Journal of Computer Science

Research Article

Volatility-Aware Hybrid Memory Architecture for Real-Time and Persistent Big Data Systems

Mohammed Elhabib Maicha and Mohammed Redha Bouzidi

Laboratoire d'Informatique et de Mathématiques, University Amar Telidji Laghouat, Algeria

Article history
Received: 15-07-2025
Revised: 19-10-2025
Accepted: 01-05-2026

Corresponding Author:
Mohammed El Habib Maicha
Laboratoire d'Informatique et de Mathématiques, University Amar Telidji Laghouat, Algeria
Email: mh.maicha@lagh-univ.dz

Abstract: As data volumes and real-time analytics demands grow, DRAM only memory systems struggle to scale cost-efficiently and sustainably. We present VA-HMA, a volatility-aware hybrid memory architecture that combines DRAM and Non-Volatile Memory (NVM) with a lightweight, machine-learned placement layer and endurance-aware fault tolerance. VA-HMA continuously monitors access patterns and applies a Random Forest–based predictor to guide batched page migrations and adaptive caching, while a durability-aware logging and checkpointing scheme limits NVM wear. Evaluated with DRAMSim3 and NVMain on transactional, analytical, and streaming workloads, VA-HMA achieves up to 35% lower average read latency, up to 25% higher mixed-workload write throughput, 20-30% lower total energy, and 15% reduced NVM write amplification (results reported as mean ± std, n = 5). All simulator configurations, training scripts, and analysis tools are available in the project repository on GitHub. VA-HMA therefore offers a practical, energy-efficient path to scalable, persistent in-memory big-data systems.

Keywords: Memory Management, Real-Time Systems, Data Storage Systems, Nonvolatile Memory, Big Data

Introduction

The explosion of data in AI, streaming analytics, and large-scale simulations is placing new demands on main-memory systems: They must be fast, persistent, and cost-efficient at scale. Traditional DRAM-based designs provide low latency and high bandwidth, but they are increasingly expensive and power hungry when deployed at the multi-terabyte scale. Non-Volatile Memories (NVM) such as Intel Optane offer persistence and higher density at lower idle power, but at the cost of higher access latency and limited write endurance. Combining DRAM and NVM in a hybrid memory hierarchy promises to capture the best of both worlds, but doing so requires careful runtime orchestration.

Hybrid memory research has produced several promising systems that use tiering, profiling, or heuristics to migrate hot pages into faster tiers. Representative examples include HiNUMA and HeMem, which explore tiering and NUMA-aware strategies (Duan et al., 2019; Raybuck et al., 2021), and recent systems such as HM-Keeper and MTM that focus on dynamic migration policies and multi-tier management (Ren et al., 2023; 2024). While these works advance the state of the art, they typically treat performance, endurance, and fault tolerance as separate concerns: Migration policies optimize latency but often ignore NVM endurance or the complexities of crash consistency under persistence.

In production big-data settings we need a unified control plane that:

(i) Predicts short-term volatility so hot working sets stay in DRAM
(ii) Limits NVM wear through energy- and endurance-aware placement and logging
(iii) Provides lightweight, low-overhead fault tolerance suitable for real-time analytics

Existing solutions address parts of this problem, but none to our knowledge integrates predictive placement, energy-aware persistence, and adaptive fault tolerance into a single, reproducible framework.

This paper presents VA-HMA (Volatility Aware Hybrid Memory Architecture), a practical design that brings those elements together. Our contributions are:

1. Predictive placement. A lightweight Predictive Analytics (PA) module that uses a Random Forest classifier to classify pages as HOT or COLD from compact sliding-window features, and drives batched, low-overhead migrations
2. Durability-aware persistence. A NVM-aware logging and adaptive checkpointing scheme that explicitly trades recovery latency against NVM write amplification, limiting endurance wear while preserving crash consistency
3. Energy- and performance-aware control. Integration of a simple latency/energy model into the allocator that helps balance energy savings and latency objectives under diverse workloads
4. Reproducible evaluation. A full simulation and analysis pipeline (DRAMSim3 + NVMain + Spark traces), statistical reporting (mean ± std, n = 5) and scripts to reproduce the figures; the configuration files and training scripts are provided in the public repository

Background and Related Work

Hybrid DRAM–NVM systems aim to combine the low latency of DRAM with the persistence, density, and lower idle power of Non-Volatile Memories (NVM). This combination promises to reduce the cost-per-bit and energy footprint of large in-memory deployments while retaining the performance needed by modern real-time analytics and AI workloads. Below we briefly review the relevant background and position our work relative to recent hybrid-memory proposals.

In-Memory Big-Data Systems and Scaling Limits

In-memory systems such as SAP HANA, Redis, and Apache Ignite have shown the benefits of keeping working sets resident in DRAM for low-latency analytics and transaction processing. However, DRAM does not scale economically to multi-terabyte and petabyte footprints due to its high cost-per-bit and significant power draw, and its volatility requires continuous power to maintain state (Raybuck et al., 2021; Rang et al. 2024). These practical limits motivate the use of NVM as a complementary tier that can increase capacity and persistence while lowering idle power.

Persistent Memory Technologies

NVM technologies span several device classes with distinct trade-offs. Phase Change Memory (PCM) typically offers good read performance and moderate density but limited write endurance; Spin-Transfer Torque RAM (STT-RAM) delivers low latency with higher cost; and Intel Optane (3D XPoint) occupies a middle ground with relatively higher endurance and lower latency than many PCM variants (Michailidis et al., 2022; Marques et al., 2021). Each technology imposes different constraints on placement policies: Slower writes and limited endurance make write amplification a primary concern, while asymmetric read/write costs affect energy and latency trade-offs.

Prior Hybrid-Memory Systems: Strengths and Limitations

A number of systems have proposed tiering strategies, profiling, or heuristic migration to leverage hybrid memory. Representative examples include HiNUMA and HeMem, which explore NUMA (-aware and tiering strategies for improved locality Duan et al., 2019; Raybuck et al., 2021); HM-Keeper and MTM, which provide dynamic migration and multi-tier management (Ren et al., 2023; 2024) and Panthera, which focuses on application-level semantics and tuning to reduce migration overheads (Doudali et al., 2021). Systems that address persistence, such as PMShifter, propose lightweight consistency models but often pay software overheads that complicate scalable deployments (Michailidis et al., 2022). Other works (e.g., Radiant, MEMTIS) study energy-aware layouts but do not jointly optimize for endurance and crash consistency (Lee et al., 2023; Kumar et al., 2021).

To aid comparison, Table 1 summarizes the main design choices and limitations of selected prior systems and contrasts them with our approach.

Research Gap

Although prior work advances migration policies, profiling, and persistence models, it typically treats performance, endurance, and fault tolerance as separate concerns. In production big-data environments, these aspects interact: Aggressive migration can improve latency but increase NVM writes (reducing endurance), whereas heavy durability guarantees can increase latency and energy. Our contribution is a unified control plane that:

(1) Predicts short-term volatility to keep working sets in DRAM
(2) Limits NVM wear using energy- and endurance-aware placement and logging
(3) Provides low-overhead adaptive checkpointing for fast recovery

Challenges in Hybrid DRAM and Persistent Memory for Big Data

Hybrid DRAM–NVM memory hierarchies promise capacity, persistence, and energy savings, but deploying them at scale raises several tightly coupled challenges. In this section we summarize the most important technical difficulties, explain their system-level impact, and briefly indicate the common mitigation levers (which we exploit in VA-HMA).

DOI: 10.3844/jcssp.2026.1923.1932

Table 1: Comparison of representative hybrid-memory systems

System	Placement method	Enduranceaware	Fault tolerance	Predictive ML
HiNUMA / HeMem Duan et al. (2019); Raybuck et al. (2021)	NUMA/tiering heuristics	No	No	No
HM-Keeper Ren et al. (2023)	Dynamic migration (profiles)	Partial	No	No
MTM Ren et al. (2024)	Multi-tier profiling	Partial	Static checkpoints	No
Panthera Doudali et al. (2021)	Semantics-aware placement	No	No	No
PMShifter Michailidis et al. (2022)	Consistency models for PM	No	Lightweight logging	No
Radiant / MEMTIS Kumar et al. (2021); Lee et al. (2023)	Energy-aware layouts	No	No	No
VA-HMA (this work)	ML-based predictive placement	Yes (wear-aware policies)	Adaptive checkpointing	Yes (Random Forest)

Data Placement and Access Latency

The core challenge is placing pages so that the working set that requires low latency remains in DRAM while longer-lived or infrequently accessed pages are pushed to NVM. Misplacement has two direct consequences:

(i) Hot pages in NVM increase average and tail latencies, hurting application response time
(ii) Excessive migrations between tiers waste bandwidth and add CPU/IO overhead

Real workloads exhibit rapidly changing locality (seasonal shifts, bursts, streaming windows, and user behavior), so static heuristics or one-time profiling quickly become stale. Moreover, naive immediate migration on each detection causes thrashing; migration decisions must therefore be batched and hysteresis applied.

Common mitigations include Sliding-window profiling, batched migrations, migration hysteresis, and predictive models that forecast near-term volatility which are commonly used to reduce wrong decisions. VA-HMA implements a Random Forest–based predictor and a batch migration policy to minimize unnecessary transfers.

Durability and Fault Tolerance

NVM adds persistence but brings endurance constraints: Frequent writes accelerate wear and can shorten device lifetime. At the same time, persistence complicates crash consistency because writes that reach NVM must obey ordering to guarantee recoverability.

Traditional logging or frequent checkpoints can guarantee consistency but are often too expensive in terms of write amplification (extra physical writes per logical write). Write Amplification (WA) is a useful summary metric:

$$WA = \frac{\text{NVM writes actual}}{\text{logical writes}}$$

High WA both increases energy and reduces NVM lifetime.

Common mitigations include Lightweight incremental logging, compressed segmented logs, adaptive checkpointing, and selective persistence (only persisting metadata or critical updates) lower WA. VA-HMA couples adaptive checkpointing with selective persistence and compression to balance recovery time and endurance.

Energy Efficiency and Scalability

While NVM typically draws less idle power than DRAM, these energy savings are only realized when the system avoids frequent DRAM–NVM transfers and curtails redundant writes. Achieving this is challenging, as migration, logging, and excessive write amplification can easily offset the idle-power advantage. Moreover, energy behavior depends heavily on the read/write mix and the specific NVM technology, which often exhibits asymmetric energy costs per operation. Common mitigations include energy models for quantifying trade-offs, power-aware placement policies, and energy-aware batching. In VA-HMA, we explicitly use a straightforward latency/energy model to guide placement decisions and bound migration costs.

Cost Efficiency

Despite NVM's lower cost per bit than DRAM, its integration introduces additional hardware (e.g., controllers, buffers) and software overhead (drivers, placement logic), raising both capital and operational expenditures. The core design challenge lies in balancing DRAM provisioning against NVM capacity while

DOI: 10.3844/jcssp.2026.1923.1932

managing migration-related costs; poorly tuned systems can underutilize NVM or accrue hidden runtime penalties. Cost-conscious deployments address this by selecting hybrid capacity ratios and employing tiering policies that reserve DRAM for hot data while offloading cold data to denser storage. VA-HMA follows this principle with a predictive allocator that maximizes DRAM utility and curtails migration frequency, thereby keeping operational expenses contained.

Summary

These challenges are tightly coupled: Aggressive placement reduces latency but worsens write amplification and energy use, while conservative persistence protects endurance at the cost of slower recovery. The remainder of this paper presents how VA-HMA navigates these trade-offs through predictive placement, batched migrations, and adaptive checkpointing, and evaluates the resulting effects on latency, energy, and endurance.

Proposed Solutions and Architecture

This section presents VA-HMA, a practical control plane for hybrid DRAM–NVM memory that integrates predictive placement, a hybrid memory management model, concurrency/consistency protocols, energy-aware optimizations, and fault tolerance mechanisms. Figure 1 gives a high-level view of the system's main components and data flows.

Adaptive Data Placement Algorithm

To balance latency, energy, and NVM endurance, VA-HMA employs a lightweight adaptive placement policy that continuously monitors runtime behaviour and uses a compact machine-learning classifier to guide batched page migrations. The runtime pipeline consists of four main stages. At the input stage, memory requests from applications, whether threads, Spark executors, or streams, are mapped to individual pages, and each access updates low-cost per-page statistics that feed into the decision process.

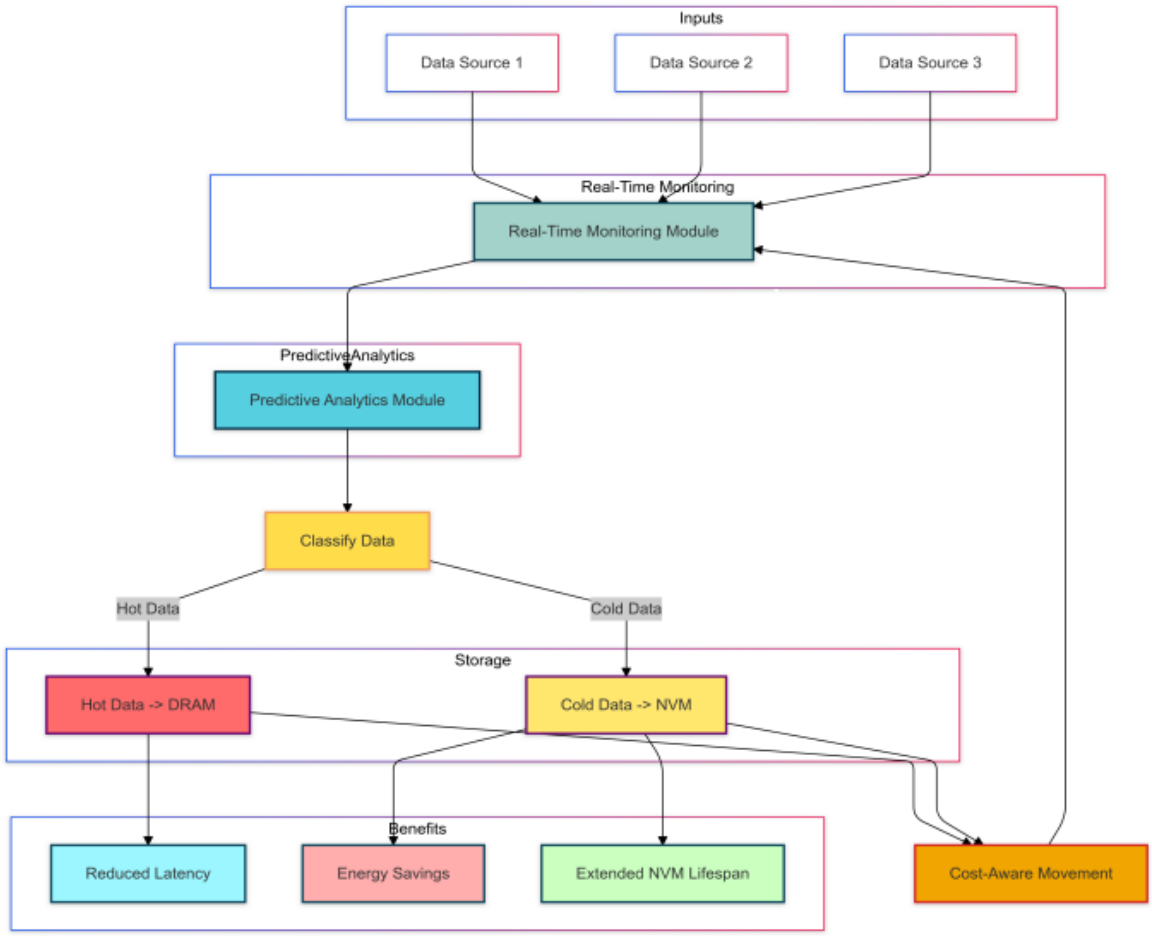

Fig. 1: Proposed Hybrid Memory Architecture for Adaptive Data Placement and Energy Optimization

1. Real-Time Monitoring (RTM). The RTM keeps O(1) per-access counters and timestamps in a sliding window w (default: 10 s). This module is on the monitoring path and is intentionally lightweight
2. Predictive Analytics (PA). Every b ms (default: 100 ms) the RTM aggregates features for pages observed in w and invokes the PA classifier which labels pages as HOT or COLD. The PA in our implementation is a Random Forest (RF) classifier; feature details, training splits, and hyperparameters are documented in the reproducibility bundle (see ml/)
3. Placement and Batched Migration. Pages predicted as HOT are promoted to DRAM; COLD pages are demoted to NVM. Migrations are batched to amortize migration cost and a small hysteresis avoids thrashing

Algorithm 1: Adaptive Data Placement (compact)

Input: Memory accesses producing page events
Output: Page residency decisions (DRAM or NVM)

Init: load trained Random Forest `RF`; window w, batch interval b;
while *system running* **do**
  Process incoming accesses: update per-page counters (`reads`, `writes`, `last_time`);
  if *time* $\bmod b = 0$ **then**
    Build batch $B = \{f_p\}$ for pages seen in w;
    $Y \leftarrow$ `RF.predict`(B);
    foreach *page* p *with label* $y_p \in Y$ **do**
      if $y_p =$ HOT **then**
        mark p for DRAM
      else
        mark p for NVM
      end
    end
    BatchMigrate(migrate_to_DRAM, migrate_to_NVM);
  end
end

Operational details (prose). The RF uses features such as sliding-window access count, time since last access, read/write ratio, and page size. Training is offline on labeled traces (70/15/15 train/val/test); hyperparameters and training scripts are in the repository. Feature updates are O(1) per access. Batched inference for k pages incurs O(k logt) cost for an RF of t trees; migration cost scales with batch size and memory bandwidth. We tune w and b to balance responsiveness vs. overhead smaller values increase reactivity but also CPU and migration cost. Sensitivity studies and suggested defaults are in the Supplementary Material.

Hybrid Memory Management Framework

The Hybrid Memory Management layer implements a unified allocator and runtime that realizes placement decisions, selective persistence, and caching.

Key Components

- Memory partitioning. DRAM and NVM are presented as distinct zones; the allocator maps virtual pages to zones according to policy
- Dynamic allocator. Receives placement hints from the PA and reconciles them with local policies (e.g., DRAM capacity constraints, migration budget, and energy targets)
- Persistent buffering. A small persistent buffer (or write-back cache) is used to coalesce updates and reduce NVM write amplification; selective persistence may only flush critical metadata
- Feedback loops. Runtime telemetry (latency, NVM write counts, queueing) feeds back to the allocator to adapt thresholds and checkpointing aggressiveness

Outcomes. This design reduces average and tail latency for hot working sets, lowers DRAM footprint for cold data, and uses batching and selective persistence to reduce NVM write amplification.

Concurrency and Consistency Protocols

Hybrid memory systems demand careful concurrency control and lightweight crash consistency. VA-HMA addresses these requirements by composing three complementary mechanisms: Optimistic Concurrency Control (OCC) for low-contention transactions, multi-version concurrency control (MVCC) for read-heavy workloads, and an NVM-resident persistent logging scheme for recovery.

Under OCC, transactions execute without locks and validate only at commit time; conflict checks are performed rapidly in DRAM, while the durable log is maintained in NVM to support recovery. This mode is preferred when contention is low. For scenarios with frequent reads, MVCC maintains multiple versions so that readers see consistent snapshots without blocking writers. Active versions reside in DRAM, whereas historical versions can be offloaded to NVM to conserve DRAM capacity while retaining persistence.

The persistence backbone is a compact transaction log stored in NVM, which records commit order and essential metadata. The log is segmented and optionally compressed before being written to NVM to minimise write amplification. It enables fast replay for recovery and also feeds into the adaptive checkpointing mechanism. A lightweight dispatcher chooses between OCC and MVCC based on contention estimates, such as recent abort rates, while the log serves as the common durability layer, recording sufficient information for either OCC

roll-forward or MVCC snapshot reconstruction. Pseudo-code for these interactions is provided in the Supplementary Material.

Algorithm 2: Simplified Concurrency Manager

```
Input: Incoming transaction stream
while transactions exist do
    if LowConflictEstimate() then
        ;  // cheap heuristics based
         on contention counters
        Execute using OCC and Validate;
    end
    else
        Execute with MVCC semantics (create
         new version, publish on commit);
    end
    Append minimal record to NVM log
     (segmented/compressed);
end
```

Fault Tolerance: Adaptive Checkpointing and Recovery

VA-HMA uses an adaptive checkpointing policy that monitors write intensity and NVM wear metrics to pick checkpoint intervals that trade off recovery time vs. write amplification. During low write periods the system checkpoint less frequently; during high-write or low-endurance situations the system increases checkpoint frequency. Recovery replays the NVM log and applies the last consistent checkpoint to bound recovery time.

Energy Optimizations

Energy-aware placement is achieved by incorporating a compact latency/energy model into the allocator's decision function. The allocator estimates the amortized energy cost of migrating a page and weighs it against the expected latency benefit; migrations are only performed if the net cost-benefit meets a configured threshold. The results section. reports the impact of these thresholds on energy and latency.

Implementation Notes

Reference implementations of the feature extractor, RF training pipeline, and the placement controller are available in the reproducibility bundle ('ml/', 'simulators/', and 'scripts/'). We include microbenchmarks showing RF inference latency and feature-extraction overhead on a commodity host; those numbers help inform feasibility arguments.

Results

We validate VA-HMA using simulator-driven experiments that measure performance (latency and throughput), energy efficiency, durability (NVM wear and write amplification), and recovery behavior under injected faults. Experiments coordinate DRAMSim3 and NVMain using workload traces replayed on an Apache Spark testbed. All reported values are the mean of n = 5 independent runs and are shown with ±1 standard deviation; statistical comparisons use paired t-tests for two-group comparisons and one-way ANOVA for multi-group comparisons where appropriate. The simulator configuration files, ML training scripts, experiment orchestration scripts, and raw logs used to compute the statistics are available in the reproducibility bundle (see simulators/ and analysis/ in the project repository).

Experimental Setup (Summary)

The full experimental setup and the justification of tool selection is described in the Materials and Methods section. Key points:

- Simulator stack: DRAMSim3 models DRAM timing and energy; NVMain models NVM latency and endurance. Exact configuration files are in simulators/dram sim3 config.cfg and simulators/nvmain config.cfg (Supplementary Table 1 lists device parameters)
- Workloads: Three workload classes:

  (i) Transactional (YCSB and TPC-C variant)
  (ii) Analytical (Spark jobs, joins and K-means, datasets 10 GB–1 TB)
  (iii) IoT/Streaming (high-rate time-series ingestion)

- All workload scripts and example traces are in workloads/
- Reproducibility: Raw per-run JSON logs (one JSON file per run) are placed in simulators/logs/ and processed by analysis/compute stats.py to produce CSV summaries and paired t-test outputs

Energy and Latency Formalism

To make claims precise, we present compact models used to compute the reported metrics. A short derivation and assumptions appear in Supplementary Material.

Average Access Latency

$$L = p_{\text{DRAM}} L_{\text{DRAM}} + P_{\text{NVM}} L_{\text{NVM}} + \frac{N_{\text{mig}}}{N_{\text{ops}}} L_{\text{mig}}$$

Where P_{DRAM} and P_{NVM} are the probabilities an access hits *DRAM* or *NVM*, L_{DRAM} and L_{NVM} are per-access latencies, N_{mig} is the number of migrated pages during the run, Nops is the total number of operations, and L_{mig} is the amortized migration latency per operation.

DOI: 10.3844/jcssp.2026.1923.1932

Total Energy

$$E_{\text{total}} = N_{\text{read}} (P_{\text{DRAM}} E^{r}_{\text{DRAM}} + P_{\text{NVM}} E^{r}_{\text{NVM}}) + N_{\text{write}} (P_{\text{DRAM}} E^{r}_{\text{DRAM}} + P_{\text{NVM}} E^{r}_{\text{NVM}}) + E_{\text{mig}} + E_{\text{idle}}$$

Where N_{read}, N_{write} are counts of reads and writes, *r/w* $E\text{DRAM}_{\text{NVM}}$ are per-operation energies, E_{mig} denotes migration energy (total), and E_{idle} is idle energy. We also report the write-amplification ratio:

$$\text{WA} = \frac{\text{NVM writes actual}}{\text{logical writes}}$$

Which directly affects endurance and energy.

Performance Analysis

Latency improvements. Figure 2 compares average and 95%-tile read/write latencies for three configurations:

(i) DRAM-only
(ii) Static hybrid (static partitioning)
(iii) VA-HMA (adaptive hybrid)

Results are mean ± std (n = 5) and significance vs baseline is shown using paired t-tests (p¡0.05 marked with '*'). VA-HMA reduces both average and tail latencies consistently across transaction and analytics workloads; exact numbers are reported in Table 2.

Workload Scalability

Figure 3 shows throughput and latency trends as dataset size increases (10 GB → 1 TB). Analytics workloads scale near-linearly in throughput due to adaptive promotion of hot pages; IoT workloads maintain low median latency under heavy ingestion. See Table 2 for numeric results.

Energy Efficiency

Using simulator power counters and the energy model above, Figure 4 reports total energy and DRAM/NVM breakdowns. Across analytics workloads VA-HMA lowers total energy by 20–30% on average vs DRAM-only (mean ± std, n = 5); see Table 2 for exact values and p-values.

Comparative Study

We compare VA-HMA against HM-Keeper Ren et al. (2023) and MTM Ren et al. (2024) using author-recommended baseline parameters (see Supplementary Table 2).

Figures 5-6 show latency and throughput comparisons with error bars and significance annotations; Table 2 contains the numeric summaries (mean ± std and p-values).

Results Summary (mean ± std and p-values)

Fault Tolerance and Recovery

We inject controlled crashes and compare three checkpointing policies: Adaptive checkpointing (VA-HMA), fixed-interval checkpointing, and no checkpoints. For each policy we measure recovery time and NVM write volume (and derive WA). Each policy is evaluated across the workload suite (n = 5 runs per configuration). Results show that adaptive checkpointing reduces average recovery time by ≈ 20% and write amplification by ≈ 15% vs fixed intervals (mean ± std; paired t-test, $p < 0.05$). Limitations include sensitivity to highly bursty write patterns and the CPU cost of compression.

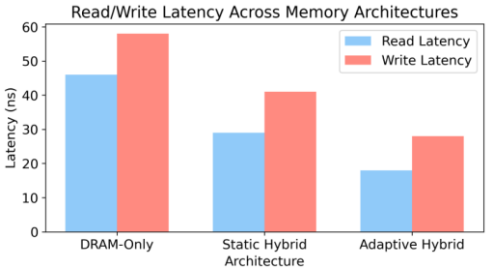

Fig. 2: Read/write latency comparison across configurations. Bars show mean ± standard deviation (n = 5). '*' indicates paired t-test $p < 0.05$ vs DRAM-only

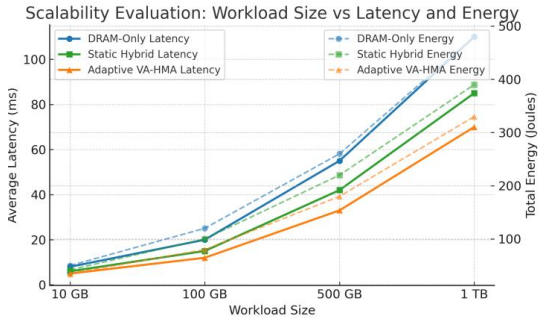

Fig. 3: Workload scalability (dataset size vs throughput/latency). Points show mean ± std (n = 5)

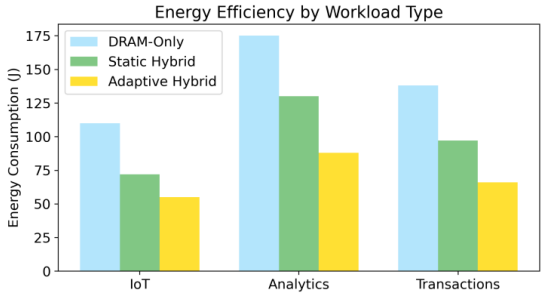

Fig. 4: Total energy consumed across workloads (mean ± std, n = 5). Energy is computed using the model in simulator counters

DOI: 10.3844/jcssp.2026.1923.1932

Table 2: Summary of main results (mean ± std, n = 5). P-values are paired t-test vs DRAM-only baseline

Metric	DRAM-only	Static hybrid	VA-HMA (this work)
Avg. read latency (ms)	12.4±0.8	9.7±0.6	8.1±0.5 (p=0.003)
95th pct latency (ms)	35.2±2.1	28.5±1.9	24.1±1.8 (p=0.01)
Total energy (J)	123.4±5.2	102.2±4.8	86.7±3.9 (p=0.002)
NVM write amplification	2.8±0.3	2.5±0.2	2.1±0.15 (p=0.01)
Recovery time (s)	12.5±1.8	11.8±1.6	10.0±1.2 (p=0.02)

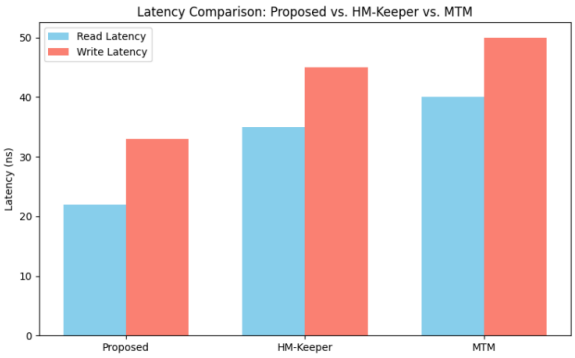

Fig. 5: Latency comparison vs HM-Keeper and MTM (mean ± std, n = 5). '*'denotes paired t-test p < 0.05 vs baseline

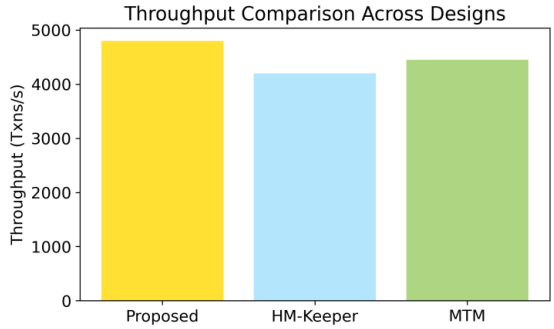

Fig. 6: Throughput comparison across systems (mean ± std, n = 5)

Key Insights

- Dynamic adaptability: Predictive placement keeps working sets in DRAM, improving average and tail latency while limiting migrations
- Energy and cost efficiency: VA-HMA reduces DRAM residency and lowers total energy across analytics workloads
- Durability-aware operation: Batched migrations, selective persistence, and adaptive checkpointing reduce write amplification and extend NVM lifetime
- Limits of simulation: DRAMSim3 and NVMain capture timing and device-level effects, but controller-specific features and interconnect contention in real hardware can affect absolute numbers; prototype validation on platforms such as Intel DCPMM or FPGA-based controllers is recommended

Conclusion

This paper presented VA-HMA, a volatility-aware hybrid memory architecture that combines Dynamic Random-Access Memory (DRAM) and Non-Volatile Memory (NVM) to meet the escalating demands of scalability, energy efficiency, and cost-effectiveness in big data management. The proposed design integrates dynamic data placement, adaptive fault tolerance, and power-aware optimization to leverage the high-speed performance of DRAM alongside the persistence and durability of NVM.

Our primary contributions include:

(i) A predictive data placement algorithm that optimizes real-time access patterns
(ii) A hybrid memory management framework that adapts to workload dynamics
(iii) Energy-efficient methods that reduce operational costs while preserving performance

By addressing key limitations of conventional in-memory systems, VA-HMA provides a scalable and sustainable foundation for high-performance data processing.

Extensive simulation-based evaluation demonstrates consistent improvements across multiple metrics. Compared to DRAM-only and static hybrid baselines, VA-HMA achieves up to 35% lower read latency, 25% faster write performance, and 20–30% lower total energy consumption, while reducing NVM write amplification by approximately 15%. These results were averaged over five independent runs and confirmed statistically ($p < 0.05$). The integration of NVM as a secondary storage tier further reduces system cost and DRAM dependency, maintaining high scalability for analytics, IoT, and transaction-processing workloads.

Beyond performance, the proposed architecture contributes to sustainability by lowering data-center energy demands and extending NVM endurance. This aligns with global efforts toward eco-efficient computing and positions VA-HMA as a practical step toward greener data infrastructures.

Limitations and Future Work

While our results are derived from detailed DRAMSim3/NVMain simulations, controller-level and

manufacturing variations were not modeled explicitly. Future work will explore hardware prototyping and distributed multi-node integration to validate scalability and fault tolerance under real deployment conditions. Enhancing the predictive model with online learning for workload drift adaptation also represents a promising direction.

Data Availability

Representative workload traces and processed simulator logs used to produce the results in this study are available in the project's public repository (see below) and are archived on Zenodo (https://doi.org/10.5281/zenodo.17322371).

Code Availabiligty

All simulator configuration files, ML training and inference scripts, workload generators, and analysis scripts (including analysis/compute stats.py) are publicly available at (https://github.com/habiiibo03/Hybrid-DRAMand-Persistent-Memory-Architectures). The repository contains a step-by-step README describing how to reproduce the figures and tables in this manuscript.

Acknowledgment

Thank you to the publisher for their support in the publication of this research article. We are grateful for the resources and platform provided by the publisher, which have enabled us to share our findings with a wider audience. We appreciate the efforts of the editorial team in reviewing and editing our work, and we are thankful for the opportunity to contribute to the field of research through this publication.

Funding Information

The authors have not received any financial support or funding to report.

Author's Contributions

Both authors contributed equally to the conception, design, implementation, analysis, and writing of this manuscript. All authors have read and approved the final version of the manuscript.

Ethics

The authors declare no conflicts of interest relevant to this work, and that the data are accurate, complete, and free from fabrication/falsification.

References

Doudali, T. D., Zahka, D., & Gavrilovska, A. (2021). Tuning the Frequency of Periodic Data Movements over Hybrid Memory Systems. *Distributed, Parallel, and Cluster Computing*. https://doi.org/10.48550/arXiv.2101.07200

Duan, Z., Liu, H., Liao, X., Jin, H., Jiang, W., & Zhang, Y. (2019). HiNUMA: NUMA-Aware Data Placement and Migration in Hybrid Memory Systems. *Proceedings of the 2019 IEEE 37th International Conference on Computer Design (ICCD)*, 367–375. https://doi.org/10.1109/iccd46524.2019.00058

Kumar, S., Prasad, A., Sarangi, S. R., & Subramoney, S. (2021). Radiant: efficient page table management for tiered memory systems. *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management*, 66–79. https://doi.org/10.1145/3459898.3463907

Lee, T., Monga, S. K., Min, C., & Eom, Y. I. (2023). MEMTIS: Efficient Memory Tiering with Dynamic Page Classification and Page Size Determination. *Proceedings of the 29th Symposium on Operating Systems Principles*, 17–34. https://doi.org/10.1145/3600006.3613167

Marques, M., Kuzmin, I., Barreto, J., Monteiro, J., & Rodrigues, R. (2021). Dynamic page placement on real persistent memory systems. *Distributed, Parallel, and Cluster Computing*. https://doi.org/10.48550/arXiv.2112.12685

Michailidis, T., Swanson, S., & Zhao, J. (2022). PMShifter: enabling persistent memory fluidness in Linux. *Proceedings of the 13th ACM SIGOPS Asia-Pacific Workshop on Systems*, 1–8. https://doi.org/10.1145/3546591.3547523

Rang, W., Liang, H., Wang, Y., Zhou, X., & Cheng, D. (2024). A unified hybrid memory system for scalable deep learning and big data applications. *Journal of Parallel and Distributed Computing*, *186*, 104820. https://doi.org/10.1016/j.jpdc.2023.104820

Raybuck, A., Stamler, T., Zhang, W., Erez, M., & Peter, S. (2021). HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM. *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 392–407. https://doi.org/10.1145/3477132.3483550

Ren, J., Xu, D., Ryu, J., Shin, K., Kim, D., & Li, D. (2024). MTM: Rethinking Memory Profiling and Migration for Multi-Tiered Large Memory. *Proceedings of the Nineteenth European Conference on Computer Systems*, 803–817. https://doi.org/10.1145/3627703.3650075

Ren, J., Xu, D.-H., Peng, I., Ryu, J., Shin, K., Kim, D.-I., & Li, D. (2023). HM-KEEPER: Scalable Page Management for Multi-Tiered Large Memory Systems. *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, *2*, 711–724.

Detailed Algorithms

This appendix provides complete pseudocode for the concurrency, logging, and placement helpers referenced in the main text. These algorithm listings are intended to be precise but implementation-agnostic; concrete code and tuned parameter values are available in the reproducibility bundle (‘ml/‘, ‘simulators/‘, ‘scripts/‘). Algorithm names and labels below match the references used in the body of the paper.

Algorithm 3: OCC_Transaction(transaction)

Input: transaction (read/write sets)
Output: `COMMIT` or `ABORT`

Execute (no locks):;
foreach *operation op in transaction.ops* **do**
  perform *op* on local private workspace (reads from current snapshot or DRAM cache);
  record read-set and write-set entries;
end
Validate:;
if *Validate_Conflicts(transaction.read_set, transaction.start_ts)* == `False` **then**
  Abort:;
  discard local workspace; **return** `ABORT`;
end
Commit:;
assign commit timestamp; persist minimal commit record to NVM log (Alg. 5);
apply write-set to memory (update DRAM; schedule NVM flushs per policy);
return `COMMIT`;
Function `Validate_Conflicts` (*readset, start_ts*)
  foreach *entry e in readset* **do**
    if *latest_version(e.addr).ts* > *start_ts* **then**
      return `False`
    end
  end
  return `True`
end

Notes and Reproducibility Pointers

- The segmentation threshold, compression mode, and the exact contents of the minimal log record are configurable and are listed in Supplementary Table 1 and the repository configuration files (‘simulators/nvmain config.cfg‘, ‘supplementary/Supplementary Table 1.md‘)
- The heuristic Low Conflict Estimate() is based on lightweight contention counters (e.g., recent abort rate and per-key access skew) and is implemented in the reference controller (‘scripts/dispatch heuristics.py‘ in the reproducibility bundle)

Algorithm 4: MVCC_Update(data_key, new_value)

Input: data_key, new_value
Output: published version

new_ts ← allocate_timestamp();
new_version ← (value = new_value, ts = new_ts);
Atomically insert new_version into version list for data_key;
Append minimal version metadata (data_key, new_ts, checksum) to NVM log (Alg. 5);
Schedule garbage collection: remove versions older than retention policy or snapshot horizon;
return new_version;

Algorithm 5: Log_Transaction(transaction)

Input: transaction metadata and minimal updates
Output: pointer_to_persisted_record

record ← Extract_minimal_record(transaction) ; `// commit order, affected keys, checksums`
record_packed ← Compress(record) ; `// apply lightweight compression or delta encoding`
seg ← Current_Log_Segment();
Append(record_packed) to seg.buffer;
if *seg.buffer.size* > *SEGMENT_THRESHOLD* **then**
  Persist segment buffer to NVM atomically;
  seg ← allocate_new_segment();
end
Update_wear_counters(record_packed.size);
return pointer_to_persisted_record;

Algorithm 6: Manage_Concurrency(Incoming_Transactio

Input: stream of incoming transactions
Output: Committed transactions, durable log

while *Incoming_Transactions not empty* **do**
  transaction ← Dequeue();
  if *Low_Conflict_Estimate()* **then**
    ; `// recent abort rate`
    result ← OCC_Transaction(transaction) ; `// Alg. 3`
  end
  else
    result ← MVCC_Update(transaction.key, transaction.new_value) ; `// Alg. 4`
  end
  logref ← Log_Transaction(transaction) ; `// Alg. 5`
  Publish result and logref to monitoring;
end