EP-ORAM: Efficient NVM-Friendly Path Eviction for Ring ORAM in Hybrid Memory

Mehrnoosh Raoufi
University of Pittsburgh
mraoufi@cs.pitt.edu

Jun Yang
University of Pittsburgh
juy9@pitt.edu

Xulong Tang
University of Pittsburgh
xulongtang@pitt.edu

Youtao Zhang
University of Pittsburgh
zhangyt@cs.pitt.edu

***Abstract*—Recent studies showed that only ORAM (oblivious RAM) can securely protect memory access patterns (i.e., data privacy) on modern computer systems. Ring ORAM is a promising ORAM protocol as it demands O(1) memory accesses for servicing each user memory request. However, Ring ORAM exhibits low memory utilization, i.e., its memory requirement is 4.8× of the protected user space. While adopting NVM (non-volatile memory) can alleviate the memory requirement, a simple implementation tends to introduce large performance degradation, preventing its adoption in practice.**

In this paper, we propose EP-ORAM, an NVM-friendly Ring ORAM implementation on DRAM/NVM hybrid memory. EP-ORAM is developed based on two key observations: (1) for tree-based Ring ORAM memory organization, saving bottom levels in NVM can dramatically reduce the DRAM memory requirement; (2) the tradeoffs among Ring ORAM operations expose design opportunities without security compromise. We, therefore, propose to save the bottom levels of the ORAM tree in NVM and shorten the path of EvictPath operation, which not only mitigates the number of NVM writes but also speeds up the execution. Our experimental results show that, under the design constraints of similar performance as the baseline that saves two bottom levels in NVM, EP-ORAM helps to save three levels in NVM, achieving 50% DRAM space reduction. In addition, EP-ORAM reduces the NVM writes by 15%.

***Index Terms*—ORAM, security, access pattern, hybrid memory**

I. Introduction

For modern computer systems, it is challenging to protect the memory access patterns between CPU and untrustworthy main memory. While CPU can be secured with low overhead [1], according to JEDEC memory standard, the memory device commands and addresses are communicated in plaintext between the on-chip memory controller and memory chips. Studies have shown that it demands ORAM (oblivious RAM) [2] to prevent information leakage from memory access patterns.

Among recent ORAM schemes [3], [4] that successfully reduced the protocol overhead to O(logN), where N is the number of protected data blocks, Ring ORAM [4] is a promising design. While its overall overhead remains O(logN), Ring ORAM achieves O(1) overhead for online accesses, i.e., the overhead to service user memory requests. Thus, Ring ORAM services user requests faster and improves the program performance, e.g., 2.7× improvement over Path ORAM [3]. The protocol maintenance operations are expensive but they are not always on the critical path.

However, Ring ORAM faces one major limitation, i.e., its low memory utilization. For a typical setting, the required DRAM space is 4.8× of the protected data [4]. Ring ORAM organizes the data blocks in a tree structure: each tree node, referred to as a bucket, consists of Z slots each of which can save one data block or dummy block. The low memory utilization comes from saving two types of dummy blocks. (1) When servicing a user memory request, Ring ORAM identifies the target tree path and fetches one block from each bucket along the path. Ring ORAM reserves S slots per bucket holding dummy data so that a bucket can service up to S accesses without reshuffling. (2) A data block, after its access, is randomly mapped to a different tree path. To prevent mapping to a tree path that has no empty slot, Ring ORAM doubles the number of memory slots that can hold user data. Therefore, on average, a tree bucket can hold (Z-S)/2 user data blocks. For a typical setting Z=12, S=7, the DRAM memory requirement of Ring ORAM is 4.8× of the protected user data.

To alleviate the low memory utilization in Ring ORAM, Cao *et al.* [5] proposed to shrink the bucket size such that the S dummy slots overlap with the slots that can save user data, which reduces the DRAM space by 34%. Raoufi *et al.* [6] proposed to exploit the dead blocks in the ORAM tree, which reduces the DRAM space by 22%. Unfortunately, even with these designs, the DRAM space requirement remains high.

With the fast advances of NVM (non-volatile memory) technologies, e.g., ReRAM (Resistive Memory) [7] and STT-RAM (Spin-Transfer Torque Memory) [8], an alternative strategy to address the high DRAM space demand is to use NVM. In particular, for a DRAM/NVM hybrid memory system, if allocating the last two levels of the ORAM tree in NVM and the rest of the levels in DRAM, the required DRAM space can be reduced by 75%. However, NVM often suffers from large memory access latency, allocating too many levels, e.g., six levels, in NVM [9] can lead to 70% performance degradation. Given the performance of Ring ORAM is already 2.2× slower than the no-ORAM implementation, the large performance degradation makes an aggressive hybrid design less appealing.

In this paper, we study the tradeoffs in DRAM/NVM hybrid design as well as among the Ring ORAM operations. By exploiting the design space exposed by these tradeoffs, we propose EP-ORAM, an efficient and NVM-friendly path eviction scheme to mitigate the high DRAM demand in Ring ORAM. In the following, we summarize our contributions.

- We study the interactions among different Ring ORAM operations. In particular, EvictPath (the operation to reshuffle tree paths) determines the stash size, the frequency of bucket level reshuffles, and the number of memory writes. We further identify the under-utilized middle levels of Ring ORAM. It exposes the opportunity to reduce the overhead of maintenance operations in Ring ORAM, in particular, the stash size.
- We propose EP-ORAM to exploit the design space exposed by our studies. EP-ORAM carefully partitions the ORAM tree between DRAM and NVM and shortens the path length during EvictPath operation, which not only reduces the number of NVM writes but also speeds up the operation.
- We evaluate the proposed design. Our experimental results show that, under the design constraints of no security compromise and similar performance as the baseline that saves two bottom levels in NVM, EP-ORAM helps to save three levels in NVM, achieving 50% DRAM space reduction. In addition, EP-ORAM reduces the NVM writes by 15%.

II. Background

A. Attack Model

In this paper, the trusted computing base only includes the processor on-chip, and everything off-chip is untrusted including memory bus and memory modules, similar to those in the literature [3], [4], [10]. The processor encrypts any data before writing to off-chip memory and decrypts the data after reading from the off-chip device. Ring ORAM is adopted to protect the memory access pattern of the user program running in the trusted computing base.

B. Ring ORAM Basics

Ring ORAM is a tree-based ORAM protocol that organizes data blocks in memory in a full binary tree. Each tree node is a Z-slot bucket. Each slot can hold either a real or a dummy block. In Ring ORAM, each bucket dedicates S slots for reserved dummy blocks, and the remaining $Z' = Z - S$ can hold real or dummy blocks. Similar to that in Path ORAM, half of the entire Z' entries in the tree can hold real blocks so that there are enough empty slots for random remapping. If the ORAM tree has L levels, it has $(2^L - 1) \times Z$ slots. There are 2^{L-1} paths from the root to the leaves. Each bucket also keeps a metadata block indicating the location of real blocks in the bucket and a *valid* flag for each block. It also maintains a *counter* to indicate how many times the bucket has been accessed. Fig. 1 depicts an example of a Ring ORAM tree with three levels.

A user data block is randomly mapped to a path in the tree. The mapping, referred to as *Position Map*, is in the protected space while the subset of frequently used entries is saved in a position map cache in the trusted base. When the user program requests a data block A, ORAM controller looks up *Position Map* to identify the path on which block A resides, accesses path l, retrieves block A, and saves block A in a small buffer called *Stash* within the trusted base. Ring ORAM keeps block A in the *Stash* until it is written back to the ORAM tree by one of the maintenance operations. There are mainly four operations; *ReadPath*, *EarlyReshuffle*, *EvictPath*, and *BackgroundEvict*. *ReadPath* is considered the online access that services the user program request. Whereas, the other three are maintenance operations to free the stash and refresh the buckets. We elaborate on these Ring ORAM operations as follows.

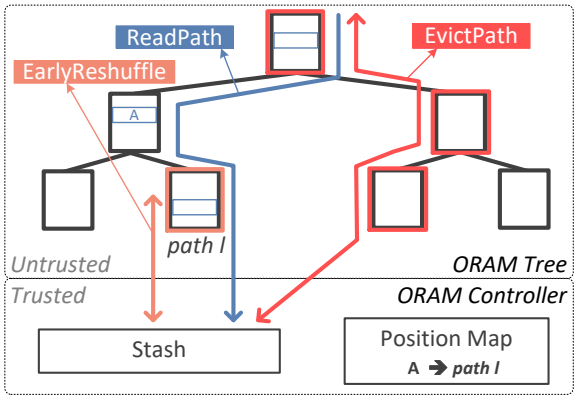

Fig. 1. Ring ORAM tree example with $L = 3$.

ReadPath. It is the path access that is to service the user memory request. The ORAM controller, after identifying the mapping from block A to path l, first accesses the metadata blocks of all buckets along path l to determine the exact location of block A. It then reads one valid block per bucket on path l — one bucket provides block A while each of the other buckets provides a valid reserved dummy block. Ring ORAM then updates the metadata of all buckets and marks all read blocks as invalid. Ring ORAM randomly maps block A to a new path, and updates *Position Map* and *Stash* accordingly. The user program can resume the execution after *ReadPath*.

EarlyReshuffle. It reshuffles a bucket on-demand. The bucket counters along path l are incremented during *ReadPath*. If any counter reaches S, Ring ORAM needs to reshuffle the bucket, assuming it runs out of valid reserved dummy blocks. To reshuffle a bucket, Ring ORAM reads the entire bucket, reshuffles/encrypts the data blocks, and writes them back to the tree. The blocks accumulated in the stash may be written back during the *EarlyReshuffle* operation.

EvictPath. It reshuffles all buckets along one path. It reads all the buckets along a path, reshuffles/encrypts them, and writes them back to the tree. The blocks accumulated in the stash may be written back during the *EvictPath* operation. Ring ORAM schedules one *EvictPath* after five online accesses in a typical setting [4]. Note that paths are chosen based on a reverse-lexicographic order so the path to be reshuffled is public information.

Since each bucket contains at most Z' real data blocks, *EarlyReshuffle* and *EvictPath* read Z' blocks but write Z blocks. Both operations update the metadata block accordingly.

BackgroundEvict. If the stash is full, Ring ORAM invokes *BackgroundEvict* to prevent the protocol from failing. This is achieved by issuing dummy *ReadPath* that reads a valid

dummy block per bucket from a randomly selected path. *BackgroundEvict* ensures the stash occupancy does not increase until the next *EvictPath* arrives. *BackgroundEvict* keeps issuing dummy *ReadPath* operations until sufficient *EvictPath* operations are executed to reduce the stash occupancy below the threshold. *BackgroundEvict* operation converts protocol correctness problem to a performance problem [3], [5].

III. EP-ORAM

In this section, we first discuss our key observations and the design space challenges. We then elaborate on the EP-ORAM scheme to address them.

A. Tradeoffs of Adopting ORAM in DRAM/NVM Memory

As explained in Section II, the ORAM tree is a full binary tree so its capacity grows exponentially. In this tree, the capacity of the last level is equal to the capacity of all the prior levels. In other words, if we save the last two levels of the ORAM tree in NVM, we can save 75% of DRAM space. NVM often suffers from large memory access latency, allocating too many levels, e.g., six levels, in NVM [9] can lead to 70% performance degradation. Given the performance of Ring ORAM is already 2.2× slower than the no-ORAM implementation, the large performance degradation makes an aggressive hybrid design less appealing.

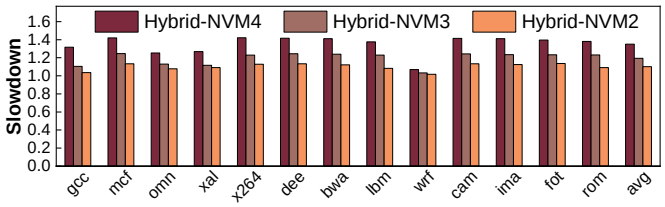

Fig. 2. Slowdown of Ring ORAM in hybrid memory compared to DRAM (Hybrid-NVMx indicates saving x bottom levels in NVM).

Assume upgrading a 4-channel DRAM to a DRAM/NVM hybrid system, we reserve one memory channel for NVM traffic. Given NVM accesses are slower than DRAM ones, we expect to experience a modest performance slowdown in such system. Fig. 2 shows the slowdown of ORAM on different hybrid configurations over the DRAM-only baseline. On average, Hybrid-NVM4 incurs 1.35× slowdown. Whereas, Hybrid-NVM3 and Hybrid-NVM2, on average, incur 1.19× and 1.10× slowdown respectively. If we bound the tolerable performance degradation to 10%, Hybrid-NVM2 is the only configuration that has an acceptable performance.

Another obstacle to adopting ORAM in a hybrid setting is the NVM writes imposed by the ORAM protocol. In a non-secure baseline each user memory request incurs at most one NVM write access. However, with ORAM, many more NVM writes are incurred per user program memory request. We studied the average number of NVM writes per user memory request for ORAM on different hybrid settings. Our experimetal results showed that Hybrid-NVM4, Hybrid-NVM3, and Hybrid-NVM2, on average, incur 17.7×, 13.3×, and 8.8× NVM writes per user request, respectively. While NVM write endurance has improved significantly in recent years, for example, ReRAM has 10^9 write endurance, i.e., 10× better than that of PCM [11]. However, more than one order of magnitude NVM write increase could still be a big concern for the hybrid memory system.

In summary, modest performance degradation and modest NVM writes may be tolerable for a hybrid memory system. As such, we set 10% performance degradation and 10× NVM write increase as the design constraints. From the above discussion, we choose Hybrid-NVM2 as the baseline for comparison, i.e., saving the last two levels of the ORAM tree in NVM.

B. Tradeoffs among Ring ORAM Operations

The three types of Ring ORAM maintenance operations help to tune the protocol to run smoothly. In particular, *EvictPath* flushes the blocks in the stash and resets the counters along the path. The frequency of *EvictPath* plays a key role in the design.

If we execute *EvictPath* less frequently, we expect to see more blocks accumulate in the stash and the counters of more buckets reach S, which triggers more *EarlyReshuffle* operations. However, *EarlyReshuffle* also flushes blocks from stash, though less effectively. If we execute *EarlyReshuffle* more frequently, we accordingly have more opportunities to flush the blocks in the stash.

BackgroundEvict serves as the last mechanism to ensure protocol correctness though it tends to introduce larger overhead. While we target minimizing the number of *BackgroundEvict*, the existence of *BackgroundEvict* operation exposes a large design space that we can explore across different Ring ORAM maintenance operations.

C. Under-utilized Middle Levels in Ring ORAM Tree

As mentioned in Section II, on average, half of Z' entries in one bucket can hold real data, and the rest is filled by dummy blocks to ensure correctness. However, for one bucket at a given time, it may contain zero to Z' real block.

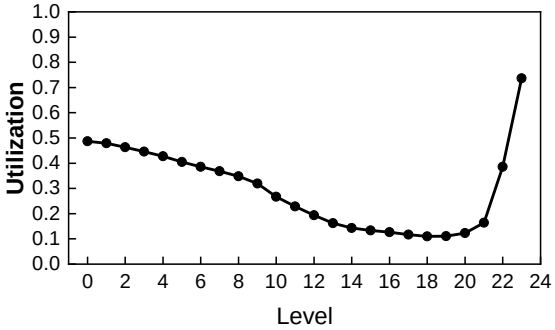

Fig. 3. Bucket utilization across Ring ORAM tree levels (Utilization=1.0 means saving Z' blocks in one bucket).

We conducted an experiment to measure the average bucket utilization across different tree levels and reported the results in Fig. 3. From the figure, we found that the bottom levels are highly utilized because they constitute the majority of the capacity. The top levels also have high utilization because they have high concentration of path overlaps [10]. For example, a block in the stash can be written back to the root regardless of

its path mapping (because all paths overlap at the root). As we move from the root to the middle levels, the utilization drops. Saving more blocks in these levels would be beneficial.

D. The EP-ORAM Design

To exploit the observations that we made in the preceding sections, we propose EP-ORAM, as shown in Fig. 4. EP-ORAM consists of two key parameters (k, h), where k indicates the number of bottom levels to be saved in NVM; and h indicates the path length that *EvictPath* is to reshuffle. (0, 24) is the all-DRAM Ring ORAM implementation while (2, 24) is the baseline that saves the two bottom levels of the ORAM tree in NVM. Here, the ORAM tree has 24 levels, indicating it has 12 GB memory space, i.e., it protects 2.5 GB of user data.

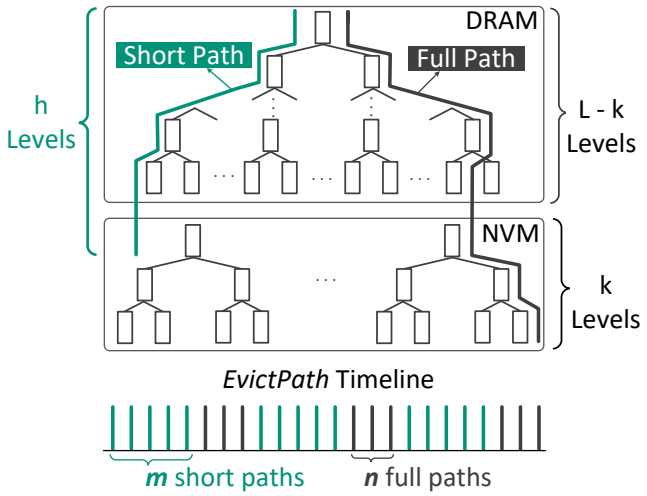

Fig. 4. An overview of the EP-ORAM design in hybrid memory system.

By choosing $h < 24$, *EvictPath* exhibits two types of path reshuffles. They differ by the number of buckets to be read, re-encrypted, and written back.

- **Full Path:** This is the default *EvictPath* operation in Ring ORAM. It reads all buckets along one tree path, i.e., $L \times Z'$ blocks, flushes the stash blocks if possible, and re-encrypts and writes $L \times Z$ back to the path. All bucket counters along the path are reset after this operation.
- **Short Path:** This is similar to full path but only applies to the top h levels. That is, the buckets in top h levels are read, re-encrypted, and written back. The last $L - h$ levels are left untouched.

The k and h values. For a DRAM/NVM hybrid system, k and h are two independent meta parameters. That is, it is possible to partially reshuffle a short path that consists of only DRAM levels, or be extended to the NVM level. On the one hand, by having the short path consisting of only DRAM levels, the partial reshuffle is fast and incurs zero NVM writes. However, for a DRAM/NVM hybrid system that has an independent memory channel for NVM, the NVM memory channel stays idle during the long *EvictPath*, which wastes the memory parallelism that can be potentially exploited to improve the effectiveness of the design. On the other hand, extending the short path to the NVM level incurs more NVM writes. It could become a major concern if the number of NVM write is too high.

Short/Full path pattern. There are two direct impacts of adopting short path *EvictPath*: (1) The buckets from bottom $L - h$ levels are left untouched and thus their counters are not reset. There is an increasing possibility of triggering *EarlyReshuffle* operations. (2) The blocks in the stash that may be flushed to the bottom levels can now be flushed to $L - h$ and above levels. This effectively increases the utilization of these levels. Due to their limited sizes, there is a possibility the blocks may not be flushed and thus stay in the stash longer. An overflow of the stash may trigger *BackgrounEvict* and degrades the overall performance. For this purpose, we propose to conduct m short path *EvictPath* and then n full path *EvictPath* and repeat continuously, as shown in Fig. 4. Note, there are five *ReadPath* operations between every two *EvictPath* invocations, the same as the baseline Ring ORAM.

To ensure the same security protection as that of the baseline Ring ORAM implementation, the invocation of short path *EvictPath* cannot be on-demand. That is, we need to statically decide a fixed pattern and apply it to all benchmarks.

Intuitively, the bigger the m value is, the more we reduce the number of NVM accesses. However, a big m value leads to a higher chance of increased space utilization of middle levels and possible stash overflow. We will study these parameters in the experiment section.

E. Security Analysis

In this section, we discuss how EP-ORAM preserves the same security guarantee as Ring ORAM.

In all tree-based ORAMs, the security guarantee is that path accesses are indistinguishable so that an attacker cannot infer any information about the user program requests. EP-ORAM does not affect the online access i.e. *ReadPath* operation so these path accesses remain indistinguishable the same way they were in Ring ORAM.

Regarding *EvictPath* operation, path lengths of evicted paths are changed in EP-ORAM but they follow a fixed pattern. Note that in Ring ORAM *EvictPath* operations are statically scheduled in a reverse-lexicographic order. Therefore, what path is going to be evicted next is public information. In the same manner, in EP-ORAM, it is public information what paths are going to be accessed in full or short length. This pattern is fixed during the execution and remains the same for all programs. Therefore, EP-ORAM does not leak any extra information to the attacker.

EarlyReshuffle activation is also public knowledge. Any bucket that is accessed S times is going to be reshuffled. EP-ORAM does not change *EarlyReshuffle* operation and thus leaks no extra information.

While EP-ORAM potentially increases the utilization in the middle tree levels, the change of the overall utilization is known but the utilization for a given bucket, i.e., how many real data blocks saved in this bucket, remains unknown to the

attackers. From what an attacker can observe from the outside, all blocks in a 12-entry bucket remains indistinguishable.

In summary, EP-ORAM preserves the security guarantee of Ring ORAM.

IV. Evaluation

We used trace-based simulation to evaluate EP-ORAM, similar to those in the literature [5], [6], [10], [12]. We used the Pin tool [13] for collecting traces from SPEC CPU2017 [14]. For each benchmark, we gathered 40 million memory access traces after skipping the warm-up phase. We ran each trace repeatedly to form a 400 million trace to place the ORAM tree into a stable state. We fed the last million accesses to USIMM [15] for DRAM access simulation. We modeled a 4-issue OoO (out-of-order) 3.2GHz processor with 160 ROB entries; a 4-channel memory with one channel is dedicated to NVM. We adopted ReRAM as the NVM and added a fixed latency of tWR = 200ns derived from [11] to simulate NVM accesses in USIMM. Table I lists the rest of the system configurations.

TABLE I
System configuration.

ORAM Configuration		Processor Configuration	
ORAM tree	24 levels	L1 D-cache	2-way 256 KB
Tree top cache	10 levels	L2 (LLC)	8-way 2 MB
Block size	64B	Mem channels	3 DRAM, 1 NVM
Stash entries	300	NVM	ReRAM (10^9 writes)

Following the typical setting in [4], we modeled a 24-level Ring ORAM tree with Z=12, and Z'=5, S=7. Given that each block is a 64B, the total ORAM tree size is $(2^{24}-1)\times12\times64$B = 12 GB. Only 50% of all Z' entries contain user data. Thus, the protected user space is 2.5 GB. We cached the top 10 levels of the tree on-chip [10]. We evaluated the following schemes.

- **All-DRAM:** it implements Ring ORAM with all tree levels being stored in DRAM.
- **Hybrid-NVM2:** it implements Ring ORAM with 2 levels in NVM and the rest of the levels in DRAM.
- **EP-ORAM:** it implements Ring ORAM and adopts EP-ORAM with k=3, and h=21. The short/full path pattern is that it uses $m=\infty$, and n=0.

A. DRAM Space, NVM Traffic and Performance Analysis

Fig. 5 compares the DRAM space demand for different schemes. From the figure, EP-ORAM reduces DRAM demand to only 1.5 GB, exhibiting 50% and 87.5% reductions over Hybrid-NVM2 and All-DRAM, respectively. This greatly alleviates the DRAM space demand in Ring ORAM designs.

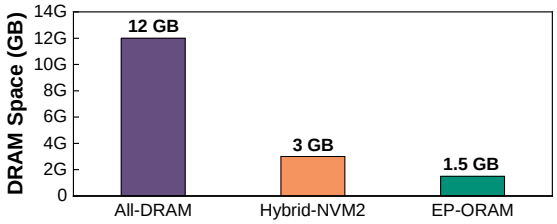

Fig. 5. DRAM space demand of different schemes.

EP-ORAM achieves large DRAM savings under the design constraints in Section III-A, i.e., 10% performance slowdown over All-DRAM, and 10× user memory requests. Fig. 6 reports the number of NVM writes in EP-ORAM with the result being normalized over Hybrid-NVM2. On average, EP-ORAM reduces the NVM write traffic by 15%. This reduction comes mainly from that short path *EvictPaths* generate no NVM writes. However, the number of *EarlyReshuffles* for buckets in NVM increases as bucket counters are not frequently reset. The reduction overweighs the increase when k=3. The number of NVM writes in EP-ORAM is 7.7× user memory requests. Fig. 7 reports the slowdown of Hybrid-NVM2 and EP-ORAM over All-DRAM baseline. On average, Hybrid-NVM2 and EP-ORAM incur 10% and 8% slowdowns over All-DRAM.

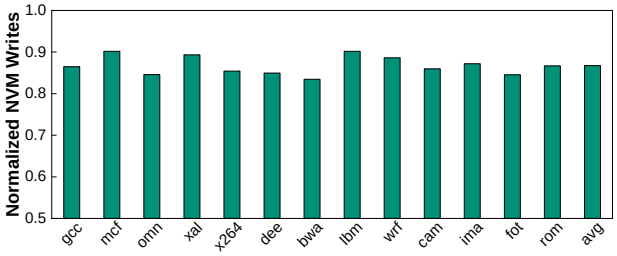

Fig. 6. NVM writes reduction of EP-ORAM compared to Hybrid-NVM2.

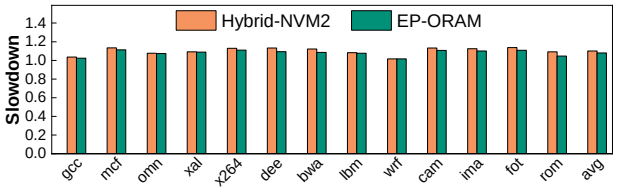

Fig. 7. Performance comparison of EP-ORAM and Hybrid-NVM2.

B. EP-ORAM Design Exploration

In this section, we study how different design choices affect EP-ORAM. Fig. 8 compares the number of NVM writes using different m and n values, with the result being normalized over Hybrid-NVM3 (i.e., saving the bottom 3 levels in NVM, and no EP-ORAM). The shaded portion of each bar indicates the amount of writes from *EvictPath* while the rest comes from *EarlyReshuffle*. For each configuration $mAnB$ indicates m=A, and n=B. From the figure, with increasing m values, the number of NVM writes from *EvictPath* decreases whereas that from *EarlyReshuffle* increases. On average, the total number of NVM writes decreases as we enlarge m value. In particular, $m\infty n0$ reduces the NVM writes to 58% of those in Hybrid-NVM3. Fig. 9 compares the execution time of different configurations of EP-ORAM with the result being normalized over Hybrid-NVM3. From the figure, $m\infty n0$ performs the best and reduces the execution time by 10%.

However, we observed that `lbm` has more NVM writes and larger slowdown at $m\infty n0$ over those at $m50n1$. This is due to more invocations of expensive *BackgroundEvict* operations.

The k, h parameters in EP-ORAM design are independent. Fig. 10 compares the following configurations: Hybrid-NVM4 and EP-ORAM with k=4, h=21. Hybrid-NVM4 incurs 35% slowdown compared to All-DRAM. EP-ORAM reduces this degradation to 11% on average. While it is slightly over our design constraint (10% performance degradation), it reduces

the DRAM space by 93.75% over All-DRAM, or 768MB rather than 12GB in All-DRAM.

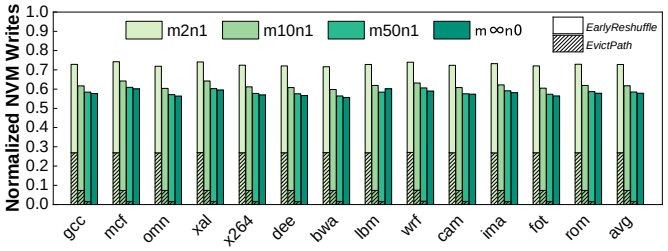

Fig. 8. NVM writes reduction with different configurations of EP-ORAM compared to Hybrid-NVM3.

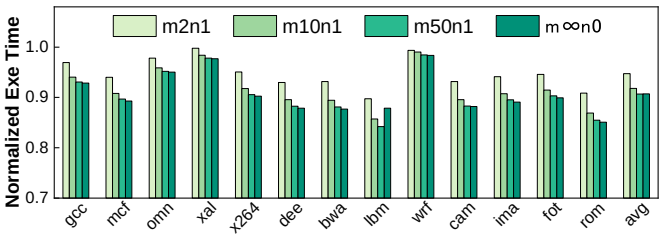

Fig. 9. Performance improvement with different configurations of EP-ORAM compared to Hybrid-NVM3.

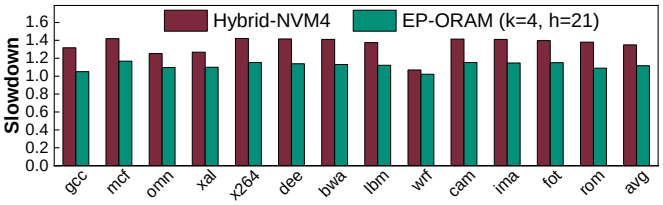

Fig. 10. Comparing EP-ORAM and Hybrid-NVM4 slowdown.

C. Utilization Analysis

To study the impact of EP-ORAM on bucket utilization, we repeated the experiment in Fig. 3 with EP-ORAM. Fig. 11 summarizes the results. From the figure, EP-ORAM makes better use of middle levels. There is a spike at level 20 because Ring ORAM aggressively writes blocks to the bottom level and levels 23 and 20 are the bottom levels for the full path and short path *EvictPath*, respectively.

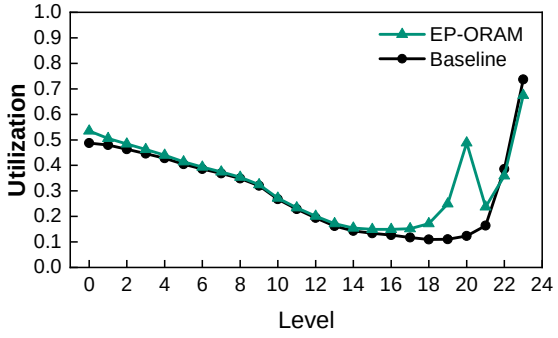

Fig. 11. Bucket utilization of EP-ORAM and the baseline across levels.

V. More Related Work

Liu *et. al.* proposed PS-ORAM [16] to offer efficient crash consistency for ORAM protocols adopted in NVM. Che *et. al.* optimized channel imbalance in Ring ORAM due to online accesses [17]. IR-ORAM [10] optimizes Path ORAM by exploiting low-utilized levels to reduce the path length. There are also other optimizations proposed on top of Path ORAM that may be adopted to Ring ORAM [12], [18], [19].

VI. Conclusion

In this paper, we propose EP-ORAM to enable efficient adoption of Ring ORAM in DRAM/NVM hybrid memory. EP-ORAM partitions the ORAM tree such that bottom levels are stored in NVM to save DRAM space. EP-ORAM identifies an opportunity in existing tradeoffs among Ring ORAM operations to shorten the EvictPath operation. EP-ORAM achieves 50% DRAM space saving and reduces NVM writes by 15%.

Acknowledgements

We thank all anonymous reviewers for their constructive comments. This work is supported in part by NSF grants #2011146, #1910413, #2154973, #1725657, and a startup funding from the University of Pittsburgh.

References

[1] I. Corp., *Intel Software Guard Extensions*, 2014. [Online]. Available: https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions.html
[2] O. Goldreich, "Towards a theory of software protection and simulation by oblivious rams," in *STOC*, 1987.
[3] E. Stefanov, M. van Dijk, E. Shi, C. W. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path oram: An extremely simple oblivious ram protocol," in *CCS*, 2013.
[4] L. Ren, C. W. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. van Dijk, and S. Devadas, "Ring ORAM: closing the gap between small and large client storage oblivious RAM," *IACR*, 2014.
[5] D. Cao, M. Zhang, H. Lu, X. Ye, D. Fan, Y. Che, and R. Wang, "Streamline ring oram accesses through spatial and temporal optimization," in *HPCA*, 2021.
[6] M. Raoufi, J. Yang, X. Tang, and Y. Zhang, "AB-ORAM: Constructing adjustable buckets for space reduction in ring oram," in *HPCA*, 2023.
[7] H.-S. P. Wong, H.-Y. Lee, S. Yu, Y.-S. Chen, Y. Wu, P.-S. Chen, B. Lee, F. T. Chen, and M.-J. Tsai, "Metal–oxide rram," *Proceedings of the IEEE*, 2012.
[8] E. Chen, D. Lottis, A. Driskill-Smith, D. Druist, V. Nikitin, S. Watts, X. Tang, and D. Apalkov, "Non-volatile spin-transfer torque ram (stt-ram)," in *DRC*, 2010.
[9] W. He, F. Wang, and D. Feng, "H2oram: Low response latency optimized oram for hybrid memory systems," in *ICCD*, 2020.
[10] M. Raoufi, Y. Zhang, and J. Yang, "IR-ORAM: Path access type based memory intensity reduction for path-oram," in *HPCA*, 2022.
[11] C. Xu, D. Niu, N. Muralimanohar, R. Balasubramonian, T. Zhang, S. Yu, and Y. Xie, "Overcoming the challenges of crossbar resistive memory architectures," in *HPCA*, 2015.
[12] R. Wang, Y. Zhang, and J. Yang, "Cooperative path-oram for effective memory bandwidth sharing in server settings," in *HPCA*, 2017.
[13] I. Corp., *Pin - A Dynamic Binary Instrumentation Tool*, 2012. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html
[14] *SPEC CPU 2017 Benchmark Suite*, 2017. [Online]. Available: https://www.spec.org/cpu2017
[15] N. Chatterjee, R. Balasubramonian, M. Shevgoor, S. H. Pugsley, A. N. Udipi, A. Shafiee, K. Sudan, M. Awasthi, and Z. Chishti, "Usimm : the utah simulated memory module," 2012.
[16] G. Liu, K. Li, Z. Xiao, and R. Wang, "Ps-oram: Efficient crash consistency support for oblivious ram on nvm," in *ISCA*, 2022.
[17] Y. Che, Y. Hong, and R. Wang, "Imbalance-aware scheduler for fast and secure ring oram data retrieval," in *ICCD*, 2019.
[18] C. W. Fletcher, L. Ren, A. Kwon, M. van Dijk, and S. Devadas, "Freecursive oram: [nearly] free recursion and integrity verification for position-based oblivious ram," in *ASPLOS*, 2015.
[19] R. Wang, Y. Zhang, and J. Yang, "D-oram: Path-oram delegation for low execution interference on cloud servers with untrusted memory," in *HPCA*, 2018.