Hercules: Enabling Atomic Durability for Persistent Memory with Transient Persistence Domain

CHONGNAN YE, MENG CHEN, QISHENG JIANG, and CHUNDONG WANG*,
School of Information Science and Technology, ShanghaiTech University, China and Shanghai Engineering Research Center of Energy Efficient and Custom AI IC, China

Persistent memory (pmem) products bring the persistence domain up to the memory level. Intel recently introduced the eADR feature that guarantees to flush data buffered in CPU cache to pmem on a power outage, thereby making the CPU cache a *transient persistence domain*. Researchers have explored how to enable the *atomic durability* for applications' in-pmem data. In this paper, we exploit the eADR-supported CPU cache to do so. A modified cache line, until written back to pmem, is a natural redo log copy of the in-pmem data. However, a write-back due to cache replacement or eADR on a crash overwrites the original copy. We accordingly develop Hercules, a hardware logging design for the transaction-level atomic durability, with supportive components installed in CPU cache, memory controller (MC), and pmem. When a transaction commits, Hercules *commits on-chip* its data staying in cache lines. For cache lines evicted before the commit, Hercules asks the MC to redirect and persist them into in-pmem log entries and *commits* them *off-chip* upon committing the transaction. Hercules lazily conducts pmem writes only for cache replacements at runtime. On a crash, Hercules saves metadata and data for active transactions into pmem for recovery. Experiments show that, by using CPU cache for both buffering and logging, Hercules yields much higher throughput and incurs significantly fewer pmem writes than state-of-the-art designs.

CCS Concepts: • **Hardware → Emerging architectures**; • **Computer systems organization → Embedded hardware**.

Additional Key Words and Phrases: Atomic Durability, Persistent Memory, Transient Persistence Domain

1 INTRODUCTION

A few companies have shipped byte-addressable *persistent memory* (pmem) products that are put on the memory bus for CPU to load and store data [1–7]. In order to popularize the use of pmem, Intel and other manufacturers have gradually upgraded architectural facilities. Intel introduced more efficient cache line flush instructions (e.g., `clwb`) to substitute the legacy `clflush` [8–10]. Cache line flush enables programmers to flush modified cache lines to the *persistence domain*, in which data can be deemed to be persistent upon a power outage [9, 11–13]. The concept of persistence domain was initially linked to the feature of Asynchronous DRAM Refresh (ADR). ADR keeps DRAM in self-refresh mode and, more important, places pmem and the write pending queue (WPQ) of memory controller (MC) in the persistence domain [14–16], as it guarantees to flush data staying in the WPQ to pmem in case of a power outage. Later Intel extended ADR as *eADR* that further manages to flush all cache lines to pmem on a crash [11, 12, 17–19]. As a result, eADR frees programmers from manually flushing cache lines to pmem. Platforms with the eADR feature are commercially available today. However, eADR factually builds a *transient persistence domain*, because the eventual persistence of data buffered in WPQ entries and CPU cache

[*] C. Wang is the corresponding author (cd_wang@outlook.com).

Authors' address: Chongnan Ye, yechn@shanghaitech.edu.cn; Meng Chen, chenmeng@shanghaitech.edu.cn; Qisheng Jiang, jiangqsh@shanghaitech.edu.cn; Chundong Wang, cd_wang@outlook.com,
School of Information Science and Technology, ShanghaiTech University, Shanghai, China and Shanghai Engineering Research Center of Energy Efficient and Custom AI IC, Shanghai, China.

1539-9087/2023/7-ART
https://doi.org/10.1145/3607473

lines is made by an uninterruptible power supply flushing all such data to pmem. It is important to note that this type of cache should not be confused with a fully persistent cache, such as the one made of emerging NVM technologies like STT-RAM [20], which is supposed to maintain factual persistence in the long run, without the need of eADR and pmem for persistence.

The advent of pmem has motivated programmers to directly operate with persistent data in pmem. It is non-trivial to enable the *atomic durability* for in-pmem data regarding unexpected system failures, e.g., a power outage. For example, inserting a key to a sorted array is likely to move existing keys that may span a few cache lines. Programmers make such an insertion into a *transaction* that shall be atomically modified as a unit. In other words, the change of cache lines for the insertion must be done in an all-or-nothing fashion. If a crash occurs, after reboot involved cache lines should either contain all keys including the new one, or retain only original keys without any movement. Programmers can use the software logging strategy to back up data for a transaction. Whereas, software logging is ineffectual. Firstly, it incurs double writes, which impair both performance and lifetime for pmem products [21–27]. Secondly, it executes extra instructions for logging and consumes more architectural resources, such as double WPQ entries and pmem spaces for data and log copies. Thirdly, software logging must explicitly enforce the ordering of persisting log copies before updating data through memory fence instructions (e.g., `sfence`) to render the backup copy reliable. The eADR helps to avoid cache line flushes but still necessitates the use of memory fences, which are costly for achieving the in-pmem atomic durability [9, 28–30].

Computer architects have explored how to enable the atomic durability in various hardware approaches for applications to gain the data consistency with pmem [14, 15, 20, 29, 31–38]. They mostly exploit either a redo or undo log copy, or both, for data to be atomically modified in a hardware-controlled transaction. Some of them considered persistent CPU caches made of non-volatile memory (NVM) technologies to keep redo log copies [20, 29]. Some others used in-pmem areas for logging and added on-chip redo or undo log buffers, or both, within the cache hierarchy [32, 33, 35–38]. Recently researchers explored the transient persistence domain of limited WPQ entries protected by the ADR feature to temporarily hold log or data copies [14, 15].

In this paper, we consider leveraging the eADR-supported CPU cache to enable the atomic durability for applications. The eADR guarantees all cache lines to be flushed back to pmem on a power outage, thereby promising substantial space in numerous megabytes to secure crash recoverability for applications. Modified data staying in a cache line is a natural redo log of the in-pmem copy. However, a normal cache replacement or the eADR on a power failure writes the cache line back to its home address. If the cache line belongs to an uncommitted transaction, the transaction cannot be recovered, as the overwrite destroys the original copy. We hence develop **Hercules** to overcome this challenge with supportive hardware components and comprehensive transactional protocols. The main points of Hercules are summarized as follows.

- Hercules makes CPU cache be both the working memory and main transaction log. It enhances a part of CPU cache lines with *transactional tags* (TransTags) and manages an in-pmem log zone holding transaction profiles and log entries for spatial extension and emergency use. It also customizes the MC between CPU and pmem to handle cache lines evicted due to cache replacement or eADR on a power-off.
- Hercules places data that programmers put in a transaction into cache lines with TransTags. On a transaction's commit, Hercules *commits on-chip* the transaction's data buffered in CPU cache by modifying TransTags. For cache lines evicted before the commit, Hercules makes the MC map and persist them to in-pmem log entries. It keeps their mappings for proper reloading until the commit, at which it *commits* them *off-chip* by changing their states in the MC. Then Hercules silently migrates them to their home addresses.
- A crash initiates the emergency use of in-pmem log zone. With eADR, Hercules dumps cache lines with TransTags and mappings in the MC into a dedicated area of log zone. To recover, it discards uncommitted transactions and carries on unfinished data write-backs for committed transactions.

Hercules exploits CPU cache to log and coalesce data updates. Only on cache replacement or power outage will Hercules passively flush cache lines, which is in contrast to prior works that proactively write undo or redo log copies to pmem for backup. As a result, Hercules both achieves high performance and minimizes pmem writes. We have prototyped Hercules within the gem5 simulator [39] and evaluated it thoroughly with micro- and macro-benchmarks. Experimental results confirm that Hercules well supports ordinary workloads of typical applications and inflicts the least writes to impact the write endurance of NVM. For example, running with prevalent workloads, Hercules yields about 89.2%, 29.2%, and 51.3% higher throughput on average than software logging, Kiln [20], and HOOP [37], while the data Hercules writes to pmem is 29.8%, 37.4%, and 1.4% that of them, respectively.

The rest of this paper is organized as follows. In Section 2, we brief the background of persistence domain and atomic durability. We show a motivational study in Section 3. We detail the design of Hercules in Section 4 and thoroughly evaluate it in Section 5. We conclude the paper in Section 6.

2 PERSISTENCE DOMAIN AND ATOMIC DURABILITY

2.1 Persistence Domain

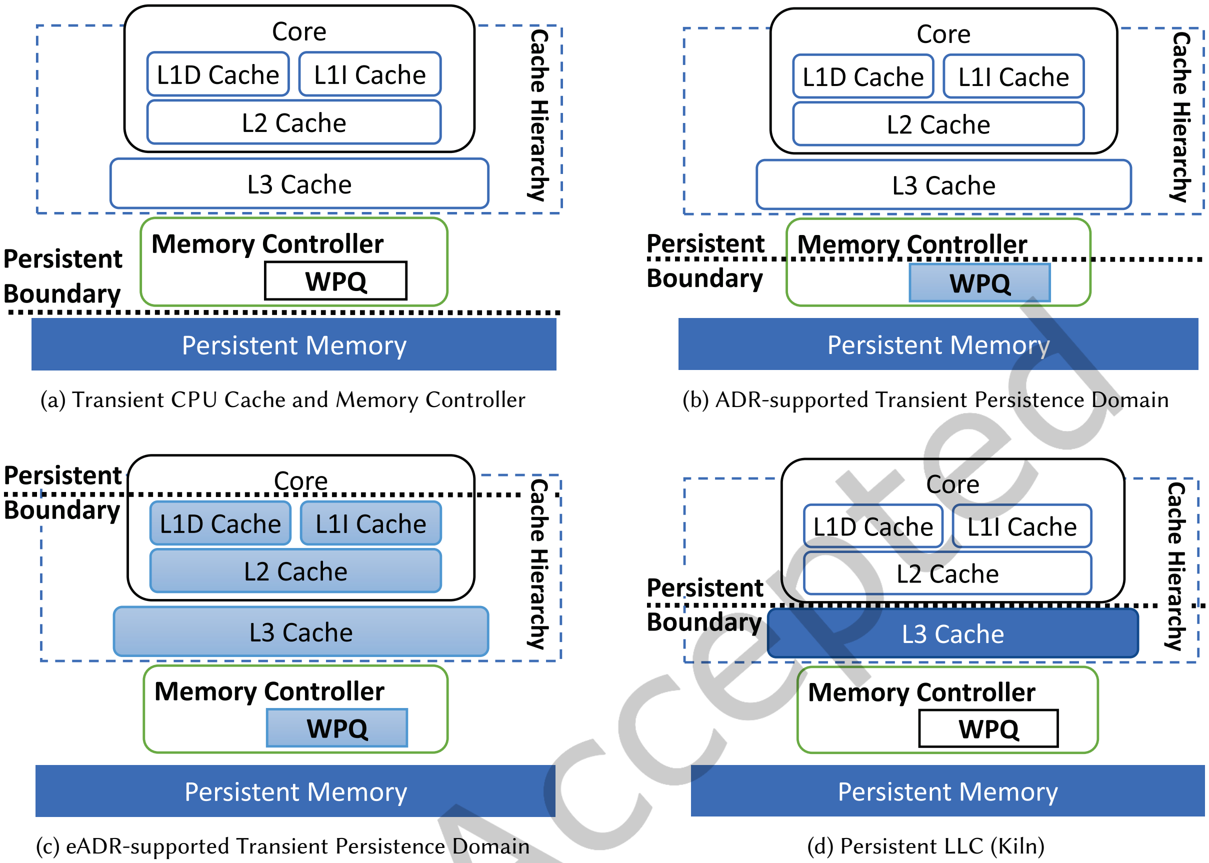
Pmem. Pmem embraces both byte-addressability and persistency. Researchers have considered building pmem with various memory technologies, such as phase-change memory [21, 23, 24, 40–43], spin-transfer torque RAM (STT-RAM) [7, 44–48], resistive RAM [25, 49–52], 3D XPoint [2, 6], and DRAM backed by flash [3–5, 53, 54]. Applications can directly load and store data with pmem [2, 9, 19, 31, 54–62].

Persistence Domain. Persistence domain is a region of computer system in which data would not be lost but retrievable when the system crashes or power failures occur [9, 17]. It conventionally includes disk drives at the secondary storage level. As shown in Figure 1a, the advent of pmem brings it up to the memory level.

ADR. The ADR further extends the persistence domain to the WPQ of MC [12, 14–16]. As shown in Figure 1b, ADR guarantees that data received at the WPQ can be flushed to pmem upon a power outage. Though, the persistence enabled by ADR is transient, as it is the pmem that eventually makes data persistent. Also, CPU cache is still volatile and cache lines would be lost on a crash. Thus, programmers must explicitly flush data staying in cache lines (e.g., `clwb`) to pmem. Flushing data from CPU cache to pmem is not only synchronized and time-consuming, but also error-prone and hurts programmability [18, 63].

eADR. Intel extended ADR as eADR which guarantees to flush all cache lines to pmem in case of a power outage by employing extra power supply [9, 17, 64]. Figure 1c captures CPU cache hierarchy with the eADR support. Alshboul et al. [18] proposed BBB that employs a **b**attery-**b**acked persist **b**uffer alongside each core's L1D cache and achieves an identical effect as eADR with much less cost. Its purpose is to establish a persistent buffer with battery backed, installed adjacent to the L1 data cache on each core. As data is written to cache, the corresponding store value is allocated in the battery-backed buffer and incorporated into the persistence domain. Through this process, BBB effectively bridges the gap between the point at which data becomes visible and the point at which it achieves persistency. BBB and eADR help programmers avoid explicit cache line flushes. More important, they make the multi-level CPU cache hierarchy provide a transient persistence domain in dozens of megabytes on top of pmem.

Whole system persistence. Motivated by the development of NVM technologies, Narayanan and Hodson [65] proposed whole-system persistence (WSP) with flush-on-fail. WSP flushes all data in the heap, stack, and thread context state from cache lines and processor registers to pmem in case of a power outage, effectively converting it into a "suspend/resume" event. WSP's developers claimed that it is *the best use* of pmem. However, the technique is limited in terms of system and application recovery from error states, and storing all intermediate state information is a challenging task for computer systems, considering issues such as out-of-order (O3) execution in multi-core CPU, continuous context switches between processes within the OS, hypervisors and



(a) Transient CPU Cache and Memory Controller

(b) ADR-supported Transient Persistence Domain

(c) eADR-supported Transient Persistence Domain

(d) Persistent LLC (Kiln)

Fig. 1. An Illustration of Persistence Domains Explored in Previous Works and This Paper

hardware/software virtualization, interactions across kernel- and user-spaces, etc. This could be one reason why Intel has chosen to enable the eADR for mainly flushing cache lines to pmem.

RTM. The idea of Intel's RTM in TSX [66] aims to provide atomicity and facilitate speculative concurrency in a shared memory system. With the support of eADR, RTM can provide atomic durability and Yi et al. [67] utilize RTM with eADR to develop HTMFS that achieves both high performance and strong consistency as a pmem file system. However, Intel TSX uses L1 cache to buffer transactional reads and writes. As a result, RTM has limitations in terms of the sizes of read and write sets for a transaction, which may cause transactions to abort. Other circumstances such as conflict and interrupts would abort transactions also. In addition, an operation (e.g., write) in the file system has a complex code path. HTMFS splits one such operation into smaller pieces and proposes a new mechanism called HOP to address the capacity limitation issue of RTM. Comparatively, the entire cache hierarchy in scores of megabytes is more capacious than L1 cache alone and, if well explored with the emerging eADR, should help to enable more generic support of atomic durability for various systems and applications beyond file system. This is factually one of the main objectives of Hercules.

2.2 Atomic Durability

The atomic durability, or failure-atomic durability, refers to the crash consistency of modifying in-pmem data in case of a crash. The insertion with an in-pmem sorted array mentioned in Section 1 is a typical transaction programmers would define with their desired semantics. A transaction must be done in an atomic (all-or-nothing) fashion. Otherwise, a half-done change may leave data in ambiguity or uncertainty after a crash.

2.2.1 Software Solution.

Modern 64-bit CPUs allow an atomic write of up to 8 bytes. Programmers bundle multiple data operations for a task in one transaction and seek software or hardware solutions. Software logging is a common technique. Programmers explicitly record original (resp. modified) data in an undo (resp. redo) log. However, software logging is not effectual with several factors. Firstly, logging incurs double writes due to writing both log and data copies [20, 35, 68, 69]. Double writes jeopardize performance and impair lifetime for NVM technologies that have limited write endurance [21–27]. Secondly, logging demands extra instruction to be executed. Log and data copies also consume more architectural resources. For example, they need double locations in pmem. If programmers use CPU cache to buffer them, they take double cache lines. Thirdly, the ordering of writing log copies prior to data must be retained by using memory fences, the cost of which, albeit the presence of eADR, is essential and substantial [28–30].

2.2.2 Hardware Designs.

The essence of gaining atomic durability is to make a backup copy before modifying data in place. In order to back up data, state-of-the-art hardware designs explore different persistence domains, which categorize them into three classes.

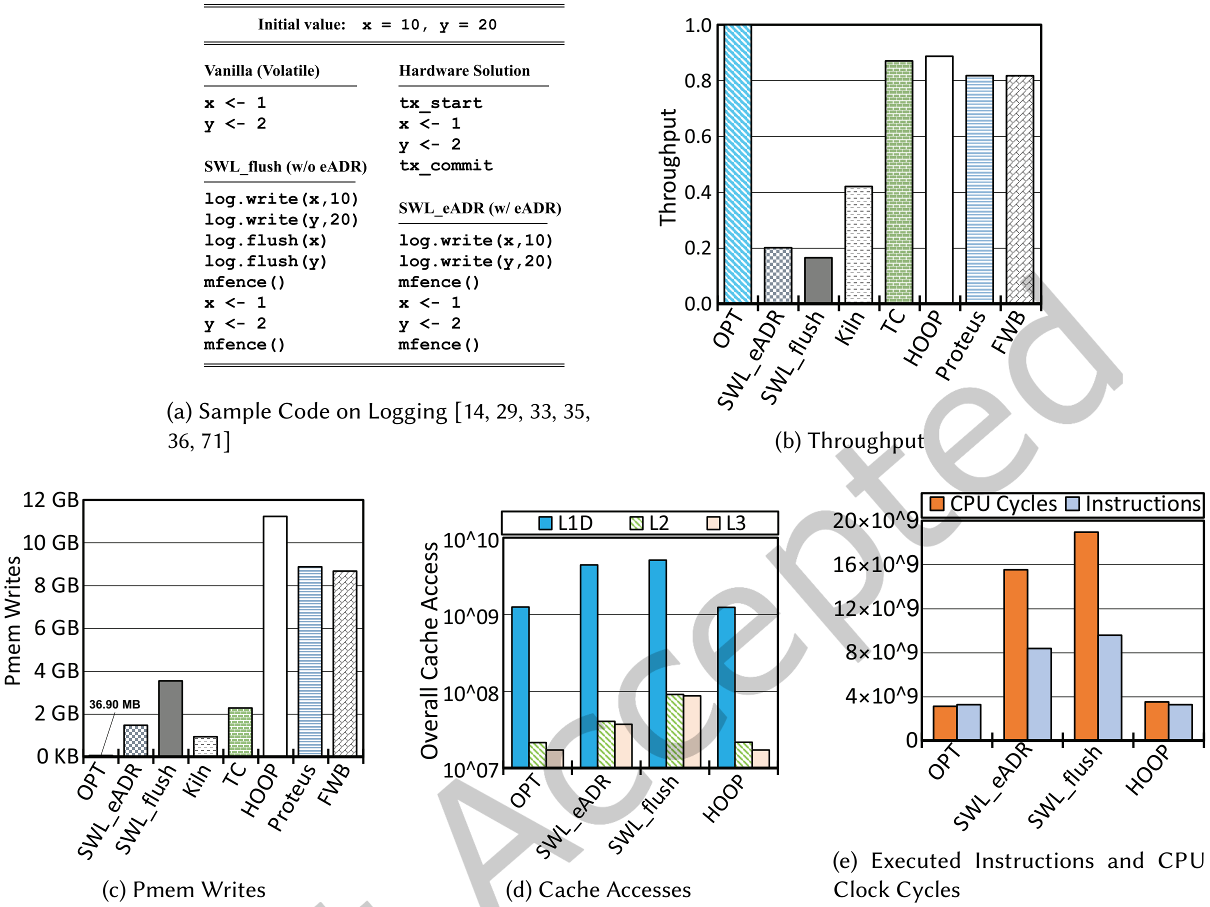
Persistent CPU cache. As shown in Figure 1d, Kiln works with a persistent last-level cache (LLC) [20]. The persistence domain covers LLC and pmem. Kiln manages redo log copies in LLC to back up in-pmem data. Later, Lai et al. [29] employed a side-path persistent transaction cache (TC) along L1 cache in each CPU core. TC is similar to the persistent buffers used in other works [18, 70]. Modified cache lines of a transaction are first-in-first-out (FIFO) put in the TC and serially written to pmem on committing a transaction.

Pmem. A few hardware designs were built on a pmem-only persistence domain. Doshi et al. [32] proposed to use a victim cache to hold evicted cache lines that would be subsequently written to an in-pmem redo log. These cache lines are eventually written to their home addresses by copying log entries via non-temporal stores. Similarly, Jeong et al. [35] proposed ReDU that utilizes a DRAM cache to hold evicted cache lines from the LLC. ReDU directly writes modified data from DRAM cache to home addresses, as it installs a log buffer alongside L1 cache to collect and write the in-pmem redo log.

Cai et al. [37] designed HOOP with a physical-to-physical address indirection layer in the MC, which helps it write modified data to a different pmem address for backup and later move data to home addresses. Joshi et al. [33] noted that the MC loads a cache line for a write request and proposed ATOM to write the loaded copy

Table 1. A Summary of State-of-the-art Hardware Designs that Provide Atomic Durability for Pmem

Persistence Domain	Representative Project	Logging Type	Pmem Writes	Read Latency	Granularity	Limitation
Pmem only	ReDU [35]	Redo	High	Low	Word	Consuming DRAM capacity
	ATOM [33]	Undo	High	Low	Cache line	Long critical path
	HOOP [37]	Redo	High	Low	Word	Limited OOP buffer
	PiCL [34]	Undo	High	High	Cache line	Does not support transaction
	FWB [36]	Redo+Undo	High	High	Word	Demanding force write back
ADR + pmem	Proteus [14]	Undo	High	Low	32B	Long critical path
	MorLog [38]	Redo+Undo	Medium	High	Word	Long commit latecy
	LAD [15]	Redo	High	Medium	Cache line	Complex commit phrase
Persistent cache + pmem	Kiln [20]	Redo	Medium	High	Cache line	High STT-RAM latency
	TC [29]	Redo	Medium	Medium	Cache line	Demanding flush TC when commit
eADR + pmem	Hercules (this paper)	Redo	Low	Low	Cache line	Requiring modify cache, MC, and pmem



```
Initial value:  x = 10, y = 20

Vanilla (Volatile)          Hardware Solution
x <- 1                      tx_start
y <- 2                      x <- 1
                            y <- 2
SWL_flush (w/o eADR)        tx_commit
log.write(x,10)
log.write(y,20)             SWL_eADR (w/ eADR)
log.flush(x)                log.write(x,10)
log.flush(y)                log.write(y,20)
mfence()                    mfence()
x <- 1                      x <- 1
y <- 2                      y <- 2
mfence()                    mfence()
```

(a) Sample Code on Logging [14, 29, 33, 35, 36, 71]

(b) Throughput

(c) Pmem Writes

(d) Cache Accesses

(e) Executed Instructions and CPU Clock Cycles

Fig. 2. A Study on Software Logging Logging with/without eADR and State-of-the-art Hardware Designs for Atomic Durability

to an in-pmem undo log in a parallelized manner. Nguyen and Wentzlaff [34] proposed PiCL that also uses the idea of undo logging with an on-chip log buffer. PiCL makes a trade-off between performance and durability by snapshotting and saving data in an epoch-based periodical checkpointing manner [45]. Ogleari et al. [36] and Wei et al. [38] both chose the undo+redo logging approach. Ogleari et al. captured data's redo and undo log copies from the in-flight write operation and the write-allocated cache line, respectively. They also used a force write-back (FWB) mechanism to control pmem writes. Wei et al. studied data encoding with hardware logging so as to only record necessary changes for a transaction's data, thereby reducing pmem writes. Both designs add on-chip undo and redo log buffers.

ADR-supported transient persistence domain. The ADR places the WPQ of MC in the transient persistence domain. Shin et al. [14] designed Proteus that considers the WPQ to keep log copies. When a transaction commits, Proteus discards relevant log copies in the WPQ and hence reduces pmem writes. Gupta et al. [15] proposed LAD that also leverages the ADR-supported MC as a staging buffer to accumulate data updates before committing a transaction. Whereas, for Proteus and LAD, the limited capability of WPQ entails a high likelihood of falling back to the use of an in-pmem log.

Table 1 summarizes the characteristics of current hardware designs and Hercules that ensure atomic durability for pmem. As to be shown, Hercules stands out from prior works by utilizing eADR-supported cache hierarchy to coalesce transaction updates, resulting in high performance, reduced pmem writes, and promising efficiency.

3 MOTIVATION

We consider leverage the transient persistence domain in CPU caches made by eADR and BBB to enable atomic durability. We have conducted a motivational study to analyze the potential gain introduced by using CPU cache to process transactions. Figure 2a captures four sample code snippets, i.e., a vanilla 'volatile' program without guarantee of atomic durability, two software undo logging versions *without* and *with* the eADR (both using CPU cache to buffer log copies), and one version using a typical hardware transactional design. Accordingly we have tailored B+-Tree with the volatile version for the optimal performance (OPT), two versions of software logging (SWL_eADR and SWL_flush), and five prior hardware designs (see Figure 2b). We note that the `mfence` instructions are necessary for software logging with the eADR (SWL_eADR). In modern processors, the CPU cache is the point of visibility (PoV) for memory consistency and cache coherence [18, 28, 30]. Generally a CPU core employs a local load/store buffer above the core's L1 cache, which hence demands the use of `mfence` instructions to keep memory writes orderly and globally visible to all cores. Currently, this load/store buffer is not part of eADR domain. By using `mfence` for SWL_eADR, logged data becomes persistent prior to data in-place updated at home addresses. SWL_eADR thus gains the recoverability to a consistent state after a crash (e.g., power outage). We run all eight designs with gem5 [39] to insert one million key-value (KV) pairs (8B/8B for K/V). We set each insertion as one transaction. Section 5 would detail our evaluation setup and methodology. Figure 2b, Figure 2c, Figure 2d, and Figure 2e show a quantitative comparison on the throughputs normalized against that of OPT, the quantity of pmem writes, and other CPU execution results, respectively. We can obtain three observations from these diagrams.

𝔒1: The eADR improves the performance of software logging and using CPU cache significantly reduces pmem writes. Comparing SWL_eADR to SWL_flush in Figure 2b tells that the avoidance of cache line flushes makes SWL_eADR gain 22.0% higher throughput. This performance improvement justifies the usefulness of transient persistence domain for atomic durability in the software logging approach. Moreover, as shown in Figure 2c, except Kiln that employs a large persistent LLC for buffering, the quantity of data written by SWL_eADR is 64.3%, 16.8%, 16.4%, and 12.9% that of TC, FWB, Proteus, and HOOP, respectively. The reason is twofold. Firstly, the eADR-supported CPU cache in sufficient megabytes holds both log and data copies over time, so SWL_eADR substantially brings down data to be written to pmem. Secondly, hardware designs mostly need to write backup copies to pmem for crash recoverability, because they have been developed without a transiently persistent CPU cache hierarchy. Therefore, hardware designs generally incur much more pmem writes than SWL_eADR.

𝔒2: Compared to software logging with eADR, hardware designs gain higher performance without the use of eADR, which indicates the potential of a new hardware design utilizing extensive CPU cache for atomic durability. Double writes make a crucial innate defect for software logging. As shown in Figure 2b, despite no explicit flush of data with eADR, SWL_eADR is still inferior to hardware designs. Without loss of generality, we take HOOP as a representative for illustration. Figure 2d and Figure 2e capture the accesses to L1D/L2/L3 caches and the number of instructions and clock cycles, for OPT, SWL_flush, SWL_eADR, and HOOP, respectively. SWL_eADR underuses the eADR-supported CPU cache, incurring 257.5%, 86.3%, and 115.4% more loads and stores to L1D, L2, and L3 caches than HOOP, respectively. HOOP conducts address indirection in the MC for hardware-controlled out-of-place backups. Consequently, it performs backup operations without using CPU cache and achieves comparable cache accesses and instructions against OPT. Due to the unawareness of CPU cache used as an ample transient persistence domain, hardware designs like HOOP, Proteus, and FWB must directly write data into pmem for backup or rely on limited WPQ entries. To sum up, SWL_eADR wastes valuable cache space despite the boost of eADR while prior hardware designs did not foresee transiently persistent CPU caches.

𝔒3: The eADR provides abundant transient persistence domain to facilitate achieving atomic durability with the potential of high performance and reduced pmem writes. As shown in Figure 2b, an evident gap still exists between OPT and hardware or software designs. The eADR-supported CPU cache is certainly a promising feature with a transiently persistent space in dozens of megabytes. As justified by our test results, SWL_eADR does not make the most out of it, while no hardware design has ever exploited it. Kiln, one using STT-RAM as the persistent LLC, implicitly manifests the potential of eADR-supported CPU cache. The throughput of Kiln is not high, partly because of the slower access latency of STT-RAM compared than that of SRAM (see Figure 2b). Yet due to the higher density of STT-RAM, Kiln's LLC can absorb more pmem writes (see Figure 2c). Additionally, platforms with the eADR feature are commercially available today, while STT-RAM-based cache is being under development.

These observations motivate us to consider how to utilize the eADR-supported CPU cache when developing a hardware design to efficiently guarantee the atomic durability for applications. A modified cache line and its in-pmem copy naturally form a pair of redo log and backup copies, which implies an opportunity for hardware logging. However, the very nature of transient persistence alludes a challenge. Let us assume that we directly use the transiently persistent CPU cache to make a redo log. In case of a cache replacement or power outage, the eADR writes a cache line back to its home address. For data belonging to an uncommitted transaction, the write-back destroys the intact backup copy in pmem and renders the transaction unrecoverable. As a result, to achieve atomic durability, we need to ensure that cache lines of an uncommitted transaction should be written elsewhere on write-backs. Also, we shall make the most out of CPU cache to simultaneously hold data and log copies for minimizing pmem writes. These summarize Hercules' main tactics and aims.

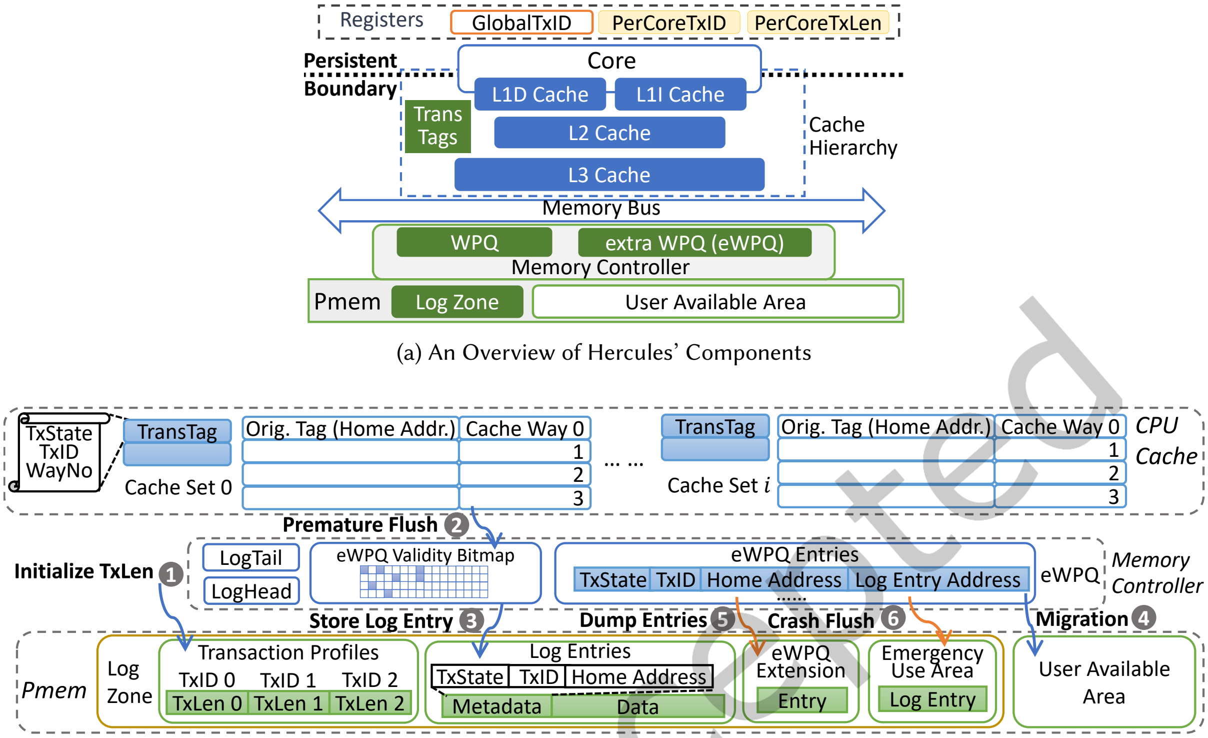
4 THE DESIGN OF HERCULES

Overview. Hercules is a hardware design to provide transactional support with a typical multi-level cache hierarchy. It makes a transiently persistent CPU cache hierarchy function both as working memory and hardware-controlled redo log. Hercules does not eagerly evict cache lines [72] but follows the conventional way of evictions upon a full cache. This is to avoid unnecessary pmem writes with regard to the limited write endurance of NVM technologies [21–27, 73]. Hercules installs *transactional tags* (TransTags) to a part of cache lines to hold data for transactions. When a transaction commits, Hercules *commits* data tracked by TransTags *on-chip* to reduce pmem writes. On evicting cache lines of an uncommitted transaction to pmem, it places them in an in-pmem log zone rather than their home addresses to avoid overwriting original data. It manages and *commits* them *off-chip* upon a committing request through managing an extended WPQ (eWPQ) in the MC. With a suite of self-contained transactional protocols, Hercules efficiently achieves atomic durability with minimized pmem writes and collaboratively works with other architectural mechanisms, such as cache replacement (LRU) and inclusion (both inclusive and non-inclusive).

4.1 Hercules' Hardware Components

Hercules uses components distributed in CPU cache, MC, and pmem to jointly control the procedure of transactions and manage the versions of data for each transaction. Figure 3a captures the main components of Hercules.

TransTag. A transaction of Hercules is a contiguous series of data operations covering one or multiple cache lines. In order to manage a transaction's data, Hercules adds TransTags per cache beside normal tags. A TransTag has `WayNo`, `TxID`, and `TxState`. TransTags in a cache set are shared among the set and a TransTag can be dynamically associated with any cache line. Once a transaction commits, the association between involved



(a) An Overview of Hercules' Components

(b) Detailed Data-paths and Operations of Hercules

Fig. 3. An Illustration of Hercules' Components and Operations

TransTags and cache lines is terminated by resetting the TxStates in the TransTag to be '0's. The WayNo in each TransTag is used to associate which cache line in the cache set Hercules is using for a transaction, which is identified with a unique 21-bit TxID. The reason why 21 bits are used for TxID is twofold. Firstly, Hercules demands extra costs such as energy, transistors, and wires to enable the atomic durability. More than 21 bits per TxID may further increase such costs and entail more challenges to achieve systematic efficiency and reliability (see Sections 5.4). Secondly, 21 bits support up to 2^{21} (≥2 million) transactions, which can satisfy the needs of typical applications at runtime.

The TxState in one bit shows if the transaction is committed or not. We call a cache line being occupied by an uncommitted transaction *transactional cache line*, of which the TxState is '1'. Otherwise, it is non-transactional without a TransTag or with TxState being '0'. In short, we configure a logical part of a cache set by filling WayNos for the use of Hercules, without fixing some cache lines for transactions. This brings in spatial efficiency and flexibility. Given a small amount of transactional data, non-transactional data can freely take cache lines with TxStates unset. Also, non-transactional and transactional data can be placed without swaps. For the illustrative example in Figure 3b, half cache lines of a four-way set (WayNo in two bits) are usable with two TransTags that cost $2 \times (21 + 2 + 1) = 48$ bits per set.

Hercules manages and uses TransTags in a similar fashion of the directory used to track cache lines for coherence [74–78]. It places TransTags alongside L1D and L2 caches in each core and has a bunch of them for the shared LLC. This organization accelerates filling and changing fields in them. The ratio of TransTags per cache set is defined as the maximum percentage of installed TransTags over all cache lines (ways) in the set ($\frac{\text{Number of TransTags installed in a set}}{\text{Number of ways existing in a set}} \times 100\%$). Note that a transactional cache line can be used with `TxState` being '1', so the ratio of TransTags addresses an upper bound of cache lines that can be simultaneously employed by Hercules for transactional data in a cache set. If the number of cache lines associated with TransTags in a set reaches such an upper bound, i.e., with all TransTags occupied with `TxState` being '1', while Hercules needs to allocate for new transactional data, cache replacement would happen to a transactional cache line in the set with regard to a replacement policy among transactional ones [79–83]. As suggested by Figure 2d, we shall consider a higher TransTags ratio for more usable space to caches that are closer to CPU, as they serve more transactional requests. The ratio of TransTags is thus practically decided by a cache's proximity to CPU. In practice, we allot TransTags to all cache lines of L1D cache. The ratio can be half or a quarter for L2 and L3 caches. We consider this ratio setting in implementation for two reasons. Firstly, the L1 cache is a crucial component that plays a significant role in securing spatial and temporal locality, as it takes the most access requests raised by CPU. A higher TransTag ratio for caches that are closer to CPU cores thus backs Hercules's efficiency of transaction processing. Secondly, cache lines evicted to L2 and LLC evidently have a lower likelihood of reuse (otherwise they would stay in L1 cache) [84, 85]. The need of maintaining a high ratio of TransTags for them lacks necessity and cost-efficiency. In other words, reducing the TransTags ratios in lower-level caches helps to save on-chip space, simplify circuit design, and decrease spatial cost for TransTags. We have a quantitative discussion on the TransTag ratio in Section 5.2.

Registers. Hercules adds a register `GlobalTxID` that is monotonically increasing, shared by all cores to compose the next transaction ID. Hercules installs `PerCoreTxID` and `PerCoreTxLen` to each CPU core, holding the current running transaction's ID and length (number of transactional cache lines), respectively. These two are critical for the context switch between threads and stay dormant for threads doing non-transactional operations. A register has 64 bits and Hercules uses the lower bits only for extension. They are kept volatile, not saved in pmem on a crash (see Section 4.3).

eWPQ. Cache lines are evicted over time. Hercules regularly writes back non-transactional ones but handles transactional ones specifically in order not to harm in-pmem original copies. It adds two registers and an extended WPQ (*eWPQ*) along WPQ in the MC. `LogHead` records the address of next available log entry in the in-pmem log zone while `LogTail` points to the last valid log entry (see Section 4.5). On evicting a transactional cache line, the MC allocates a log entry to put data of the cache line by atomically fetching and increasing `LogHead` by one, which is not time-consuming. Hercules retains that cache line's metadata in an eWPQ entry, including `TxID`, `TxState`, and and the mapping from home address to log entry's address. The eWPQ structure comprises a validity map and 512 entries by default. The validity map is used to efficiently allocate and deallocate eWPQ entries. Searching among eWPQ entries follows the mature addressing manner used for store buffer (SB), line fill buffer (LFB), and WPQ that are widely employed in existing architectures [18, 86, 87].

Log zone. Figure 3b shows four areas of in-pmem log zone. The first area (transaction profiles) is an array of transaction lengths (`TxLens`) indexed by `TxIDs`. Hercules keeps the runtime length of a transaction in the corresponding thread's context while it only allows two legal values for the in-pmem `TxLen`, i.e., an initial zero and a non-zero eventual length. It uses the atomic change of `TxLen` to be non-zero to mark the commit of a transaction (see Sections 4.2 and 4.3). Using a transaction's `TxID` to index and find the transaction's `TxLen` is fast and brings about lock-free parallel loads/stores for concurrent transactions. We set a `TxLen` in 4B, so a transaction can cover up to 2^{32} (≥4 billion) cache lines.

In the second area, a log entry keeps a transactional cache line evicted from LLC with its data and metadata, including the home address and `TxState`. The third area of eWPQ extension is used to store evicted eWPQ entries in case that there are overwhelming evicted transactional cache lines, which, however, rarely happen in accordance with our tests on practical workloads (see Section 5.2 for more detail). The next area of emergency use is used upon an unexpected crash. We show a contiguous log zone in Figure 3b while it can be partitioned to support concurrent accesses with distributed MCs [15].

4.2 Hercules' Transaction

Primitives. Like prior works [14, 20, 29, 33, 36, 37], Hercules has three primitives for programmers to proceed a transaction, i.e., `tx_start`, `tx_commit`, and `tx_abort` to respectively start, commit, and abort a transaction in applications. Listing 1 shows main steps that Hercules composes for these primitives.

Let us first illustrate how Hercules proceeds a transaction in an *optimistic* situation, i.e., 1) the transaction manages to commit, 2) all cache lines of the transaction stay in CPU cache until the commit, i.e., with no cache line evicted in the entire course of transaction, 3) Hercules can find a cache line with TransTag available whenever needed, and 4) no crash occurs.

```
switch (primitive) {
  primitive: tx_start
    Initialize TxLen to 0.
    Initialize PerCoreTxLen to 0.
    Initialize PerCoreTxID with the value of GlobalTxID.
    Atomically increase GlobalTxID.
  primitive: tx_commit
    Store PerCoreTxLen to TxLen in pmem atomically.
    Reset the TxState for transactional lines and eWPQ entries of the corresponding
    transaction.
  primitive: tx_abort
    Invalidate all transactional lines of the transaction.
    Invalidate transaction's eWPQ entries and recycle the corresponding log entries.
    Set all corresponding transactional lines' TxState to 0.
}
```

Listing 1. Pseudocode of Handling Transactional Primitives

Optimistic procedure. On a `tx_start`, Hercules atomically fetches a `TxID` from the `GlobalTxID` and increments the register by one for subsequent transactions. Hercules finds its entry with `TxID` in the array of transaction profiles and initialize the `TxLen` to be zero (➊ in Figure 3b). Until `tx_commit` is encountered, Hercules manages and processes cache lines for data that programmers put in the transaction. It adopts the write-allocate caching policy. Modifying data causes a cache miss if the data is not in CPU cache and Hercules allocates a cache line with TransTag before loading the cache line from pmem. It then fills `TxID`, `WayNo`, and data and sets the `TxState` as '1'. If data already stays in a clean cache line, Hercules obtains and configures a TransTag. Given a dirty cache line which might be originally non-transactional or belong to a committed transactions, Hercules sends that line to the next level in the memory hierarchy, e.g., L1D to L2, before getting a TransTag, in order not to taint the latest update (see 'Write-allocate at L1' event shown at Line 2 in Listing 2). When the transactional cache line is evicted to the lower level, Hercules first sends the older dirty non-transactional version to the next lower-level cache or pmem in an asynchronous manner, without any considerable stall incurred to access latency [64, 88]. In case of a crash, the eADR flushes cache lines in the reverse order of, say, L3, L2, and L1, since the latest updated data stays in higher-level caches that are closer to CPU cores. As a result, the dirty non-transactional cache lines in the lower level would be firstly persisted to their home addresses. The upper-level transactional cache line, if committed, refills the home address; if not, Hercules writes it to the area of emergency

```
switch (event) {
  event: Write-allocate at L1
    if (The line is non-transactional line) {
      Atomically increase PerCoreTxLen.
      if (Line's dirty bit is 1) {
        Send a copy of the cache line to the next level.
      }
    }
    Set the transaction information in TransTag.
    Write the data.
  event: Write victim at L2 or L3
    if (No space to place data) {
      Process replacement event at L2 or L3.
    }
    if (Line's TxState is 0 and dirty bit is 1) {
      Send a copy of the cache line to the next level.
    }
    Set the transaction information in TransTag.
    Write the data.
  event: Write victim to pmem
    if (Victim's TxState is 1) {
      if (eWPQ is full) {
        Process replacement event at eWPQ.
      }
      Find an entry in eWPQ.
      Increase register LogHead in MC and allocate an entry in log zone.
      Write the victim to log zone.
      Write transaction information in eWPQ.
    } else {
      Write the victim to its home address.
    }
}
```

Listing 2. Pseudocode of Handling Write Request

use (to be presented). This rules out any inconsistency and uses lower-level caches for staging to further reduce pmem writes.

Hercules follows generic rules in programming transactions. It disallows nested or overlapped transactions in one thread, so at most one transaction is ongoing within a thread. Following previous works [14, 20, 32, 33, 37], programmers may consider concurrency control mechanisms like locks or semaphores between transactions in multi-threading programs.

When a thread enters a transaction for the first time, the transaction's TxID and length are used to fill PerCoreTxID and PerCoreTxLen of the running CPU core, respectively. PerCoreTxLen is incremented by one every time Hercules is going to launch a transactional update on an uncovered cache line. If a context switch occurs, the values of PerCoreTxID and PerCoreTxLen are saved as part of the thread's context for an afterward execution. On committing a transaction, Hercules atomically sets the in-pmem TxLen with PerCoreTxLen and resets the TxState to be '0' for each transactional cache line to make it visible to other threads.

The foregoing procedure shows that Hercules efficiently handles a transaction and *commits on-chip*. The eADR guarantees committed cache lines would be flushed to their home addresses in case of a crash. Next we present how Hercules handles conditions not covered in the optimistic circumstance.

Premature flush. The eADR enables Hercules to write back data on cache replacements rather than explicit cache line flushes, so updates to a cache line are coalesced and pmem bandwidths are saved. Hercules handles the write-back of a non-transactional cache line with the MC's WPQ in the ordinary way. For an evicted cache line recorded in a TransTag, if `TxState` is '0', i.e., being non-transactional, the MC writes back the cache line to the home address. If `TxState` is '1', Hercules initiates a *premature flush* with the MC's eWPQ (❷ in Figure 3b). As shown in Figure 3b, the eWPQ is made of eWPQ entries and an eWPQ validity bitmap to track the validity status of each eWPQ entry. The MC finds a free eWPQ entry for the evicted transactional cache line and allocates a log entry by atomically fetching and increasing the `LogHead`. The MC copies `TxID`, `TxState`, home address, and log entry's address to the eWPQ entry and asynchronously writes back the transactional cache line in the log entry (❸ in Figure 3b).

Hercules employs the eWPQ both for logging uncommitted data and loading proper data. When a thread resumes execution, it may use a cache line that has been prematurely flushed. The cache line may be from any transaction that is already committed or this resumed transaction. Hercules references cache lines regarding their home addresses. On a load request, the MC checks if the target address matches any eWPQ entry and simultaneously tests the `TxState`. Given a match and '1' `TxState`, if the `TxID` in this TransTag is the same as ongoing `TxID` upon comparison, Hercules gets a potential hit. A match with '0' `TxState` is also likely a hit, and no match results in a miss. If an eWPQ entry's `TxState` is '0', Hercules should have committed the entry's corresponding cache line off-chip while the cache line may still stay in the log zone and wait for a migration. Hercules employs a background thread to migrate such committed-off-chip cache lines to their home addresses. We would present the details of commit off-chip and migration later.

Regardless of a hit or miss, once receiving a request, the MC starts loading the cache line from the home address. A hit at the eWPQ fetches the corresponding log entry and halts the load from home address. MC checks the full address stored in the log entry and forwards it to the CPU cache in case of a true match. When CPU cache receives the log entry, MC nullifies the matching eWPQ entry and validity bit in the bitmap. A miss continues the load of cache line from home address. Hercules rules out the possibility of CPU starts using obsolete data loaded from home address when the MC gets a hit at the eWPQ. The reason is that eWPQ stays in the integrated MC which is much closer to CPU cores with much shorter access latency than pmem sitting on the memory bus. In a very rare event that the eWPQ becomes full, Hercules needs to perform a search among the eWPQ extension. It waits for the completion of search such that CPU would not use any obsolete data loaded from pmem. Note that loading data simultaneously from microarchitectural buffers (e.g., store buffer or line fill buffer) and memory hierarchy (cache or main memory) is a classic optimization tactic. In fact, there have been mature architectural and microarchitectural techniques for speculation and prefetch upon loading data [89–92], which Hercules can make use of. Listing 3 shows Hercules's procedure of data loading from pmem. In addition, an access from an ongoing transaction may happen to an eWPQ entry with mismatched `TxID` and '1' `TxState`. Hercules aborts that transaction with an exception.

The other reason for employing the eWPQ is to commit and migrate cache lines that have been prematurely flushed. A transaction may commit without reusing all or part of transactional cache lines that have been evicted to pmem. Hercules exploits the eWPQ to deal with them. There are two ways to deal with such cache lines. One is to load them into CPU cache for committing on-chip and store them to home addresses by cache replacements. The other one is to *commit* them *off-chip* by resetting the `TxStates` in corresponding eWPQ entries and migrating them from log entries to home addresses via non-temporal stores. We choose the second way to reduce cache pollution. For efficiency, we periodically scan eWPQ entries for data migrations. The period is configurable, set to be every three million instructions in our tests. A completion of migrating a log entry clears the validity bit for

```
switch (event) {
  event: Read hit at private L1
    Load finishes.
  event: Read hit at private L2 or L3
    if (The line is transactional cache line) {
      Load its TransTag exclusively to L1.
    }
    Load cache line to L1.
  event: Read hit at the other core's private cache
    if (The line is transactional cache line) {
      Process event tx_abort.
    } else {
      Load cache line.
    }
  event: All cache miss, load data from pmem
    if (eWPQ is full) {
      Search eWPQ and eWPQ extension area.
      if (eWPQ hit) {
        Load data from log zone.
      } else {
        Load data from home address.
      }
    } else {
      Start eWPQ searching and home address loading parallelly.
      if (eWPQ hit) {
        Stop home address loading.
        Load data from log zone.
      } else {
        Load data from home address.
      }
    }
  event: No space to place data in L1
    Process replacement event at L1.
}
```

Listing 3. Pseudocode of Handling Read Request

the eWPQ entry (❹ in Figure 3b). A load request that happens before the reset of validity bit still fetches data from the log entry.

Previous works have justified the efficacy and efficiency of using a part of address for cache management [79–82]. We accordingly devise a compact eWPQ entry that holds one bit of TxState and three fields of 63 bits evenly partitioned for TxID, home address, and log entry's address. Our evaluation shows that transactions of typical applications are empirically small and generally take few to dozens of cache lines (see Section 5.2), so a home address and a TxID in overall 42 bits are sufficient for indexing. If duplicate matches occur, an eWPQ entry leads to a log entry that holds the full home address to rule out ambiguity. We manage the eWPQ like a fully associative cache and set a default size of 4KB for 512 entries, which are ample to serve ordinary workloads found in typical applications (see Section 5.2). We believe a larger eWPQ is practically viable [15, 16]. Yet we take into account the very low likelihood of a full eWPQ, wherein Hercules evicts the least-recently-used (LRU) entries to an in-pmem eWPQ extension area that is ten times larger than the eWPQ (❺ in Figure 3b). If a request misses in the eWPQ, Hercules checks the eWPQ extension area with a target home address to properly fetch the corresponding log entry.

Flush on a power-off. When a power-off occurs, Hercules flushes WPQ, `LogHead`, `LogTail`, eWPQ, and then all cache lines to pmem. Hercules writes non-transactional cache lines to home addresses. In case of a crash, it dumps transactional ones to the area of emergency use in the log zone with home addresses and TransTags (❻ in Figure 3b). These metadata and data are useful for recovery (see Section 4.3).

Cache replacement. Transactional and non-transactional cache lines flexibly share a cache set. In general, Hercules considers an effective algorithm [79–83] to select a victim for eviction, but entitles a higher priority to transactional ones for staying in the cache hierarchy, because this will reduce the likelihood of premature flushes that lead to pmem writes and weaken the endurance of pmem. Yet Hercules has differential tactics at different levels of cache hierarchy.

For L1 Cache, we follow standard cache replacement for Hercules to gain both spatial and temporal locality. In particular, Hercules assigns identical priority to transactional and non-transactional cache lines in a cache set. Non-transactional cache line can replace and evict a transactional one at L1 cache with a mature replacement algorithm based on, say, frequency, recency, and/or reuse distance [79–83]. For L2/L3 caches, as mentioned, Hercules gives higher priority to transactional cache lines for staying in each cache set. In other words, non-transactional cache lines are more preferred to be replaced. The reason why Hercules does so is twofold. Firstly, we find that many cache lines (blocks) reach LLC with extremely low possibility of reuse, i.e., *dead blocks*. This has been justified by prior works as well [84, 85, 93]. The eviction of dead non-transactional cache lines helps to prevent transactional cache lines from being prematurely flushed to in-pmem log. Secondly, the replacement strategy in Hercules is similar to that of Intel Cache Allocation Technology (CAT) [94, 95], which enables a certain amount of cache lines (ways) in a cache set to be reserved for the use of specific applications. We believe that users who utilize Hercules to conduct transactions are aiming for higher throughput, lower service latency, and fewer pmem writes. Therefore, prioritizing and retaining transactional cache lines is rational and useful. Listing 4 presents the policy of cache replacement for Hercules.

```
switch (event) {
  event: Replacement at L1
    Find victim through normal LRU.
  event: Replacement at L2 or L3
    if (Reach TransTag ratio limit) {
      Replace a transactional victim through LRU.
    } else {
      Replace a non-transactinal victim through LRU.
    }
  event: Replacement at eWPQ
    Find an eWPQ victim through LRU.
}
```

Listing 4. Pseudocode of Handling Replacement Request

Transaction abortion. A transaction may abort due to various events like exception, fault, or running out of memory. For example, when an odd program continually insists on demanding overwhelming pmem space, it is possible that Hercules consumes up all pmem space due to the limitations imposed by the OS and physical pmem device. Hercules aborts the transaction and terminates the program. In implementation, Hercules could set a threshold (e.g., 80% of overall pmem capacity) to earlier detect such anomalous or malicious behaviors.

Hercules also aborts a transaction if a transactional cache line is to be fetched from another core, through a comparison against TxID and TxState in the TransTag. Such a mechanism is similar to Intel's RTM in TSX [66]. On an abortion, data recorded in the TransTags and eWPQ entries with TxIDs matched and TxStates being '1's are invalidated and discarded, incurring no harm to original data.

4.3 The Crash Recoverability of Hercules

Hercules puts a specific flag in the log zone to mark a normal shutdown or not. The flag is not set if any transactional cache line is saved to the area of emergency use on power-off. Regarding an unset flag, Hercules recovers at the transaction level in order to support applications recovering with semantics. As modifying TxStates of multiple cache lines cannot be atomic, Hercules atomically sets the TxLen to commit a transaction. In recovery, it fetches eWPQ, LogHead, and LogTail into the MC and scans transaction profiles.

Hercules discards transactions with zero TxLens. As to a committed one, Hercules scans the area of emergency use to find out cache lines with TxIDs matched and TxStates being '1's. Hercules moves them to their home addresses. In addition, some cache lines of the transaction might have been prematurely flushed before the commit, being tracked by the eWPQ, but not migrated yet prior to the power-off. That explains why Hercules has saved the entire eWPQ. If an entry with a matching TxID is valid in the eWPQ validity bitmap, Hercules migrates the mapped log entry and then clears the corresponding validity bit. LogTail may be moved after the migration. Once moving all such cache lines is completed, Hercules resets the transaction's TxLen to be zero. This atomic write rules out ambiguity if a crash takes place in an ongoing recovery. After resetting GlobalTxID and clearing the eWPQ, Hercules is ready to recommence new transactions.

4.4 In-pmem Log Space Management

Garbage collection (GC) on log entries. Concurrent transactions commit at different time and take up discontinuous log entries. Committed log entries become invalid, scattered across the log zone. Figure 4 shows how Hercules cleans them up. LogHead and LogTail frame a window of log entries Hercules is using while the eWPQ tracks all valid ones. When LogTail has not moved for a while, numerous invalid entries might accumulate in the window. If Hercules monitors that the distance between LogTail and LogHead is greater than a threshold, e.g., 2^{20}, it will initiate a GC (❶❷❸ in Figure 4). Hercules fetches a chunk (e.g., 32) of successive log entries starting at LogTail (❶). It appends valid uncommitted ones to the locations pointed by LogHead and updates corresponding eWPQ entries (❷). Only after updating each eWPQ entry will Hercules move LogHead by one. Then it slides LogTail over the chunk to the next valid entry (❸). Hercules orderly performs these steps to preclude any crash inconsistency. Note that the system may crash in a GC, particularly when a power outage has happened after a movement, which results in a moved log entry existing both at LogHead and LogTail. Such a log entry can be ignored as it belongs to an uncommitted transaction with TxLen being zero, without any loss of Hercules' crash recoverability.

Log extension. In a very low likelihood with normal workloads, excessive transactional data might occasionally overfill the entire CPU cache in dozens of megabytes or even flood the log zone. Hercules continues cache placements and replacements to swap in and out data, respectively, to proceed transactions. We can configure a log zone in gigabytes or even larger. In case that such a large space is still to be used up, we extend the log zone by using the end part of default log zone to store indirect indexes to the space in a new log zone allocated on-demand elsewhere. This is like the strategy of indirect blocks used by file systems to manage big files [67, 96]. We also enhance the eWPQ with more entries and extend an eWPQ entry with one more bit to tell the MC if it needs to do indirect references or not to find actual data for a prematurely flushed cache line.

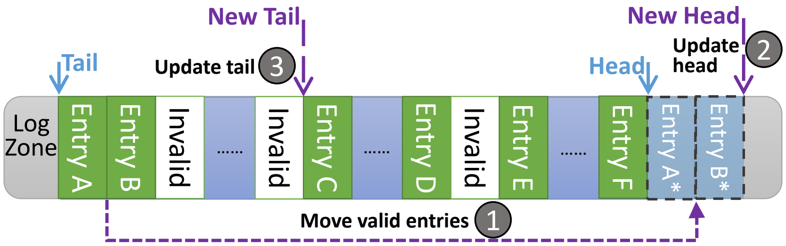


Fig. 4. Garbage Collection on Log Entries

4.5 Discussion

4.5.1 *The Interactions between Hercules and Cache Inclusion and Coherence Policies.*

Cache Inclusion. Hercules is capable of working with inclusive and non-inclusive cache architectures except for exclusive cache, as it is imperative to forward a modified cache line to the subsequent level when a transactional cache line encounters a dirty, previously non-transactional copy (see Section 4.2). This mechanism aligns with the definition of inclusive and non-inclusive multi-level caches but rules out the exclusivity of a cache line. Note that, Hercules applies exclusion to each TransTag, which means that an evicted transactional cache line, once reloaded into a higher-level cache, its TransTag as well as cache line will be removed from the lower-level cache. The reason of doing so is twofold. Firstly, an exclusive housing of TransTags substantially reduces the spatial cost of keeping duplicate cache lines with TransTags across cach levels. Secondly, it helps to reduce microarchitectural actions. Let us illustrate with an example. Given a transactional cache line loaded to a higher level, e.g., L2⟶L1D, for read purpose, an inclusive cache hierarchy has it at both levels holding the latest version. When the transaction commits, Hercules needs to reset TxStates twice for the same cache line across levels. Given a transactional cache line loaded to a higher level for updating, cache coherence protocols like MESI or MESIF help an inclusive cache hierarchy to invalidate the older version at L2 upon a modification, but that costs a microarchitecture-level coherence state transition ($M \Rightarrow I$). Hercules's exclusion circumvents these inessential state resets or transitions.

Cache Coherence. Hercules collaboratively interacts with cache coherence protocols like MESI or MOESI. TransTags and the directory for cache coherence [74–76, 78] share similarities in use and we can integrate them to jointly track cache lines. Hercules does not affect the sharing of non-transactional cache lines. As to transactional ones, the concurrency control of applications and architectural TxIDs in TransTags prevent other cores from modifying or fetching uncommitted versions of them. Once a cache line is committed, the cache coherence mechanism helps to broadcast the update information to other locations in remote cores' private caches or shared levels in the memory hierarchy by sending state transition messages. In all, Hercules inflicts no harm to cache coherence and leverages state transitions to keep cache lines that have been involved in transactions up-to-date for availability.

4.5.2 *The Granularity, State Reset, and Isolation Schemes of Hercules's Transactions.*

Transaction Granularity. A cache line is the unit transferred between CPU cache and memory, so Hercules chooses it as the unit for transactional operations. Using programmer-defined variables is more fine-grained but must incur higher cost and complexity.

State Reset. Because cache lines of a transaction are likely to be scattered in a multi-level cache/memory hierarchy, all prior designs commit them with concrete efforts for atomic durability [20, 29, 33, 35–37]. For example, TC forcefully persists data in the side-path cache to pmem [29] while Kiln flushes down cache lines from upper-level volatile caches [20]. HOOP migrates all data staying in its out-of-place (OOP) buffer installed in the MC to its in-pmem OOP region [37]. FWB has to wait for the drain of current log updates [36]. Comparatively, Hercules' commit is much more efficient and lightweight, as it just sets `TxLen` and resets `TxStates` for cache lines that a transaction covers. Since TransTags form a structure similar to the directory for cache coherence used to track and transit states for cache lines, Hercules employs an auxiliary circuit to select ones with a `TxID` and clear their `TxStates`. For data that might be prematurely flushed before the commit, Hercules uses the auxiliary circuit to notify the integrated MC and wait for the completion of resetting `TxStates` in relevant eWPQ entries. Generally these resets can be swiftly done like state transitions for cache coherence. We preset a uniform state reset latency in which Hercules is supposed to finish, with a discussion presented in Section 5.3.

Isolation. Not all pmem systems supporting transactions provide thread-atomicity (isolation) [97]. As mentioned, like using prior hardware designs [14, 20, 32, 33, 37, 64], with Hercules programmers are responsible for the isolations between threads through concurrency control methods (e.g., locks or semaphores). Hercules also offers an additional safeguard by aborting transactions if a transactional cache line is to be fetched from another core. In addition, if false sharing happens to a cache line belonging to an uncommitted transaction, Hercules would abort the transaction to retain transactional granularity of cache line at the architecture level. As a result, developers using Hercules for atomic durability need to attend the placement and separation of data in order to rule out unexpected aborts and keep smooth execution. In particular, developers shall be responsible for handling recurrent aborts appropriately to guarantee a forward progress with regard to their awareness of semantic conflicts and contentions on shared transactional data [8, 66]. Hercules helps them to catch each abort such that they can implement a mechanism with retry/ignore to properly attain their targets.

4.5.3 The Support of System Software for Hercules.

Currently, the extensive utilization of pmem is still being explored. Adjustments to the operating system (OS) are required not only for Hercules but also for state-of-the-art hardware designs discussed in the paper. For example, the compiler and OS should be augmented to recognize and compile primitives (e.g., `tx_start` and `tx_commit`) introduced by Hercules-like hardware designs. During the recovery process, the OS needs to perform address mapping before Hercules-like hardware designs can access and manage the log area to recover data. More than that, when the log zone is used up, Hercules would extend it with the assistance of OS for space allocation.

5 EVALUATION

We implement Hercules with gem5 in the syscall emulation (SE) mode and 'classic caches' model. Table 2 captures the settings of CPU with three-level inclusive caches that align with prior works [14, 20, 29, 34, 37]. We configure that the pmem embraces a longer read latency than write latency according to recent studies on Intel Optane DC memory [59, 61]. An SRAM or DRAM buffer built in pmem efficiently absorbs write requests while a read request is likely to directly load data from slower NVM. We configure an in-pmem log zone in 256MB. We set default TransTag ratios to be 100%/50%/25% for L1D/L2/L3 caches. As L1D cache impacts the most on performance in CPU cache hierarchy, in order to make a fair and strict evaluation on Hercules, we reduce L1D size from 32KB to 30KB for Hercules by evenly removing some ways in cache sets within gem5 to counterbalance the spatial cost of it. We further estimate the spatial and energy costs for Hercules in Section 5.4. For a cache line access involving a TransTag, we increase the tag latency by 30% as extra time cost. We set the state reset latency in ten clock cycles by default with a discussion in Section 5.3. We also set ten clock cycles for searching the eWPQ to check if a cache line has been prematurely flushed.

Table 2. System Configuration

Component	Setting	Remarks
Processor	3GHz, out-of-order, 8 cores, ROB size=192 [75, 98, 99], issue/write-back/commit width=8, MESI protocol	Generic
L1I Cache	32KB, 4-way, 2-cycle latency	
L1D Cache	32KB, 4-way, 2-cycle latency	
L2 Cache	256KB, 8-way, 8-cycle latency	
LLC	16MB, 16-way, 30-cycle latency	
Pmem	Read/write latency=150ns/100ns [26, 61], capacity=512GB, single channel, read/write buffer size=64	
Smaller L1D	30KB, 4-way, 2/2 cycles read/write latencies	Hercules
Side-path TC	4KB, FIFO, 40/50 cycles read/write latencies	TC [29]
STT-RAM LLC	64MB, 16-way, 40/50 cycles read/write latencies	Kiln [20]

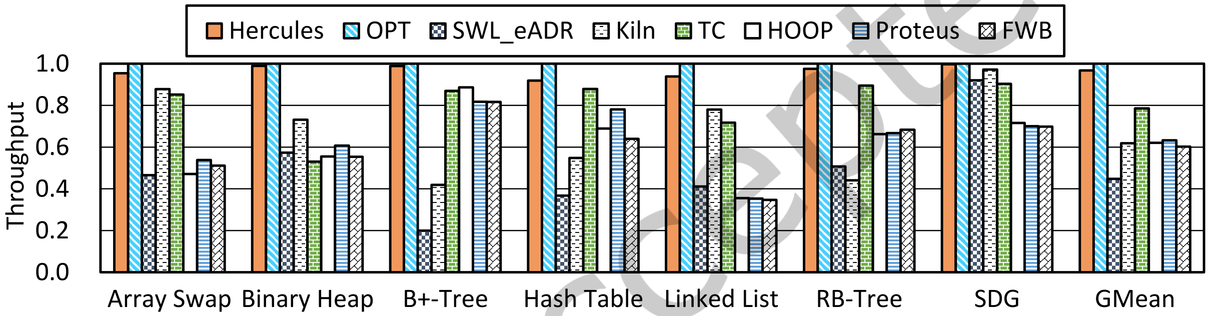


Fig. 5. A Comparison on Throughput of Micro-benchmarks (Normalized against OPT's)

Table 3 lists micro- and macro-benchmarks we use. We consider ones that have been widely used in prior works [16, 20, 33, 35–37]. These prevalent benchmarks are not only used for evaluating architectural supports for atomic durability but also for testing file systems and pmem libraries [57, 67, 100, 101]. Our evaluation methodology strictly follows prior works to configure them. We run one million transactions with each benchmark.

Table 3. Benchmarks Used in Evaluation

Category	Benchmark	Remarks
Micro-benchmarks	Array Swap	Swaps two elements in an array
	Binary Heap	Inserts/deletes entries in a binary heap
	B+-Tree	Inserts/deletes KV pairs in a B+-Tree
	Hash Table	Inserts/deletes KV pairs in a hash table
	Linked List	Inserts/deletes entries in a linked list
	RB-Tree	Inserts/deletes KV pairs in a RB-Tree
	SDG	Inserts/deletes edges in a scalable large graph
Macro-benchmarks	TPC-C	OLTP workload (New-order transactions)
	YCSB	80%/20% of write/read

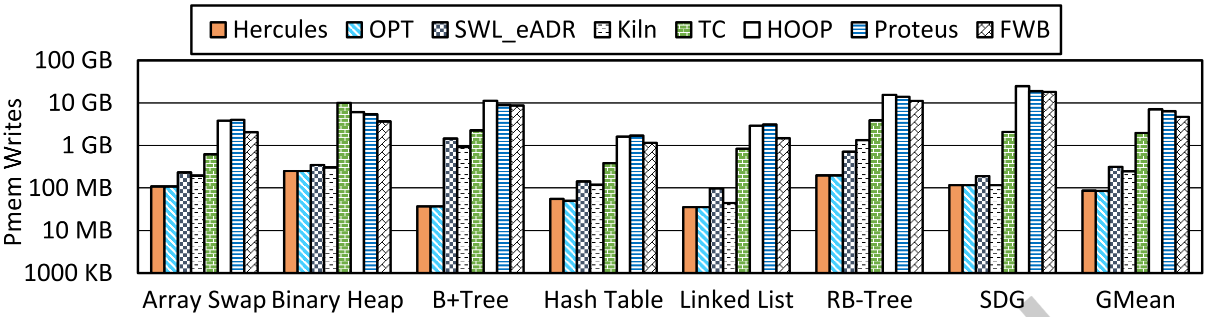


Fig. 6. A Comparison on Pmem Writes of Micro-benchmarks (Y Axis in Logarithmic Scale)

We compare Hercules to state-of-the-art hardware designs, including Kiln, TC, HOOP, Proteus, FWB, and PiCL. They represent different approaches to guarantee atomic durability for in-pmem data (see Section 2.2). As PiCL is with periodical checkpointing, we discuss it separately at the end of Section 5.1. In Table 2, the STT-RAM LLC and side-path cache are for Kiln and TC, respectively, both configured in line with STT-RAM's characteristics [20, 102]. Software logging with eADR (SWL_eADR) is also compared while software logging with cache line flushes is omitted for brevity. By using memory fence instructions, SWL_eADR persists original data in an in-pmem log prior to updating data in place (undo log). It is necessary to insert memory fence instructions to guarantee the ordering between logging actions and in-place updating, since without them, a reordered store sequence of, say, logging after in-place updating may cause the logged copy to lose trustworthiness and recoverability [9]. In order to maintain the atomic durability of data structures, it is crucial to log any critical changes made to the structure. Take B+-Tree and RB-Tree for example. An insertion to B+-Tree and RB-Tree may incur node split and self-balancing, respectively. All such structural changes should be logged and tracked for recovery, which we have taken into full consideration in implementing SWL_eADR as well as other state-of-the-art designs.

5.1 Micro-benchmark

Throughput. We normalize the throughputs (txn/μs, transactions per microsecond) of all designs against that of OPT, which demonstrates the optimal performance without the involvement of logging, cache line flush, memory fence, or anything else for the guarantee of atomic durability. As shown in Figure 5, we include the geometric mean of throughputs over benchmarks for a high-level overview [14, 20, 29, 34, 37]. Hercules achieves comparable performance to OPT. It significantly outperforms prior works with on average 89.2%, 29.2%, 15.2%, 51.3%, 48.0%, and 57.0% higher throughput than SWL_eADR, Kiln, TC, HOOP, Proteus, and FWB, respectively. Hercules leverages CPU cache hierarchy to absorb and coalesce transactional updates and mainly commits them on-chip. It gains superior efficacy in handling continuous transactions with such a spacious transient persistence domain.

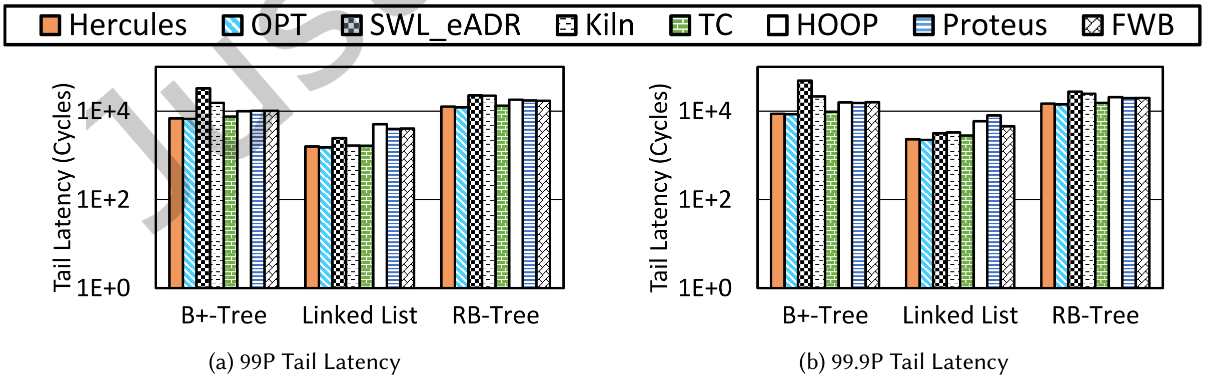
The throughputs of Kiln and TC are limited by two factors. One is due to STT-RAM's longer write/read latencies. The other one is that Kiln and TC have enforced limits in using persistent caches, such as using a small side-path cache [29] or taking a fall-back path to write pmem for backup in case of an almost full request queue [20]. Other hardware designs, such as HOOP, Proteus, and FWB, exploit hardware components like undo/redo log buffers, WPQ, or pmem to compose and persist backup copies. Continuous transactions keep limited log buffers or WPQ entries being fully occupied over time. They are hence inferior to Hercules that leverages the extensive CPU cache hierarchy to absorb and process data.

Figure 5 exhibits different observations across benchmarks, as their transactions are with different semantics and complexities. For example, two insertions with Linked List and RB-Tree differ a lot. Unlike prior designs varying significantly across benchmarks, Hercules shows consistently superior performance. Its strong robustness is mainly accredited to CPU cache it exploits. A multi-level cache hierarchy has been proved to be effectual for various workloads over decades.

Pmem writes. To minimize pmem writes is another goal of Hercules. Figure 6 captures the quantity of pmem writes caused in running transactions with micro-benchmarks (Y axis in the logarithmic scale). Hercules significantly reduces pmem writes. On average, the data it writes is 29.8%, 37.4%, 5.3%, 1.4%, 1.5%, and 2.1% that of other designs in the foregoing order, respectively. Write-backs of Hercules only happen upon normal cache replacement or power-off, so it performs pmem writes in a passive and lazy way. SWL_eADR incurs double writes. Kiln takes a fall-back path that forcefully sends cache lines to pmem with an overflowing request queue. TC has a similar fall-back path to write pmem when its side-path cache is almost full. Also, whenever a transaction commits, TC issues relevant stores to pmem. Other hardware designs explicitly write backup copies to pmem. HOOP, for example, does address indirection at the MC and migrates data between pmem locations over time for out-of-place updating. Proteus leverages the WPQ for buffering to reduce pmem writes, but the limited capacity of WPQ impedes its efficacy. By using CPU cache in numerous megabytes to absorb and coalesce data updates, Hercules effectively minimizes pmem writes.

Tail latencies. Without loss of generality, we record 99P/99.9P (99- and 99.9-percentile) tail latencies for transactions and show them for three benchmarks in Figure 7a and 7b, respectively, with Y axes in the logarithmic scales. These results further justify Hercules' efficacy as it makes much shorter tail latencies with various workloads. Take B+-Tree for example. Hercules' 99.9P tail latency is 40.5% and 55.1% shorter than that of Kiln and HOOP, respectively.

A comparison to the approach of checkpointing. PiCL checkpoints data periodically in pmem for recovery without forming transactions [34, 45]. In Figure 8 we present the number of clock cycles and the quantity of pmem writes Hercules and PiCL have done to complete all operations for each benchmark. PiCL's checkpointing is like undo-logging all data at the start of every epoch, which, albeit being simplistic, causes on average 78.3% more time and 34.1× pmem writes than Hercules. Hercules only covers data programmers place in fine-grained transactions and leverages CPU cache to log and buffer them. This is why Hercules costs both much less time and dramatically fewer pmem writes than PiCL.



(a) 99P Tail Latency

(b) 99.9P Tail Latency

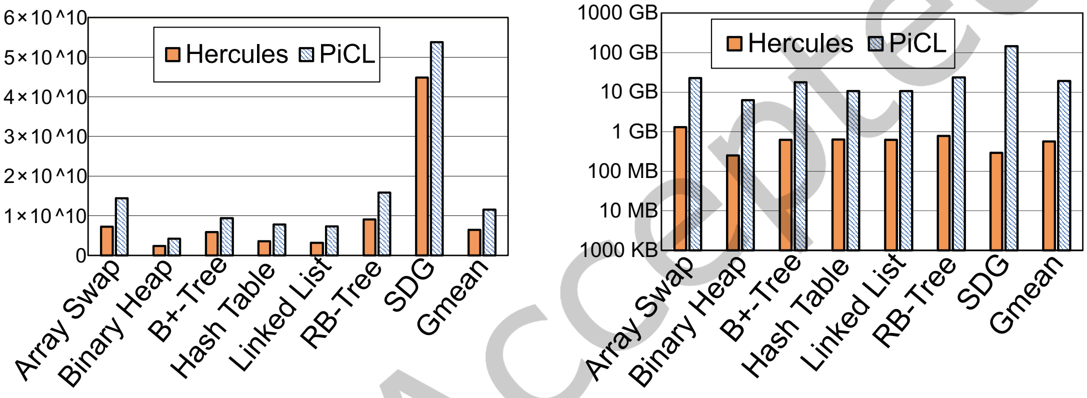
Fig. 7. A Comparison on the 99P and 99.9P Tail Latencies

Table 4. A Summary of TxLens for benchmarks

Benchmark	Array Swap	Binary Heap	B+-Tree	Hash Table	Linked List	RB-Tree	SDG
Min. TxLen	7	8	15	3	12	2	22
Max. TxLen	9	1,021	143	7	18	71	149
Avg. TxLen	8.0	11.6	34.4	6.2	12.5	50.6	33.8

5.2 Tests for Premature Flush and TransTag Ratios

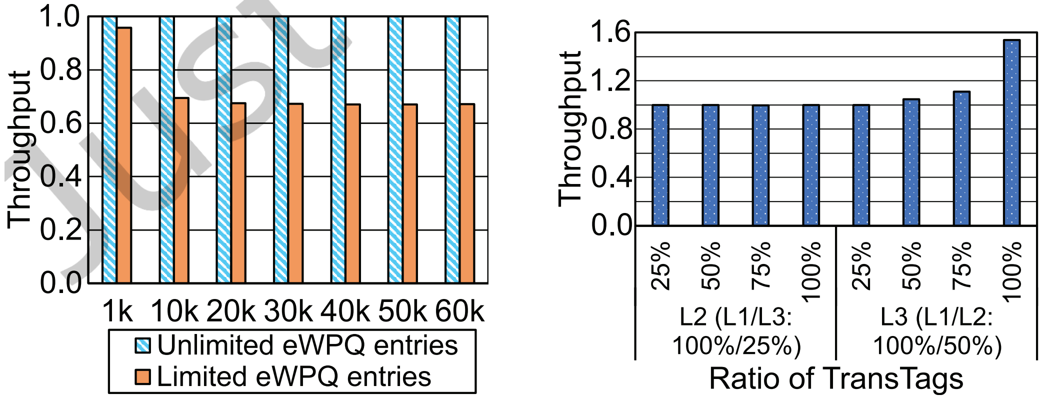
We summarize TxLens on finishing all transactions for benchmarks in Table 4. Modern cache hierarchy has no difficulty in putting few to dozens of cache lines. Hercules hence effectually suits ordinary workloads and almost



(a) CPU Clock Cycles

(b) Pmem Writes

Fig. 8. A Comparison between Hercules and PiCL



(a) Varying Transaction Sizes

(b) Impact of TransTag Ratios

Fig. 9. The Throughputs with Artificial Huge Transactions

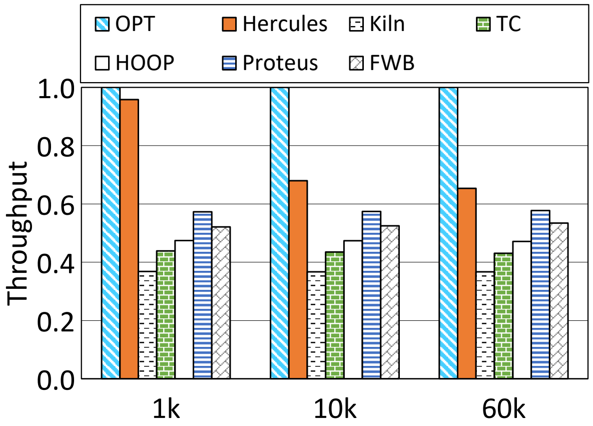


Fig. 10. A Comparison on Artificial Huge Transactions with the Scale-down Configuration

all transactions can be committed on-chip. Meanwhile, ordinary transactions in small sizes are hardly affected by the change of TransTag ratios and seldom cause premature flushes or GC.

To thoroughly evaluate Hercules, we run unrealistic artificial tests executed in a shrunk cache hierarchy. We scale down L1D/L2/L3 caches by 8× to be 3840B/32KB/2MB and synthesize test cases in which a huge transaction is composed of massive write-intensive operations and keeps doing with continuous caches lines in a read-modify-write fashion. Such an access pattern showed spatial locality on cache lines being updated in the huge transaction. We set the length of a transaction to be 1k, 10k, 20k to 60k (k: $\times 10^3$) and run with unlimited and 512 eWPQ entries, respectively. Figure 9a shows the throughputs normalized against that with unlimited eWPQ. For a huge transaction involving tens of thousands of cache lines, CPU cache and eWPQ would be saturated and Hercules must use the in-pmem eWPQ extension. The impact of premature flushes grows up. For example, the throughput with 512 eWPQ entries declines by 32.8% at the 60k case.

In order to further comprehensively evaluate the robustness of Hercules as well as state-of-the-art software and hardware designs, we reduce the size of cache hierarchy and hardware resources used to support transactions for Hercules and other designs by a scale-down of eight times. A shrunk cache hierarchy more impacts the efficiency of designs that rely on caching to aggregate modified data for transactions, such as Hercules, Kiln, and TC. Figure 10 captures the results of throughput for all in running aforementioned artificial huge transactions with overwhelming data. Figure 10 clearly demonstrates that, even with the scale-down configuration, Hercules still outperforms other state-of-the-art designs with, for example, 13.3% and 51.8% greater throughput than that of Proteus and TC, respectively.

We also run unrealistically huge transactions when varying TransTag ratios at L1D, L2, and L3. Without loss of generality, we illustrate with the 30k test case upon changing TransTag ratios at L2 and L3. In Figure 9b, we normalize the throughputs against that of default 100%/50%/25% ratios. There is an evident uptrend along an increasing ratio at L3. A 2MB L3 cache has 32,768 cache lines. An increased ratio means more space to house the working set of 30k case. Only 100% ratio at L3 manages to fit all 30,000 cache lines of 30k and yields the highest throughput without premature flush.

5.3 The Impacts of Factors on Hercules

State reset latency. We set the default state reset latency per transaction as ten clock cycles to toggle `TxStates` of aggregated TransTags with auxiliary circuit. We can deploy circuits in different complexities to gain different latencies. Figure 11 shows the throughput (txn/μs) curves on running micro-benchmarks when we vary the latency duration from the ideal (zero) to 90 cycles. With a longer latency, the throughputs of Linked list, Hash Table, Array Swap, and Binary Heap decrease more severely than those of B+-Tree, RB-Tree, and SDG. The curves in Figure 11 complement TxLens in Table 4. The throughput of a benchmark with smaller transactions is surely more sensitive to the increase of state reset latency, since a smaller transaction itself takes less execution time. Take RB-Tree and Linked List for comparison again. An insertion to RB-Tree is generally more complicated and involves 50.6 transactional cache lines per transaction. Yet an insertion to Linked List deals with 12.5 cache lines per transaction on average. As a result, increasing the state reset latency more affects lighter benchmarks like Linked List.

WPQ size. We further test when varying the size of WPQ. Without loss of generality, we choose Linked List and present throughputs of all designs in Figure 12. Hercules' bottleneck is not on write-backs via the WPQ to pmem, so its performance has no evident fluctuations. This justifies the robustness of Hercules in utilizing extensive CPU cache for atomic durability. A larger WPQ helps prior works like HOOP and Proteus yield performance improvements. Proteus leverages the WPQ of MC for transactional operations while HOOP depends on the MC to do address indirection and data movements. They hence benefit more from an increase of WPQ entries.

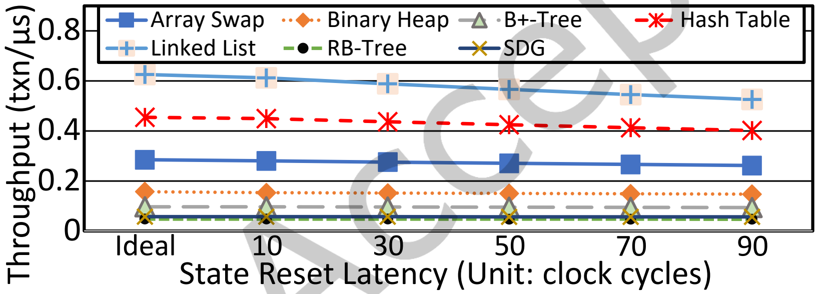


Fig. 11. The Impact of State Reset Latency

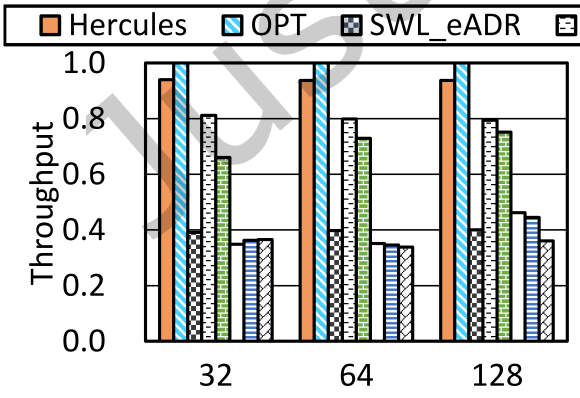


Fig. 12. The Impact of WPQ Size on Linked List

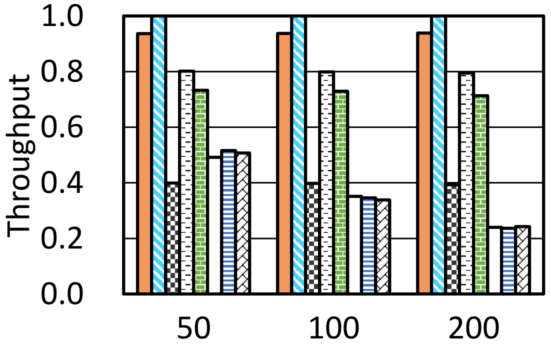


Fig. 13. The Impact of Pmem Latency (ns) on Linked List

Pmem latency. The default write latency of pmem is 100ns. We vary it to emulate different pmem products. Figure 13 captures the throughputs of all designs again on Linked List with three write latencies. By keeping all transactional data in CPU caches without incurring pmem writes, Hercules is unaffected by varying pmem latencies. Notably, the performances of prior works that rely on writing data to pmem for backup badly degrade. In summary, leveraging the capacious transient persistence domain made of CPU caches enables Hercules with high adaptability to various pmem products.

Table 5. Miss Rate of an Ordinary Thread that is Competing for Cache Lines

Structure	B+-Tree		RB-Tree	
Contending with	Transactional thread	Ordinary thread	Transactional thread	Ordinary thread
L1	59.51%	57.88%	39.25%	38.89%
L2	80.27%	80.25%	71.29%	71.23%
L3	1.24%	1.24%	21.58%	21.60%

Cache contention. To demonstrate the impact of Hercules's cache replacement strategy on performance for ordinary threads, we conduct an experiment involving two threads competing for cache space. In particular, we run an ordinary thread without transactions that is continuously operating with B+-Tree (resp. RB-Tree). In the meantime, we engage the other contending thread, which is either an ordinary non-transactional thread doing with another B+-Tree (resp. RB-Tree) or a transactional thread relying on Hercules for performing transactions with the second B+-Tree (resp. RB-Tree), in running on the same core. Table 5 presents the cache miss rates for the ordinary thread when it simultaneously executes with the other contending thread. A comparison between either tree's two columns clearly conveys that the replacement strategy of Hercules hardly affects the performance of non-transactional ordinary thread.

Table 6. The Spatial Cost of Hercules

TransTags (Unit: KB)			eWPQ (Unit: Bytes)		
L1D	L2	L3	Entries	Validity Bitmap	`LogHead` & `LogTail`
1.29 (per core)	6.25 (per core)	208	4,096	64	16

5.4 Recoverability, Energy and Spatial Costs of Hercules

Recovery. We tailor gem5's *checkpoint* function to save all metadata and data into the log zone backed by files and emulate a crash by encountering a simulator magic instruction. Hercules manages to recover properly and resume execution.

Spatial cost. Table 6 summarizes the overall spatial cost of Hercules such that removing 2KB at L1D per core is sound to evaluate it. Similar to on-chip buffers used by prior works [18, 29, 34–36, 38], TransTags incur the main on-chip spatial cost for Hercules. Due to the space limitation, we brief an estimate with a 4-way L1D cache in 30KB. As we use all L1D cache lines, a TransTag needs 22 bits (21-bit `TxID` and 1-bit `TxState`) without `WayNo`. The original metadata per line, such as cache tag and state, takes at most 48 bits for a VIPT cache [98, 103]. TransTags thus cost 3.9% ($\frac{22}{48 + 64 \times 8} \times 100\%$) more space. This also explains why we increase the tag latency by 30% on accessing a transactional cache line ($\frac{22}{48 + 22} \times 100\% \approx 31.4\%$). Similarly we estimate the proportions of

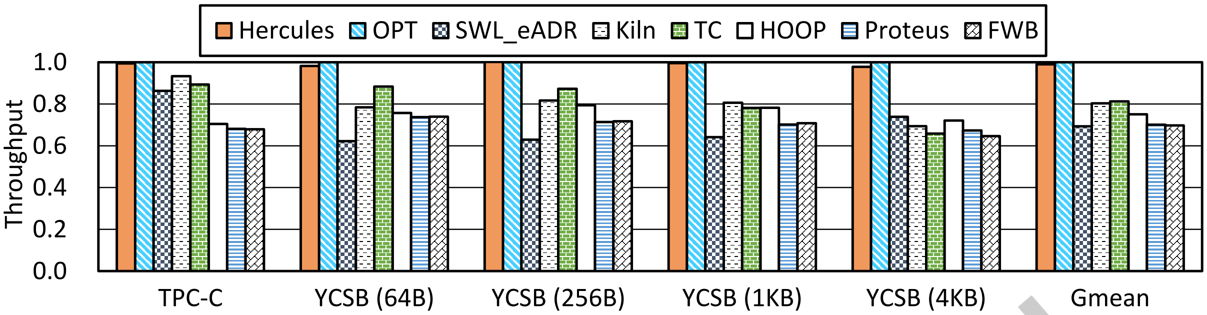


Fig. 14. A Comparison on Throughput of Macro-benchmarks with One Thread (Normalized against OPT's)

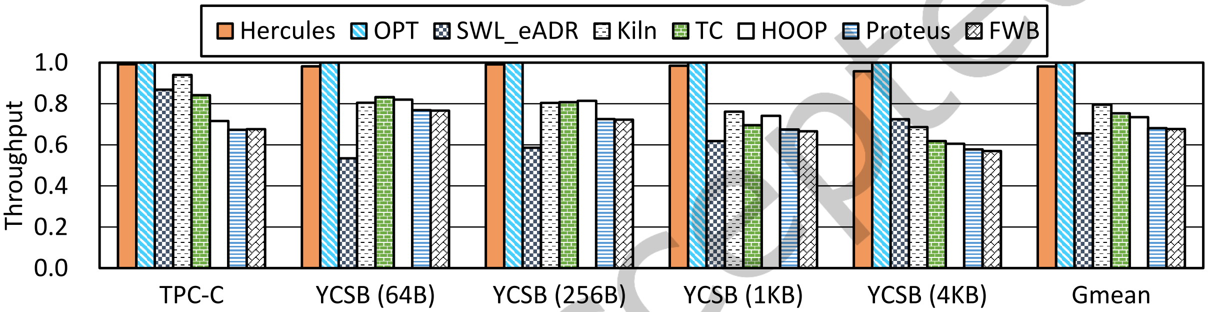


Fig. 15. A Comparison on Throughput of Macro-benchmarks with Four Threads (Normalized against OPT's)

TransTags at L2 and L3 to be 2.2% and 1.2%, respectively. We also estimate the spatial cost in the integrated MC for Hercules. The metadata for a transactional cache line in each eWPQ entry requires 64 bits of storage (1 bit for TxState, 21 bits for TxID, 21 bits for home address, and the last 21 bits for the address of log entry). With 512 entries by default, the total size of eWPQ entries is 4KB. Additionally, Hercules needs 64B, 8B, and 8B for the eWPQ Validity Bitmap, LogHead, and LogTail, respectively. In all, the spatial cost of Hercules is insignificant.

Energy cost. If a crash occurs, besides the eADR's ordinary flushes, Hercules persists the TransTag and home address that are estimated as at most 10B (e.g., $\geq \frac{22+48}{8}$ at L1D) for a transactional cache line. The energy costs per store from L1D, L2, and L3 caches to pmem on a crash are respectively $11.839nJ$/B, $11.228nJ$/B, and $11.228nJ$/B [18, 104]. The base cost of flushing the entire cache hierarchy is thus $214.831mJ$ for 8-core CPU. Regarding 3840/16384/65536 transactional cache lines at L1D/L2/L3 with eight cores, all of them are dirty and uncommitted in the worst case. The cost to flush TransTags and home addresses for them is about $9.653mJ$. Hercules also needs to identify transactional cache lines by comparing WayNo in each TransTag and all comparisons cost about $84.045nJ$ with an up-to-date comparator taking $0.98pJ$ per comparison [105]. Overall, Hercules maximally brings about 4.49% ($\frac{9.653+84.045\times10^{-6}}{214.831}\times 100\%$) extra energy cost, which is practicable in upgrading the eADR. In addition, a TxID in more bits may increase such extra cost to be beyond 5.0% or even greater, and also impose further challenges on designing and producing chips with efficiency and reliability [99, 106–108].

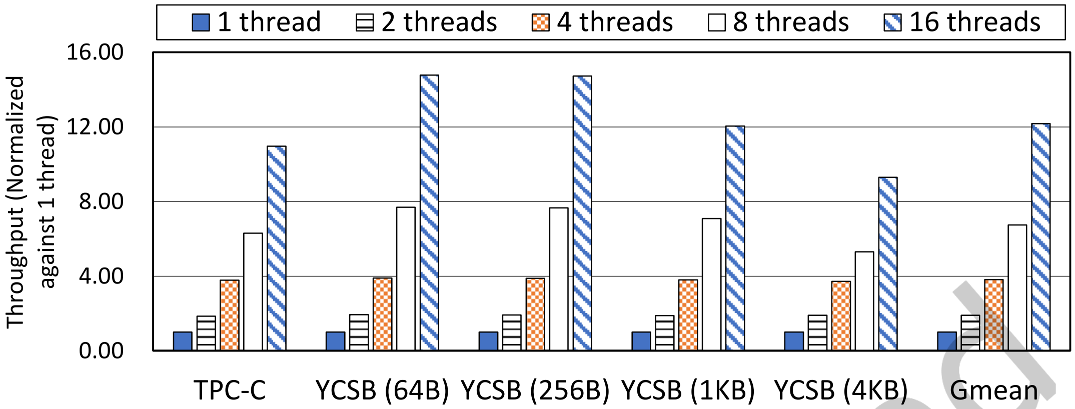


Fig. 16. Scalability Experiment with Varying Threads on Macro-benchmarks (Normalized against 1 thread's)

5.5 Macro-benchmark

We utilize TPC-C and YCSB workloads from WHISPER [16, 35, 109] to evaluate Hercules for two purposes. Firstly, we measure the performance and robustness of Hercules with more sophisticated workloads of realistic applications. Secondly, we run with multi-threads to test the scalability of Hercules in serving concurrent transactions. TPC-C's New-order follows the de facto semantics. As to YCSB, we vary the value size for a comprehensive test.

As shown in Figure 14, on dealing with more complicated transactions of macro-benchmarks, Hercules still yields a higher throughput than SWL_eADR, Kiln, TC, HOOP, Proteus, and FWB by 39.2%, 20.5%, 19.0%, 31.5%, 41.1%, and 41.7% on average, respectively. Furthermore, with a larger value, the performance gap between Hercules and prior designs generally becomes wider. The root cause is Hercules' robust design. It leverages the sufficient CPU cache to take in transactions. The eADR renders a value almost persistent in CPU cache. A larger value does not greatly increase the cost of persisting the value, especially on the critical path. Comparatively, the double writes of logging again hinder SWL_eADR from achieving high performance. As to Kiln and TC, a larger value imposes more burdens in using FIFO queues and side-path cache, respectively, and triggers more executions through fall-back paths that severely affect their throughputs. For FWB and Proteus, larger transactions still make immense data updates that continually run out of their log buffer and WPQ entries, respectively. As to HOOP, larger values consume more pmem bandwidths and incur longer time for the MC to wait for the completion of data migrations.

The results with running four threads in Figure 15 justify the scalability of Hercules, which produces 43.7%, 20.1%, 27.8%, 31.6%, 42.9%, and 43.7% higher throughput, respectively, than prior designs on average. A multi-level private/shared cache hierarchy with the set-associativity management implies an innate scalability to support multi-threading with multi-cores. Hercules gains scalability accordingly. Threads may share `GlobalTxID` register at the start of a transaction and eWPQ outside of the critical path for prematurely flushed cache lines. Though, getting a `TxID` can be swiftly done in an atomic operation while the probability of massive synchronizations under a spacious cache hierarchy is low. For prior designs, the contention on resources between multi-threads is much fiercer. Take HOOP for illustration again. As it depends on the MC for out-of-place data updates, multiple threads contend flushing data through the MC. This offsets the effect of CPU cache for concurrency and makes the MC a busy synchronization point being shared at runtime, thereby limiting HOOP's scalability.

We evaluate the scalability of Hercules through an experiment on macro-benchmarks running with varying threads. The result presented in Figure 16 demonstrates that Hercules exhibits a well scalability throughout handling both TPC-C and YCSB workloads. For example, the average throughput in geometric mean with 2 to 16 threads for Hercules is 1.9×, 3.8×, 6.8×, and 12.2×, respectively, that of one thread. We note that TPC-C and YCSB with large values have complex operations and long code paths. This is likely to result in fierce contentions for transactional cache lines and premature flushes via the eWPQ, as mentioned in Section 5.2. These findings suggest that Hercules gains both efficacy and scalability in leveraging CPU cache hierarchy as well as multi-core CPU for transactional support. Compared to state-of-the-art hardware designs that have been built on components in much lower capacities, say, the WPQ used by Proteus or side-path cache of TC, Hercules makes significantly higher scalability.

6 CONCLUSION

Researchers are revolutionizing computer architecture with promising features to facilitate the use of pmem. Among them, the eADR radically changes the fact that CPU cache hierarchy has been practically volatile for decades. In this paper, we propose Hercules, a systematic hardware design leveraging the transient persistence domain made of CPU cache in scores of megabytes to enable the transaction-level atomic durability for in-pmem data. Hercules has comprehensive control logics and data-paths installed in CPU cache, MC, and pmem. It provides transactional primitives and protocols to define and proceed transactions. Hercules well serves typical applications. Experiments confirm that it significantly outperforms prior works with higher performance. Hercules also substantially minimizes pmem writes with ample CPU cache buffering data.

ACKNOWLEDGEMENT

This work was jointly supported by National Key R&D Program of China No. 2022YFB4401700, Nature Science Foundation of Shanghai under Grants No. 22ZR1442000 and 23ZR1442300, and ShanghaiTech Startup Funding.

REFERENCES

[1] SK Hynix. SK Hynix developed the world's highest density 16GB NVDIMM. https://news.skhynix.com/sk-hynix-developed-the-worlds-highest-density-16gb-nvdimm/, October 2014. Accessed: 06-15-2022.

[2] Intel. 3D XPoint^{TM}: A breakthrough in non-volatile memory technology. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html. Accessed: 04-22-2022.

[3] Micron. NVDIMM: Persistent memory performance. https://media-www.micron.com/-/media/client/global/documents/products/product-flyer/nvdimm_flyer.pdf?rev=0c295086bb4c43729b89f369219259bc, December 2017. Accessed: 06-15-2022.

[4] Dell. Dell EMC NVDIMM-N persistent memory: user guide. https://dl.dell.com/topicspdf/nvdimm_n_user_guide_en-us.pdf, February 2021. Accessed: 06-15-2022.

[5] Hewlett Packard Enterprise. HPE NVDIMMs. https://www.hpe.com/psnow/doc/c04939369.html, November 2021. Accessed: 06-15-2022.

[6] Intel. Intel® Optane^{TM} memory - responsive memory, accelerated performance. https://www.intel.com/content/www/us/en/products/details/memory-storage/optane-memory.html, July 2022. Accessed: 07-13-2022.

[7] Everspin. Everspin releases highest density MRAM products to create fastest and most reliable non-volatile storage class memory. https://www.everspin.com/sites/default/files/Everspin%20Releases%20Highest%20Density%20MRAM%20Products%20FINAL%20041216.pdf, April 2016. Accessed: 07-10-2022.

[8] Intel. Intel® 64 and IA-32 architectures software developer manuals. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html. Accessed: 05-12-2022.

[9] Steve Scargall. *Programming persistent memory: A comprehensive guide for developers.* APress, Berlin, Germany, 1 edition, 2020.

[10] Vaibhav Gogte, Aasheesh Kolli, and F. Thomas Wenisch. *A Primer on Memory Persistency.* Number 1935-3243 in Synthesis Lectures on Computer Architecture. Springer Cham, 1 edition, 2022.

[11] Shashank Gugnani, Arjun Kashyap, and Xiaoyi Lu. Understanding the idiosyncrasies of real persistent memory. *Proc. VLDB Endow.*, 14(4):626–639, dec 2020.

[12] Kyeongmin Cho, Sung-Hwan Lee, Azalea Raad, and Jeehoon Kang. Revamping hardware persistency models: View-based and axiomatic persistency models for Intel-x86 and Armv8. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming*

Language Design and Implementation, PLDI 2021, pages 16–31, New York, NY, USA, 2021. Association for Computing Machinery.

[13] Sujay Yadalam, Nisarg Shah, Xiangyao Yu, and Michael Swift. ASAP: A speculative approach to persistence. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 892–907, 2022.

[14] Seunghee Shin, Satish Kumar Tirukkovalluri, James Tuck, and Yan Solihin. Proteus: A flexible and fast software supported hardware logging approach for NVM. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, pages 178–190, New York, NY, USA, 2017. Association for Computing Machinery.

[15] Siddharth Gupta, Alexandros Daglis, and Babak Falsafi. Distributed logless atomic durability with persistent memory. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, pages 466–478, New York, NY, USA, 2019. Association for Computing Machinery.

[16] Xijing Han, James Tuck, and Amro Awad. Dolos: Improving the performance of persistent applications in ADR-supported secure memory. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, pages 1241–1253, New York, NY, USA, 2021. Association for Computing Machinery.

[17] Intel. eADR: New opportunities for persistent memory applications. https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html, January 2021. Accessed: 07-15-2022.

[18] Mohammad Alshboul, Prakash Ramrakhyani, William Wang, James Tuck, and Yan Solihin. BBB: Simplifying persistent programming using battery-backed buffers. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 111–124, 2021.

[19] Mazen Alwadi, Vamsee Reddy Kommareddy, Clayton Hughes, Simon David Hammond, and Amro Awad. Stealth-persist: Architectural support for persistent applications in hybrid memory systems. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 139–152, 2021.

[20] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P. Jouppi. Kiln: Closing the performance gap between systems with and without persistence support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 421–432, New York, NY, USA, 2013. Association for Computing Machinery.

[21] Moinuddin K. Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, and Bulent Abali. Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 14–23, 2009.

[22] Engin Ipek, Jeremy Condit, Edmund B. Nightingale, Doug Burger, and Thomas Moscibroda. Dynamically replicated memory: Building reliable systems from nanoscale resistive memories. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 3–14, New York, NY, USA, 2010. Association for Computing Machinery.

[23] Nak Hee Seong, Dong Hyuk Woo, and Hsien-Hsin Lee. Security refresh: Protecting phase-change memory against malicious wear out. *IEEE Micro*, 31(1):119–127, January 2011.

[24] Rujia Wang, Lei Jiang, Youtao Zhang, and Jun Yang. SD-PCM: Constructing reliable super dense phase change memory under write disturbance. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 19–31, New York, NY, USA, 2015. Association for Computing Machinery.

[25] Mohammad Khavari Tavana, Amir Kavyan Ziabari, and David Kaeli. Block cooperation: Advancing lifetime of resistive memories by increasing utilization of error correcting codes. *ACM Trans. Archit. Code Optim.*, 15(3), aug 2018.

[26] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. FlatStore: An efficient log-structured key-value storage engine for persistent memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1077–1091, New York, NY, USA, 2020. Association for Computing Machinery.

[27] Stephen Longofono, Seyed Mohammad Seyedzadeh, and Alex K. Jones. Virtual coset coding for encrypted non-volatile memories with multi-level cells. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1128–1140, 2022.

[28] Youyou Lu, Jiwu Shu, Long Sun, and Onur Mutlu. Loose-ordering consistency for persistent memory. In *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, pages 216–223, 2014.

[29] Chun-Hao Lai, Jishen Zhao, and Chia-Lin Yang. Leave the cache hierarchy operation as it is: A new persistent memory accelerating approach. In *Proceedings of the 54th Annual Design Automation Conference 2017*, DAC '17, New York, NY, USA, 2017. Association for Computing Machinery.

[30] Sara Mahdizadeh Shahri, Seyed Armin Vakil Ghahani, and Aasheesh Kolli. (Almost) fence-less persist ordering. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 539–554, 2020.

[31] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutlu. ThyNVM: Enabling software-transparent crash consistency in persistent memory systems. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 672–685, New York, NY, USA, 2015. Association for Computing Machinery.

[32] Kshitij Doshi, Ellis Giles, and Peter Varman. Atomic persistence for SCM with a non-intrusive backend controller. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 77–89, 2016.

[33] Arpit Joshi, Vijay Nagarajan, Stratis Viglas, and Marcelo Cintra. ATOM: Atomic durability in non-volatile memory through hardware logging. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 361–372, 2017.

[34] Tri M. Nguyen and David Wentzlaff. PiCL: A software-transparent, persistent cache log for nonvolatile main memory. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-51, pages 507–519. IEEE Press, 2018.

[35] Jungi Jeong, Chang Hyun Park, Jaehyuk Huh, and Seungryoul Maeng. Efficient hardware-assisted logging with asynchronous and direct-update for persistent memory. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-51, pages 520–532. IEEE Press, 2018.

[36] Matheus Almeida Ogleari, Ethan L. Miller, and Jishen Zhao. Steal but no force: Efficient hardware undo+redo logging for persistent memory systems. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 336–349, 2018.

[37] Miao Cai, Chance C. Coats, and Jian Huang. HOOP: Efficient hardware-assisted out-of-place update for non-volatile memory. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 584–596, 2020.

[38] Xueliang Wei, Dan Feng, Wei Tong, Jingning Liu, and Liuqing Ye. MorLog: Morphable hardware logging for atomic persistence in non-volatile main memory. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA '20, pages 610–623. IEEE Press, 2020.

[39] Gem5. The gem5 simulator. https://www.gem5.org/. Accessed: 01-30-2022.

[40] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable DRAM alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 2–13, New York, NY, USA, 2009. Association for Computing Machinery.

[41] Benjamin C. Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Engin Ipek, Onur Mutlu, and Doug Burger. Phase-change technology and the future of main memory. *IEEE Micro*, 30(1):143–143, 2010.

[42] Moinuddin K. Qureshi, Andre Seznec, Luis A. Lastras, and Michele M. Franceschini. Practical and secure PCM systems by online detection of malicious write streams. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, pages 478–489, USA, 2011. IEEE Computer Society.

[43] Mohammad Arjomand, Mahmut T. Kandemir, Anand Sivasubramaniam, and Chita R. Das. Boosting access parallelism to PCM-based main memory. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 695–706. IEEE Press, 2016.

[44] Wujie Wen, Yaojun Zhang, Mengjie Mao, and Yiran Chen. State-restrict MLC STT-RAM designs for high-reliable high-performance memory system. In *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, pages 1–6, New York, NY, USA, 2014. Association for Computing Machinery.

[45] Ping Chi, Cong Xu, Tao Zhang, Xiangyu Dong, and Yuan Xie. Using multi-level cell STT-RAM for fast and energy-efficient local checkpointing. In *Proceedings of the 2014 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '14, pages 301–308. IEEE Press, 2014.

[46] Hengyu Zhao, Linuo Xue, Ping Chi, and Jishen Zhao. Approximate image storage with multi-level cell STT-MRAM main memory. In *Proceedings of the 36th International Conference on Computer-Aided Design*, ICCAD '17, pages 268–275. IEEE Press, 2017.

[47] Armin Haj Aboutalebi and Lide Duan. RAPS: Restore-aware policy selection for STT-MRAM-based main memory under read disturbance. In *2017 IEEE International Conference on Computer Design (ICCD)*, pages 625–632, 2017.

[48] Xiaochen Guo, Mahdi Nazm Bojnordi, Qing Guo, and Engin Ipek. Sanitizer: Mitigating the impact of expensive ECC checks on STT-MRAM based main memories. *IEEE Transactions on Computers*, 67(6):847–860, June 2018.

[49] Cong Xu, Dimin Niu, Naveen Muralimanohar, Rajeev Balasubramonian, Tao Zhang, Shimeng Yu, and Yuan Xie. Overcoming the challenges of crossbar resistive memory architectures. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 476–488, 2015.

[50] Lunkai Zhang, Brian Neely, Diana Franklin, Dmitri Strukov, Yuan Xie, and Frederic T. Chong. Mellow writes: Extending lifetime in resistive memories through selective slow write backs. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 519–531. IEEE Press, 2016.

[51] Tzu-Hsien Yang, Hsiang-Yun Cheng, Chia-Lin Yang, I-Ching Tseng, Han-Wen Hu, Hung-Sheng Chang, and Hsiang-Pang Li. Sparse ReRAM engine: Joint exploration of activation and weight sparsity in compressed neural networks. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, pages 236–249, New York, NY, USA, 2019. Association for Computing Machinery.

[52] Teyuh Chou, Wei Tang, Jacob Botimer, and Zhengya Zhang. CASCADE: Connecting RRAMs to extend analog dataflow in an end-to-end in-memory processing paradigm. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, pages 114–125, New York, NY, USA, 2019. Association for Computing Machinery.

[53] SMART Modular Technologies. Advanced memory - DDR4 NVDIMM. https://www.smartm.com/api/download/fetch/17, January 2022. Accessed: 06-15-2022.

[54] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. NV-Tree: Reducing consistency cost for NVM-based single level systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 167–181, USA, 2015. USENIX Association.

[55] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 91–104, New York, NY, USA, 2011. Association for Computing Machinery.

[56] Ren-Shuo Liu, De-Yu Shen, Chia-Lin Yang, Shun-Chih Yu, and Cheng-Yuan Michael Wang. NVM Duet: Unified working memory and persistent store architecture. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 455–470, New York, NY, USA, 2014. Association for Computing Machinery.

[57] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 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, pages 105–118, New York, NY, USA, 2011. Association for Computing Machinery.

[58] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. DudeTM: Building durable transactions with decoupling for persistent memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 329–343, New York, NY, USA, 2017. Association for Computing Machinery.

[59] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. Characterizing and modeling non-volatile memory systems. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 496–508, 2020.

[60] R. Madhava Krishnan, Jaeho Kim, Ajit Mathew, Xinwei Fu, Anthony Demeri, Changwoo Min, and Sudarsun Kannan. *Durable Transactional Memory Can Scale with TimeStone*, pages 335–349. Association for Computing Machinery, New York, NY, USA, 2020.

[61] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, Santa Clara, CA, February 2020. USENIX Association.

[62] Alexandro Baldassin, João Barreto, Daniel Castro, and Paolo Romano. Persistent memory: A survey of programming support and implementations. *ACM Comput. Surv.*, 54(7), July 2021.

[63] Jinglei Ren, Qingda Hu, Samira Khan, and Thomas Moscibroda. Programming for non-volatile main memory is hard. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, APSys '17, New York, NY, USA, 2017. Association for Computing Machinery.

[64] Taiyu Zhou, Yajuan Du, Fan Yang, Xiaojian Liao, and Youyou Lu. Efficient atomic durability on eADR-enabled persistent memory. In Andreas Klöckner and José Moreira, editors, *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, PACT 2022, Chicago, Illinois, October 8-12, 2022*, pages 124–134. ACM, 2022.

[65] Dushyanth Narayanan and Orion Hodson. Whole-system persistence. In Tim Harris and Michael L. Scott, editors, *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012*, pages 401–410. ACM, 2012.

[66] Intel. Restricted transactional memory overview. https://www.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/intrinsics/intrinsics-for-avx2/intrinsics-for-tsx/intrinsics-for-restrict-transactional-mem-ops/restricted-transactional-memory-overview.html, March 2023. Accessed: 03-09-2023.

[67] Jifei Yi, Mingkai Dong, Fangnuo Wu, and Haibo Chen. HTMFS: strong consistency comes for free with hardware transactional memory in persistent memory file systems. In Dean Hildebrand and Donald E. Porter, editors, *20th USENIX Conference on File and Storage Technologies, FAST 2022, Santa Clara, CA, USA, February 22-24, 2022*, pages 17–34. USENIX Association, 2022.

[68] Eunji Lee, Hyokyung Bahn, and Sam H. Noh. Unioning of the buffer cache and journaling layers with non-volatile memory. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 73–80, San Jose, CA, February 2013. USENIX Association.

[69] Gang Liu, Kenli Li, Zheng Xiao, and Rujia Wang. PS-ORAM: Efficient crash consistency support for oblivious RAM on NVM. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, pages 188–203, New York, NY, USA, 2022. Association for Computing Machinery.

[70] Per Ekemark, Yuan Yao, Alberto Ros, Konstantinos Sagonas, and Stefanos Kaxiras. TSOPER: Efficient coherence-based strict persistency. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 125–138, 2021.

[71] Youyou Lu, Jiwu Shu, and Long Sun. Blurred persistence: Efficient transactions in persistent memory. *ACM Trans. Storage*, 12(1), January 2016.

[72] Hsien-Hsin S. Lee, Gary S. Tyson, and Matthew K. Farrens. Eager writeback - a technique for improving bandwidth utilization. In *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 33, Monterey, California, USA, December 10-13, 2000*, pages 11–21. ACM/IEEE Computer Society, 2000.

[73] Moinuddin K. Qureshi, Michele Franceschini, Ashish Jagmohan, and Luis A. Lastras. Preset: Improving performance of phase change memories by exploiting asymmetry in write times. In *39th International Symposium on Computer Architecture (ISCA 2012), June 9-13, 2012, Portland, OR, USA*, pages 380–391. IEEE Computer Society, 2012.

[74] Li Zhao, Ravi Iyer, Srihari Makineni, Don Newell, and Liqun Cheng. NCID: A non-inclusive cache, inclusive directory architecture for flexible and efficient cache hierarchies. In *Proceedings of the 7th ACM International Conference on Computing Frontiers*, CF '10, page 121–130, New York, NY, USA, 2010. Association for Computing Machinery.

[75] Daniel Molka, Daniel Hackenberg, Robert Schone, and Wolfgang E. Nagel. Cache coherence protocol and memory performance of the Intel Haswell-EP architecture. In *Proceedings of the 2015 44th International Conference on Parallel Processing (ICPP)*, ICPP '15, page 739–748, USA, 2015. IEEE Computer Society.

[76] Yan Solihin. *Fundamentals of Parallel Multicore Architecture*. Chapman and Hall/CRC, Berlin, Germany, 1st edition, 2015.

[77] Fan Yao, Milos Doroslovacki, and Guru Venkataramani. Are coherence protocol states vulnerable to information leakage? In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 168–179, 2018.

[78] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. Attack directories, not caches: Side channel attacks in a non-inclusive world. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 888–904, 2019.

[79] Carole-Jean Wu, Aamer Jaleel, Will Hasenplaugh, Margaret Martonosi, Simon C. Steely, and Joel Emer. SHiP: Signature-based hit predictor for high performance caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 430–441, New York, NY, USA, 2011. Association for Computing Machinery.

[80] Jinchun Kim, Elvira Teran, Paul V. Gratz, Daniel A. Jiménez, Seth H. Pugsley, and Chris Wilkerson. Kill the program counter: Reconstructing program behavior in the processor cache hierarchy. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 737–749, New York, NY, USA, 2017. Association for Computing Machinery.

[81] Kunal Korgaonkar, Ishwar Bhati, Huichu Liu, Jayesh Gaur, Sasikanth Manipatruni, Sreenivas Subramoney, Tanay Karnik, Steven Swanson, Ian Young, and Hong Wang. Density tradeoffs of non-volatile memory as a replacement for SRAM based last level cache. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 315–327, 2018.

[82] Subhash Sethumurugan, Jieming Yin, and John Sartori. Designing a cost-effective cache replacement policy using machine learning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 291–303, 2021.

[83] Diya Joseph, Juan L. Aragón, Joan-Manuel Parcerisa, and Antonio González. TCOR: A tile cache with optimal replacement. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 662–675, 2022.

[84] Chandrashis Mazumdar, Prachatos Mitra, and Arkaprava Basu. Dead page and dead block predictors: Cleaning tlbs and caches together. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 507–519, 2021.

[85] Samira Manabi Khan, Yingying Tian, and Daniel A. Jiménez. Sampling dead block prediction for last-level caches. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 175–186, 2010.

[86] Seunghee Shin, James Tuck, and Yan Solihin. Hiding the long latency of persist barriers using speculative execution. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*, pages 175–186. ACM, 2017.

[87] Per Ekemark, Yuan Yao, Alberto Ros, Konstantinos Sagonas, and Stefanos Kaxiras. TSOPER: efficient coherence-based strict persistency. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2021, Seoul, South Korea, February 27 - March 3, 2021*, pages 125–138. IEEE, 2021.

[88] Wikipedia contributors. Write buffer. https://en.wikipedia.org/w/index.php?title=Write_buffer&oldid=1067314254, March 2023. Accessed: 03-01-2023.

[89] Yong Chen, Surendra Byna, Xian-He Sun, Rajeev Thakur, and William Gropp. 2008 international conference on parallel processing september 8-12, 2008 portland, oregon exploring parallel I/O concurrency with speculative prefetching. In *2008 International Conference on Parallel Processing, ICPP 2008, September 8-12, 2008, Portland, Oregon, USA*, pages 422–429. IEEE Computer Society, 2008.

[90] Jih-Kwon Peir, Shih-Chang Lai, Shih-Lien Lu, Jared Stark, and Konrad Lai. Bloom filtering cache misses for accurate data speculation and prefetching. In Kemal Ebcioglu, Keshav Pingali, and Alex Nicolau, editors, *Proceedings of the 16th international conference on Supercomputing, ICS 2002, New York City, NY, USA, June 22-26, 2002*, pages 189–198. ACM, 2002.

[91] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 88–105, 2019.

[92] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-privilege-boundary data sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 753–768, New York, NY, USA, 2019. Association for Computing Machinery.

[93] Priyank Faldu and Boris Grot. Leeway: Addressing variability in dead-block prediction for last-level caches. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 180–193, 2017.

[94] Intel. Introduction to cache allocation technology in the intel® xeon® processor e5 v4 family. https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-cache-allocation-technology.html, March 2023. Accessed: 03-04-2023.

[95] Lucia Pons, Vicent Selfa, Julio Sahuquillo, Salvador Petit, and Julio Pons. Improving system turnaround time with intel cat by identifying llc critical applications. In *Euro-Par 2018: Parallel Processing: 24th International Conference on Parallel and Distributed Computing, Turin, Italy, August 27-31, 2018, Proceedings 24*, pages 603–615. Springer, 2018.

[96] Xiaojian Wu and A. L. Narasimha Reddy. SCMFS: A file system for storage class memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, New York, NY, USA, 2011. Association for Computing Machinery.

[97] Alexandro Baldassin, João Barreto, Daniel Castro, and Paolo Romano. Persistent memory: A survey of programming support and implementations. *ACM Comput. Surv.*, 54(7), jul 2021.

[98] Tianhao Zheng, Haishan Zhu, and Mattan Erez. SIPT: Speculatively indexed, physically tagged caches. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 118–130, 2018.

[99] Ajeya Naithani and Lieven Eeckhout. Reliability-aware runahead. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 772–785, 2022.

[100] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid Volatile/Non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, Santa Clara, CA, February 2016. USENIX Association.

[101] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. SplitFS: reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pages 494–508. ACM, 2019.

[102] Asit K. Mishra, Xiangyu Dong, Guangyu Sun, Yuan Xie, N. Vijaykrishnan, and Chita R. Das. Architecting on-chip interconnects for stacked 3D STT-RAM caches in CMPs. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 69–80, New York, NY, USA, 2011. Association for Computing Machinery.

[103] Mayank Parasar, Abhishek Bhattacharjee, and Tushar Krishna. SEESAW: Using superpages to improve VIPT caches. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 193–206, 2018.

[104] Dhinakaran Pandiyan and Carole-Jean Wu. Quantifying the energy cost of data movement for emerging smart phone workloads on mobile platforms. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 171–180, Oct 2014.

[105] Xiyuan Tang, Linxiao Shen, Begum Kasap, Xiangxing Yang, Wei Shi, Abhishek Mukherjee, David Z. Pan, and Nan Sun. An energy-efficient comparator with dynamic floating inverter amplifier. *IEEE Journal of Solid-State Circuits*, 55(4):1011–1022, 2020.

[106] D. Bertozzi, L. Benini, and G. De Micheli. Error control schemes for on-chip communication links: the energy-reliability tradeoff. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(6):818–831, 2005.

[107] Benton H. Calhoun, Yu Cao, Xin Li, Ken Mai, Lawrence T. Pileggi, Rob A. Rutenbar, and Kenneth L. Shepard. Digital circuit design challenges and opportunities in the era of nanoscale CMOS. *Proceedings of the IEEE*, 96(2):343–365, 2008.

[108] Lillian Pentecost, Alexander Hankin, Marco Donato, Mark Hempstead, Gu-Yeon Wei, and David Brooks. NVMExplorer: A framework for cross-stack comparisons of embedded non-volatile memories. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 938–956, 2022.

[109] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. Association for Computing Machinery.