Hot-Page-Aware Checkpointing for Flash SSDs

Geunhyun Park
Graduate School of Data Science
Seoul National University
Seoul, Republic of Korea
ghpark0116@snu.ac.kr

Sang-Won Lee*
Graduate School of Data Science
Seoul National University
Seoul, Republic of Korea
swlee69@snu.ac.kr

***Abstract*—Flash-based SSDs have become the dominant storage medium for modern database management systems. Although SSDs provide high random read throughput, sustained writes can generate greater internal device overhead than reads due to flash management activities, even when individual write requests are acknowledged quickly. This asymmetry creates performance challenges for checkpointing, a core DBMS mechanism that bounds crash recovery time and controls log growth. Specifically, in write-intensive OLTP workloads, checkpointing generates substantial background write traffic, which can become a bottleneck on SSDs. Additionally, a small number of hot pages often remain dirty across many checkpoint intervals and are repeatedly selected for checkpoint flushing, leading to redundant writeback that yields little durable progress while consuming device bandwidth and increasing write-induced interference. These redundant writes exacerbate the performance issues by causing noticeable latency that degrades transactional throughput and overall system efficiency. To better align checkpointing with SSD characteristics, we present a novel hot-page-aware checkpointing technique that reduces redundant writes, ultimately improving SSD endurance and overall DBMS performance. Our lightweight per-page mechanism tracks consecutive checkpoint selections and defers unnecessary flushes. We implement this in MySQL/InnoDB and evaluate it using TPC-C workloads. Experiment results show that it improves transaction throughput by 1.58× compared to Vanilla MySQL.**

***Index Terms*—checkpointing, OLTP, database system, flash SSDs**

I. Introduction

Solid State Drives (SSDs) have become the de facto storage medium for modern database management systems (DBMSs) [1], [2]. This shift is particularly pronounced in the online transaction processing (OLTP) market, where organizations have rapidly migrated from hard disk drives (HDDs) to SSDs to benefit from improved power efficiency and substantially higher random I/O operations per second (IOPS). Despite this progress, SSDs have a fundamental read–write asymmetry that distinguishes them from HDDs: reads are typically faster and less disruptive than writes. Writes incur additional costs due to out-of-place updates, flash translation layer (FTL) indirection, garbage collection, and write amplification, which can collectively reduce effective bandwidth and increase tail latency [3]. This asymmetry becomes particularly problematic in write-intensive OLTP workloads, where updates are typically highly skewed and a small set of hot pages absorbs a disproportionate share of writes [4], [5]. An et al. shows that the write working set, pages written more than once within a time window, accounts for only 0.54% (300 MB) of all accessed pages on average [6]. As a result, DBMS components that schedule and issue background writes can dominate end-to-end transaction performance by inducing substantial I/O contention and throttling the high read throughput that is the key advantage of flash storage [7].

Checkpointing in DBMSs generates substantial write traffic and serves critical durability functions. It preserves data durability and limits log size by periodically flushing dirty pages from the buffer pool to persistent storage. Importantly, while maintaining these durability guarantees is vital, checkpointing also influences performance by bounding crash recovery time. However, it can exert significant write pressure by forcing background flushes that contend with foreground reads, consuming device bandwidth, and potentially amplifying SSD internal work. Moreover, checkpointing may repeatedly flush hot pages, leading to redundant writebacks when the same page is persisted multiple times within a short interval. These redundant writes accelerate SSD wear, reducing device lifetime, and could saturate the SSD. Consequently, frequent flushing of hot pages not only delays latency-sensitive read requests but also degrades overall transaction throughput, turning fast-read flash storage into a write-bound bottleneck [8]. Nonetheless, our proposed method ensures recovery-time objectives remain intact, reassuring risk-averse stakeholders.

We propose a hot-page-aware checkpoint optimization that explicitly accounts for the performance asymmetry of SSDs. The design goal is to reduce unnecessary checkpoint-induced writes for hot pages while preserving correctness and maintaining practical recovery behavior. By decreasing redundant writeback to the SSD, the DBMS reduces write-induced interference and frees device bandwidth for read operations, enabling faster page fetches on demand [6]. This shift is particularly beneficial for OLTP workloads where transaction execution frequently depends on timely reads from the storage layer (e.g., cache misses, secondary index traversals, and occasional cold-page accesses). In our approach, limiting hot page checkpoint writes is not merely an optimization of background I/O. It is a mechanism for improving the overall balance of device utilization, translating into higher transaction throughput.

The contributions of this paper are as follows.

- Problem characterization: We identify and motivate hot

*Sang-Won Lee is the corresponding author.

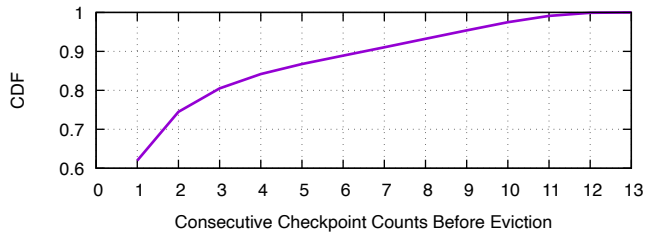

Fig. 1. Cumulative Distribution of Consecutive Checkpoint Count Before Buffer Cache Eviction

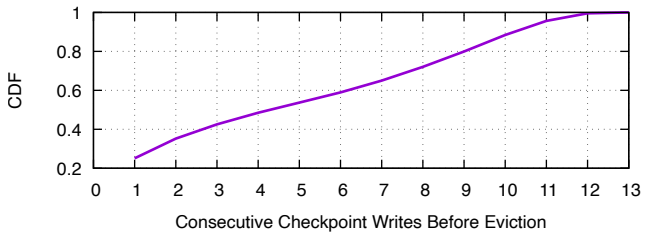

Fig. 2. Cumulative Distribution of Consecutive Checkpoint Writes Before Buffer Cache Eviction

page checkpointing as a source of redundant write traffic under skewed update workloads, and explain why this behavior is particularly costly on SSDs due to write amplification and read-write interference.
- Hot-page-aware checkpoint optimization: We introduce a checkpointing strategy that reduces checkpoint writeback for hot pages.
- System implementation in MySQL [9]: We integrate the optimization into MySQL, demonstrating that hot page awareness can be incorporated without altering transactional semantics.
- Performance impact: We show that reducing checkpoint hot page writes decreases SSD write volume and improves the DBMS's ability to read pages quickly, yielding higher overall transaction performance.

II. Background and Motivation

A. I/O Asymmetry in SSDs

NAND flash memory exhibits an inherent read-write asymmetry because writing a page takes significantly longer than reading one. Specifically, flash memory enforces an erase-before-write constraint at the block granularity, while read operations occur at the page granularity. In-place overwrite is not possible, so an SSD typically updates a logical page by writing new data to a new physical location and marking the old version as invalid. This out-of-place update process is managed by the flash translation layer (FTL), which is responsible for address mapping, wear leveling, and garbage collection. When free space becomes scarce, garbage collection reclaims blocks by copying valid pages elsewhere and erasing entire blocks, introducing additional internal writes and latency variability. Consequently, write traffic can trigger write amplification, and sustained write load can consume controller bandwidth and device-level parallelism that would otherwise serve reads. Therefore, SSD read IOPS often exceed write IOPS by several times, depending on the individual flash storage and the level of over-provisioning. In some cases, the read-write gap can exceed an order of magnitude (e.g., more than 13×) [8], [10].

B. Database Checkpointing

Most modern DBMSs rely on a buffer pool to cache frequently accessed pages in memory. A transaction modifies a page in the buffer pool, which turns the page dirty as it now differs from its on-disk version [11]. To preserve durability and support crash recovery, DBMSs use write-ahead logging (WAL) to record updates in a persistent log before flushing the dirty page to disk [12].

Checkpointing bridges in-memory updates and persistent storage by periodically flushing dirty pages from the buffer pool to the persistent storage. It establishes a log position up to which all updates are guaranteed to be recoverable by ensuring that the corresponding dirty pages have been written back to persistent storage to a sufficient extent [13]. The goals of checkpointing are to reduce recovery time by limiting the log replay range and to ensure database durability by establishing a log position up to which all updates are recoverable [14].

C. Checkpointing Hot Pages

To quantify the prevalence of hot-page persistence under checkpointing, we characterize how long dirty pages remain resident across consecutive checkpoint intervals before they are evicted from the buffer cache. To investigate this in OLTP workloads, we track the number of consecutive checkpoints before buffer cache eviction for each 4KB page during 2-hour runs of the TPC-C [15] benchmark, a widely used OLTP benchmark that exhibits intensive random read and write operations. Our experiments were performed with 500 warehouses (54 GB) database and 32 client threads. We modified MySQL/InnoDB to collect this data under a 50% buffer cache setting. Figure 1 shows that most pages are only briefly exposed to checkpointing. Specifically, around 62% experience checkpointing exactly once, and by the third checkpoint, the cumulative fraction reaches approximately 81%. Thus, most pages do not remain resident long enough for frequent checkpoint writeback. However, a smaller segment persists across many checkpoint intervals, indicating hot-page residency. For example, Figure 2 illustrates pages that stay through 5 or more consecutive checkpoints account for roughly 51% of checkpoint writes, and those with 10 or more checkpoints contribute about 20%. This indicates that checkpoint-induced write traffic is highly skewed towards a small number of long-resident pages. This observation motivates hot-page-aware checkpointing in reducing redundant writebacks on flash SSDs.

D. Motivation

Checkpointing is essential for durability and bounded crash recovery. However, it can become a dominant source of

Algorithm 1 Hot-Page-Aware Checkpointing Algorithm

```
1: function CHECKPOINT
2:     for each page ∈ flush_list do
3:         if page.ckpt_count < max_ckpt_count then
4:             page.ckpt_count ← page.ckpt_count + 1
5:             Remove page from flush_list
6:         else
7:             FLUSH(page)
8:         end if
9:     end for
10: end function
```

background write traffic in WAL-based DBMSs. By flushing dirty pages to advance the recovery point and control log growth, the DBMS injects sustained writes into the storage subsystem. On flash SSDs, where read operations are much quicker than write operations, this persistent writing interferes by consuming bandwidth and resources, which reduces transaction processing speed. Yet, current checkpointing logic is largely 'flash-agnostic.' It prioritizes memory management and recovery bounds, without considering the impact of writing on flash storage.

The problem is exacerbated under skewed OLTP workloads that repeatedly update a small working set. In such settings, a small subset of hot pages accounts for most dirty-page generation. Checkpointing may flush these hot pages multiple times within short intervals because they are quickly dirtied again. Redundant writebacks not only contribute little to durable progress but also significantly increase total write volume. This elevated write volume leads to greater SSD wear and write-induced interference, which delays incoming reads. Ultimately, such behavior causes multiple writes of the same logical page within a short interval, increasing write amplification without meaningfully decreasing future checkpoint load.

These observations motivate hot-page-aware checkpointing that prioritizes efficient writeback on flash storage. By reducing unnecessary checkpoint writes for pages that are likely to be re-dirtied soon, a DBMS benefits from lower write pressure, better preservation of SSD resources for reads, and improved overall transaction performance. At the same time, this approach maintains acceptable checkpoint advancements for practical recovery time and log-size control. In summary, the motivation for optimizing checkpoints for hot pages is not to eliminate checkpointing costs, but to improve the efficiency of checkpoint I/O on flash storage.

III. Implementation

A. Overview

We implement a hot-page-aware checkpointing mechanism designed to reduce redundant writes on flash SSDs. Our key observation is that pages repeatedly chosen for checkpointing are often hot. These pages remain dirty across multiple checkpoint intervals and are likely to be updated again. Since SSD writes are more disruptive than reads, these redundant writes consume device bandwidth and controller resources, delaying reads and reducing transaction throughput. Accordingly, our method does not define hotness by the number of write accesses. Instead, it focuses on pages that persist in the dirty state across successive checkpoint intervals and are repeatedly chosen as checkpoint flush targets, since these pages are the most likely to incur redundant checkpoint writeback. Our goal is to minimize checkpoint-triggered writes for such hot pages while preserving WAL correctness, bounding recovery time, and controlling log growth.

We integrate a hot-page deferral mechanism into the checkpoint flush pipeline. We do not modify foreground update logic or the WAL protocol. Instead, we act when a dirty page is eligible and selected for the checkpoint flush target. While other flushes, such as LRU flush, continue without restrictions, we selectively defer checkpoint-triggered writes. Algorithm 1 illustrates the architecture and the procedure of hot-page-aware checkpointing policy.

B. Hot-Page-Aware Checkpointing

We implement hot-page awareness as a lightweight per-page mechanism integrated directly into checkpoint-driven flushing. Each dirty page maintains a `checkpoint_count` field, which records how many consecutive checkpoint selections have resulted in deferral rather than writeback. A global configuration parameter, `max_checkpoint_count`, bounds the number of times a page may be deferred before it must be flushed.

The mechanism is intentionally minimally intrusive. When a page is inserted into the flush list or other dirty-page tracking structures, it remains a normal candidate for checkpoint flushing, and we continue to use the existing list for compatibility with other flushing policies. The hot-page logic is applied only when the checkpoint subsystem selects a page as a flush target. At that point, the system checks the page's `checkpoint_count`. If the count is below `max_checkpoint_count`, the system increments the counter, removes the page from the checkpoint candidate set for that cycle, and skips issuing the checkpoint write. If the count reaches the threshold, the page is flushed, and the counter is reset.

This policy targets the common case in write-intensive OLTP workloads where a small subset of pages remains dirty across multiple checkpoint intervals and is repeatedly selected for checkpoint flushing. In such cases, immediately flushing the page often yields only limited, temporary progress, because the page is likely to be dirtied again soon while still incurring the full cost of a storage write. By deferring checkpoint writes for these persistently selected pages within a bounded budget, the system shifts checkpoint effort toward dirty pages that are more likely to remain clean after writeback, thereby improving checkpoint I/O efficiency.

The design preserves compatibility with existing buffer management and correctness guarantees. Deferral applies only to checkpoint-triggered flushes; other mechanisms, such as LRU-based flushing, eviction, or explicit single-page flushes, may still write the page at any time under memory pressure.

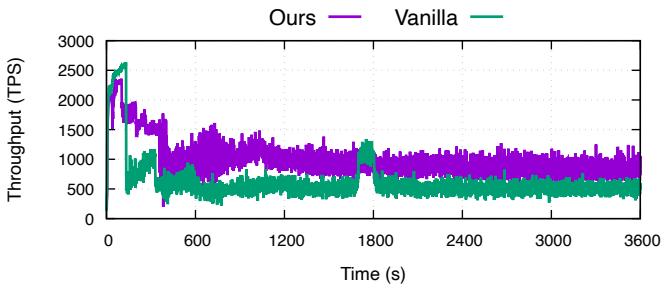

Fig. 3. TPC-C Throughput (Buffer Size = 50%)

TABLE I
PERFORMANCE METRICS (TPC-C, BUFFER SIZE = 50%)

Metrics	Vanilla	Ours
Average TPS	630	995
Write/tx (KB)	67.3	34.3
Read/tx (KB)	19.1	16.6
Write/Sec. (MB)	41.4	33.3
Read/Sec. (MB)	11.7	16.1
User CPU (%)	26.5	40.6
LRU Flush (%)	0.9	56.9
Checkpoint Flush (%)	99.1	43

To avoid stale state, `checkpoint_count` is reset whenever a page is successfully flushed or evicted. Since the counter is updated only when a page is chosen for checkpoint flushing, runtime overhead remains low and depends on checkpoint selection frequency rather than on every read or write. Counter updates use the existing synchronization already required for flush decisions.

Overall, the policy reduces redundant checkpoint write-back while maintaining bounded persistence delay. Once the deferral limit is reached, the system forces the page to be written, ensuring eventual checkpoint progress and preserving WAL-based recovery guarantees. Under these constraints, the approach trades slightly delayed data persistence for improved checkpoint I/O efficiency.

C. Recovery

Crash recovery in a WAL-based DBMS relies on the invariant that log records describing updates reach stable storage before the corresponding data pages are persisted. This enables the system to reconstruct a consistent state after failure by replaying the log beyond the last checkpointed recovery point. A checkpoint advances this recovery point. It does so by ensuring the effects of updates up to a given log position have been propagated to stable storage to a sufficient extent. This limits the amount of redo required and bounds recovery time. Our hot-page-aware checkpointing does not change these recovery invariants or the WAL protocol. It still guarantees eventual persistence through a bounded deferral budget. It also allows other flush paths to write pages at any time. As a result, correctness is preserved and recovery proceeds normally. Checkpoint progress continues to constrain log growth and the redo range.

D. Prototype Implementation

We implement the hot-page-aware flush policy inside MySQL/InnoDB by extending the existing dirty-page and checkpoint flushing pipeline. We preserve InnoDB's WAL semantics and its current flush paths. InnoDB maintains dirty pages in buffer-pool structures, and drives writeback through background flush threads under several triggers, including checkpoint advancement, LRU pressure, and explicit single-page flushes. Our policy intervenes only at the checkpoint-driven path. This intervention happens immediately after a page is selected as a candidate for checkpoint flushing and just before the flush operation is dispatched to the I/O subsystem. These changes localize the hot-page logic to the checkpoint flushing decision, requiring only minimal additions to per-page metadata and list handling. Our approach preserves InnoDB's correctness and operational behavior while reducing redundant checkpoint writeback on flash SSDs.

IV. PERFORMANCE EVALUATION

A. Experimental Setup

All experiments were conducted on a Linux server equipped with an Intel Xeon Silver 4216 CPU (2.10GHz, 16 cores, 32 threads) and 128GB of main memory. We used a Samsung SSD 970 EVO Plus 250GB for database storage and a Samsung SSD 970 EVO 1TB for database log storage, with the ext4 file system in direct I/O mode. All NVMe SSDs are connected to the host via a PCIe interface. We configured the database page size to 4KB, and the number of concurrently running client threads to 32. We used tpcc-mysql to execute the TPC-C workload. TPC-C is an industry standard OLTP benchmark that generates a high volume of random read and write transactions. For all experiments, we initialized the database size to 54GB (500 warehouses).

B. Overall Performance

Figure 3 shows Transactions Per Second (TPS) over a 1-hour execution with 50% buffer cache configuration. After an initial warm-up with transient fluctuations, both systems reach a steady state, where our hot-page-aware checkpointing consistently sustains higher throughput than vanilla MySQL. Throughout the hour, our method delivers 1.58× the baseline's average throughput. The gap is persistent over time, indicating that the improvement is not limited to short-lived bursts but reflects a sustained reduction in checkpoint-induced interference during normal OLTP execution.

C. Performance Metrics

Table I summarizes performance metrics for a 50% buffer pool setting, comparing Vanilla MySQL with our hot-page-aware checkpointing. Our approach increases average throughput from 630 TPS to 995 TPS. This improvement coincides with a substantial reduction in write traffic per transaction, decreasing from 67.3 KB to 34.3 KB (49% lower), while read/tx also drops slightly from 19.1 KB to 16.6 KB. At the

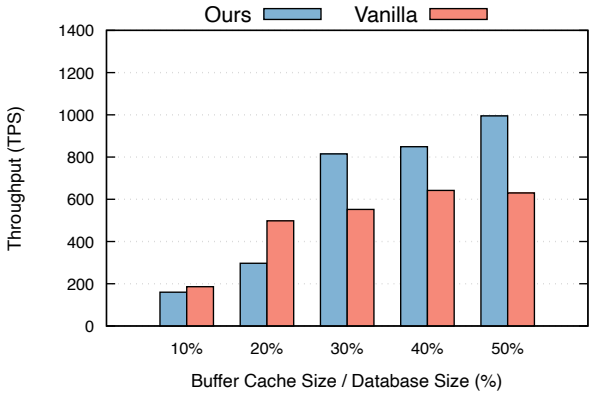

Fig. 4. TPC-C Throughput: Varying Buffer Cache Sizes

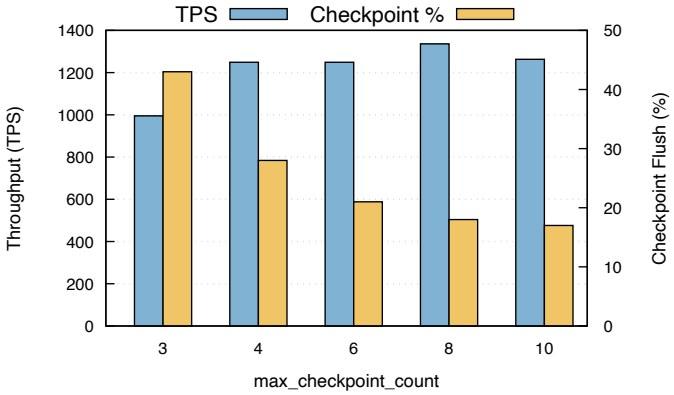

Fig. 5. TPC-C Throughput and Checkpoint Flush Proportion: Varying max_checkpoint_count

device level, sustained write bandwidth decreases from 41.4 MB/s to 33.3 MB/s (20% lower), whereas sustained read bandwidth increases from 11.7 MB/s to 16.1 MB/s (38% higher), consistent with reduced write-induced interference. The resource breakdown further shows that CPU utilization increases from 26.5% to 40.6%, indicating that the system shifts from being I/O-constrained toward doing more useful work. Finally, the flush composition changes significantly. Vanilla MySQL spends 99.1% of its flushing time on checkpoint and only 0.9% on LRU flush, whereas our design reduces the checkpoint flush share to 43% and increases the LRU flush share to 56.9%. This shift aligns with the policy's intent to defer redundant checkpoint flushing of hot pages and rely more on cache-management-driven flushing. Overall, these metrics indicate that hot-page-aware checkpointing reduces unnecessary write traffic and alleviates write-induced contention on flash SSDs, translating device bandwidth savings into improved OLTP execution efficiency.

D. Effects of Buffer Cache Size

Figure 4 shows TPC-C throughput as the buffer pool size increases from 10% to 50% of the database size. At small cache sizes (10% and 20%), both systems are heavily constrained, and throughput remains low. However, once the buffer pool reaches 30% and above, our hot-page-aware checkpointing consistently outperforms Vanilla MySQL, and the gap widens as memory increases. In particular, our throughput rises sharply at 30% and continues to improve through 50%, whereas Vanilla MySQL scales more modestly and begins to plateau.

At small buffer cache sizes (10% and 20%), our method underperforms vanilla because the system is dominated by buffer eviction rather than redundant checkpoint writeback. For example, checkpoint flushing accounts for 64% of write traffic when the buffer cache is 30% of the database size, but only 22% when the buffer cache is reduced to 20%. Under this severe memory pressure, many dirty pages are evicted before they remain resident across multiple checkpoint intervals, limiting the opportunity for checkpoint deferral to coalesce repeated updates. As a result, the overhead of hot page tracking and repeated reconsideration of deferred pages is not amortized. In addition, deferring checkpoint writes under a small buffer can reduce the availability of clean frames and increase urgent buffer eviction flushing, further lowering TPS. Once the buffer cache becomes large enough to retain hot pages across multiple checkpoints, the reduction in redundant writeback outweighs this overhead, leading to higher throughput than vanilla.

Overall, the results indicate that our approach benefits increasingly from larger buffer pools, where it can retain hot pages longer and reduce redundant checkpoint writeback, translating into higher sustained OLTP throughput at medium-to-large cache sizes.

E. Effects of Max Checkpoint Count

Figure 5 evaluates the sensitivity of our approach to `max_checkpoint_count`, which bounds the number of consecutive checkpoint selections a dirty page may defer before it must be flushed. As `max_checkpoint_count` increases, the fraction of flushing attributable to checkpointing decreases monotonically, dropping from about 43% at 3 to 28% at 4, 21% at 6, 18% at 8, and 17% at 10. This trend confirms that a larger deferral budget more aggressively suppresses repeated checkpoint writeback for long-lived dirty pages, thereby reducing checkpoint-induced background I/O.

The throughput response, however, is not monotonic. TPS rises from about 1,000 at `max_checkpoint_count` = 3 to about 1,250 at 4 and 6, and reaches its maximum of about 1,340 at 8, before declining slightly to about 1,260 at 10. Relative to `max_checkpoint_count` = 3, the peak at 8 improves throughput by roughly 35%. This result is important because it shows that minimizing checkpoint flush share alone is not sufficient to maximize performance. Although `max_checkpoint_count` = 10 yields the smallest checkpoint-flush fraction, it does not deliver the highest TPS. In other words, once redundant checkpoint writes have been sufficiently reduced, further deferral provides diminishing returns and can slightly hurt steady-state execution, likely by shifting pressure to other parts of the system, such as

TABLE II
RECOVERY PERFORMANCE (REDO LOG SIZE = 15GB)

Recovery Time (s)	Vanilla	Ours
Analysis	1,655	1,761
Redo	240	706
Undo	0.6	0.3
Total	1,895.6	2,467.3

delayed writeback or buffer replacement. Overall, the figure indicates that moderate-to-high deferral budgets are most effective, and that `max_checkpoint_count` = 8 provides the best balance in our setting between suppressing checkpoint interference and sustaining OLTP throughput.

F. Recovery Performance

Table II quantifies the recovery cost of hot-page-aware checkpointing for the setting `max_checkpoint_count` = 3 under a 15 GB redo log. End-to-end recovery time increases from 1,895.6 seconds to 2,467.3 seconds (30.2% higher). The overhead is not uniform across phases. The analysis phase grows only modestly, from 1,655 s to 1,761 s, whereas the redo phase increases substantially, from 240 s to 706 s. In contrast, undo remains negligible in both configurations. This breakdown shows that the recovery penalty is not caused by additional rollback work, but by a larger redo range that must be replayed after a crash.

This behavior is consistent with the design of our method. By deferring checkpoint writeback for repeatedly selected hot pages, the system reduces redundant background writes during steady-state execution, but it also advances the durable checkpoint boundary more conservatively. As a result, more recent updates remain to be processed during recovery, thereby increasing redo time. Importantly, although redo is nearly tripled, total recovery time does *not* increase by the same factor because recovery is dominated by analysis in both systems. In Vanilla MySQL, analysis accounts for most of the restart cost, and it remains the largest component even with our method. Thus, the main effect of hot-page-aware checkpointing is to shift recovery composition toward more redo work rather than to slow down every recovery phase uniformly.

Overall, Table II highlights an explicit systems trade-off. Hot-page-aware checkpointing improves steady-state performance by eliminating unnecessary checkpoint-induced writes, but it does so by retaining more dirty state across checkpoint intervals, which increases the amount of work left for crash recovery. We defer adaptive checkpoint deferral control, which jointly optimizes runtime throughput and recovery time, to future work.

V. CONCLUSION

This paper identified a fundamental mismatch between conventional checkpointing and flash SSD behavior driven by read–write asymmetry. We showed that dirty-page dynamics in OLTP workloads are highly skewed, with a small subset of hot pages persisting across many checkpoint intervals and being repeatedly selected for checkpoint flushing. This behavior leads to redundant writeback that provides limited durable progress while increasing write pressure and write-induced interference.

To address this inefficiency, we proposed a hot-page-aware checkpointing technique that defers checkpoint-triggered flushes for repeatedly selected pages using a lightweight per-page counter and a bounded deferral limit. The design preserves WAL correctness and remains compatible with existing flush paths such as LRU/eviction and single-page flush, ensuring that memory management and operational constraints are not compromised. Overall, our results indicate that incorporating hot-page awareness into checkpointing is a practical step toward improving checkpoint I/O efficiency on flash storage.

ACKNOWLEDGMENT

This work was supported by the National Research Foundation of Korea (NFR) under project BK21 FOUR (Dept. of Data Science, SNU) and by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (RS-2024-00436569).

REFERENCES

[1] R. Appuswamy, G. Graefe, R. Borovica-Gajic, and A. Ailamaki, "The five-minute rule 30 years later and its impact on the storage hierarchy," *Commun. ACM*, vol. 62, no. 11, p. 114–120, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3318163

[2] G. Haas and V. Leis, "What Modern NVMe Storage Can Do, and How to Exploit it: High-Performance I/O for High-Performance Storage Engines," *Proc. VLDB Endow.*, vol. 16, no. 9, p. 2090–2102, May 2023. [Online]. Available: https://doi.org/10.14778/3598581.3598584

[3] G. Wu and X. He, "Reducing SSD read latency via NAND flash program and erase suspension," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, ser. FAST'12. USA: USENIX Association, 2012, p. 10.

[4] S. T. Leutenegger and D. Dias, "A modeling study of the TPC-C benchmark," in *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '93. New York, NY, USA: Association for Computing Machinery, 1993, p. 22–31. [Online]. Available: https://doi.org/10.1145/170035.170042

[5] C. Ruemmler and J. Wilkes, "A trace-driven analysis of disk working set sizes," Hewlett-Packard Laboratories, Tech. Rep., 1993. [Online]. Available: https://api.semanticscholar.org/CorpusID:60477850

[6] M. An, J. Park, T. Wang, B. Nam, and S.-W. Lee, "NV-SQL: Boosting OLTP Performance with Non-Volatile DIMMs," *Proc. VLDB Endow.*, vol. 16, no. 6, p. 1453–1465, Feb. 2023. [Online]. Available: https://doi.org/10.14778/3583140.3583159

[7] M. An, S. Im, D. Jung, and S.-W. Lee, "Your read is our priority in flash storage," *Proc. VLDB Endow.*, vol. 15, no. 9, p. 1911–1923, May 2022. [Online]. Available: https://doi.org/10.14778/3538598.3538612

[8] M. An, I.-Y. Song, Y.-H. Song, and S.-W. Lee, "Avoiding Read Stalls on Flash Storage," in *Proceedings of the 2022 International Conference on Management of Data*, ser. SIGMOD '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1404–1417. [Online]. Available: https://doi.org/10.1145/3514221.3526126

[9] M. Team, "mysql-server." [Online]. Available: https://github.com/mysql/mysql-server

[10] B. Lee, M. An, and S.-W. Lee, "LRU-C: Parallelizing Database I/Os for Flash SSDs," *Proc. VLDB Endow.*, vol. 16, no. 9, p. 2364–2376, May 2023. [Online]. Available: https://doi.org/10.14778/3598581.3598605

[11] R. Ramakrishnan, *Database Management Systems*, 3rd ed. McGraw-Hill, 2002.

[12] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM Trans. Database Syst.*, vol. 17, no. 1, p. 94–162, Mar. 1992. [Online]. Available: https://doi.org/10.1145/128765.128770

[13] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[14] A. Silberschatz, H. Korth, and S. Sudarshan, *Database System Concepts*, 7th ed. McGraw-Hill Education, 2019.

[15] Percona, "tpcc-mysql." [Online]. Available: https://github.com/Percona-Lab/tpcc-mysql