Extending DONUTS: A Comparative Evaluation of Cache Replacement Policies for Efficient Crash Consistency in Non-Volatile Memory

Carlos Eduardo M. Brant[1], **Kleber Kruger**[1]

[1]Campus de Coxim – Universidade Federal de Mato Grosso do Sul (UFMS)
79.400-000 – Coxim – MS – Brazil

carlos.brant@ufms.br, kleber.kruger@ufms.br

Abstract. *Non-volatile memory (NVM) has emerged as a promising technology to replace or complement DRAM in computer systems. However, ensuring crash consistency is a critical challenge in NVM-based systems, as partial updates to data structures can leave the system in an inconsistent state upon failure. This work extends DONUTS, a software-transparent checkpointing mechanism, by integrating additional cache replacement policies beyond the original LRU-Read strategy. Evaluations using SPEC CPU2006 and Splash2 benchmarks demonstrate that alternative cache replacement policies maintain a runtime overhead of approximately 2%, while offering superior efficiency for certain workloads.*

1. Introduction

Non-volatile memory (NVM) is an emerging technology with significant potential for integration into computer system memory hierarchies [Chen 2016]. Unlike DRAM, which loses data upon a power failure, NVM technologies such as Phase-Change Memory (PCM), Spin-Transfer-Torque RAM (STT-RAM), Resistive RAM (ReRAM), and 3D XPoint offer data persistence while providing access latencies that approach those of DRAM [Prasad et al. 2022]. Key benefits of NVM adoption include persistence at the main memory level, eliminating the need for secondary storage access and associated serialization overhead; reduced access latencies for read and write operations; lower energy consumption and higher storage density, enabling systems to achieve greater memory capacity with reduced power dissipation per unit volume [Chen 2016, Kruger et al. 2023a].

Despite their advantages, NVM-based systems face substantial challenges. A primary issue is maintaining data consistency in the presence of system failures [Kruger et al. 2021]. In contrast to conventional systems, which do not require crash consistency guarantees due to their volatile main memory, NVM systems must ensure memory updates can be recovered to a consistent checkpoint state. For instance, an atomic operation updating two cache lines in NVM may corrupt data structures if the system fails after only one line has been persisted, violating atomicity guarantees.

Additional complications include the impracticality of rewriting entire application code bases using transactional memory libraries or persistent memory APIs. As shown in Figure 1, to update the value of a node in a linked list, the code must be rewritten with explicit instructions to demarcate the beginning and end of an atomic operation. This approach imposes significant development overhead, restricting NVM adoption to specially

```
1 void updateNode(int old_value, int
      new_value)
2 {
3   NVHeap *nv = NVHOpen("foo.nvheap");
4   NVList::VPtr a = nv->GetRoot<NVList::
      NVPtr>();
5   AtomicBegin {
6     while(a != nullptr) {
7       if(a->get_value() == old_value)
8         a->set_value(new_value);
9       a = a->get_next();
10    }
11  } AtomicEnd;
12 }
```

(a) Software-based

```
1 void updateNode(int old_value, int
      new_value)
2 {
3     struct List *a = root_pointer;
4     while(a != nullptr) {
5         if(a->value == old_value)
6             a->value = new_value;
7         a = a->next;
8     }
9 }
```

(b) Software-transparent

Figure 1. Examples of software-based and software-transparent implementations, where the latter does not require changes to the application's source code.

written applications. Another challenge is the need for crash consistency mechanisms that do not introduce excessive performance and energy overhead, a requirement frequently driven by the nature of current applications [Chakrabarti et al. 2014].

DONUTS addresses these challenges through a checkpoint strategy that integrates consistency mechanisms into cache replacement policies [Kruger et al. 2021]. Rather than treating checkpointing as an additional mechanism triggered by periodic events, DONUTS recognizes that cache eviction events can be used as checkpointing opportunities. By coordinating evictions with epoch boundaries via modified replacement policies, DONUTS achieves dynamic epoch intervals without requiring explicit periodic timers or instruction counters, allowing for dynamic epoch sizes that adapt to the program's behavior across different execution phases.

This work extends DONUTS by implementing and evaluating multiple cache replacement policies beyond the original LRU-Read. We aim to address the following research questions:

1. How do traditional cache replacement policies affect checkpoint frequency and distribution?
2. Can cache replacement policies derived from traditional models yield better performance-consistency trade-offs under varying application workloads?

To answer both research questions, the specific objectives of this work include implementing variants of traditional replacement policies within DONUTS and conducting comprehensive evaluations in single and multi-core scenarios. These evaluations analyze the relationship between replacement policy behavior, checkpoint characteristics, and system overhead metrics. Evaluations show that LRU, even in memory-intensive workloads, results in an average runtime overhead of less than 2% across all benchmarks.

This paper is organized as follows: Section 2 reviews background concepts, including crash consistency, logging techniques, the DONUTS architecture, and cache replacement policies. Section 3 explains our implementation and Section 4 presents our evaluation model and respective results, with an emphasis on runtime overhead and checkpoint frequency. Finally, Section 5 reviews related work, and Section 6 summarizes this paper and discusses future directions.

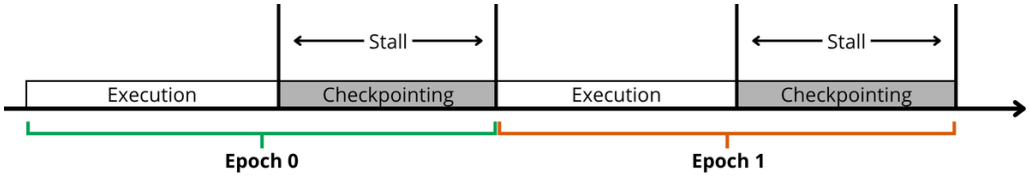
2. Background

2.1. Crash Consistency in NVM-Based Systems

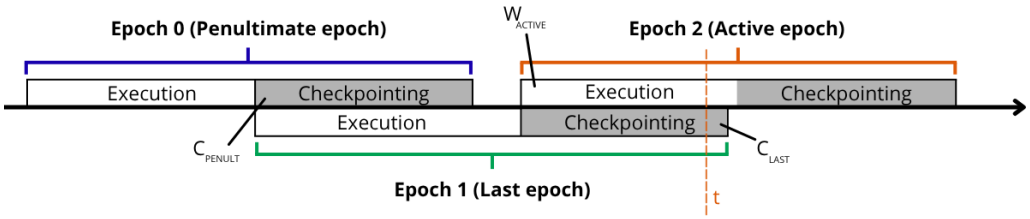
Crash consistency is the property ensuring systems can recover to a valid, consistent memory state after unexpected failures [Kruger et al. 2023a]. In NVM-based systems, crash consistency requires periodic checkpoints (durable snapshots of system state serving as recovery points). Each checkpoint represents a consistent memory image to which the system can revert in the event of a failure.

The epoch-based checkpointing model divides the system execution into time intervals called epochs, with each bounded by two consecutive checkpoints [Kruger et al. 2021]. Each epoch comprises three distinct phases, which are:

- **Execution phase**: Applications execute normally, reading and writing to cache via conventional load/store instructions. Modified cache blocks acquire dirty status, indicating divergence from their persistent versions.
- **Commit phase**: At epoch termination, the system initiates a commit event, marking all dirty blocks as persistence candidates and resetting their status to clean, enabling their replacement by subsequent epoch data.
- **Persistence phase**: While the current epoch executes, dirty blocks from committed epochs are written asynchronously to NVM in the background, producing a durable checkpoint.



(a) Epoch model without overlapping



(b) ThyNVM epochs model with overlapping

Figure 2. Epoch model examples.

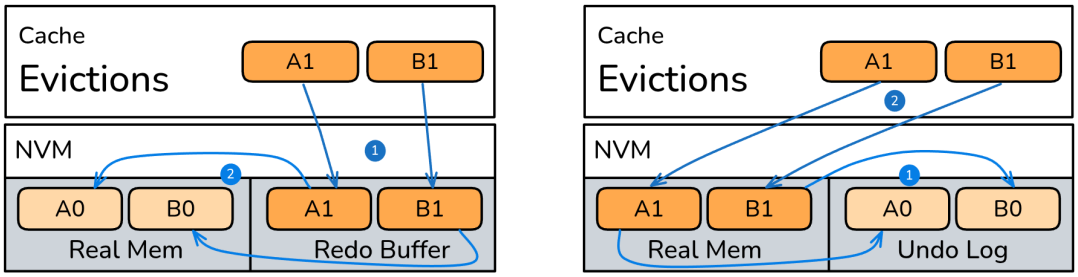
Scheduling these phases effectively is critical for system performance. Sequential execution of these phases without overlaps (Figure 2a) incurs severe performance degradation. Analysis from ThyNVM shows that alternating between execution and checkpointing phases without overlap can consume up to 35.4% of execution time in memory-intensive workloads [Ren et al. 2015]. This bottleneck becomes more pronounced with large cache sizes. For instance, write-backs from 64 MB L3 + 128 MB L4 caches in contemporary server processors can require 480 ms, resulting in significant system stalls. Thus, modern crash consistency mechanisms address this limitation by allowing the current epoch to proceed in parallel with the persistence of previous epochs (Figure 2b).

DONUTS extends this concept by establishing persistence in the background with dynamic epoch boundaries based on cache occupancy thresholds rather than fixed time or instruction intervals [Kruger et al. 2021].

2.2. Write-Ahead Logging

To ensure consistent states in NVM-based systems, crash consistency mechanisms usually persist program data to non-volatile memory using a logging technique. Write-ahead logging (WAL) constitutes the predominant technique, which is categorized into two variants, as shown in Figure 3:

- **Redo Logging**: Upon cache eviction, modified data is temporarily buffered in a redo-log region. This data is written to its original NVM address only after the redo-log entry becomes durably persistent. During recovery, the system executes pending entries, reapplying modifications to recover the most recent consistent state. The primary disadvantage is the indirection required to locate updated values, as the system must consult redo-log mappings to determine if data has been redone or still resides in standard locations.
- **Undo Logging**: Upon cache eviction, the original data (before modification) is first read from its standard NVM address and persisted to an undo-log region. Modified data is subsequently written to its original location. On recovery, the system applies undo-log entries, reverting incomplete updates to restore the last consistent checkpoint. Undo-logging provides direct access semantics (guaranteeing all data in its standard location is consistent) but requires anticipatory logging before modifications reach main memory.



(a) Redo-logging (b) Undo-logging

Figure 3. Redo and undo logging models.

DONUTS employs undo-logging with the critical distinction of performing logging operations via processing-in-memory (PIM) mechanisms. Instead of performing logging operations via the processor's memory controller, DONUTS logs directly within memory hardware using Processing in Memory (PIM) with slight modification to the DDR standard protocol, reducing processor-memory bandwidth bottlenecks by up to two times [Kruger et al. 2023b].

2.3. Cache Replacement Policy

Cache replacement policies are responsible for managing the allocation and use of space in cache memories. By definition, caches are high-speed memories, but with significantly less capacity than main memory. Due to this limitation, they quickly reach their

maximum capacity. Thus, when a cache miss occurs and it is necessary to load a new block of data into the cache, the controller must decide which existing block is evicted to make room. The replacement policy defines the algorithm used to make this decision [John L. Hennessy 2017].

In DONUTS, the choice of replacement policy directly influences system characteristics relevant to crash consistency. For instance, the LLC cache miss rate indirectly determines the epoch length and checkpoint frequency [Kruger et al. 2023a]. Policies that retain dirty blocks longer naturally extend epoch duration, reducing checkpoint frequency. Conversely, policies that aggressively evict data may trigger checkpoint events more frequently, which increases memory bandwidth consumption but potentially reduces the data per checkpoint.

2.3.1. Least Recently Used (LRU)

The Least Recently Used (LRU) policy evicts the block that has not been accessed for the longest time. As illustrated in Figure 4, this policy maintains an ordering of blocks based on access recency, typically utilizing a stack or linked list structure [John L. Hennessy 2017]. In crash consistency contexts, LRU may evict modified blocks simply because they are the least recently used, potentially increasing checkpoint frequency in workloads with scattered access patterns [Nguyen and Wentzlaff 2018].

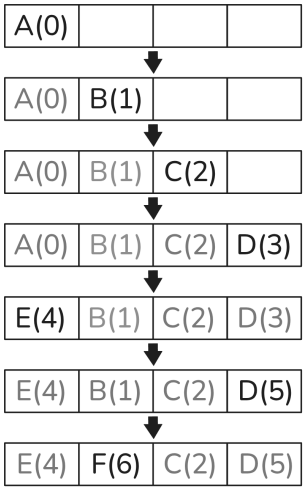


Figure 4. An example of LRU with the access sequence A B C D E D F.

2.3.2. Most Recently Used (MRU)

The Most Recently Used (MRU) policy evicts the most recently accessed block, as shown in Figure 5. While this approach is suboptimal for workloads exhibiting temporal locality, it proves effective in scenarios with sequential or scanning access patterns, where recently used blocks are unlikely to be reused in the near future [John L. Hennessy 2017]. However, in NVM systems, MRU may lead to frequent evictions of actively written (dirty) blocks, potentially increasing checkpoint overhead in write-intensive applications [Wang et al. 2021].

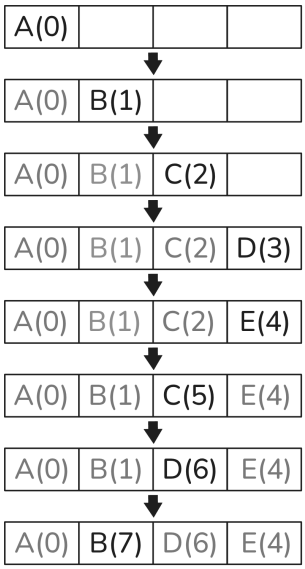


Figure 5. An example of MRU with the access sequence A B C D E C D B.

2.3.3. Not Most Recently Used (NMRU)

The Not Most Recently Used (NMRU) policy selects a victim from all blocks except the most recently used one, often implemented with a single bit per set to track the MRU block. It approximates LRU behavior with reduced hardware overhead [John L. Hennessy 2017]. In consistency-aware setups, NMRU may balance retention of modified blocks while avoiding the full complexity of true LRU, stabilizing epoch durations.

2.3.4. Not Recently Used (NRU)

The Not Recently Used (NRU) policy uses a reference bit per block to track recent accesses. Blocks with cleared reference bits are prioritized for eviction, with periodic bit resets to age references. NRU provides a low-overhead approximation suitable for large caches [John L. Hennessy 2017]. In crash consistency, its aging mechanism could help retain frequently modified dirty blocks longer, reducing unnecessary checkpoints.

2.3.5. Pseudo-LRU (PLRU)

Pseudo-LRU (PLRU) approximates LRU using a tree of bits or a bit vector, requiring fewer bits than true LRU ($\log_2(ways)$ bits per set). As illustrated in Figure 6, the algorithm updates a binary tree to point away from recently used ways during accesses [John L. Hennessy 2017]. In the context of NVM checkpointing, PLRU offers implementation efficiency and effective locality capture, which can lead to fewer evictions of dirty blocks compared to random policies [Nguyen and Wentzlaff 2018].

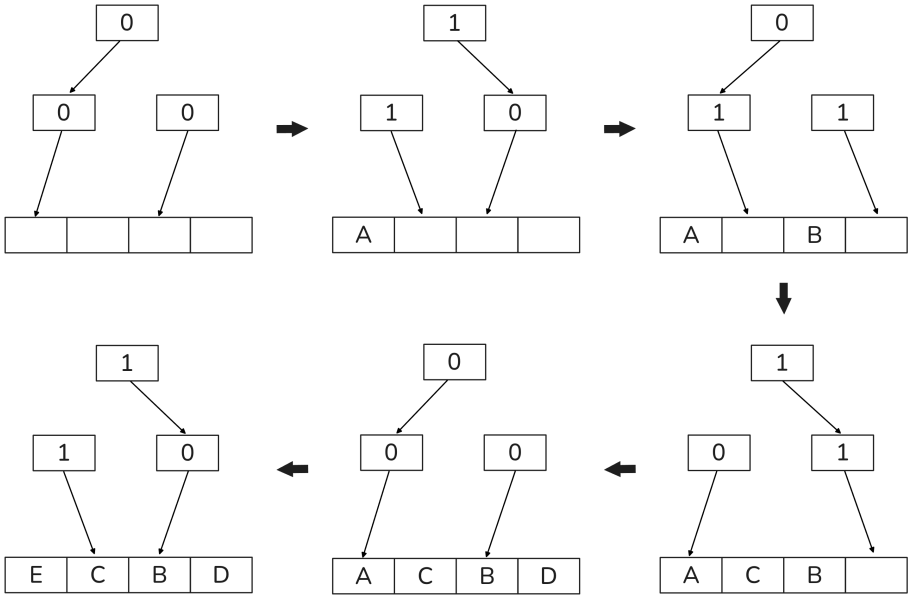


Figure 6. An example of PLRU with the access sequence A B C D E.

2.3.6. Static Re-Reference Interval Prediction (SRRIP)

Static Re-Reference Interval Prediction (SRRIP) assigns re-reference prediction values (RRPV) to blocks, evicting those with the highest RRPV. On insertion, blocks receive an intermediate RRPV, decreased on hits and increased on aging [John L. Hennessy 2017, Jaleel et al. 2010]. SRRIP improves on LRU for thrashing workloads. In NVM systems, its prediction mechanism may better retain frequently re-referenced dirty blocks, reducing checkpoint triggers in interval access patterns [Wang et al. 2021].

2.3.7. Round Robin

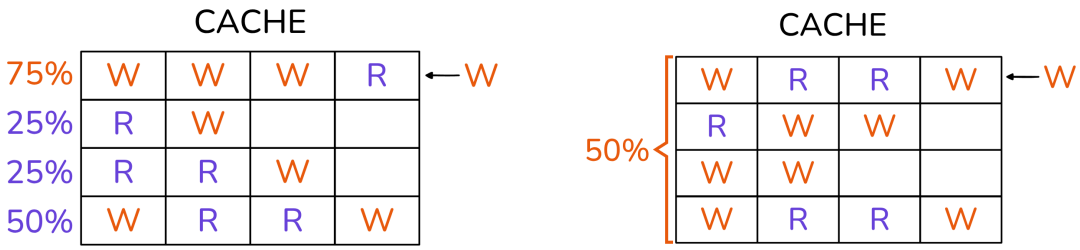
The Round Robin policy cycles through cache ways in a fixed order, evicting the next way in sequence regardless of access history. It offers simple implementation but ignores locality [John L. Hennessy 2017]. For crash consistency, its predictable cycling could lead to uniform distribution of evictions, potentially balancing checkpoint frequency across sets but performing poorly on localized writes [Kruger et al. 2023a].

2.3.8. Random

The Random policy selects a victim block arbitrarily, typically using a pseudo-random number generator. It requires minimal hardware and provides resistance to pathological access patterns that degrade deterministic policies [John L. Hennessy 2017]. In consistency contexts, Random may produce variable epoch lengths, making checkpoint behavior less predictable but robust against adversarial workloads [Baldassin et al. 2020].

2.4. DONUTS

DONUTS is a hardware mechanism that provides software-transparent crash consistency through asynchronous checkpoints, mainly coordinated by events triggered by cache replacement policy [Kruger et al. 2021]. Unlike mechanisms based on fixed-size epochs, DONUTS establishes dynamic epoch boundaries, performed by four distinct conditions:



(a) Cache set-associative threshold (b) Cache capacity threshold

Figure 7. Conditions that trigger the checkpoint in DONUTS.

a) **Cache set-associative threshold**: when a cache set-associative reaches a specified percentage of dirty blocks (Figure 7a), signaling modified data saturation in that set.
b) **Cache capacity threshold**: when the total cache reaches a defined percentage of dirty blocks (Figure 7b), indicating overall cache saturation.
c) **Timeout**: when a specified duration elapses since the last completed checkpoint (e.g., 50 ms), ensuring bounded checkpoint intervals for applications with sparse modification patterns.
d) **Periodic instructions**: when a maximum number of instructions is reached (e.g., 50 million instructions), acting as a fallback mechanism.

When the system reaches one of these conditions, DONUTS commits the current epoch and marks all dirty blocks for asynchronous persistence, effectively establishing dynamic epoch boundaries adapted to actual application behavior rather than predetermined schedules [Kruger et al. 2021].

Figure 8. DONUTS cache replacement policy.

The original DONUTS implementation utilizes LRU-Read as the cache replacement policy, a strategic modification of the traditional LRU algorithm. The Figure 8 shows

the behavior when the associative-set threshold is set to 100%. The green arrows above the cache lines indicate which block is allocated at each step. The fundamental distinction from traditional LRU is that LRU-R victimizes only clean blocks, retaining dirty blocks within the cache for extended periods. This design directly supports crash consistency by minimizing premature evictions of modified data, thereby extending epoch duration and reducing checkpoint frequency [Kruger et al. 2021]. On the right side of each set is the current percentage of its dirty blocks. The crucial moment occurs at step 10, when the set becomes 100% dirty. A block must be evicted, but instead of immediately evicting (which risks losing non-persisted modifications), DONUTS first performs a commit event (CM) at step 11a. During this event, all dirty blocks are marked for asynchronous persistence in the background. Their flags are cleared, turning them into clean blocks again. Only then, at step 11b, can the cache resume normal operation and safely perform the requested allocation.

3. Methodology

We implemented all cache replacement policies using the in Sniper Simulator 8.1, a widely adopted, cycle-accurate, and multi-core x86 simulator that supports in-order, out-of-order, and SMT core models [Carlson et al. 2014, Carlson et al. 2011]. Sniper models the execution of instructions in the processor, memory hierarchy, coherence protocol, and DRAM memory controllers. DONUTS extends the simulator by adding support for NVM models, allowing for different timing for read and write operations in main memory.

Sniper contains seven replacement policies: LRU, PLRU, SRRIP, Random, NMRU, NRU, and Round-Robin. To add new models to the system, we inherit the abstract class CacheSet and then implement its virtual methods, modifying the function `getReplacementIndex` (see Figure 9), called every time the cache needs to evict a block.

```
1  UInt32 getReplacementIndex(CacheCntlr* cntlr)
2  {
3     // Try to find any clean victim using the original policy logic
4     const auto index = ::CacheSetNMRU::getReplacementIndex(cntlr);
5
6     // Check if the dirty blocks count reach the threshold defined
7     // if so, throws a checkpoint event
8     if (countDirtyBlocks() >= m_threshold)
9        cntlr->checkpoint(CheckpointReason::CACHE_SET_THRESHOLD, m_index);
10
11    // After commit there will be clean blocks, so return the index
12    return index;
13 }
```

Figure 9. Function that gets the index for replacement.

Our new cache replacement policy behaves identically to the original algorithms (explained in Section 2). The difference is that it does not choose dirty blocks, and it triggers a checkpoint event when the threshold is reached. Therefore, dirty blocks are marked for asynchronous persistence in the background and revert to a clean state, i.e., they are unmodified again. The implementations are available at github.com/donuts-nvm.

To automate our experiments, we developed a script to run all benchmarks using the task-spooler application according to a test configuration file. We also needed to up-

date the benchmark run scripts to run in a Python 3 environment. These implementations are available at github.com/zbrant/benchmarks.

4. Evaluation

4.1. Simulation Environment and System Configuration

Our experiments were conducted using the configuration summarized in Table 1. The main system parameters include:

Component	Configuration
Processor	Xeon X5550 Gainestown
Frequency	2.66 GHz
L1-I Cache	Private, 32 KB, 4-way, 64B block, 4 cycle hit
L1-D Cache	Private, 32 KB, 8-way, 64B block, 4 cycle hit
L2 Cache	Private, 256 KB, 8-way, 64B block, 8 cycle hit
L3 Cache	Shared, 2 MB, 16-way, 64B block, 30 cycle hit
NVM (STT-RAM)	Read latency: 10 ns; Write latency: 50 ns
Cache Set Threshold	100%
Cache Capacity Threshold	75%
Timeout	50 ns
Max Instructions per Epoch	30 Millions

Table 1. System Configuration

The system configuration reflects parameters commonly used in crash-consistency studies, enabling a direct and fair comparison [Kruger et al. 2023a]. The NVM characteristics were set according to the STT-RAM persistent memory model [Mishra et al. 2024]. The evaluation compares a baseline system (without crash-consistency mechanisms, using a standard LRU replacement policy, and no checkpointing) representing a 0% overhead scenario. This baseline is contrasted against the DONUTS architecture evaluated under five LLC replacement policies: LRU (Least Recently Used), MRU (Most Recently Used), NMRU (Not Most Recently Used), NRU (Not Recently Used), and Random. All DONUTS variants apply the epoch thresholds identified as optimal in prior sensitivity analyses [Kruger et al. 2021], ensuring the best performance/consistency trade-off. These thresholds are:

- Cache capacity limit at `75%`: this threshold offers an effective balance between checkpoint frequency and runtime overhead. Lower values prematurely terminate epochs, resulting in an unnecessarily high checkpoint rate. In contrast, higher values (e.g., above 80%) substantially increase checkpoint latency due to the larger volume of dirty blocks that need to be persisted.
- Cache set threshold at `100%`: prior experiments [Kruger et al. 2021] showed that lower thresholds force early epoch termination, increasing checkpoint frequency by up to 3.8× in some workloads. Setting this parameter to 100% minimizes the overhead associated with frequent epochs triggered within short time intervals.
- Timeout in `50ms`: default value to prevent long epochs in low-write execution phases.

- Maximum instructions per epoch in `30M`: this value is compatible with other projects [Nguyen and Wentzlaff 2018] and serves to ensure that the epoch size does not exceed several instructions.

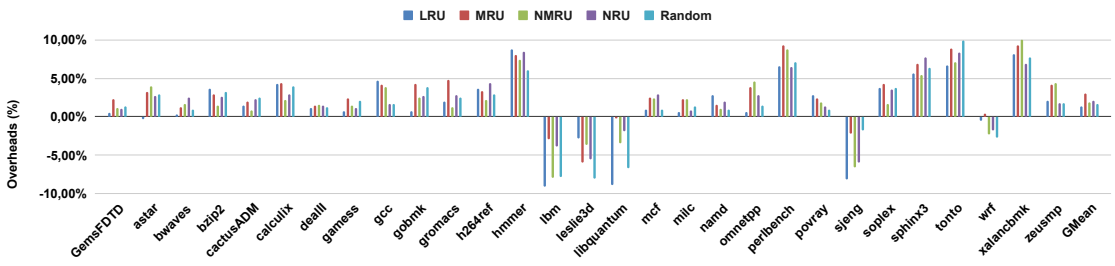
Finally, to evaluate the system under workloads with distinct behaviors, we selected benchmarks that span diverse application domains, computational characteristics, and memory access patterns. Each application was executed for one billion instructions, which provides sufficient statistical coverage for robust performance analysis while maintaining simulation feasibility. This instruction count captures representative phases of each workload without imposing prohibitive simulation time [P and Mishra 2024]:

- SPEC CPU2006: a comprehensive single-core benchmark suite comprising sequential workloads with a wide range of memory intensities, from compute-bound applications (e.g., povray, gcc) to memory-bound ones (e.g., libquantum, bwaves).
- SPLASH-2: a multi-threaded benchmark suite capturing parallel execution behavior, shared-memory communication, and synchronization patterns.

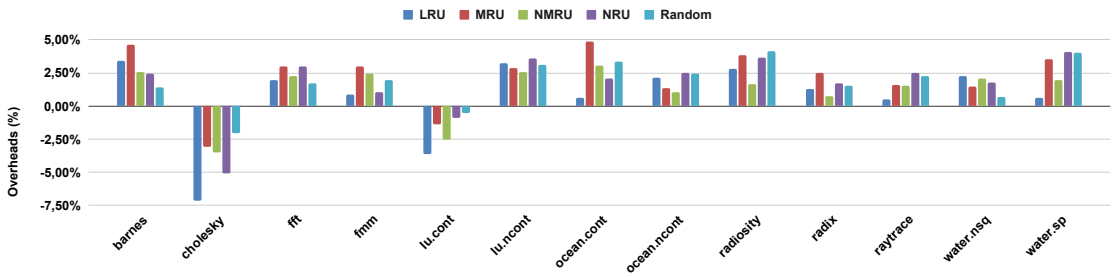
4.2. Results and Analysis

This section reports the evaluation of DONUTS configured with the LRU, NRU, NMRU, MRU, and Random replacement policies developed as part of this work.

4.2.1. Runtime Overhead



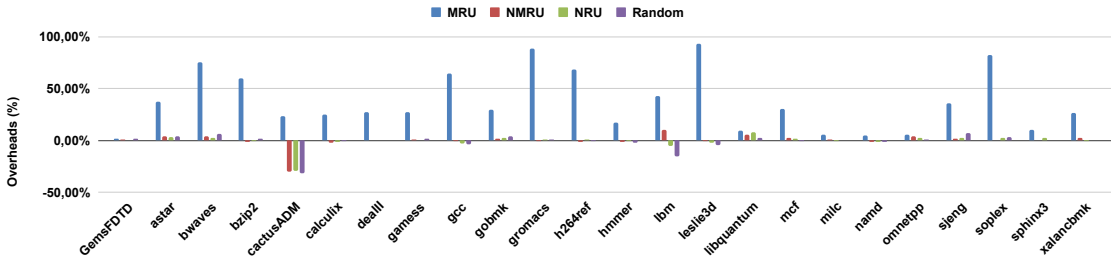
(a) Single core applications.



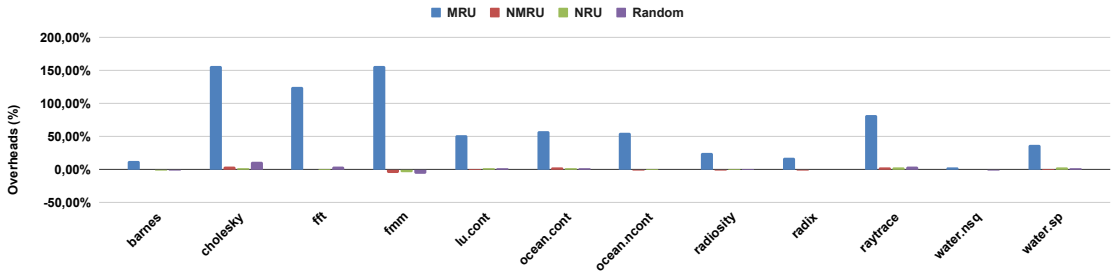
(b) Multicore applications.

Figure 10. Runtime overhead using baseline.

The experimental results derived from the benchmark suites are presented in Figure 10 and Figure 11, which illustrate representative scenarios. In these plots, the x-axis



(a) Single core applications.



(b) Multicore applications.

Figure 11. Runtime overhead using LRU as baseline.

denotes the benchmark applications, while the y-axis depicts the percentage overhead incurred by each replacement policy.

Figure 10 presents the runtime overhead of DONUTS with the five evaluated replacement policies, normalized to the baseline system with no crash-consistency mechanism. All policies exhibit very low average overhead: the geometric mean across SPEC CPU2006 benchmarks ranges from 1.38% (LRU) to 3% (MRU), while Splash2 multicore workloads show even lower values, ranging from 0.64% (LRU) to 2.15% (MRU).

LRU achieves the lowest overhead in both single and multi-core scenarios. This result is expected because workloads with strong temporal locality in their write patterns (common in SPEC CPU2006 applications such as *libquantum, lbm, mcf, milc, bwaves* and *GemsFDTD*) benefit from LRU retaining recently written (dirty) blocks longer in the cache. Since recently modified blocks are, by definition, the most recently used, LRU naturally delays their eviction, extending epoch duration and reducing checkpoint frequency (one of the primary sources of overhead in DONUTS). Conversely, MRU consistently shows the highest overhead because it aggressively evicts recently accessed blocks, which, in write-intensive phases, are often the dirty blocks. This forces frequent set-level commits (Cache Set Threshold = 100%), increasing the checkpoint rate and logging traffic.

Figure 11 shows the result for runtime overhead using LRU as the baseline. Overall, the data reinforce that policies with strong recency bias deliver the best performance in DONUTS because they align with the mechanism's goal of maximizing dirty blocks retention.

4.2.2. Checkpoint Frequency Analysis

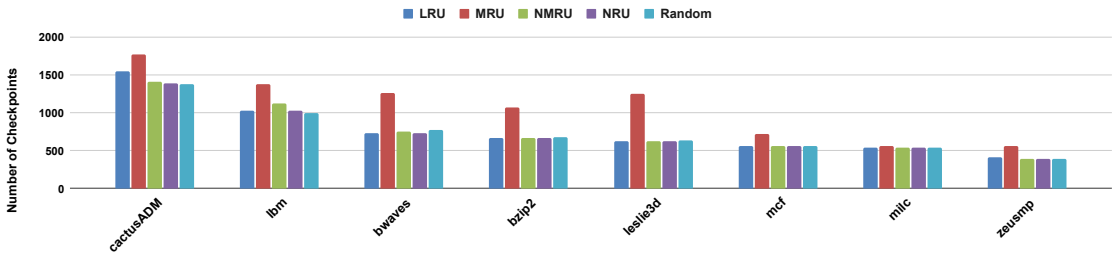


Figure 12. Checkpoint on single core applications.

Figure 12 shows the eight applications that created the most checkpoints. NRU produces the fewest checkpoints across all benchmarks shown, but the average difference between pairs for LRU, MRU, NMRU, and Random is only 55 checkpoints. This result is explained by their tendency not to choose recently written blocks in the cache: dirty lines remain among the most recently used, delaying set saturation and therefore the cache set threshold trigger. MRU generates significantly more checkpoints by immediately making recently written blocks candidates for eviction, which rapidly fills sets with dirty lines, thereby forcing frequent commits even when the working set would fit comfortably in the cache under a recency-aware policy. Some applications have created only 30 checkpoints across all tests because of another threshold: the instruction count. As a result, these applications only trigger checkpoints when this threshold is reached, leading to just 30 checkpoints during the tests.

5. Related Works

5.1. DONUTS Differentiation and Processing-in-Memory Integration

The DONUTS approach is based on integrating crash consistency directly with cache replacement policy events via processing-in-memory operations [Kruger et al. 2021]. Unlike prior systems that perform logging via a memory controller on the processor, DONUTS exploits DDR protocol extensions to implement logging within memory hardware. This architectural decision provides advantages like:

- Bandwidth efficiency: Logging operations execute in parallel within memory, consuming only internal memory bandwidth rather than saturating processor-memory interconnect.
- Natural epoch boundaries: Cache replacement events provide organic checkpointing opportunities rather than timer-imposed ones, resulting in epoch durations that adapt to application behavior.
- Asynchronous persistence with group-commit semantics: Following database system concepts, DONUTS batches checkpoint writes to amortize persistence overhead, reducing the average by block latency.
- Protocol compatibility: The standard DDR approach keep the compatibility with diverse NVM technologies (PCM, STT-RAM, ReRAM) without requiring specific technology modifications [Prasad et al. 2022].

5.2. Comparison with Existing Crash Consistency Mechanisms

In recent years, industry and academia have promoted extensive research projects about crash consistency mechanisms in NVM-based systems, with implementations going from purely software-based approaches to fully hardware-integrated solutions. A survey of contemporary mechanisms reveals distinct design approaches [Chen 2016].

Type	Project	Technique	Log type[1]	Epoch-based[2]	SW transp.[3]	PIM[4]
Software	Kindle	Process Persistence / Checkpoint	Redo	-	-	-
	LACT	Checkpointing	-	-	✓	-
	shadow-stack	Checkpointing	-	-	-	-
	EasyCrash	Selective Persistence	-	-	✓	-
	PMVerify	Verification	-	-	-	-
	Hybrid-PGAS	Transaction	Undo	-	-	-
Hardware	LightWSP	Logging	Redo	✓	✓	-
	JASS	Checkpointing	-	✓	✓	-
	iMIV	Integrity Verification	-	-	✓	✓
	Prosper	Checkpointing	-	✓	✓	-
	Milestone	Logging	Undo+redo	-	✓	-
	HM-ORAM	ORAM	-	-	✓	-
	DONUTS	Logging	Undo	✓	✓	✓
	Steins	Recovery	-	-	✓	-
	SpecPMT	Logging	Redo	-	-	-
	Silo	Logging	Undo+redo	-	✓	-
	Escort	Checkpointing	-	✓	✓	-

[1] Type of logging employed undo/redo/hybrid; ”-” if no traditional logging.
[2] Indicates if the project uses periodic/timer-driven checkpoints or epochs-based mechanism.
[3] Whether the mechanism is transparent to applications (no modifications needed).
[4] Usage of processing-in-memory.

Table 2. Summary of newer crash consistency mechanisms

Table 2 presents an overview of recent crash-consistency mechanisms. Some proposals adopt purely software-based or runtime/framework strategies that require no hardware modifications. For example, EasyCrash [Al Qayyam et al. 2025], which leverages analytical models with selective clwb/sfence, and Escort, which extends CpNvm through background execution. Other mechanisms incorporate micro-architectural enhancements, dedicated on-board logic, or hardware timers, such as, JASS, which integrates DRAM scrubbers and predictors, and Silo/SpecPMT [Zhang and Hua 2023, Ye et al. 2023], which introduces on-chip log buffers.

5.3. Ideas for Future Studies

Previous studies have shown that DONUTS can achieve significant gains, reducing log write volume by up to 42% when employing the LoW strategy in applications with moderate write density workloads. This reduction occurs because LoW avoids unnecessary log operations, which would otherwise degrade the NVM's internal bandwidth, generating performance overhead and higher energy consumption. On the other hand, in workloads with high write density, performing log operations only at the time of epoch persistence can create bottlenecks in the write queue, introducing delays that impair overall system performance. Thus, adopting a hybrid strategy that dynamically selects the log mode

based on an epoch predictor, alternating between LoR and LoW according to the behavior of each program region, becomes attractive. The central idea is to use the execution history to guide the choice of log mode: previously executed regions characterized by high write frequency should adopt LoR mode, while the others can operate under LoW mode. Determining the appropriate thresholds for this decision represents the main challenge to be addressed in future work.

Other idea is to explore adaptive approaches that dynamically adjust policies based on epoch characteristics, as well as the development of strategies explicitly designed for crash consistency rather than adapting policies conceived for cache performance. Validating these approaches on different PIM-based logging mechanisms, not investigated in this work, also shows promise.

6. Conclusion

This work extended DONUTS by incorporating support for additional traditional replacement policies (MRU, NMRU, NRU, and Random) beyond the original LRU-Read strategy. Furthermore, it conducted a systematic evaluation of the impact of these strategies across different workloads and single-core and multi-core benchmarks, providing empirical evidence to aid in selecting more appropriate policies for epoch-based NVM checkpointing mechanisms.

In summary, the obtained results contribute to a better understanding of the tradeoffs involved in crash consistency in persistent memory systems, offering valuable guidance for selecting appropriate consistency mechanisms for diverse application domains and workload profiles.

References

[Al Qayyam et al. 2025] Al Qayyam, B., Alshboul, M., Abdalfattah, N., and Jararweh, Y. (2025). Efficient crash-safe sorting for systems with non-volatile main memory. *Future Generation Computer Systems*, page 108229.

[Baldassin et al. 2020] Baldassin, A., Murari, R., Carvalho, J. a. P. L. d., Araujo, G., Castro, D., Barreto, J. a., and Romano, P. (2020). Nv-phtm: An efficient phase-based transactional system for non-volatile memory. *Euro-Par 2020: Parallel Processing*, pages 477–492.

[Carlson et al. 2011] Carlson, T. E., Heirman, W., and Eeckhout, L. (2011). Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, New York, NY, USA. Association for Computing Machinery.

[Carlson et al. 2014] Carlson, T. E., Heirman, W., Eyerman, S., Hur, I., and Eeckhout, L. (2014). An evaluation of high-level mechanistic core models. *ACM Trans. Archit. Code Optim.*, 11(3).

[Chakrabarti et al. 2014] Chakrabarti, D., Boehm, H.-J., and Bhandari, K. (2014). Atlas: Leveraging locks for non-volatile memory consistency. *ACM SIGPLAN Notices*, 49:433–452.

[Chen 2016] Chen, A. (2016). A review of emerging non-volatile memory (nvm) technologies and applications. *Solid-State Electronics*, 125:25–38.

[Jaleel et al. 2010] Jaleel, A., Theobald, K. B., Steely, S. C., and Emer, J. (2010). High performance cache replacement using re-reference interval prediction (rrip). *SIGARCH Comput. Archit. News*, 38(3):60–71.

[John L. Hennessy 2017] John L. Hennessy, David A. Patterson, C. K. (2017). *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 6th edition.

[Kruger et al. 2021] Kruger, K., Azevedo, R., and Pannain, R. (2021). Donuts: Um eficiente método de checkpointing em memórias não voláteis. *Proceedings of the Anais do XXII Simpósio em Sistemas Computacionais de Alto Desempenho*, pages 120–131.

[Kruger et al. 2023a] Kruger, K., Pannain, R., and Azevedo, R. (2023a). Donuts: An efficient method for checkpointing in non-volatile memories. *Concurrency and Computation: Practice and Experience*, 35:e7574.

[Kruger et al. 2023b] Kruger, K., Pannain, R., and Azevedo, R. (2023b). Using logging-on-write to improve non-volatile memory checkpoints via processing-in-memory. *2023 IEEE 35th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 68–77.

[Mishra et al. 2024] Mishra, S., Gohil, B. N., and Ray, S. (2024). A survey on persistent memory indexes: Recent advances, challenges and opportunities. *Journal of Systems Architecture*, 151:103140.

[Nguyen and Wentzlaff 2018] Nguyen, T. M. and Wentzlaff, D. (2018). Picl: A software-transparent, persistent cache log for nonvolatile main memory. *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 507–519.

[P and Mishra 2024] P, A. and Mishra, D. (2024). Kindle: A comprehensive framework for exploring os-architecture interplay in hybrid memory systems. In *2024 IEEE International Symposium on Workload Characterization (IISWC)*, pages 262–272.

[Prasad et al. 2022] Prasad, B., Parkin, S., Prodromakis, T., Eom, C.-B., Sort, J., and MacManus-Driscoll, J. L. (2022). Material challenges for nonvolatile memory. *APL Materials*, 10(9):090401.

[Ren et al. 2015] Ren, J., Zhao, J., Khan, S., Choi, J., Wu, Y., and Mutlu, O. (2015). Thynvm: enabling software-transparent crash consistency in persistent memory systems. *Proceedings of the 48th International Symposium on Microarchitecture*, pages 672–685.

[Wang et al. 2021] Wang, Z., Choo, C.-H., Kozuch, M. A., Mowry, T. C., Pekhimenko, G., Seshadri, V., and Skarlatos, D. (2021). Nvoverlay: Enabling efficient and scalable high-frequency snapshotting to nvm. *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 498–511.

[Ye et al. 2023] Ye, C., Xu, Y., Shen, X., Sha, Y., Liao, X., Jin, H., and Solihin, Y. (2023). Specpmt: Speculative logging for resolving crash consistency overhead of persistent memory. pages 762–777.

[Zhang and Hua 2023] Zhang, M. and Hua, Y. (2023). Silo: Speculative hardware logging for atomic durability in persistent memory. pages 651–663.