Last Path Caching: A Simple Way to Remove Redundant Memory Accesses of Path ORAM

Naoki Fujieda, Ryo Yamauchi and Shuichi Ichikawa
Department of Electrical and Electronic Information Engineering,
Toyohashi University of Technology
fujieda@ee.tut.ac.jp (Naoki Fujieda),
ichikawa@tut.jp (Shuichi Ichikawa)

***Abstract*—Oblivious RAM (ORAM) is a technique to hide the access pattern of data to untrusted memory along with their contents. Path ORAM is a recent lightweight ORAM protocol, whose derived access pattern involves some redundancy that can be removed without the loss of security. In this paper, we introduce *last path caching*, which removes the redundancy of Path ORAM with a simpler protocol than an existing scheme. By combining two caching strategies, our technique showed only 0.2% performance loss from the existing one, while keeping the determinacy of the derived access pattern.**

I. Introduction

Memory encryption [5] is a common technique for secure processors to prevent information leakage from a data bus to an external memory being observed [11], [13]. However, information does not only be leaked from the data themselves, but also from their access pattern, i.e. the sequence of memory addresses accessed by the processor [6]. For example, some of search queries to an email repository [6] and an SQLite database [7] can be distinguished by observing access pattern. Memory encryption hides the contents, while it cannot hide the access pattern.

Oblivious RAM (ORAM) [4] is a technique to hide the data and their access pattern by shuffling data and adding dummy memory accesses. Path ORAM [10] is a recent lightweight ORAM protocol, which was proposed with a hardware implementation called PHANTOM [7]. However, its overhead on the bandwidth is still high for practical use.

This study focuses on the redundancy of Path ORAM. A **path** is a set of memory blocks read and written through an ORAM access. In Path ORAM, two consecutive paths has an overlapped region. Writing to and reading from such regions can be removed as redundant memory accesses without the loss of security.

To deal with this redundancy, this paper introduces *last path caching*, which has a simpler procedure than an existing scheme, called Fork Path ORAM [15]. Last path caching also has an advantage on security: the derived access pattern is completely independent of the original access pattern, while Fork Path ORAM might reflect it in a specific condition. In the following sections, we evaluate ORAM systems and show that our method achieves almost the same performance as the existing one by combining two different caching strategy.

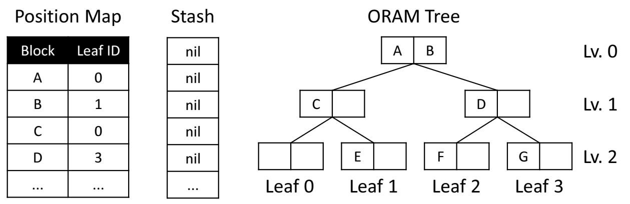
II. Path ORAM

A. Organization of Path ORAM

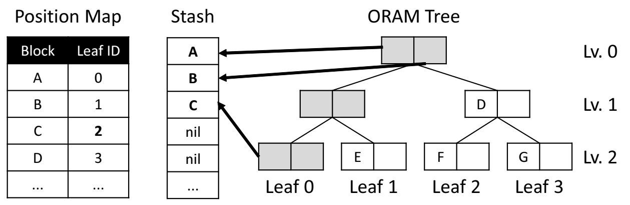
Path ORAM [15] is a lightweight ORAM protocol, and PHANTOM [7] is the first hardware implementation of Path ORAM on FPGAs. The main data structure of Path ORAM consists of an ORAM tree, a stash, and a position map, shown in Fig. 1 (a).

The **ORAM tree** is a binary tree of encrypted data, which is mapped to an external, untrusted memory. Assume the number of blocks storing actual data to be N, the height of the tree L is set to approximately $\log_2 N$, and the levels are numbered from level 0 (root) to level L (leaf). Each leaf has its own ID from 0 (left) to $2^L - 1$ (right). Each node, which is sometimes called a bucket, holds Z blocks. The number of blocks in the ORAM tree, including dummy data, is calculated as $(2^{L+1} - 1)Z$. The ORAM tree is accessed based on a path from the root to a leaf. In this paper, the path to the leaf x is denoted as $\mathcal{P}(x)$.

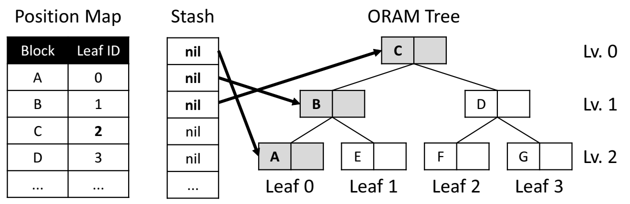
The **stash** is an on-chip cache of the ORAM tree that temporarily keeps blocks being read from the tree. Some blocks might not be written back to the tree during an ORAM



(a) Initial status.



(b) Step 3: reading $\mathcal{P}(0)$ for block C.



(c) Step 4: writing $\mathcal{P}(0)$ back.

Fig. 1. Organization and working example of Path ORAM.

access. The stash must be large enough that the probability of shortage of the stash due to such blocks is negligible.

The **position map** is a lookup table between the block address given from the processor and the path (or leaf ID) that the corresponding block belongs. A block with a leaf ID of x is found either in the buckets on $\mathcal{P}(x)$ or in the stash. The capacity of the position map is NL bits. If it is too large to be stored on the chip, a recursive approach is applied where the position map is managed by another Path ORAM [15].

B. *ORAM Access*

An ORAM access in Path ORAM is divided into four steps. It is also illustrated with an example in Fig. 1, where the requested block from the processor is B. This example assumes that $L = 2$ and $Z = 2$. In the ORAM tree, actual data blocks are labeled by A, B, ..., and G, while dummy blocks are denoted by blanks. Before a request comes, blocks A and B are placed at the root and C is in the left node of level 1. Blocks A, B, C, and D have their respective leaf IDs of 0, 1, 0, and 3.

Step 1 is a search for the stash. If the requested block is found in the stash, it is immediately accessed by the processor and the following steps are omitted. In this example, this step has no effect because there are no valid blocks in the stash.

Step 2 is a lookup and an update of the position map. The leaf ID of the requested block (x in the following steps) is read from the corresponding entry in the position map. It is then replaced by a random number from 0 to $2^L - 1$ so that the block can be remapped to another path. In the example, this step reveals that the block C is stored on $\mathcal{P}(0)$. It will be remapped to $\mathcal{P}(2)$ after the ORAM access.

Step 3 is a read of a path. All blocks in the nodes on $\mathcal{P}(x)$ are read from external memory and decrypted. Actual data blocks are moved to the stash. Now that the requested block is in the stash, it is accessed by the processor. In Fig. 1 (b), all blocks on $\mathcal{P}(0)$, painted in gray, are read and the blocks A, B, and C are stored into the stash.

Step 4 is a write back of the path. It applies the following processes to all the nodes on $\mathcal{P}(x)$, in order from the leaf to the root. First, it picks out blocks in the stash whose path includes the target node. If Z or more blocks are extracted, Z blocks chosen from them are encrypted and written to external memory. Otherwise, all the extracted blocks and dummy block(s) are written after encryption. To put it briefly, blocks in the stash are written back to buckets as near as possible to the leaf. In Fig. 1 (c), at first, blocks with paths that includes leaf 0 are extracted. Since only $\mathcal{P}(0)$ includes leaf 0, only the block A is picked. The block A and a dummy block are written back to leaf 0. The left node of level 1 is included by $\mathcal{P}(0)$ and $\mathcal{P}(1)$. The block B and a dummy block are written there. The remaining block C is written to the root with a dummy block. Note that some blocks might remain in the stash after this step, though all blocks are written back to the ORAM tree in this example.

With these steps, a request from the processor (that misses in the stash) becomes a sequence of a read from and a write to a random path. Therefore, the access pattern derived from Path ORAM is a sequence of accesses to random paths, which is completely independent of the original access pattern from the processor.

III. Redundancy of Path ORAM

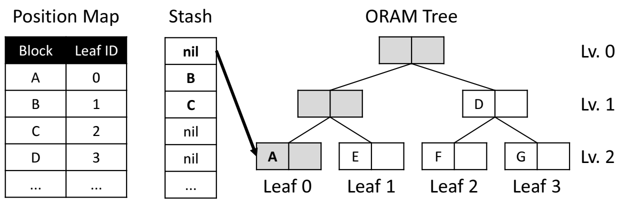
A. *Example of Redundant Access*

Suppose a request to the block E comes after the example shown in Fig. 1 (c). Since the block E is located in the leaf 1, all blocks along $\mathcal{P}(1)$ is read from external memory. The point is that the blocks read from the root and the left node of level 1 were written back and removed from the stash in the previous ORAM access. If they remained valid on the chip, the access to these nodes would be omitted. If the block B, instead of the block E, were requested under the same condition, the whole access to the ORAM tree would be omitted because the block B was read to the stash in the previous ORAM access.

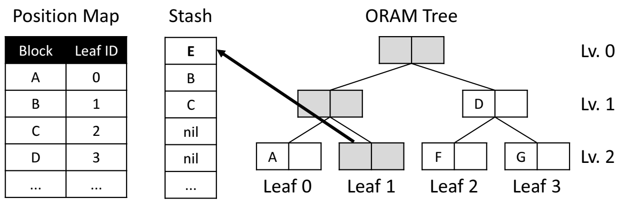
An observation from this example is twofold: (1) the paths of two successive ORAM accesses overlap to some extent, and (2) the preceding path-write-back and the subsequent path-read of the overlapped part are redundant. The sequence of the accessed paths has no information about the original access pattern. The redundant parts of ORAM access can be safely removed as far as the way of removal is also independent of the original access pattern.

B. *Path Merging*

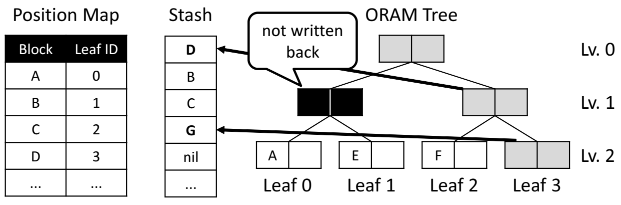
Fork Path ORAM [15] was proposed by Zhang *et al.* to remove the redundant memory accesses in Path ORAM. The



(a) Writing non-overlapped part of $\mathcal{P}(0)$.



(b) Reading non-overlapped part of $\mathcal{P}(1)$.



(c) Problem after a hit of the block B in the stash.

Fig. 2. Working examples of path merging in Fork Path ORAM.

basis of Fork Path ORAM is that, if the path that will be accessed in the next ORAM access is known before the path-write-back step, the overlapped part between the current and the next paths can be determined.

Figure 2 shows an example of Fork Path ORAM, where a path-read from $\mathcal{P}(0)$ has been completed and the next ORAM request to the block E has arrived. Since the block E belongs to $\mathcal{P}(1)$, the non-overlapped parts between the two paths are the leaf 0 and the leaf 1. In the path-write-back, only leaf 0 is written back (Fig. 2 (a)). Similarly, only leaf 1 is read from the ORAM tree in the next path-read (Fig. 2 (b)).

If the next requested block is not known, a dummy request is inserted. In the original Path ORAM, it only has to be known before completing the path-write-back. This difference causes extra dummy requests in Fork Path ORAM, which may affect the performance. However, even though a dummy request has been inserted, there is a chance to replace it silently with an incoming real request. At the time a real request comes, it can be safely replaced if the series of write-back access that has been issued is the same as the sequence that would have been issued when it had been inserted from the beginning. This supplementary technique is called dummy label replacing [15].

To increase the overlap of paths, Fork Path ORAM also adapts a reordering of ORAM requests. It has a request window with a fixed number of ORAM requests. The request that has the nearest path to the currently accessed path in the window is selected as the next request. If the number of real requests is smaller than the window size, dummy requests are inserted to fill the window.

Although Fork Path ORAM completely removes the access to the overlapped parts of paths, it has two major weaknesses. The first one arises from property of the dummy requests. An application with a small number of memory request is more sensitive to the access latency and easily affected by the dummy requests. Its performance will be much worse by adapting Fork Path ORAM due to extra dummy requests. The dummy label replacing technique solves the problem to some extent; however, it must be carefully considered that it is worth the cost of making the ORAM controller much more complicated.

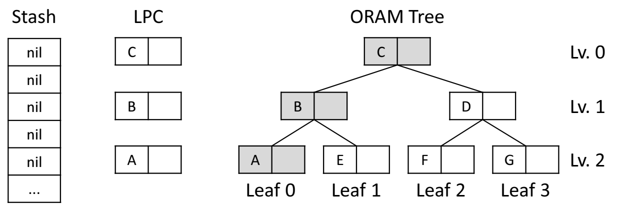
The second weakness, which is more critical, security-related problem, is the loss of the determinacy of the derived access pattern. Suppose the blocks B and D are requested in order after Fig. 1 (b). The write back step is the same as the previous example shown in Fig. 2 (a). Since the block B is found in the stash, the controller then skips the path access for the block B and starts to search for the block D. The only overlapped node between the previous path ($\mathcal{P}(0)$) and the block D's path ($\mathcal{P}(3)$) is the root. The non-overlapped part, the left node of level 1 and the leaf 3, are read to the stash (Fig. 2 (c)). The problem is that the left node of level 1, shown in black, has not been written back. It comes from skipping the path access for the block B. If only the block D had been requested after Fig 1 (b), the black node would have been written back! It leaks the information that there was a hit in the stash on the way, which may be a hint for the original access pattern from the processor. Excluding such ORAM accesses from the path merging may be possible by checking if the next (or incoming) request will hit in the stash, though it will catastrophically increase the complexity of the controller.

IV. Last Path Caching

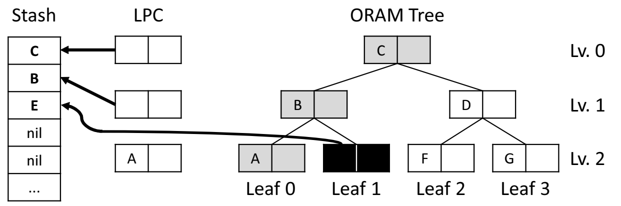
In this paper, we propose last path caching as an alternative technique to remove the redundant memory accesses, which has a simpler procedure than Fork Path ORAM. Figure 3 (a) illustrates its principles of operations. An additional on-chip cache, Last Path Cache (LPC), is placed between the ORAM tree and the stash. It stores the previously written path from the stash. Altough it can be write-through or write-back, it is shown as a write-through cache in Fig. 3 (a). The characteristics of the last path caching vary according to the type of cache. In the following subsections, we introduce three schemes: WT, WB, and WB/WT Hybrid.

A. WT Scheme

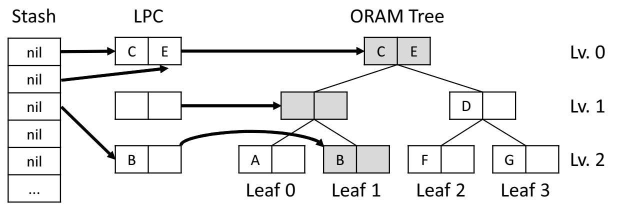
Figure 3 shows the organization and a working example of the WT (*write-through*) scheme. There, all levels of the LPC is set to write-through. It means that all the blocks in the LPC have been also written to the ORAM tree and they are present in the tree. In the example, if the block E is requested (Fig. 3 (b)), the overlapped part (including B and C) is supplied from the LPC, while the non-overlapped part (including E) is read from the ORAM tree. The non-overlapped part in the LPC (including A) is simply ignored. On the write back, the



(a) Last Path Cache between the ORAM tree and the stash.

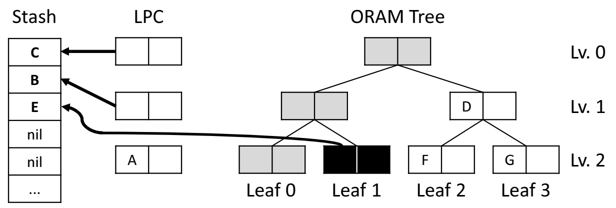


(b) Reading $\mathcal{P}(1)$ for the block E.

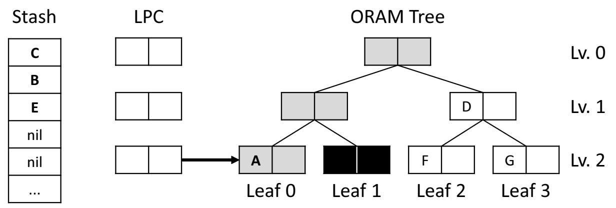


(c) Writing $\mathcal{P}(1)$ back, along with the LPC.

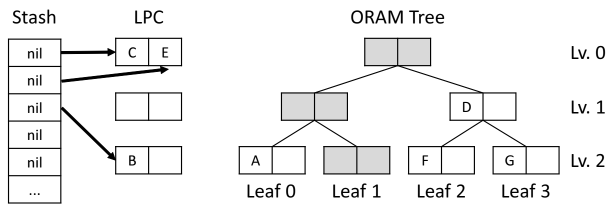
Fig. 3. Organization and behavior of last path caching (*write-through*).



(a) Reading $\mathcal{P}(1)$ for the block E.



(b) Writing unread LPC blocks to the ORAM tree.



(c) Writing stash blocks to the LPC.

Fig. 4. Behavior of the last path caching (*write-back*).

whole path are written to both the ORAM tree and the LPC, as shown in Fig. 3 (c).

An advantage of the WT scheme is that no physical caches are actually required. The blocks written to the LPC are those that have been invalidated in the stash. Since their data have not been overwritten in the stash at this point, they can be simply reactivated if needed. It requires a table indexed by a level that memorizes which blocks in the stash have been written to the corresponding level, along with the reactivation logic. They are expected to much smaller than the actual cache.

A weakness of the WT scheme is that the redundant write accesses are not removed. It means the reduction of the memory accesses becomes only a half of that achieved by Fork Path ORAM, when ignoring the effect of the extra dummy requests.

B. WB Scheme

Figure 3 describes the WB (*write-back*) scheme, where all levels of the LPC is set to write-back. All blocks in the LPC are dirty: they do not exist in the ORAM tree and they must be written to the tree before invalidation. The read step (Fig. 4 (a)) works in a similar way as the WT scheme. The blocks that were read from the LPC are invalidated, while those that were not read (i.e. the non-overlapped part in the LPC) remain valid. Before the write-back step, all valid blocks in the LPC are written back to the ORAM tree (Fig. 4 (b)). Blocks in the stash are then written back to the LPC, rather than the ORAM tree (Fig. 4 (c)). From the viewpoint of the derived access pattern with the WB scheme, it corresponds to a postponement of the path-write-back until the next ORAM access.

An advantage of the WB scheme is that the sequence of memory accesses becomes deterministic. The write-back step of Fork Path ORAM [15] requires the overlapped region between the current and the next path. Delaying the write back with the LPC mitigates the requirements to the overlap between the current and the previous path. This removes the uncertainty due to the next path.

A weakness of the WB scheme is that the time between reading a block in the ORAM tree and writing back to the same block gets longer. DRAM is usually used as external memory, which leverages locality of access with a row buffer. Writing blocks immediately after reading increases the chance for the blocks to hit in the row buffer. The delay of the write back decreases the possibility of the row buffer hit, which degrades the effective bandwidth of the memory.

C. WB/WT Hybrid Scheme

Here, we review the weak points of the above-mentioned schemes. The weakness of the WT scheme gets more serious near the root: such nodes are more likely to be included in the overlapped region. The weakness of the WB scheme becomes significant near the leaf: since the path-write-back begins from the leaf, the relative increase of the delay gets larger in nodes close to the leaf.

The last scheme, called WB/WT hybrid scheme, combines them to complement each other. To be more precise, with a threshold t, a part of the LPC from the root to the level $t-1$

TABLE I
TRACES FOR EVALUATION, WHICH ARE SORTED BY MPKI (MISS PER KILO INSTRUCTION) OF THE LAST LEVEL CACHE.

Name	MPKI	Name	MPKI
face (facesim)	1.22	comm3	6.41
comm5	1.51	leslie (leslie3d)	7.75
black (blackscholes)	1.79	comm1	9.09
freq (freqmine)	2.69	ferret	13.00
comm4	3.74	comm2	14.99
fluid (fluidanimate)	4.09	libq (libquantum)	19.16
stream (streamcluster)	4.90	mummer	24.23
swapt (swaptions)	5.80	tigr	32.40

TABLE II
SYSTEM PARAMETERS.

Processor Cores	
Core Type	4-way, out-of-order
# of Cores	4
Core Frequency	3.2 GHz
Last-level Cache	512 kB/core
DRAM and Memory Controller	
# of DRAM Channels	4
DRAM Frequency	800 MHz
Peak Throughput	51.2 GB/s
ORAM Controller	
Data Block Size	64 B
Height of Tree (L)	23
# of Slots per bucket (Z)	4
# of Levels for Treetop Cache	3
ORAM Hit Latency	40 ns
AES Circuit Latency	25 ns
AES Circuit Throughput	204.8 GB/s

is set to write-back and the rest (i.e. from the level t to the leaf) is set to write-through. A physical cache is required only for the write-back levels. Data in the stash can be reused for the write-through levels as we had explained in Section IV.A.

V. EVALUATION

A. Methodology

In this section, the performance of the proposed schemes is evaluated with a trace-based simulation environment. Sixteen traces from single-thread applications of the Memory Scheduling Championship [12] are used, Applications, listed in Table I, are composed of PARSEC, SPEC CINT2006, BioBench, and commercial (*comm*) workloads. Requests in the traces have been filtered in advance by a last level cache with 512 kiB of capacity and 64-byte data blocks [1]. They are simulated with a modified version of a DRAM simulator usimm [1] where an ORAM controller is added between the processor and the DRAM controller. Main system parameters are summarized in Table II. All four cores executes the same program. The first 400 million instructions are fast-forwarded to warm ORAM up. The following 100 million instructions are used for the measurement. The performance index is the sum of the number of cycles to execute them. The number of ORAM accesses, the number of DRAM accesses, and the miss rate in row buffers of DRAM on write are also measured for reference.

The following five ORAM variants are evaluated:

- the original Path ORAM [10] (*Normal*) as the baseline of performance,
- the WB scheme (*WB*),
- the WT scheme (*WT*),
- the WB/WT hybrid scheme (*WB/WT*), and
- Fork Path ORAM [15] (*Fork*) for the comparison.

In the WB/WT hybrid, the threshold is set to 8 unless otherwise mentioned: the LPC blocks from the root to the level 7 are set to write-back. We also evaluate a case of adapting a reordering of ORAM requests [15]. The size of the window is set to 4 or 8 requests, which is denoted as Q4 or Q8, respectively. In order to prevent starvation, the movement to find the next path is limited to one-way: the next request is selected from those with higher leaf ID than the current one. If no such requests are found, a request with the smallest leaf ID is selected.

TABLE III
RESULT OF EVALUATION WITHOUT REORDERING.

Criterion	WB	WT	WB/WT	Fork
# of Cycles	+0.5%	-2.2%	-3.9%	-4.0%
# of DRAM Accesses	-8.3%	-4.2%	-8.3%	-8.3%
DRAM Row Buffer Miss Rate on Write	+89.9%	0.0%	+7.3%	+7.3%

B. Evaluation Result without Reordering

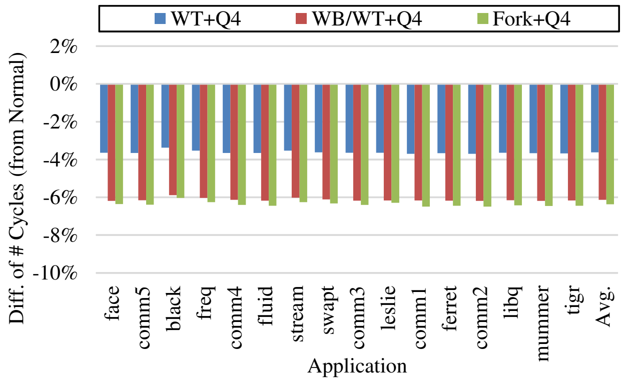
Table III summarizes the evaluation results in the case the reordering of requests is not applied. The results are shown by the relative difference from *Normal*. The average values of all the traces are shown because they show almost the same result. The WB scheme decreased the performance by 0.5%. While the number of DRAM accesses was reduced, the miss rate in row buffers was almost doubled, which widely increased the time to process an ORAM request. In the WT scheme, the performance gain was 2.2%, almost half of that in the WB/WT hybrid or Fork Path. The reduction in DRAM accesses was also halved. This result confirms the discussion about the weakness of the WT scheme in Section IV. The result of the WB/WT hybrid almost matched that of Fork Path. The increased row buffer miss came from the omission of access to levels near the root. Such blocks are also likely to hit in row buffers because they fit in a single row buffer per DRAM channel.

C. Evaluation Result with Reordering

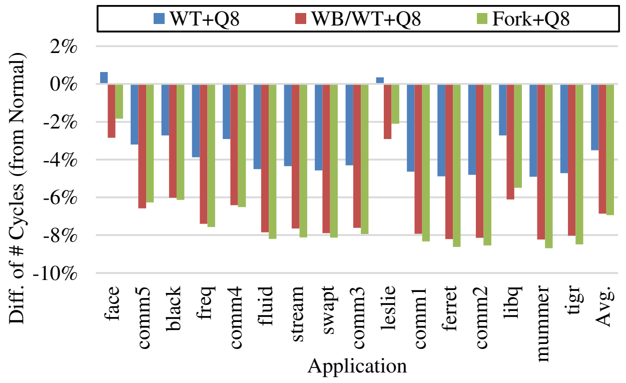
Figure 5 shows the relative execution time where the reordering of requests is applied. Graphs (a) and (b) correspond the cases where the size of the window is 4 and 8, respectively. The X-axis is the shortened name of trace, while the Y-axis is the difference of the relative number of executed cycles. Avg. stands for the average of all the traces. The result of the WB scheme is omitted in this evaluation.

When the window size was 4 (Fig. 5 (a)), Almost the same result among the applications was observed, in similar to Table III. The average performance loss of *WB/WT+Q4* over *Fork+Q4* was only 0.2 percentage points.

On the other hand, when the window size increased to 8 (Fig. 5 (b)), the difference of the relative performance



(a)



(b)

Fig. 5. Result of performance evaluation with reordering window size of 4 (Q4) and 8 (Q8).

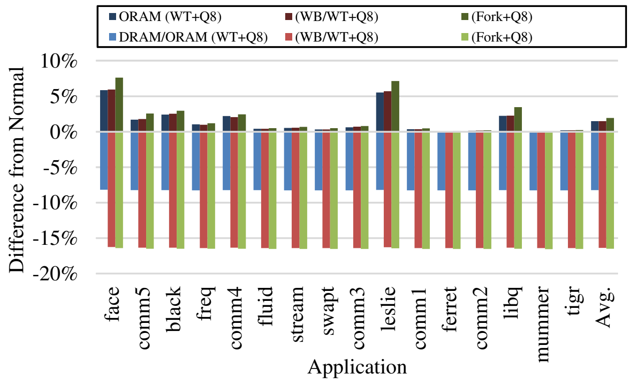


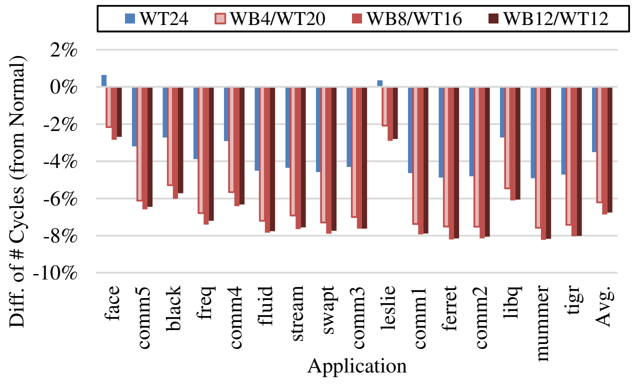
Fig. 6. Breakdown of the difference of the number of DRAM access.

among the applications was shown. The performance gain got smaller in the applications that had fewer cache misses, with exceptions of *leslie* and *libq*. The advantage of the WB/WT hybrid scheme over Fork Path was observed in such applications. The average increase of performance was almost the same between them: *WB/WT+Q8* was 6.86% and *Fork+Q8* was 6.94%.

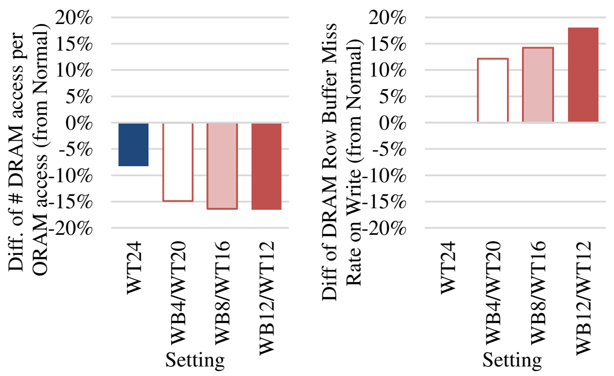
Detailed analysis on the difference of the number of DRAM accesses was made in Fig. 6. Dark bars correspond to the difference (increase) of the number of ORAM accesses, while light bars mean the difference (decrease) of the number of DRAM accesses per ORAM access. The product of two numbers makes the number of DRAM accesses.

The increase of ORAM accesses came from dummy accesses due to the shortage of requests from the processor. The number of dummy requests became large when there were few cache misses (i.e. an application on the left, *leslie*, or *libq* was running). Dummy requests with Fork Path increased by 0.4 points on average over the proposed schemes. The increased dummy requests came from earlier deadline to decide the next path, which had been pointed out in Section III.B.

The number of DRAM accesses per ORAM access corresponded to the number of the redundant memory accesses actually omitted. Its difference between *WB/WT* and *Fork* was only 0.1 points on average. When the paths of two successive requests went too close, their overlapped region was partially stored in the write-through levels of the LPC where write to external memory could not be removed. However, the result implied that it was rare case.



(a) The difference of the number of cycles.



(b) The difference of indices in DRAM access.

Fig. 7. The effect of the threshold in WB/WT Hybrid Scheme.

D. Effect of Threshold in Hybrid Scheme

Figure 7 depicts the result of a sensitivity study to the threshold of the WB/WT hybrid scheme. Similarly to Fig. 5, Graph (a) shows the difference of the number of executed cycles for each application. The name of setting is expressed as *WBn/WTm*, *WBn*, or *WTm*, where n levels of the LPC from the root are set to write-back and the other $m(= 24 - n)$ levels are write-through. Note that *WT24* is identical to the WT scheme. Graph (b) summarizes the difference of the number of DRAM accesses (left) and the difference of the miss rate in row buffers (right). In similar to Table III, the average of each of them is only shown because both of them did not vary with application.

From Fig. 7, the peak of the reduction of execution time was found at *WB8* for the all applications. The reduction of DRAM accesses per ORAM access almost reached at the upper limit at *WB8*, while the row buffer miss steadily rose by increasing the threshold. Thus, the WT and WB schemes were balanced at this point.

VI. Related Work

After the algorithm [10], the first implementation [7], and an exploration of design space [9] of Path ORAM were presented, many optimization techniques and ORAM architectures similar to Path ORAM have been proposed, including Fork Path ORAM [15] explained in Section III.

The PHANTOM implementation [7] applied another caching technique, called treetop caching, to the ORAM tree. It caches all blocks on a few levels of the tree from the root. It is achieved by skipping access to these levels and leaving blocks, which would be moved to the tree, be untouched in the stash. Since nodes near the root are the most frequently accessed, the treetop caching leverages spatial locality of the ORAM tree, while our last path caching utilizes temporal locality. Though the effect of these two caching techniques are partially overlapped, using both of them gives higher performance. According to our evaluation with a similar environment to that shown in Section V.A, a 3-level treetop caching increased the performance by 5.7% on average. The average performance gain of *WB/WT+Q8* was 6.9% (see Section V.C). The number of blocks required for the cache was almost the same between them. The combination of these two caches achieved 7.5% of the performance gain. Detailed analysis on their combination is left as future work.

Ring ORAM [8] is an ORAM architecture based on Path ORAM. In the original Path ORAM, if a node is included in the accessed path, all blocks in the node are read out to the stash. Ring ORAM limits the number of blocks read by an ORAM access to only one per node. As a result, the total number of read blocks becomes unproportional to Z, the number of blocks for each node. However, it may reduce the efficiency in capacity because of an increased number of dummy blocks. Although it was reported that it did not affect in a practical use [8], the detail has not been reported. The idea of separate treatment of the path-read and the path-write-back was also seen in RAW ORAM [3]; its goal was to reduce the cost of encryption, which is not related to our research.

When the position map is too large and the recursive approach is applied as explained in Section II.A, Freecursive ORAM [2] is effective. It introduces a PosMap Lookaside Buffer (PLB), a cache to the position map, which greatly reduces the ORAM accesses caused by looking up the position map. Although the recursive approach is not currently considered in our research, the proposed schemes do not interfere the use of the PLB.

Prefetch is often useful when a running program has high spatial locality. In Path ORAM, blocks can be prefetched by making multiple blocks always mapped to the same path. This technique is called super block [9]. PrORAM [14] extends it to a dynamic approach: it merges adjacent blocks into a super block or unmerges super blocks on the fly. It showed higher performance than the static approach by unmerging useless super blocks. These approaches are also orthogonal to the proposed schemes.

VII. Conclusion

This paper proposed a technique to remove the redundant memory accesses in Path ORAM, which was simpler than existing Fork Path ORAM and was determinate in the access pattern derived by ORAM. Our evaluation showed that the performance loss from the existing method was no more than 0.2%. The advantage of the reduced dummy accesses was also observed when ORAM requests were not frequently arrived.

Our future work includes a detailed analysis on the applicability of the proposed technique, especially with a larger cache to the ORAM tree. We are also planning to implement the corresponding controller on FPGAs to verify the simplicity of the last path caching.

Acknowledgement

This study was partially supported by JSPS Grants-in-Aid for Scientific Research (KAKENHI), Grant Number 26870278 and 16K00072.

References

[1] 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," University of Utah, Tech. Rep. UUCS-12-002, 2012.

[2] 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 *Proc. ASPLOS '15*, 2015, pp. 103–116.

[3] 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 *Proc. FCCM '15*, 2015, pp. 215–222.

[4] O. Goldreich, "Towards a Theory of Software Protection and Simulation by Oblivious RAMs," in *Proc. STOC '87*, 1987, pp. 182–194.

[5] M. Henson and S. Taylor, "Memory Encryption: A Survey of Existing Techniques," *ACM Comput. Surv.*, vol. 46, no. 4, pp. 53:1–53:26, 2014.

[6] M. S. Islam, M. Kuzu, and M. Kantacioglu, "Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation," in *Proc. NDSS '12*, 2012.

[7] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, and D. Song, "PHANTOM: Practical Oblivious Computation in a Secure Processor," in *Proc. CCS '13*, 2013, pp. 311–324.

[8] L. Ren, C. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. van Dijk, and S. Devadas, "Constants Count: Practical Improvements to Oblivious RAM," in *Proc. USENIX Security '15*, 2015, pp. 415–430.

[9] 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 *Proc. ISCA '13*, 2013, pp. 571–582.

[10] 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 *Proc. CCS '13*, 2013, pp. 299–310.

[11] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "AEGIS: architecture for tamper-evident and tamper-resistant processing," in *Proc. ICS '03*, 2003, pp. 160–171.

[12] The Journal of Instruction Level Parallerism. 3rd JILP Workshop on Computer Architecture Competitions (JWAC-3): Memory Scheduling Championship (MSC). [Online]. Available: http://www.cs.utah.edu/~rajeev/jwac12/

[13] D. L. C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," in *Proc. ASPLOS-IX*, 2000, pp. 168–177.

[14] X. Yu, S. K. Haider, L. Ren, C. Fletcher, A. Kwon, M. van Dijk, and S. Devadas, "PrORAM: Dynamic Prefetcher for Oblivious RAM," in *Proc. ISCA '15*, 2015, pp. 616–628.

[15] 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 *Proc. MICRO '15*, 2015, pp. 102–114.