2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA) | 978-1-6654-2235-2/20/$31.00 ©2021 IEEE | DOI: 10.1109/HPCA51647.2021.00012

Streamline Ring ORAM Accesses through Spatial and Temporal Optimization

Dingyuan Cao*†, Mingzhe Zhang†, Hang Lu†, Xiaochun Ye†‡, Dongrui Fan†, Yuezhi Che§ and Rujia Wang§
*Tsinghua University, Beijing, China
†State Key Laboratory of Computer Architecture, ICT, CAS, Beijing, China
‡State Key Laboratory of Mathematical Engineering and Advanced Computing, China
§Illinois Institute of Technology

cdy17@mails.tsinghua.edu.cn, {zhangmingzhe, luhang, yexiaochun, fandr}@ict.ac.cn, yche3@hawk.iit.edu, rwang67@iit.edu

***Abstract*—Memory access patterns could leak temporal and spatial information in a sensitive program; therefore, obfuscated memory access patterns are desired from the security perspective. Oblivious RAM (ORAM) has been the favored candidate to eliminate the access pattern leakage through randomly remapping data blocks around the physical memory space. Meanwhile, accessing memory with ORAM protocols results in significant memory bandwidth overhead. For each memory request, after going through the ORAM obfuscation, the main memory needs to service tens of actual memory accesses, and only one real access out of them is useful for the program execution. Besides, to ensure the memory bus access patterns are indistinguishable, extra dummy blocks need to be stored and transmitted, which cause memory space waste and poor performance.**

In this work, we introduce a new framework, String ORAM, that accelerates the Ring ORAM accesses with *Spatial* and *Temporal* optimization schemes. First, we identify that dummy blocks could significantly waste memory space and propose a compact ORAM organization that leverages the real blocks in memory to obfuscate the memory access pattern. Then, we identify the inefficiency of current transaction-based Ring ORAM scheduling on DRAM devices and propose an effective scheduling technique that can overlap the time spent on row buffer misses while ensuring correctness and security. With a minimal modification on the hardware and software, and negligible impact on security, the framework reduces 30.05% execution time and up to 40% memory space overhead compared to the state-of-the-art bandwidth-efficient Ring ORAM.

***Index Terms*—Ring ORAM, Performance, Space Efficiency, Security**

I. Introduction

As protecting data security and privacy becomes increasingly critical, modern computing systems start to equip trusted hardware to protect the computation and data from various attacks. For example, we see industrial standard trusted computing modules such as Trusted Computing Module (TPM) [1], eXecute Only Memory (XOM) [2], Trusted Execution Technology (TXT) [3], Intel SGX [4], AMD SME [5], ARM TrustZone [6], as well as academic secure processor prototypes such as AEGIS [7], ASCEND [8]. It is clear to see that the current trusted computing base (TCB) is still small and limited, which includes part of the processor plus partial memory space. The majority of components in the system, such as the main memory, storage device, and the interconnections, are still vulnerable to various attacks. For example, if the memory access patterns are leaked to the adversary, the program that is being executed may be inferred through the Control Data Flow Graph reconstruction [9]. Researchers have also discovered that on a remote data storage server with searchable encryption, access patterns can still leak a significant amount of sensitive information [10]. Some recent attacks show that neural network architecture [11] and RSA keys [12] can be reconstructed through the memory access patterns.

To completely remove the potential leakage through the memory access pattern we need to obfuscate the access patterns that the adversary may observe. Oblivious RAM (ORAM), as a cryptographic approach, provides a complete set of access and remap operations that can randomly move the data blocks in the memory to a different physical address after each access [13]. The basic idea of ORAM is to utilize dummy blocks for obfuscation. For each memory access, dummy blocks are fetched together with the real block. After each memory access, the location of the accessed block is reassigned so that the temporal and spatial access pattern can be hidden. As a result, an outside attacker cannot infer the type of access or whether the user is accessing the same data repeatedly. Because of the redundancy of dummy blocks, ORAM comes with a high overhead, which motivates the ORAM designers to optimize its theoretical performance. Through decades of advances in cryptography, tree-based ORAMs show great potential to be adopted in main memory systems with relatively efficient bandwidth and storage overhead. For example, Path ORAM [14] translates one memory access into a full path operation, and has been made into hardware prototype [15] and integrated with SGX [16]. Further, Ring ORAM [17] optimizes the online data read overhead by selectively reading blocks along the path, and reduces overall access bandwidth by $2.3\times$ to $4\times$ and online bandwidth by more than $60\times$ relative to the Path ORAM [17].

Mingzhe Zhang, Rujia Wang and Dingyuan Cao have equal contribution. This work was performed while Dingyuan Cao was an undergraduate research intern at ICT, CAS. This work is supported in part by National Natural Science Foundation of China grants No. 62002339, No. 61732018, the Strategic Priority Research Program of the Chinese Academy of Sciences under grant No. XDB44030200, the Open Project Program of the State Key Laboratory of Mathematical Engineering and Advanced Computing (No. 2019A07) and the CARCH Innovation Project (CARCH4506).

978-1-6654-2235-2/21/$31.00 ©2021 IEEE
DOI 10.1109/HPCA51647.2021.00012

Authorized licensed use limited to: INSTITUTE OF COMPUTING TECHNOLOGY CAS. Downloaded on December 11,2021 at 10:41:19 UTC from IEEE Xplore. Restrictions apply.

While Ring ORAM is efficient in terms of theoretical bandwidth overhead ($log(N)$, where N is the total data blocks in the ORAM tree), when implemented on real memory system, we identify that both *space* and *access* efficiency require further optimization. Since we need to reserve and store abundant dummy blocks in the memory or storage devices, the off-chip memory effective utilization rate decreases significantly. Also, Ring ORAM accesses show biased locality during different access phases, even with the optimal subtree layout is applied. The read path operation selectively read blocks along a path, leading to relatively low row buffer hit rate; eviction operation reads and then writes on a full path and shows much better utilization of the subtree layout. As a result, implementing Ring ORAM on main memory devices requires further optimizations with hardware implications to minimize additional overhead.

This motivates us to rethink the existing ORAM design: can we achieve access and space efficiency while ensuring the memory access pattern is well obfuscated? In this work, we propose String ORAM with a set of schemes that are aiming at achieving such goals. We first quantitatively analyze the memory space waste of state-of-art ORAMs due to the dummy blocks. Then, we dig into the behavior of DRAM bank during each ORAM access, and find the opportunity to squeeze more command into one access. Based on such observations, we propose several schemes, which can: 1) minimize the memory space waste and reshuffle overhead by reusing real data blocks, 2) improve the ORAM operation performance by reducing memory bank idle time, 3) allow us to achieve both space and access efficiency with a fully integrated architecture. Our contributions are as follows:

- We present an in-depth study of memory space and access inefficiency caused by dummy data blocks in current ORAM design.
- We propose innovative protocol side modifications that can hide the access pattern by utilizing existing massive real data blocks. The more compact protocol can reduce the memory space utilization inefficiency.
- We propose a slight modification on the DRAM command scheduler, which can issue PRE and ACT command in advance. This helps to minimize DRAM bank idleness and return real blocks faster.
- We combine the two optimization approaches through architectural integration and evaluate our String ORAM framework with state-of-the-art optimizations.
- We show the evaluation results of improvement in performance, queuing time, and row buffer miss rate. We also evaluate the hardware modification overhead and security of our design.

II. Background

A. Threat Model

In this work, the system equips the secure and tamper-resistance processor, which is capable of computing without information leakage [14], [17]–[19]. The off-chip memory systems are vulnerable to access pattern attacks, such as physically monitoring the visible signals on the printed circuit boards (including the motherboard and memory modules). With commodity DRAM DIMMs in the system, the address bus, the command bus, and the data bus are separate. The memory controller sends out the pair of addresses and data to the DRAM with corresponding DRAM command, such as precharge, read/write, and activate. Therefore, the attacker can still sniff critical information through the address and command bus, even when the data bus is encrypted. By observing the access patterns such as access frequency, access type (read or write), and also the repeatability of accesses to the same location, the attacker can obtain some leaked sensitive information in the program [10].

With the emerging memory technologies and interfaces, adding additional trusted components to the memory DIMMs or data path can partially reduce the attack surfaces, therefore, reducing the need for security protection. For example, Secure DIMM [20] assumes that the entire ORAM controller can be moved from the processor to the memory module side. D-ORAM [21] assumes that the memory module is still not trusted, but we can leverage a secure delegator on the motherboard to facilitate the ORAM accesses. Meanwhile, ObfusMem [22] and InvisiMem [23] assume that the entire memory modules are trusted with logic inside and only care about the access pattern in between. However, such designs always require substantial modifications to the memory hardware or interface, which are less general approaches to hide the memory access pattern. Therefore, in our threat model, we still assume the memory device is out of the trusted boundary.

B. Basics of ORAM

Oblivious RAM [13] is a security primitive that can hide the program's access pattern and eliminate information leakage accordingly. The basic idea of ORAM is to access more blocks than the actual data we need, and shuffle the address space so that the program address appears to be random. With the ORAM controller in the secure processor, one memory access from the program is translated into an ORAM-protected sequence. ORAM protocol guarantees that any two ORAM access sequences are computationally indistinguishable. In other words, ORAM physical access pattern and the original logical access pattern are independent, which hides the actual data address with the ORAM obfuscation. Since all ORAM access sequences are indistinguishable, an attacker cannot extract sensitive information through the access pattern.

Tree-based ORAM schemes, such as Path ORAM [14] and Ring ORAM [17], have greatly improved the overall access and reshuffle efficiency through cryptographic innovations. Tree-based ORAM schemes are also the building blocks of several advanced ORAM frameworks, such as Obliviate [24], Taostore [25] and Zerotrace [16]. In this work, we focus on one of the most bandwidth-efficient tree-based ORAM, Ring ORAM [17].

Figure 1 shows the control and memory layout of Ring ORAM. The memory is organized as a binary tree, which

Authorized licensed use limited to: INSTITUTE OF COMPUTING TECHNOLOGY CAS. Downloaded on December 11,2021 at 10:41:19 UTC from IEEE Xplore. Restrictions apply.

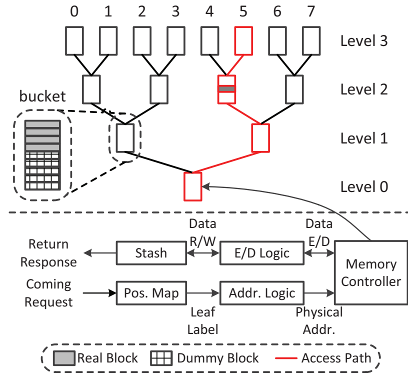


Fig. 1. An example 4-level Ring ORAM structure with $Z = 4$ and $S = 5$.

Data	Metadata			
	Index	Valid?	Real?	Counter
	0	1	1	2→3
Access →	1	1→0	0	
	2	1	0	
	3	1	1	
	4	1	0	
	5	0	1	
	6	1	1	
	7	0	0	

Real Block / Dummy Block / Accessed

Fig. 2. Ring ORAM bucket details ($Z = 4, S = 4$).

has $L + 1$ levels – the root of the tree is at level 0 while the leaves are at level L. And each node of the tree is a bucket that can store multiple real and dummy data blocks. All data blocks are encrypted indistinguishably so that an adversary cannot differentiate dummy blocks from real ones. Each leaf node has an one-to-one correspondence to a path ℓ that goes to the leaf node from the root, so there are 2^L paths in total (path 0, 1, ..., $2^L - 1$). On the controller side, the ORAM interface consists of several components: stash, position map, address logic, and encryption/decryption logic. The stash is a small buffer that temporarily stores data blocks fetched from the ORAM tree. The position map is a lookup table that maps program addresses to data blocks in the tree. In the position map, each data block corresponds to a path id ℓ, indicating that it is situated in a bucket along the path ℓ.

In the Ring ORAM construction, each bucket on the binary tree node has $Z + S$ slots and a small amount of metadata. In these slots, Z slots store real data blocks, and S slots store dummy blocks. Figure 2 shows the bucket organization of Ring ORAM. In this example, we have a bucket with $Z = 4$ and $S = 4$, and each bucket has additional metadata fields such as *index*, *valid*, *real*, and a *counter*. Bit *valid* identifies whether this block has been accessed, bit *real* identifies which blocks in the bucket are real blocks, and the *counter* records how many times this bucket has been accessed. For example, in Figure 2, a dummy block at index 1 (real bit is 0) has been accessed, so its valid bit changes to 0, and the counter increases by 1. For every bucket in the tree, the physical positions of the $Z + S$ real and dummy blocks are permuted randomly when the *counter* exceeds S.

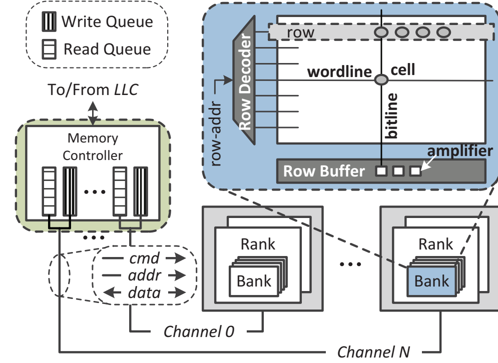


Fig. 3. The DRAM-based main memory organization.

The Ring ORAM operations are summarized as below:

- *Read path* operation reads and decrypts the metadata of all buckets along the path ℓ, to determine which bucket contains the block of interest. Then the ORAM controller selects one block to read per bucket along the path. The selection is randomly based on the metadata: a block that has been read before cannot be reread (by checking the *valid* bits). The bucket that contains the block of interest returns the real block, while all other buckets along this path return a dummy block. The blocks accessed in each bucket are marked invalid.
- *Eviction* operation is issued after every A times of *read path* operations. For each bucket along the path, it reads all the Z real blocks, permutes them, and writes $Z + S$ blocks back. The sole purpose of an eviction operation is to push blocks back to the binary tree from the stash. Ring ORAM adopts a deterministic eviction order, reverse lexicographic order so that consecutive eviction paths have fewer overlapped buckets [17].
- *Early reshuffle* is needed to ensure that each bucket is properly randomly shuffled. Each bucket can only be touched at most S times before early reshuffle or eviction because, after S accesses to a bucket, all dummy blocks have been invalidated. Early reshuffle operation reads and writes buckets that have been accessed S times and reset the metadata fields.

C. Basics of DRAM

The structure of the DRAM-based main memory system is as shown in Figure 3. In modern DRAM, a *bank* is the finest granularity that can be accessed in parallel (referred to as bank-level parallelism [26]). A group of banks in different DRAM chips consist of a *rank* and operate in lockstep. Finally, several banks are organized into one channel, which shares one set of physical links (consisted of command, address, data buses) to communicate with the memory controller. Note that the banks in different channels can be accessed in parallel (referred to as channel-level parallelism), while the banks in one channel have contention at the physical link.

The top-right portion of Figure 3 (in the blue box) shows the structure of a bank. In each bank, a two-dimensional array of DRAM cells is used to store data, and a *row buffer*

Authorized licensed use limited to: INSTITUTE OF COMPUTING TECHNOLOGY CAS. Downloaded on December 11,2021 at 10:41:19 UTC from IEEE Xplore. Restrictions apply.

consisting of several amplifiers is connected to the array. The array includes a series of rows, each of which can be selected via a wordline. The array also has multiple columns, and the memory cells in each column are connected to an amplifier in the row buffer via a shared bitline. The wordlines and the bitline together determine the data to be read or written.

The middle-left portion of Figure 3 (in the green box) presents the basic structure of the memory controller. In the memory controller, one read queue and one write queue are allocated for each channel. The memory access requests from the Last Level Cache (LLC) are stored in the corresponding queues according to their target addresses and issued when the target DRAM bank is idle. Note that, once the read/write request queues are full, the memory controller stops to receive the incoming requests, which probably causes the pipeline stall at the processing core [27].

To serve the memory requests, the memory controller issues different commands to a bank according to its status. The commands are defined as follow:

- Activate(**ACT**): selects a row and copies the entire row content to the row buffer.
- Read/Write(**RD/WR**): accesses the data in the row buffer according to the column address.
- Precharge(**PRE**): de-activates the row buffer and copies its data back to the array.

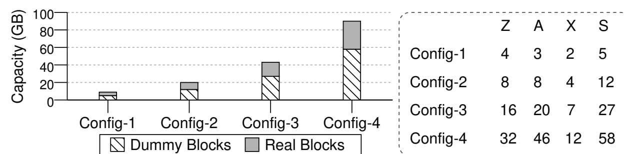
Note that, the **ACT** command can only be sent to a bank in a precharged state, i.e., the previous row buffer content is cleared. In general, there are two types of schemes for bank access: the close-page policy and the open-page policy. With the close-page policy, the **PRE** command is sent immediately after the **RD/WR** is done. Such a method removes the **PRE** command from the critical path, but it also misses the opportunity of utilizing the locality at the row buffer. On the contrary, the open-page policy allows the row buffer to keep its data after the **RD/WR** commands. In this way, consecutive requests to the same row can be served one after another without **PRE** and **ACT**. However, if the request is a row buffer miss, the **PRE** and **ACT** must be sent before the data can be accessed. Such a worst-case situation is referred to as a *row buffer conflict*. Although the open-page policy may cause more latency for the worst-case access, it also has the opportunity to accelerate memory access if the row buffer conflict rate is low. In this paper, we assume that the DRAM modules use the open-page policy.

III. MOTIVATION

In this section, we first discuss the inefficiency of Ring ORAM when it is implemented on DRAM-based system in the perspective of low space utilization and high performance overhead. Then we present our observations and the optimization opportunity for Ring ORAM.

A. Memory Space Waste Due to Dummy Blocks

As we introduced earlier in Section II-B, the memory organization with ORAM protection is padded with dummy blocks. The excessive dummy blocks in the memory are needed to hide where real data blocks are stored. As a result, the overall memory space utilization is reduced. We calculate the occupied capacity for real and dummy blocks in different Ring ORAM settings, as shown in Figure 4. Here, the definition of parameter Z, S, and A are described in Section II-B, which refer to the number of real blocks in a bucket, the number of dummy blocks in a bucket, and the eviction frequency. The relationship between S and A is theoretically defined by the equation $S = A + X$ $(X \geq 0)$ in [17]. With a larger S than A, we can ensure the early reshuffle operations don't happen too frequently. All configurations have been proved to meet the security requirements of Ring ORAM and are most theoretically bandwidth-efficient [17]. To quantitatively show the capacity overhead of dummy blocks, we set the height of the ORAM tree as $L = 23$, and block size as 64 Byte. We observe the following rules from the experimental analysis:



	Z	A	X	S
Config-1	4	3	2	5
Config-2	8	8	4	12
Config-3	16	20	7	27
Config-4	32	46	12	58

Fig. 4. The memory space utilization of Ring ORAM [17] with different configurations. The height of the ORAM tree is 23 ($L = 23$), and each block is 64 Byte.

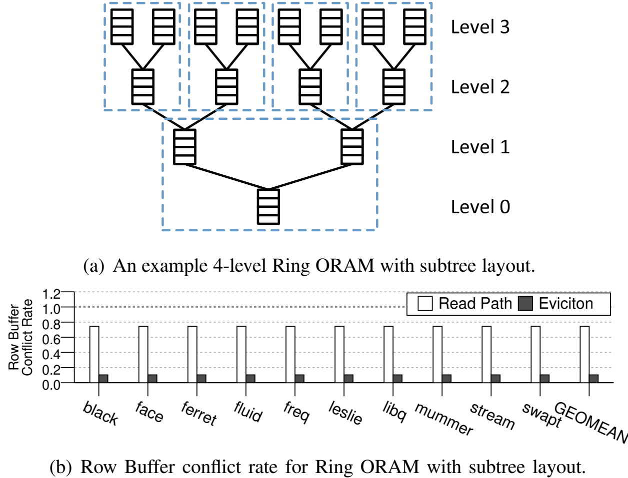
1) The real blocks capacity grows linearly from *Config-1* to *Config-4*, with the Z value increase from 4 to 32, and the actual capacity grows from 4GB to 32GB. This is because the definition of Z decides how many real blocks are stored in the ORAM tree.

2) The dummy blocks capacity grow **above** linearly from *Config-1* to *Config-4*, due to the S value increase from 5 to 58. The reason for the S value needs to be this large is because of the equation defined above. Theoretically, these A and S pairs can achieve the best overall bandwidth [17]; however, the memory space waste is unacceptable – with $Z = 32$ and $S = 58$, the ORAM tree requires 58GB extra memory space for dummy blocks to provide $32GB$ capacity for real blocks. Such a configuration only has a memory space efficiency of 35.56% (proportion of real blocks capacity over total memory capacity allocated to the ORAM).

B. Row Buffer Conflicts with Selective Read Operation

Ring ORAM utilizes a tree-based logical structure to ensure its theoretical security and efficiency. When the logical tree is mapped to the physical DRAM memory space, we need to consider how to utilize the parallelism in the memory system during an ORAM access. Also, different address bit stripping schemes could result in distinct path access patterns. Subtree layout [19] is considered as the most efficient address mapping for tree-based ORAM organization, which maximizes the row buffer utilization, especially for full path accesses (such as Path ORAM). The core idea of subtree layout is to group the data blocks in a subtree and map them to row buffer as a whole. As shown in Figure 5(a), the 4-level ORAM tree is horizontally divided into two layers of subtrees. Assuming

Authorized licensed use limited to: INSTITUTE OF COMPUTING TECHNOLOGY CAS. Downloaded on December 11,2021 at 10:41:19 UTC from IEEE Xplore. Restrictions apply.



(a) An example 4-level Ring ORAM with subtree layout.

(b) Row Buffer conflict rate for Ring ORAM with subtree layout.

Fig. 5. The ineffectiveness of sub-tree layout for Ring ORAM.

that each subtree's blocks are in the same row, then accessing a full path can be translated into 2 row accesses (16 memory accesses in total). In this case, only 2 of them are row buffer misses, and the remaining 14 blocks are all fast row buffer hits. The row buffer conflict rate is relatively low in this case.

Although Ring ORAM is also tree-based, the unique read path operation degrades the benefits of the subtree layout. Only one block per bucket is fetched each time, so the total data blocks transferred are reduced. Considering the same tree configuration, as shown in Figure 5(a), a Ring ORAM read operation will bring only 4 blocks in total. In this case, half of the accesses are row buffer hits, and the other half are row buffer misses. Therefore the row buffer conflict rate is increased. Such a scenario would be exaggerated when we have a multi-channel multi-bank memory system. Our experiment found that on a four-channel memory system, the row buffer conflict rate during the selective read path operation is significantly higher than the full path eviction operation. Figure 5(b) illustrated the biased locality on row buffer during these two distinct phases. During the read path operation, the row buffer conflict rate is around 74%; however, the full path eviction operation has a much lower conflict rate of 10%. Therefore, we find that the subtree layout is exceptionally effective for full path operation, but not enough for accelerating the selective read path operation in the Ring ORAM. The read path operation is always a critical operation during the execution, so its performance impact is obvious.

C. Idle Bank Time with Transaction-based Scheduling

Next, we discuss how ORAM accesses are translated into DRAM commands and scheduled by the DRAM memory controller. After checking the position map, the ORAM controller will generate the physical addresses of the data block to be fetched along the selected path. The memory controller then actually translates the access sequences into memory requests that contain memory commands and addresses that memory DIMMs are capable of understanding. For conventional pro-

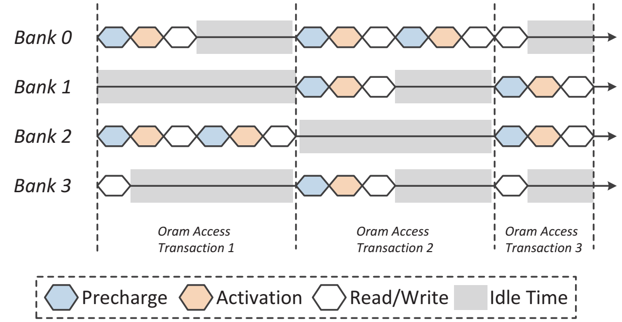


Fig. 6. The illustration of the idle time for the ORAM based on a 4-bank DRAM.

Algorithm 1: Transaction-based scheduler algorithm

Input: $\mathtt{i}$: current ORAM access transaction number
n: current cycle
Output: Issue command to the DRAM module

1 **while** *not end of the program* **do**
2   **if** *memory controller can issue command at cycle n* **then**
3   check memory command queue;
    if *has commands $\in$ transaction $\mathtt{i}$* **then**
4   issue the command based on FR-FCFS;
5   **else**
6   Continue;
7   **end**
8   **end**
9   **if** *no commands $\in$ transaction $\mathtt{i}$* **then**
10     $\mathtt{i}$++;
11   **end**
12   n++;
13 **end**

grams without ORAM's protection, once the requests are in the memory controller's queue, the PRE, ACT, or RD/WR can be freely scheduled based on the bank or channel idleness to maximize the performance. However, one single ORAM access now consists of multiple data block accesses, and they must be issued to the memory in-order and atomically. We refer to such scheduling as transaction-based scheduling [28], where the transaction means all the memory requests for the same ORAM operation. Figure 6 shows the example of three ORAM accesses to multiple memory banks. Within each ORAM access transaction, the commands include not only the actual RD/WT commands but also the PRE and ACT commands due to the bank conflicts.

As a result, when the memory controller is issuing the memory requests, it has to follow the transaction-based timing constraints. The transaction-based scheduling algorithm is described in Algorithm 1. The $i+1$-th ORAM access must wait for the i-th access completion before it is scheduled out to the memory. We can observe mixed commands sent to random memory channels and banks within each ORAM

Authorized licensed use limited to: INSTITUTE OF COMPUTING TECHNOLOGY CAS. Downloaded on December 11,2021 at 10:41:19 UTC from IEEE Xplore. Restrictions apply.

transaction due to the random selective read path operation. Since the ACT and PRE are also attached to their own ORAM transaction, such commands can only start at the beginning of each transaction when there is a bank conflict. The simple transaction-based scheduling would cause abundant wasted time on memory banks. We define the memory bank idle time as the average duration each bank stops receiving memory command due to the transaction-based scheduling barrier. In Figure 6, we can observe that when some memory banks have a higher workload than the others, although the idle banks are ready to issue the ACT or PRE commands, they are not able to do so, such as bank 1 in ORAM access 1, and bank2 in ORAM access 2.

To summarize, we identify that current ORAM transaction-based scheduling can cause significant bank idleness, especially when the read path operation causes a high row buffer conflict rate, as explained in the prior section. The PRE and ACT commands do not return any data back to the processor. Therefore, if we can free the scheduling of them from the transaction, we can significantly improve the memory bank utilization and overlap the row buffer conflicts.

D. Design Opportunities

Based on the three observations above, we identify the following design opportunities:

1) Typically, the Ring ORAM requires that the $S \geqslant A$, which provides abundant dummy blocks for the *read path* operations at the cost of storage waste. If we can reduce the S and allow part of real blocks to be accessed as dummy blocks, the memory space efficiency can be improved significantly.

2) Subtree layout can significantly promote the access efficiency under the open-page policy for full path read or write operation. However, it is not efficient for selective read path operation. Therefore, if we can change the row buffer management scheme for the read path, we are able to minimize the performance impact of high row buffer conflict rate and long critical path delay.

3) Transaction based ORAM scheduling technique ensures the correctness of ORAM protocol; however, when it comes to the command-level scheduling, we find it is less desired to group the PRE and ACT within the current ORAM access transaction. If we can schedule such commands earlier, we have a higher chance to utilize the idle bank and hide the latency caused by row buffer conflicts. In this case, without reducing or changing the number of row buffer conflicts, we preserve the security and correctness of ORAM while improving the performance.

The mentioned approaches, in turn, improve the efficiency of ORAM access from spatial and temporal aspects. The next section describes the details of our spatial optimization through a compact ORAM bucket design and temporal optimization with a proactive bank management scheme.

IV. Design

Our ORAM framework, *String ORAM*, reduces the wasted memory space, the average memory request queuing time, and the row buffer pollution. The framework consists of: a) a compact ORAM bucket organization and updated access protocol; b) a new scheduler aiming at reducing bank idle time caused by transaction based scheduling; c) an integrated architecture that can support efficient memory utilization and access for ORAM.

A. Compact Bucket (CB): A Compact ORAM Organization and Access Protocol

Based on our motivations in Section III-A, the majority of the allocated space for the ORAM protected program stores dummy data blocks, which significantly reduces the usability of the limited main memory space. As shown in Figure 7 (a), Ring ORAM reserves S dummy blocks per data bucket so that it can support at most S dummy accesses before a *reshuffle* operation. Meanwhile, the rest of Z real blocks may remain untouched, if there is no real data access in this bucket.

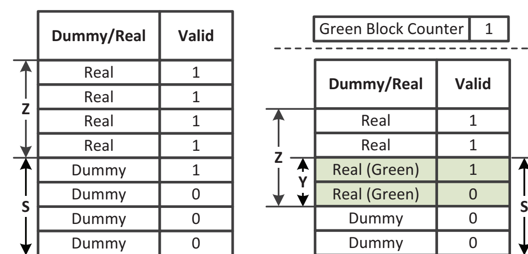


Fig. 7. The equivalent *Compact Bucket* design

Ideally, we want to minimize the dummy blocks in the bucket. A simple take is to reduce the value of S directly. However, if we only have a few dummy blocks per bucket, the reshuffle would happen very frequently, and the overhead would be significant. To reduce the value S, and ensure the reshuffles happen at a similar frequency, our idea is to borrow the real blocks that are already in the bucket, and treat them as *green* blocks, as shown in Figure 7 (b).

The Compact Bucket(CB) organization in Figure 7 (b) can support equivalent accesses to the bucket, compared with (a). Here, we reduce the number of reserved dummy blocks to $S-Y$, where Y is the number of real blocks in the bucket that can be served as dummy data during a read path access. We define such blocks as *green* blocks, and value Y as *CB rate*. Therefore, with the help of Y green blocks, we can achieve the same number of operations per bucket as in Figure 7 (a). In this example ($Z = 4$, $S = 4$, $Y = 2$), we limit the number of real data blocks that can be fetched as dummy blocks to 2. And with the additional 2 dummy blocks reserved in the bucket, this bucket can still support up to 4 accesses before path eviction or early reshuffle.

ORAM accesses with CB. To facilitate the accesses to CB, we slightly modify the metadata in the bucket. In the original Ring ORAM, we use a counter per bucket to hold how many accesses have been made to the bucket, and one bit per block to record whether it is a real or dummy block. As we need to limit the number of green blocks in a bucket, we need to have

Authorized licensed use limited to: INSTITUTE OF COMPUTING TECHNOLOGY CAS. Downloaded on December 11,2021 at 10:41:19 UTC from IEEE Xplore. Restrictions apply.

a *green block counter* to record how many green blocks have been touched. The counter size is comparably small, which is at $log_2(Y)$. When there is a read path operation, the block selection can freely choose a dummy block in the bucket or a real block if the green counter value is less than Y. During the eviction and reshuffle, the green counter values are reset, just like other metadata in the bucket.

Choosing the right Y and managing the stash overflow. The following questions arise when we modify the Ring ORAM into a compact format. First, can we set the value of Y as big as possible? Second, what is the consequence of setting a big Y? Third, how do we determine the best Y for a given ORAM configuration? Clearly, with CB, we are bringing more than one real block per read path operation, and this adds the burden on the stash. With the same size of stash, using an aggressive CB configuration with a large Y value can cause the stash fill quickly. To address the stash overflow problem, we adopt background eviction [29], which was initially proposed for Path ORAM. When the stash size reaches a threshold, the background eviction is triggered, and it halts the execution and starts to write blocks in the stash back to the ORAM. However, at this point, we may not meet the Ring ORAM eviction frequency A. If the ORAM controller issues the eviction directly without following the eviction frequency, it may leak information such as the stash is almost full, as we see consecutive eviction patterns instead of multiple read path then eviction pattern. Therefore, dummy read path operations (reading specifically dummy blocks) have to be issued until the desired interval A is reached and eviction operation is called. In this way, our background eviction does not change the access sequences and prevents such leakage.

Due to the high overhead of background eviction, it is recommended to have a modest Y value that triggers less or almost no background eviction. We analyze the tradeoffs in the result sections with different Y selections and various stash sizes.

CB benefits summary. As the spatial optimization in our framework, the space efficiency brought by CB is obvious. If we reserve Y real blocks in one bucket as green blocks, we can reduce the space overhead by Y blocks per bucket. If the value of Y is properly chosen (without triggering too many background eviction), the additional benefits of this scheme are that the number of dummy blocks that need to be read and written during the eviction/reshuffle phase is reduced, as well as the number of blocks per path that needs to be permuted. Thus, the time spent on eviction and reshuffle is reduced, and this can in turn accelerate the read path operation. ORAM accesses will experience much shorter request queuing time in the memory controller.

B. Proactive Bank (PB): A Proactive Memory Management Scheme for ORAM Scheduler

As reported in section III-B, the read path and eviction operation in Ring ORAM show distinct memory access locality. The selective block read cannot fully leverage the locality benefits from the subtree layout, therefore, a large portion of the memory accesses during the read path phase are row buffer conflicts. This means the row buffer inside each memory bank needs to be closed then opened frequently with PRE and ACT commands. Moreover, we find that due to the transaction-based scheduling, the PRE and ACT cannot be issued ahead of each transaction.

We propose a proactive bank (PB) scheduler that separates the PRE and ACT commands from the ORAM transaction during the command scheduling. Algorithm 2 shows our modified scheduling policy. Instead of staying idle and waiting for all commands for the current transaction i finished, the PB scheduler scans the memory command queue to see if any PRE or ACT coming from $i + 1$ can be issued ahead. In this case, when the current transaction is finished, the next transaction can directly start with RD or WR. In other words, the long row buffer miss penalty is hidden through latency overlapping.

Algorithm 2: PB scheduler algorithm

```
Input: i: current ORAM access transaction number
       n: current cycle
Output: Issue command to the DRAM module
1  while not end of the program do
2      if memory controller can issue command at cycle n
         then
3          check memory command queue;
           if has commands ∈ transaction i then
4              issue the command based on FR-FCFS;
5          else if has command ∈ transaction i+1 then
6              if meet inter-transaction row buffer conflict
                 and the command is PRE or ACT then
7                  issue the command;
8              end
9          else
10             Continue;
11         end
12     end
13     if no commands ∈ ORAM transaction i then
14         i++;
15     end
16     n++;
17 end
```

By revisiting the example in motivation, with PB scheduler, some of the PREs and ACTs can be issued by the memory controller ahead of the current ORAM transaction, as shown in the Figure 8. These commands are marked with a red outline. Clearly, the reason that such PREs and ACTs can be done ahead is that such row buffer conflicts are inter-transaction. As a result, whenever these ORAM transactions are in the memory request queue, the PREs and ACTs are able to be issued. Our PB scheduler does not fetch the PREs and ACTs that are caused by intra-transaction conflicts to the same bank. For example, in ORAM access 2, the second set of PRE and ACT are still issued in-order. As we do not change the access sequences for each ORAM access, such

Authorized licensed use limited to: INSTITUTE OF COMPUTING TECHNOLOGY CAS. Downloaded on December 11,2021 at 10:41:19 UTC from IEEE Xplore. Restrictions apply.

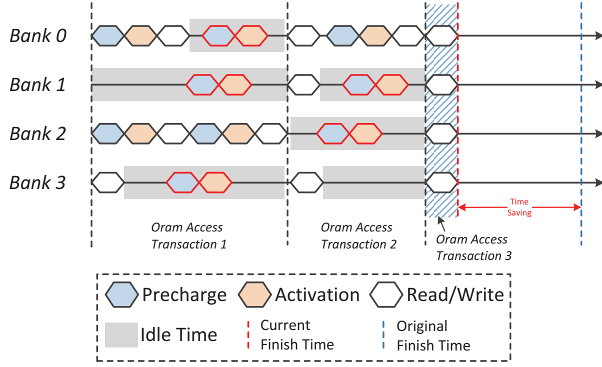


Fig. 8. The illustration for the timing behavior of PB.

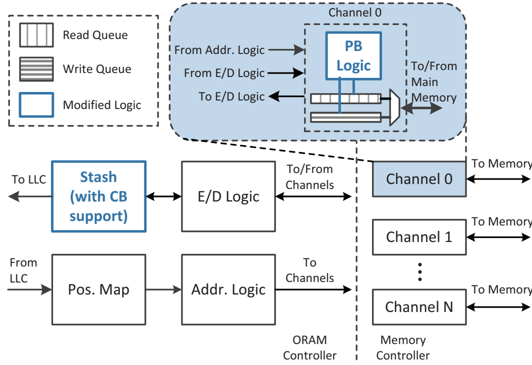


Fig. 9. The architecture overview.

intra-transaction conflicts are inevitable.

Impact on access sequence. By scheduling the memory command PREs and ACTs out of ORAM transaction, these commands' issue time will be earlier than the original time. PB scheduling only affects when such commands are issued, but not changing the command order or causing asynchronous data read or write. The actual RD and WR commands that carry data are still obeying the transaction-based access sequences. Besides, the row addresses associated with PRE and ACT are public information, scheduling them ahead does not change original addresses nor leak any information.

PB benefits summary. PB optimizes the Ring ORAM accesses in the temporal aspect. We separate the non-data related commands from the original transaction through proactive command scheduling, hence utilizing the bank idle time to prepare fast data access for the next ORAM transaction. The row buffer miss latency can be hidden through a multi-channel multi-bank memory system. Not only do we reduce the idle time on the memory system, but also shorten the read path latency.

C. Architecture Integration

To support the proposed spatial and temporal optimizations, we slightly modify the ORAM interface, bucket structure, and the DRAM command scheduler. Figure 9 shows the overall hardware architecture of our proposed framework. We highlighted the modified changes on the ORAM controller and memory controller.

For the CB scheme, we modify the bucket structure and add the green block counter to record how many green block accesses have been made to the bucket and limit the maximum to Y. In addition, the ORAM controller needs to be able to issue background eviction to mitigate the potential stash overflow caused by an aggressive Y value.

For the PB scheme, there are no modifications to the DRAM interface or DIMM side. The modification is a very lightweight scheduling policy that can be incorporated with the DRAM controller. The PB scheduler in the DRAM controller only needs to work with the ORAM interface to know the current ORAM access number and scan the command queue to decide which command to be issued.

V. Security Analysis

In this section, we discuss the security implications of our proposed performance optimization framework.

Claim 1: CB does not leak access pattern information or cause stash overflow. Compact Bucket (CB) aims at reducing the bucket size and allows real blocks to be used as dummy blocks during the read path operation. Therefore, it is possible to bring more than one real data block into the stash, which is different from the original Ring ORAM's protocol. The stash is within the security boundary, therefore the extra real data block inside of the stash does not leak any information. If the additional real blocks brought into the stash are serviced by other memory requests before eviction, the timing differences of execution the program may differ. We argue that this is not a critical issue since the prior ORAM prefetch work [29] also brings more than one real block per read request. Moreover, without the superblock scheme, in our experiment, it is rare to see the green blocks brought into stash will be consumed by other memory requests before eviction. To completely remove such leakage potential, we can force the green blocks not to be directly fetched by other requests from the stash. The other issue is the filling speed for the stash could be faster and cause stash overflow. We discuss that through leakage-free background eviction (use dummy read path operations to reach the eviction interval), we can keep the stash occupancy low. The relationship between stash size, CB rate(Y) and performance are presented in Section VII.

Claim 2: PB does not leak access pattern information during the scheduling. Proactive Bank (PB) is a lightweight memory scheduler that is easy to implement and only modify the issue time of non-data related commands(PREs and ACTs) on the memory bus. The memory access sequences on the bus, including the number of requests/commands, the order of requests/commands, are entirely remained unchanged with PB scheduler. The addresses associated with PRE and ACT are public information: PRE closes a bank and only contains the last accessed bank information, which is known since the bank has been previously accessed; ACT contains the row address for next transaction's access, which is also public as long as the path id is determined. Whether an ORAM

Authorized licensed use limited to: INSTITUTE OF COMPUTING TECHNOLOGY CAS. Downloaded on December 11,2021 at 10:41:19 UTC from IEEE Xplore. Restrictions apply.

TABLE I
PROCESSOR CONFIGURATION

Frequency	3.2GHz
# Cores	4
Core	5-stage pipeline, OoO execution support ROB size: 128; Retire width: 4; Fetch width: 4
Last Level Cache	4MB
Cacheline Size	64B

TABLE II
MEMORY SUBSYSTEM CONFIGURATION

Memory Controller	
# Memory Channel	4
Read Queue	64 entries
Write Queue	64 entries per channel
Address Mapping	row:bank:column: rank:channel:offset
DRAM Module	
Specification	DDR3-1600
Memory Capacity	8GB (per channel)
Ranks per Channel	1
Banks per Rank	8
Rows per Bank	16384
Columns (cachelines)	128 per row
Row Buffer Capacity	4KB

transaction can be accelerated depends on the bank idleness and the transaction access distribution, which is also random and public. Therefore, the memory access pattern is still computationally indistinguishable for the attacker.
Claim 3: Combining CB and PB does not leak information. Combining both schemes does not introduce additional information leakage. CB and PB try to streamline Ring ORAM accesses from two distinct directions, and combining them will only increase the performance benefits from the spatial and temporal aspects.

VI. METHODOLOGY

We implement our proposed *String ORAM* framework on USIMM [30], which supports cycle-accurate simulation for a detailed DRAM-based memory system. Based on this platform, we simulate a CMP system with the parameters of the state-of-art commercial processor, and the detailed configurations are as shown in Table I. For the memory subsystem, we follow the JEDEC DDR3-1600 specification to simulate a DRAM module with 4 channels, and each channel has 8 banks. The total capacity of the DRAM module is 32GB. The address mapping follows the order of "`row:bank:column: rank:channel:offset`", which follows the subtree layout to maximize the row buffer locality [19]. The detailed parameters for the memory subsystem are shown in Table II.

We use 10 memory-intensive applications for the evaluation. The applications are selected from PARSEC 3.0, SPEC and BIOBENCH. For each benchmark, a methodology similar to Simpoint is used to generate the trace file consisted of 500 million instructions out of 5 billion instructions. The applications and corresponding traces are also used in the MSC contest [31]. The applications are described as Table IV.

TABLE III
THE DEFAULT *String ORAM* CONFIGURATIONS

Inherited ORAM Model	Ring ORAM [17]
Stash Size	500
Data Block Size	64Byte
Binary Tree Levels (L+1)	24
Tree Top Cache Levels	6
Real Blocks per Bucket (Z)	8
Dummy Block per Bucket (S)	12
CB Rate (Y)	8

The configuration of ORAM in our framework is shown in Table III. The ORAM tree is set as $Z = 8$, $S = 12$, $Y = 8$, and $L = 23$, with a total size of 20GB and fits into our simulated memory system. The default stash size is set at 500. In Section VII, we will provide further discussion on the impact of the stash size and the *CB Rate(Y)*.

TABLE IV
WORKLOADS AND THEIR MPKIS.

Suite	Workload	MPKI	Workload	MPKI
PARSEC	black	4.58	face	10.37
	ferret	10.42	fluid	4.72
	freq	4.42	stream	5.57
	swapt	5.16		
SPEC	leslie	9.45	libq	20.20
BIOBENCH	mummer	24.07		

VII. RESULTS

To evaluate the *String ORAM*, we conduct a series of experiments. We first compare the performance of our proposed schemes with the baseline Ring ORAM, both in execution time, memory request queuing time and bank idle time. After that, we provide a sensitivity study on the CB rate with a thorough impact analysis on stash size and background eviction rate. Lastly, we discuss the broader applicability of our proposed schemes.

A. Performance

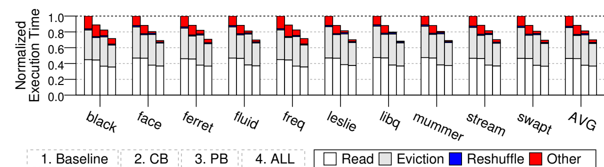


Fig. 10. Normalized execution time.

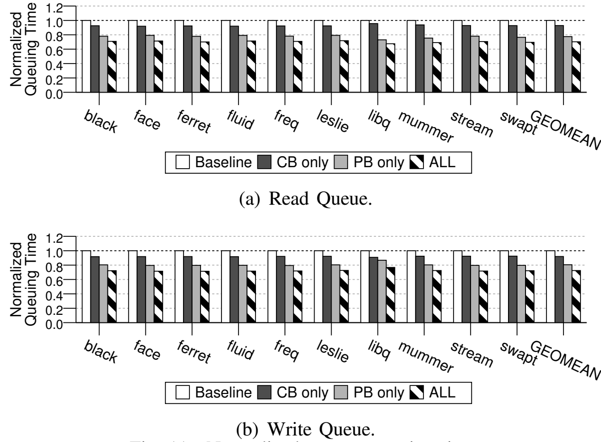
We use the total execution time (including all operations: read path, eviction, early reshuffle, and other related operations) to denote the system performance. As shown in Figure 10, the individual CB scheme improves the performance by 11.72% as the average. This is because the CB scheme reduces the number of total blocks on the path so that eviction operations take a shorter time to finish. Besides, the PB provides a more significant performance improvement than CB. On average, the execution time is decreased by 18.87%. Such improvement is achieved by moving ACT and PRE command from future ORAM access to occupy idle bank while waiting for current ORAM access finishes. Finally,

Authorized licensed use limited to: INSTITUTE OF COMPUTING TECHNOLOGY CAS. Downloaded on December 11,2021 at 10:41:19 UTC from IEEE Xplore. Restrictions apply.

when we consider the combination of CB and PB, the total performance improvement achieves 30.05%.

In addition, Figure 10 also shows that the CB, PB, and CB+PB schemes provide similar performance improvement range across different applications (the variation of all results is less than 0.38%). This indicates that our proposed schemes work for different applications and prevent information leakage from the execution time variance.

B. Queuing Time



(a) Read Queue.

(b) Write Queue.

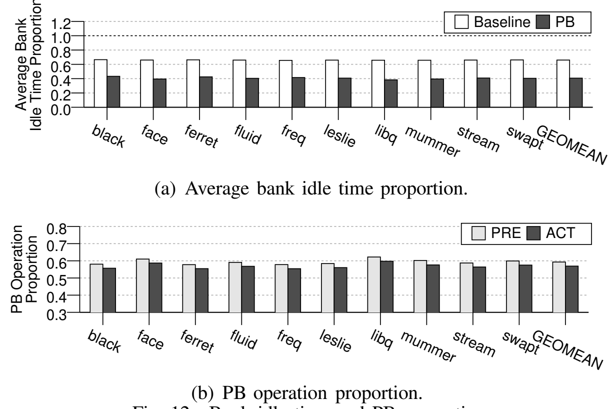
Fig. 11. Normalized request queuing time.

Figure 11 presents the memory request queuing time of different schemes. We can see that CB provides similar queuing time reduction for the read queue (10.41%) and write queue (11.83%). The reason is that CB alleviates the access overhead to the memory channel by reducing the number of memory accesses in eviction operation, which allows both queues to have more opportunities to service read path operation. On the other hand, the queuing time reduction of the read queue caused by PB is higher than the write queue (22.53% vs. 19.46%). Because PB directly reduces the performance overhead of read path operations, as the read requests can be completed more quickly. Since write operations only happen during eviction and reshuffle, the queuing time reduction of the read queue indirectly helps the write queue to gain benefit. Overall, the CB and PB scheme together reduce the queuing time of the read queue & write queue by 32.87% and 31.30%.

C. Bank Idle Time Reduction

Figure12(a) shows the average bank idle time before and after applying PB scheme. Originally, as discussed in previous sections, DRAM banks suffer from an imbalanced workload, causing bank idleness while waiting for other banks to finish current ORAM access. This idle time takes up 65.99% of the total execution time. Through the PB scheme, the idle time of the bank is greatly reduced to 40.72% of execution time, enabling bank to serve more requests than before.

Our experiments also suggest that 59.31% PRE and 56.93% ACT can be issued earlier than its own transaction, as shown in Figure12(b). These commands were overlapped with the critical path in each transaction, and as a result, data blocks



(a) Average bank idle time proportion.

(b) PB operation proportion.

Fig. 12. Bank idle time and PB proportion.

TABLE V
CB CONFIGURATIONS AND CORRESPONDING SPACE SAVING.
$(Z = 8, S = 12, L = 23)$

	CB rate	Total Memory Space (GB)	Dummy Block Percentage
Baseline	$Y = 0$	20	60%
Config-1	$Y = 2$	18	55.6%
Config-2	$Y = 4$	16	50%
Config-3	$Y = 6$	14	42.9%
Config-4 (default)	$Y = 8$	12	33.3%

can be read directly at the beginning of the transaction. The remaining commands that cannot be fetched earlier are mainly caused by intra-transaction bank conflicts.

D. CB Sensitivity Analysis

We further evaluate the effectiveness of CB with different configurations of Y. As shown in Table V, the five configurations represent different compact rates corresponding to different memory space efficiency. A higher compact rate can reduce the occupied total memory space significantly, as well as the dummy block percentage. If we want to have extreme storage efficient ORAM construction, we may choose a higher Y value, at the cost of more frequent background evictions.

While CB is not performance optimization oriented, we can still observe some performance gain through a more compact bucket design. The performance gain of CB mainly comes from the eviction phase, as we reduce the number of blocks that need to be read and written. We show the performance with different CB rates in Figure 13. When the stash size is at 500, which does not cause additional background eviction, *Config-4* with $Y = 8$ achieves the best performance. The CB scheme with $Y = 2$ to 8 has a total execution time reduction from 2.02% to 11.72%. When combining the PB scheme, the performance improvement between $Y = 2$ to 8 increases from 20.79% to 30.05%.

Figure 13 also shows the green blocks fetched per read. With CB, $0.17 \sim 3.26$ green blocks are brought into the stash per read path, on average. Therefore, Y cannot be set too aggressively if we don't want the stash to be filled too quickly. In the next section, we study the stash fill rate with different stash sizes.

Authorized licensed use limited to: INSTITUTE OF COMPUTING TECHNOLOGY CAS. Downloaded on December 11,2021 at 10:41:19 UTC from IEEE Xplore. Restrictions apply.

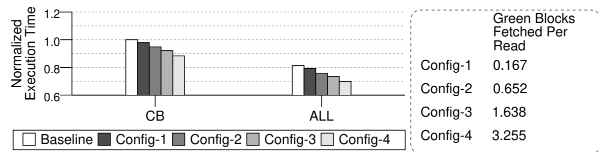


Fig. 13. The sensitivity study to the CB compact rate.

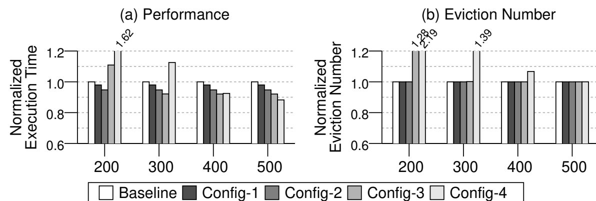
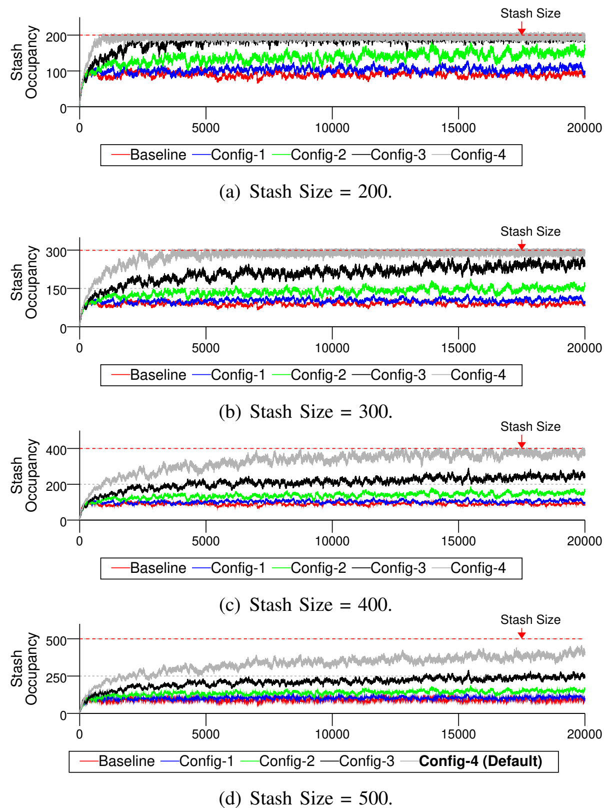


Fig. 14. Stash size v.s performance.



(a) Stash Size = 200.

(b) Stash Size = 300.

(c) Stash Size = 400.

(d) Stash Size = 500.

Fig. 15. Run-time stash occupancy with different stash size configurations

E. Stash Size v.s. Eviction Overhead

We further analyze the relationship between the stash size and additional background eviction operations with the CB scheme. Figure 14 and 15 show additional eviction operations with different stash sizes and the dynamic stash occupancy with different Y. Clearly, a smaller stash can be filled faster by the additional fetched real blocks, yet a larger one can mitigate this issue.

From the results, we observe that, although the stash occupancy increases with the Y value, in practice, the stash is not blown up with effective reverse lexicographical eviction scheme. By properly selecting the stash size, we can mitigate the background eviction overhead when the Y is large. For example, when the stash size is too small, i.e., 200, $Y \geq 6$ starts to cause background evictions. However, when we have a relatively large stash size, such as 500, even with $Y = 8$, the background eviction is not triggered during the simulated time. The enlarged stash size is still considered very small ($64B \times 500 = 32KB$) and bounded.

VIII. Related Work

We see an increasing number of architectural optimizations on ORAM recently. Firstly, since ORAM protocols generate massive data movement, with trusted ORAM logic closer to the memory, we can significantly reduce the data movement between the processor and the memory. For example, SecureDIMM [20] implements a PIM-like structure that mitigates the transfer overhead by adding ORAM logic to the DRAM module. D-ORAM [21] moves the ORAM controller on board and minimizes the bandwidth interference with other applications. Secondly, the effective data fetch per ORAM access can be improved by locality-aware schemes. PrORAM [29] dynamically merges or separates the consecutive data accesses with superblocks and then implements a locality aware prefetcher for ORAM. Multi-range ORAM [32] proposes to store range data within a path and reduce the required accesses. Thirdly, the dummy blocks in ORAM protocols have a high impact on the overall performance. Fork Path [18] focuses on the overlapped data during the path ORAM accesses and proposes to cache the content instead of writing them back. Similarly, Shadow Block [28] utilizes the dummy blocks to store the additional copies of the real data, which transforms the dummy accesses to the prefetching for real blocks. The Relaxed hierarchical ORAM [33] reforms the ORAM to a layered construction and uses small ORAM as the cache of full ORAM. Lastly, efficient ORAM scheduling schemes have been explored to maximize memory system utilization. For example, CP-ORAM [34] focuses on fairness between ORAM applications and normal applications through an application-aware scheduler. A channel imbalance-aware scheduler [35] was proposed to minimize the channel imbalance for Ring ORAM read operation. Our proposed String ORAM is a new framework focusing on the memory space waste due to dummy blocks and memory idle time due to ineffective locality improvement schemes designed for Path ORAM.

Instead of using ORAM, other memory access pattern obfuscation techniques are proposed to achieve lower overhead. InvisMem [23] and ObfusMem [22] use the logic layer on HMC and the bridge chip on NVM to implement the memory obfuscation function and reduce the overhead of issuing multiple dummy accesses per memory request. Note that, such implementations require the memory device to be partially trusted. Memcloak [36] claims to store the same data block multiple times at different addresses to achieve the obfuscated memory accesses. Compared to our design, we limit the TCB boundary while improving memory space utilization instead of storing multiple copies of real data around.

Authorized licensed use limited to: INSTITUTE OF COMPUTING TECHNOLOGY CAS. Downloaded on December 11,2021 at 10:41:19 UTC from IEEE Xplore. Restrictions apply.

IX. Conclusions

In this paper, we present *String ORAM*, a framework that accelerates the Ring ORAM accesses through an integrated architecture with spatial and temporal optimizations. Through extensive experiments, we identify that dummy blocks in Ring ORAM protocols cause significant memory space waste. Further, we find that current locality optimization schemes are less effective for Ring ORAM read operation. Therefore, we first present a compact ORAM bucket design (CB), which brings two folds of benefits: reduced memory space with fewer dummy blocks, and reduced evict path overhead with fewer blocks to shuffle. Then, we present a proactive ORAM access scheduler (PB) on the DRAM controller, which minimize the bank idle time without modifying the access sequences of ORAM. Next, we show the integrated String ORAM architecture that supports our designs. Lastly, we evaluated our proposed framework in terms of security, performance gain, queuing time reduction, memory bank idle time reduction.

References

[1] S. Bajikar, "Trusted platform module (tpm) based security on notebook pcs-white paper," *Mobile Platforms Group Intel Corporation*, vol. 1, p. 20, 2002.

[2] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," *Acm Sigplan Notices*, vol. 35, no. 11, pp. 168–177, 2000.

[3] D. Grawrock, *The Intel safer computing initiative: building blocks for trusted computing*. Intel Press Hillsboro, 2006, vol. 976483262.

[4] S. Johnson, V. Scarlata, C. Rozas, E. Brickell, and F. Mckeen, "Intel® software guard extensions: Epid provisioning and attestation services," *White Paper*, vol. 1, pp. 1–10, 2016.

[5] D. Kaplan, J. Powell, and T. Woller, "Amd memory encryption," *White paper*, 2016.

[6] "Introducing arm trustzone," https://developer.arm.com/ip-products/security-ip/trustzone, accessed: 2019-03-30.

[7] G. E. Suh, C. W. O'Donnell, and S. Devadas, "Aegis: A single-chip secure processor," *IEEE Design & Test of Computers*, vol. 24, no. 6, pp. 570–580, 2007.

[8] L. Ren, C. W. Fletcher, A. Kwon, M. van Dijk, and S. Devadas, "Design and implementation of the ascend secure processor," *IEEE Transactions on Dependable and Secure Computing*, 2017.

[9] X. Zhuang, T. Zhang, and S. Pande, "Hide: an infrastructure for efficiently protecting information leakage on the address bus," in *ACM SIGPLAN Notices*, vol. 39, no. 11. ACM, 2004, pp. 72–84.

[10] M. S. Islam, M. Kuzu, and M. Kantarcioglu, "Access pattern disclosure on searchable encryption: Ramification, attack and mitigation," in *in Network and Distributed System Security Symposium (NDSS*. Citeseer, 2012.

[11] X. Hu, L. Liang, S. Li, L. Deng, P. Zuo, Y. Ji, X. Xie, Y. Ding, C. Liu, T. Sherwood *et al.*, "Deepsniffer: A dnn model extraction framework based on learning architectural hints," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 385–399.

[12] T. M. John, "Privacy leakage via write-access patterns to the main memory," 2017.

[13] O. Goldreich, "Towards a theory of software protection and simulation by oblivious rams," in *Proceedings of the nineteenth annual ACM symposium on Theory of computing*. ACM, 1987, pp. 182–194.

[14] E. Stefanov, M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path oram: an extremely simple oblivious ram protocol," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 299–310.

[15] C. W. Fletcher, L. Ren, A. Kwon, M. Van Dijk, E. Stefanov, D. Serpanos, and S. Devadas, "A low-latency, low-area hardware oblivious ram controller," in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2015, pp. 215–222.

[16] S. Sasy, S. Gorbunov, and C. W. Fletcher, "Zerotrace: Oblivious memory primitives from intel sgx." *IACR Cryptology ePrint Archive*, vol. 2017, p. 549, 2017.

[17] L. Ren, C. W. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. Van Dijk, and S. Devadas, "Constants count: Practical improvements to oblivious ram." in *USENIX Security Symposium*, 2015, pp. 415–430.

[18] X. Zhang, G. Sun, C. Zhang, W. Zhang, Y. Liang, T. Wang, Y. Chen, and J. Di, "Fork path: improving efficiency of oram by removing redundant memory accesses," in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015.

[19] L. Ren, X. Yu, C. W. Fletcher, M. Van Dijk, and S. Devadas, "Design space exploration and optimization of path oblivious ram in secure processors," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013, pp. 571–582.

[20] A. Shafiee, R. Balasubramonian, M. Tiwari, and F. Li, "Secure dimm: Moving oram primitives closer to memory," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 428–440.

[21] R. Wang, Y. Zhang, and J. Yang, "D-oram: Path-oram delegation for low execution interference on cloud servers with untrusted memory," in *High Performance Computer Architecture (HPCA)*, 2018.

[22] A. Awad, Y. Wang, D. Shands, and Y. Solihin, "Obfusmem: A low-overhead access obfuscation for trusted memories," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, pp. 107–119.

[23] S. Aga and S. Narayanasamy, "Invisimem: Smart memory defenses for memory bus side channel," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, pp. 94–106.

[24] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee, "Obliviate: A data oblivious filesystem for intel sgx." in *NDSS*, 2018.

[25] C. Sahin, V. Zakhary, A. El Abbadi, H. Lin, and S. Tessaro, "Taostore: Overcoming asynchronicity in oblivious data storage," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 198–217.

[26] C. J. Lee, V. Narasiman, O. Mutlu, and Y. N. Patt, "Improving memory bank-level parallelism in the presence of prefetching," in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2009, pp. 327–336.

[27] L. Zhang, B. Neely, D. Franklin, D. Strukov, Y. Xie, and F. T. Chong, "Mellow writes: Extending lifetime in resistive memories through selective slow write backs," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2016, pp. 519–531.

[28] X. Zhang, G. Sun, P. Xie, C. Zhang, Y. Liu, L. Wei, Q. Xu, and C. J. Xue, "Shadow block: Accelerating oram accesses with data duplication," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 961–973.

[29] X. Yu, S. K. Haider, L. Ren, C. Fletcher, A. Kwon, M. van Dijk, and S. Devadas, "Proram: dynamic prefetcher for oblivious ram," in *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*. IEEE, 2015, pp. 616–628.

[30] N. Chatterjee, R. Balasubramonian, M. Shevgoor, S. Pugsley, A. Udipi, A. Shafiee, K. Sudan, M. Awasthi, and Z. Chishti, "Usimm: the utah simulated memory module," *University of Utah, Tech. Rep*, 2012.

[31] "2012 memory scheduling championship (msc)," http://www.cs.utah.edu/ rajeev/jwac12/, accessed: 2018-11-01.

[32] Y. Che and R. Wang, "Multi-range supported oblivious ram for efficient block data retrieval," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 369–382.

[33] C. Nagarajan, A. Shafiee, R. Balasubramonian, and M. Tiwari, "Relaxed hierarchical oram," in *The 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS)*, 2019.

[34] R. Wang, Y. Zhang, and J. Yang, "Cooperative path-oram for effective memory bandwidth sharing in server settings," in *High Performance Computer Architecture (HPCA)*, 2017.

[35] Y. Che, Y. Hong, and R. Wang, "Imbalance-aware scheduler for fast and secure ring oram data retrieval," in *2019 IEEE 37th International Conference on Computer Design (ICCD)*. IEEE, 2019, pp. 604–612.

[36] W. Liang, K. Bu, K. Li, J. Li, and A. Tavakoli, "Memcloak: Practical access obfuscation for untrusted memory," in *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, 2018, pp. 187–197.

Authorized licensed use limited to: INSTITUTE OF COMPUTING TECHNOLOGY CAS. Downloaded on December 11,2021 at 10:41:19 UTC from IEEE Xplore. Restrictions apply.