2017 IEEE International Symposium on High Performance Computer Architecture

Cooperative Path-ORAM for Effective Memory Bandwidth Sharing in Server Settings

Rujia Wang† Youtao Zhang§ Jun Yang†
† Electrical and Computer Engineering Department § Computer Science Department
University of Pittsburgh
{ruw16,youtao,juy9}@pitt.edu

***Abstract*—Path ORAM (Oblivious RAM) is a recently proposed ORAM protocol for preventing information leakage from memory access sequences. It receives wide adoption due to its simplicity, practical efficiency and asymptotic efficiency. However, Path ORAM has extremely large memory bandwidth demand, leading to severe memory competition in server settings, e.g., a server may service one application that uses Path ORAM and one or multiple applications that do not. While Path ORAM synchronously and intensively uses all memory channels, the non-secure applications often exhibit low access intensity and large channel level imbalance. Traditional memory scheduling schemes lead to wasted memory bandwidth to the system and large performance degradation to both types of applications.**

In this paper, we propose CP-ORAM, a Cooperative Path ORAM design, to effectively schedule the memory requests from both types of applications. CP-ORAM consists of three schemes: *P-Path*,*R-Path*, and *W-Path*. *P-Path* assigns and enforces scheduling priority for effective memory bandwidth sharing. *R-Path* maximizes bandwidth utilization by proactively scheduling read operations from the next Path ORAM access. *W-Path* mitigates contention on busy memory channels with write redirection. We evaluate CP-ORAM and compare it to the state-of-the-art. Our results show that CP-ORAM helps to achieve 20% performance improvement on average over the baseline Path ORAM for the secure application in a four-channel server setting.

I. Introduction

With the fast adoption of cloud computing paradigm, it becomes increasingly important to prevent information leakage from programs running on untrusted cloud servers. Secure processor designs, e.g., XOM [12] and TPM [11], can encrypt and secure the program code, the user data and its execution flow. However, sensitive information may still be extracted through memory access sequences [32], [19]. Studies showed that completely stopping information leakage from memory access patterns requires ORAM (Oblivious RAM) [9], [10], a cryptographic primitive that often incurs large performance overhead. The recent advance on ORAM proposed Path ORAM [23], a simple and practical ORAM protocol that has greatly improved asymptotic efficiency.

Cloud service providers often consolidate multiple applications on one physical server to reduce power and energy consumption, and to maximize system resource utilization. This is also preferred when executing a secure application that adopts Path ORAM — Path ORAM converts each memory access from the secure application to tens to hundreds of memory accesses, which would leave most system resources idle if the secure application monopolizes the server. Therefore, it is natural to consolidate one secure application with one or multiple non-secure applications on one physical server. Unfortunately, due to the extreme memory access intensity in Path ORAM, it is challenging to effectively schedule memory requests from both types of applications. In particular, Path ORAM synchronously distributes its memory accesses across all memory channels, while non-secure applications have much lower access intensity and memory requests exhibit significant imbalance at the channel level. Adopting traditional memory scheduling schemes often wastes large memory bandwidth and introduces large performance degradation to both types of application.

In this paper, we propose CP-ORAM, a Cooperative Path ORAM design, to address the above issues. We summarize our contributions as follows.

- We study the co-run interference between secure and non-secure applications and analyze the root causes of the ineffectiveness in adopting traditional memory scheduling. To our knowledge, this is the first paper that focuses on memory scheduling for Path ORAM in server settings.
- We propose CP-ORAM that consists of three cooperative scheduling schemes for effective memory bandwidth sharing. *P-Path* is designed to assign and enforce scheduling priority during the co-run. *R-Path* maximizes channel utilization by proactively scheduling read operations from the following Path ORAM access. *W-Path* mitigates contention on busy channels by writes redirection.
- We evaluate CP-ORAM and compare it to the state-of-the-art. Our experimental results show that CP-ORAM achieves an average of 20% performance improvement over the baseline Path ORAM for the secure application in a server setting with four channels.

In the rest of the paper, we present the ORAM background in Section II. We motivate the design and elaborate the three schemes in Section III. We discuss the experimental methodology and the results in Section IV and V, respectively. Additional related work is discussed in Section VI. We conclude the paper in Section VII.

2378-203X/17 $31.00 © 2017 IEEE
DOI 10.1109/HPCA.2017.9

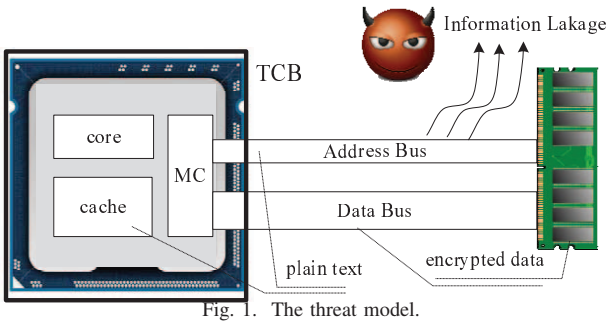


Fig. 1. The threat model.

II. Background

In this section, we discuss the threat model, briefly review the ORAM and Path ORAM algorithms, and then present the server setting of our baseline.

A. Threat Model

The threat model adopted in this paper follows that in previous work [8], [19], [31]. With the processor being the only hardware component within the trusted computing base (TCB) [11], an adversary can access the data stored in main memory and the data communicated on address and data buses. Such access includes potential physical access to the hardware and may be enhanced by specially designed devices and tools, e.g., bus traffic analyzer [26].

To ensure security, the data in the main memory are encrypted with architectural assisted security enhancements [24], [29], which not only ensure data secrecy and integrity but also minimize performance overhead to the system. However, as shown in Figure 1, accessing user data needs to have plain text physical addresses sent to the memory modules, making encryption alone designs insufficient. For example, by tampering with the memory address bus, an adversary can extract important information from the observed access patterns, e.g., the files being accessed in a cloud file system [28], [22], the disease/specialist information being looked up in a medical application [4], and the queries being executed on a database [2]. Even when both code and data are unknown to the adversary, previous work has demonstrated a control flow graph (CFG) fingerprinting technique to identify known pieces of code solely based on the address trace [32].

The security focus in this paper is on preventing information leakage from address access patterns.

B. Oblivious RAM

Oblivious RAM (ORAM) [9], [10] is a cryptographic primitive for preventing information leakage from memory access sequences. ORAM conceals the access pattern from an application by continuously shuffling and re-encrypting the memory data after each access. An adversary, while still being able to observe all the memory addresses transmitted on the bus, has negligible probability to extract the real access pattern.

Path ORAM [23] was recently proposed as a practical ORAM implementation. In Path ORAM, the unsafe memory is structured as a balanced binary tree, where each node is referred to as a bucket that can hold `Z` blocks (a block is

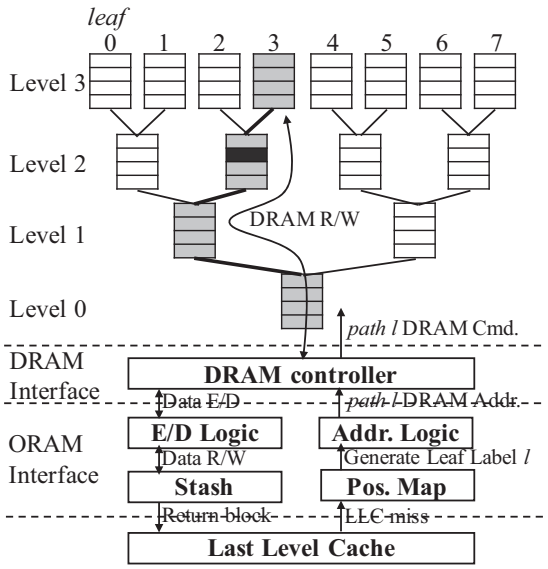


Fig. 2. The Path ORAM scheme.

often of the size of a cacheline, i.e., 64B). The tree has `L+1` levels — the root of tree is at level `0` while the leaves are at level `L`. The total number of data blocks that an ORAM tree can hold is $N=Z*(2^{L+1}-1)$. For the example in Figure 2, we have `L=3`, `Z=4`, and `N=60`.

The blocks in the Path ORAM tree can be either real data blocks or dummy blocks. The dummy blocks are introduced as space filler, which may be replaced with real data blocks if needed. An adversary cannot differentiate dummy blocks from real ones as encryption hides the contents of the blocks.

The ORAM interface for Path ORAM consists of a stash, a position map, the address logic and encryption/decryption logic. The stash is a small buffer that stores up to `C` data blocks from the ORAM tree. The position map is a lookup table that maps program addresses to data blocks in the tree. A Path ORAM tree has 2^L paths from the root to different leaves. Given an LLC (last level cache) miss, the program address is first sent to the position map to determine on which path the requested block is stored. Assuming the block is on path `l`, the address logic determines the actual DRAM physical address using a static mapping table. Then the physical address is sent to DRAM controller and translated to DRAM device commands, such as `PRE`, `ACT`, `RD`, and `WR`, to perform actual memory operations.

Accessing a memory data block starts with the search for the block in the stash. A stash hit terminates the search and returns the block while a stash miss results in a Path ORAM access to the unsafe memory. Assuming that the block is on path `l`, each ORAM access consists of two phases — read phase and write phase.

- In the read phase, all the data blocks on path `l` are read and decrypted, and stored in the stash. For an LLC read miss, the requested block is then returned. For an LLC write miss, the requested block gets updated in the stash. After the access, Path ORAM maps the requested block

to a new path `l'` while all other fetched blocks are still associated with their original paths.

- In the write phase, data blocks are encrypted with new keys and written to the buckets along path `l`. These buckets are greedily filled in with blocks from the stash. Path ORAM uses the order from the leaf to the root with blocks pushed as many as possible down to the leaf.
  The write phase may write back a block from another path if (i) this block is currently in the stash; (ii) the target bucket is on the overlapped portion of the two paths; and (iii) there are free space in the target bucket.

Path ORAM changes the block-to-path mapping after each access such that the memory accesses from user application, even if being very regular, are randomized, which effectively prevents information leakage. Path ORAM further eliminates access temporal pattern by issuing accesses at fixed rate, even if there are no actual accesses [15], [7].

At the architectural level, several schemes have been proposed to improve Path ORAM performance. Ren et al.[19] optimize block mapping using sub-tree layout, which maximizes row buffer hit for ORAM accesses. In addition, they save the top of the Path ORAM tree, i.e., the most frequently used blocks, in a small on-chip cache to improve performance. Zhang et al. [31] eliminate unnecessary memory accesses if consecutive path accesses have overlap.

C. Server Setting

With fast technology scaling, modern computer servers usually have abundant system resources. For example, the server in Figure 3 adopts a chip-multiprocessor that supports the concurrent execution of multiple threads, and four memory channels each of which can transmit data at 12.8GB/s (DDR3-1600). Cloud service providers often consolidate multiple applications on one physical server to reduce power and energy consumption, and to maximize system resource utilization.

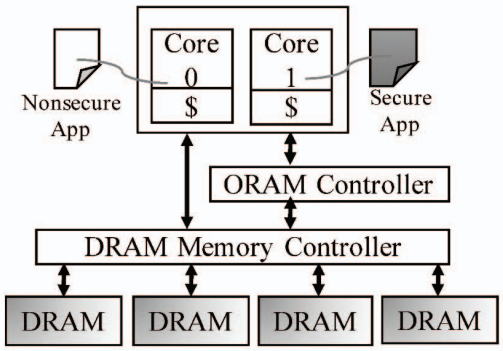


Fig. 3. The baseline server setting.

In this paper, the baseline secure processor uses two cores — one is dedicated to execute secure application that demands data encryption to protect data secrecy and Path ORAM to prevent information leakage from access patterns; the other is to execute one non-secure application without data encryption and Path ORAM. These two types of applications are referred to as **S-App** and **NS-App**, respectively, in this paper. We evaluate the setting with more cores running more than one S-App and NS-App applications in the experimental section. To simplify discussion, we assume that each core has private cache. In the case if a shared last level cache is used, it demands additional security enhancements to prevent potential timing channels and side channels between two applications [27], [6], [14], [13].

The S-App and NS-App applications share the main memory through address partition — for simplicity and without considering the space occupied by the OS, each application may take half of the space of each bank from each channel. There is no physical address overlap between two applications. As a comparison, an alternative design is to share the memory through channel partition — S-App uses a subset of channels while NS-App uses the rest. As we will show in the next section, channel partition is sub-optimal because the channels allocated to NS-App tend to be under-utilized while those allocated to S-App tend to be overloaded.

In this paper, we assume that the ORAM memory space is 4GB and each bucket stores 4 blocks such that the tree has 24 levels (`L=23`). We adopt sub-tree layout to spread ORAM memory accesses across all four memory channels. We also adopt tree top caching that cache top 10 levels in a 256KB cache ($4*64*(2^{10}-1) \approx 256KB$), which eliminates around 42% accesses to the memory. We observe negligible path overlap beyond top 10 levels from consecutive path accesses [31].

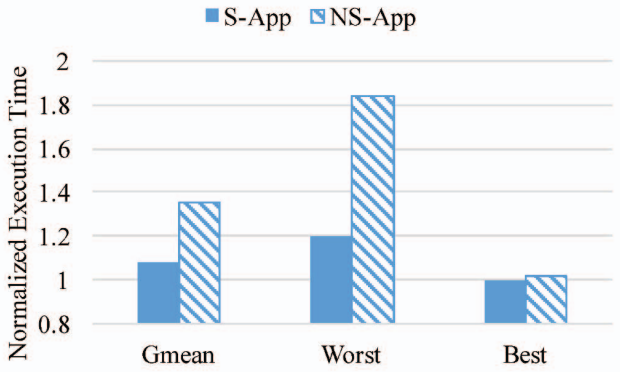


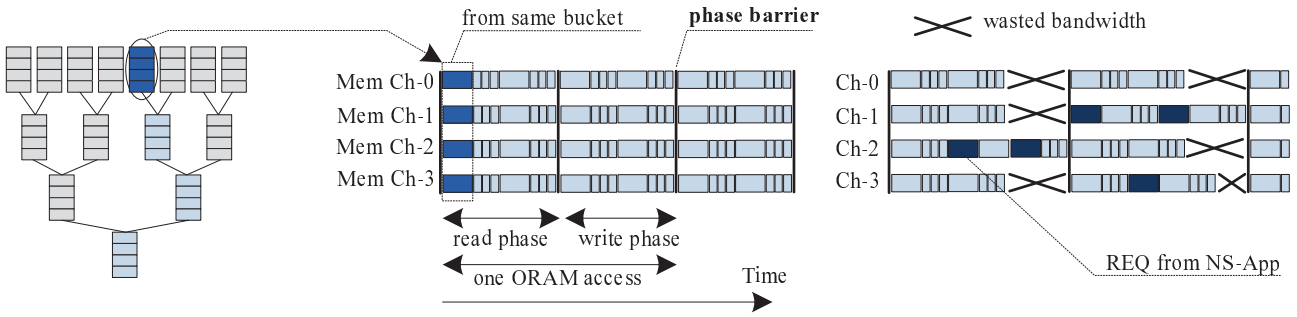
Fig. 4. The S-App and NS-App co-run leads to large performance degradation.

III. The CP-ORAM Design Details

In this section, we first motivate the CP-ORAM design and then develop three schemes to improve memory bandwidth utilization in server settings.

A. Motivation

The Path ORAM is an extremely memory intensive protocol — [19] showed that a secure application that adopts Path ORAM (i.e., S-App) consumes almost all of the system peak memory bandwidth and introduces 10-100× slowdown over the native execution without adopting ORAM. Consequently, the processor shall be left mostly idle if S-App monopolizes the server. Non-secure applications (i.e., NS-Apps), even those



(a) Path ORAM has synchronous progress across channels (b) Requests from NS-App waste memory bandwidth

Fig. 5. S-App has phase barriers and wastes memory bandwidth if out of synchronization.

that are traditionally categorized as memory intensive, have much low memory access intensity. Therefore, it is natural to consolidate S-App and NS-App applications on one server to improve resource utilization and save energy consumption.

We study the performance degradation when two applications, one S-App and one NS-App, co-run on one server. Figure 4 reports the average (`Gmean`), worst (`Worst`), and best (`Best`) results, which are normalized to the solo execution of each application. `Gmean` averages the results of a suite of workloads. The worst (and best) case considers the summed degradation percentages in different co-runs. The system settings are listed in Section IV. From the figure, we have two observations.

- In `Best`, both types of applications have little performance degradation. This indicates that consolidation has the potential to significantly improve resource utilization in server setting.
- In `Worst`, both types of applications suffers from large degradation. NS-App suffers more percentage degradation than S-App does, i.e., 80% vs 20% degradation when comparing to the solo run.
  While S-App has smaller percentage degradation, the memory bandwidth utilization is greatly reduced. Given that S-App often utilizes almost full peak memory bandwidth [19] and its performance largely depends on the memory performance in solo execution, a 20% degradation in `Worst` indicates that the system allocates 10.2GB/s (=20%$\times$ 4$\times$12.8GB/s) memory bandwidth to the co-run NS-App. However, the MPKI (memory accesses per kilo instructions) of the NS-App in `Worst` is 24, i.e., 0.2GB/s bandwidth demand at most, which is much lower than the actually allocated amount.

To exploit the consolidation potential while mitigating the performance degradation, we analyze the co-run in details and identify the root causes of the degradation to each type of the applications.

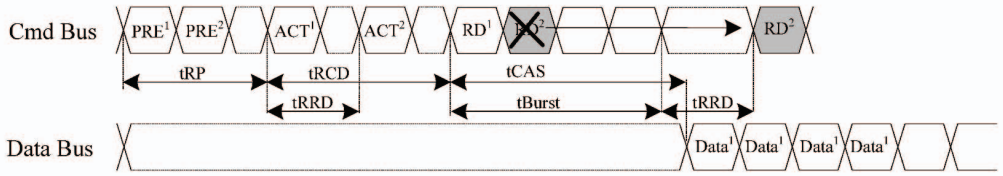
1) Root Cause of S-App Degradation: The S-App, to maximize channel utilization, maps the blocks in each bucket to different channels [19]. This leads to the synchronous progress across different channels, as shown in Figure 5(a). [19] showed that the memory utilization is close to peak bandwidth with synchronized progress. In Path ORAM, each ORAM access includes a read phase and a write phase. For security reasons, the write phase cannot start before the read phase completes. Also, a new ORAM access cannot start before the preceding one completes. Therefore, the end of each phase effectively becomes the synchronization barrier for all channels. This is referred to as **phase barrier** in this paper.

In Figure 5(b), if one channel is slowed down due to scheduling the memory requests from NS-App, other channels need to wait even if they finish early. This leads to large memory bandwidth waste and significantly slows down S-App because the performance of the latter depends mainly on memory performance.

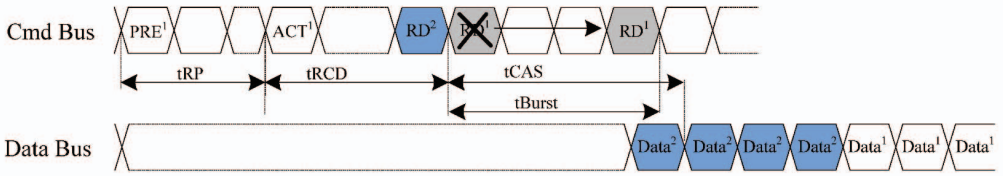
TABLE I
SELECTIVE DRAM INTERNAL TIMING CONSTRAINT(AT 800MHZ)

Timing	Cycles	Description
tRP	11	Row Precharge. The time interval that it takes for a DRAM array to be precharged for another row access.
tRCD	11	Row to Column command Delay. The time interval between row access and data ready at sense amplifiers.
tRRD	5	Row activation to Row activation Delay. The minimum time interval between two row activation commands to the same DRAM device. Limits peak current profile.
tFAW	32	Four (row) bank Activation Window. A rolling time-frame in which a maximum of four bank activations can be engaged.
tCAS	11	Column Access Strobe latency.The time interval between column access command and the start of data return by the device.
tBurst	4	Data burst duration. The time period that data burst occupies on the data bus.

2) Root Cause of NS-App Degradation: As discussed, the memory access intensity of NS-App is much lower than that of S-App. Given that the performance degradation of NS-App comes mainly from memory bandwidth competition, we next study the root cause by studying the memory scheduling details. Path ORAM adopts open page policy for better performance [19] — servicing a read request normally needs a `PRE` command to close the current row in the target bank, a



(a) 1 cycle delay of PRE^2 results in (tRRD+tBurst-1) cycle delay of RD^2



(b) A row buffer hit from S-App (i.e., RD^2) defeats NS-App priority

Fig. 6. The co-run timing examples of two applications (CMD^i is the command for Ri (i=1 or 2)).

ACT command to activate the row to be accessed, and a RD command to fetch the requested data. The baseline adopts FR-FCFS (first-ready first-come-first-serve) [17], [21], [20] with the goal to maximize memory throughput. Table I lists a subset of timing constraints and their default values in the discussion.

Figure 6 presents the memory bus timing for a co-run scenario with two applications. In the example, we assume two read requests R1 and R2 are from different types of applications, arrive at the memory controller at the same time, and demand data from different banks.

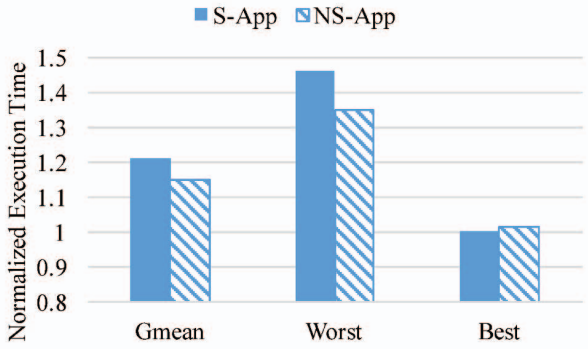


Fig. 7. The co-run interference when NS-App has high scheduling priority.

Since there is no constraint restricting precharge commands, the memory controller sends out PRE^1 and PRE^2 consecutively. PRE^1 is sent first because it has higher priority (which will be elaborated later in this section). However, due to timing constraints in JEDEC standard, ACT^1 needs to be tRP after PRE^1 while RD^1 needs to be tRCD after ACT^1. The interference enlarges due to additional constraints, e.g., ACT^2 has to be tRRD after ACT^1 and no more four ACT can be issued within tFAW time window; RD^2 needs to at least tBurst after RD^1 to avoid conflict on data bus. In summary, a later scheduled request tends to suffer from larger delay.

Since the baseline does not assign priority to either application, S-App and NS-App compete for the memory bandwidth based on their request arrival time. However, S-App translates each user memory access to tens of physical memory accesses, which gain more opportunities to grab the system resources once the latter become idle. An NS-App request, when arriving at the memory controller, often gets delayed because the requested resources, e.g., the memory channel, are busy.

To mitigate the extremely biased request arrival in S-App and NS-App co-run, we have to assign high priority to NS-App to ensure that it gains sufficient opportunities to have its requests timely processed.

Scheduling NS-App with high priority. We re-do the experiment for Figure 4 with the scheduling priority assigned to NS-App. The results are summarized in Figure 7. From the figure, NS-App still suffers from 15% performance degradation on average. In addition, S-App shows much bigger degradation, rising to 21% on average and 46% in the worst case.

We analyze the memory scheduling details in the new experiment. We identify that it is the high row buffer hits in Path ORAM[19] that defeat simple priority allocation during scheduling.

Figure 6(b) illustrates the command sequence when servicing an NS-App request. Assuming R1 is from NS-App and is a row buffer miss, the memory controller sends out PRE, ACT, and RD to service the request. In the example, a request R2 (from S-App) arrives after R1. However, R2, when it is a row hit, which is frequent for S-App, can send out its RD^2 command before RD^1, without PRE and ACT, and leads to extra delay to NS-App. In this case, NS-App still suffers from large performance degradation.

From above discussion, it is clear that while it has great potential to consolidate S-App and NS-App on one physical server, simple scheduling schemes, either with or without priority, tend to introduce large performance degradation and memory bandwidth waste.

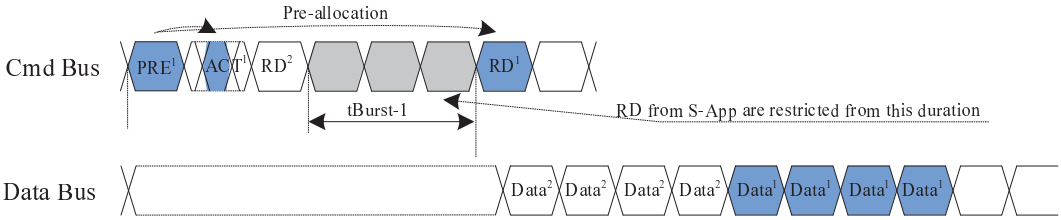


Fig. 8. P-Path pre-allocates bus slots to enforce priority allocation for NS-App.

B. P-Path: Enforce Scheduling Priority through Resource Pre-Allocation

To address the large performance loss in the co-run, we propose CP-ORAM that consists of three cooperative memory scheduling schemes — P-Path, R-Path, and W-Path. We elaborate P-Path in this section and R-Path and W-Path in following sections.

P-Path is designed to assign scheduling priority to NS-App and enforce this priority through resource pre-allocation. The simple allocation in Section III-A, while prioritizing `PRE`, cannot prevent `ACT` and `RD` commands from being delayed by S-App requests. The P-Path scheme addresses this issue by resource pre-allocation. That is, when the memory controller sends out `PRE` for an NS-App read request, it also allocates address, data, and command bus slots for its coming `ACT` and `RD` commands. S-App requests, even if they are row buffer hits, cannot be scheduled if otherwise leading to resource conflict. Figure 8 illustrates that an S-App `RD` for a row buffer hit cannot be scheduled within (tBurst-1) cycles from the pre-allocated `RD` slot.

While pre-allocation helps to enforce priority allocation for NS-App, it may degrade S-App performance significantly. Therefore, as shown in Algorithm 1, P-Path proportionally pre-allocates channel and bus resources based on a given threshold `th` (percentage value) and pre-allocates for every `th×10` out of 10 row buffer miss requests from NS-App. The threshold `th` is statically determined in this paper. We leave it as our future work to develop dynamic threshold adjustment for better trade-off among resource utilization and performance improvement. The algorithm gives NS-App higher scheduling priority, i.e., if an S-App request/device command and an NS-App request/device command are ready at the same time, the priority is always given to NS-App. The algorithm is enabled only for row buffer misses from NS-App, the requests with row buffer hits do not need resource pre-allocation.

Algorithm 1: P-Path Scheduling Algorithm

Input: PreAllocation Threshold `th` (percentage value)
Output: Issue proper command to DRAM
Parameter: `cycle`: program cycle;
`cnt`: row buffer misses from NS-App

```
1  while not end of program do
2      if can issue memory commands at cycle then
3          check channel command queue;
4          if has commands from NS-App then
5              issue the command;
6              if the command is PRE, i.e., being a row
                buffer miss then
7                  cnt = cnt++ % 10 ;
8                  if cnt < th×10 then
9                      pre-allocate address, command, and
                        data bus slots for ACT and RD/WR;
10                 end
11             end
12         else
13             issue S-App command;
14         end
15     else
16         continue;
17     end
18     cycle++;
19 end
```

C. R-Path: Maximize Memory Bandwidth Utilization using Next Read

From the analysis in Section III-A, the performance degradation of S-App comes mainly from the phase barriers. That is, the progress of multiple memory channels may lose synchronization due to co-run interference; if the slowest channel has not reached its phase barrier, other faster channels have to wait, leading to significant waste of memory bandwidth. In this section, we propose R-Path to maximize memory bandwidth utilization using the read operations from next ORAM access.

R-Path is designed to schedule memory operations across phase barrier to improve bandwidth utilization. Figure 9 elaborates what operations may be promoted without tampering with the correctness as well as the security of Path ORAM. In the example, we assume two consecutive Path ORAM accesses need to access blocks `b1` and `b2` on paths `l1` and `l2`, respectively. Along path `l1`, there might exist a block `b3` from path `l3`. Paths `l1` and `l3` have significant overlap while paths `l1` and `l2` only overlap at a level close to the root. For simplicity, we assume `b1` and `b2` are mapped to paths `lx` and `ly`, respectively, after the accesses and these two paths have no overlap with `l1` and `l3` other than the root bucket (so that they are ignored from discussion). Since Path ORAM tries to push blocks as deep as possible in the write phase, `b3` may replace `b1`'s place. Given that we do not know if `b3` exists until we read all blocks from `l1`, the write phase is tightly data dependent on the read phase. Therefore, we cannot schedule

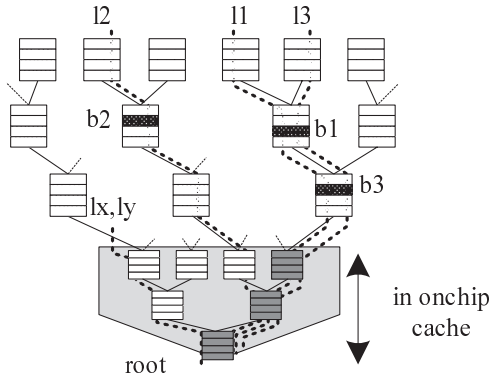


Fig. 9. R-Path safely promotes reads from the next ORAM access.

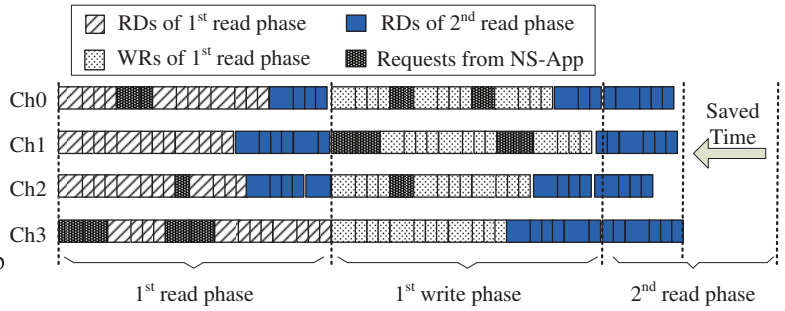


Fig. 10. R-Path improves memory bandwidth utilization and speeds up program execution.

operations from the write phase to the read phase in the same ORAM access.

Interestingly, the blocks read during the read phase of the second ORAM access, i.e., reading path `l2`, do not have data dependence. In this example, path `l1` and `l2` overlap at a level that is close to the root bucket. Since the top 10 levels of the tree are cached, accessing the overlapped buckets does not lead to memory accesses. Thus, it is safe to schedule read operations from the second ORAM access to improve the bandwidth utilization. This effectively prefetches data blocks from the read phase of the next ORAM access.

To ensure the correctness of Path ORAM, R-Path works as follows.

- R-Path only prefetches data blocks from the read phase of its immediate next ORAM access and stores the prefetched blocks in a small read buffer. The prefetched blocks cannot be sent to stash directly as, otherwise, some blocks may be selected for write back to path `l1` in write phase. The latter is infeasible when there is no prefetch. In this paper, each ORAM path stores 14 (=24-10) levels of the tree in memory. Therefore, the prefetch buffer needs to hold at most 56 blocks. The prefetched data are copied to the stash after Path ORAM has determined what to write for the write phase of the current access.
- R-Path creates a dummy access if there is no ORAM access in the queue, similar to that in [15], [7].
  If the 2nd ORAM access overlaps with the 1st one in memory, i.e., at a level $\geq$ 10, we enable the fork path optimization [31] and disable R-Path. Our experimental results show that the probability of the former is very low due to its low probability after large tree top caching.

As shown in Figure 10, R-Path promotes read operations from the 2nd ORAM read to the 1st read or write phases, which improves the memory bandwidth utilization of the 1st read and write phases, and shortens the read length of the 2nd ORAM access.

D. W-Path: Mitigating Write Traffic on Busy Channels

R-Path shortens the length of read phase by promoting some of its operations to preceding phases. It cannot reduce the length of write phase because write operations cannot be promoted across the read/write phase barrier in one ORAM access. In this section, we elaborate the W-Path design to mitigate the co-run interference in the write phase of Path ORAM access.

Intuitively, W-Path schedules write operations across phase barriers by deferring them to future write phases. In this way, write operations can also be moved across the phase barriers, but not the read/write phase barrier in the same access. Figure 11 illustrates how it works.

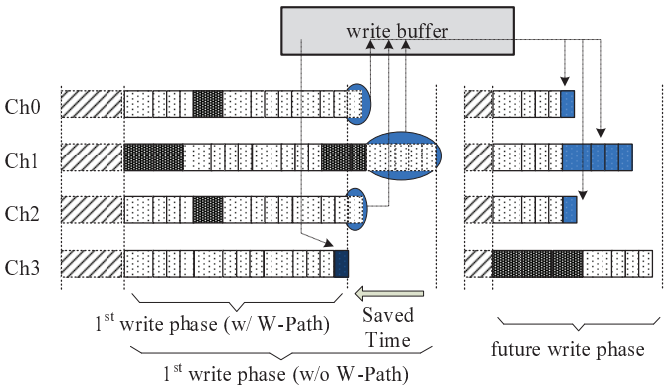


Fig. 11. W-Path uses a write buffer to schedule write operations across phase barriers.

- W-Path is triggered when one or multiple memory channels finish their write operations for the current ORAM access. W-Path searches for the data blocks in the write buffer that need to be written to the fast channels and starts to expunge them. Their slots in the write buffer are then identified as empty entries.
- Given that one ORAM access writes back 14 blocks to each channel. We integrate a 56-block write buffer. The write buffer reserves 8 entries for each channel and let all channels compete for the rest of the entries.
- W-Path aggressively uses the write buffer to shorten the length of the write phase. If a channel has data blocks in its ready queue, and the write buffer has empty reserved entries for this channel, W-Path moves one block from the corresponding channel queue to the write buffer. If

the reserved entries in the write buffer for the channel are full, W-Path moves blocks only if doing so helps to reduce the length of the write phase.
- The write buffer is looked up during the read phase such that a hit in the write buffer gets the data from the write buffer directly.

As a comparison to R-Path that can move read operations across phase barriers to the read and write phases of the preceding ORAM access, W-Path temporarily buffers the write operations in the write buffer such that they are defer to future ORAM accesses.

E. Architectural Enhancements

Figure 12 presents an overview of CP-ORAM design. The shaded boxes indicates the components that are either added or enhanced. We enhance the DRAM controller to enable fine-grained priority enforcement. The read buffer (for R-Path) and write buffer (for W-Path) are added into the ORAM controller. In this paper, each buffer can store up to 56 blocks, i.e., 4KB each. In the read phase, for the current ORAM access, the write buffer is looked up with matched blocks sent to stash. At the end of write phase, i.e., the beginning of the next read phase, the blocks in read buffer are sent to stash. Both operations are in parallel with the accessing of remaining blocks (of the current ORAM access) from the memory. In the write phase, the write buffer is visited when a block need to be swapped out. Since the write buffer is very small (14 entries per channel), finding a block to be written out is relatively fast, and imposes negligible accessing overhead.

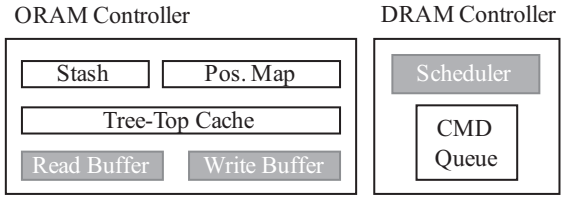


Fig. 12. Architectural enhancements for CP-ORAM.

The secure processor can switch between secure mode and non-secure mode. The former demands encryption and Path ORAM while the latter has no security enforcement. In this paper, we assume there is no address space overlap between S-App and NS-App, and one bit is communicated to the memory controller to differentiate S-App memory accesses from NS-App ones.

F. Security Analysis

ORAM prevents information leakage with address randomization – it assigns a new physical address to each program address accessed. Our cooperative designs do not modify this strategy and thus do not compromise the security guarantee of Path ORAM.

The first scheme, P-Path, modifies the scheduling priority based on a simple counter. The resource pre-allocation policy only affects the completion time but dose not reveal additional pattern information. The second scheme, R-Path, prefetches data blocks from the following ORAM access. R-Path is enabled only if the paths of two accesses do not overlap at a level in memory. Therefore, the data blocks from the 2nd path need to be read anyway. The exception is that the second ORAM access becomes ready after the read phase the preceding one — in this case, a dummy access shall be inserted, which delays the second ORAM access by one ORAM access length. As discussed in [31], this does not degrade security. Due to large top cache we use in the paper, we did not observe the merging opportunity as in [31]. The W-Path does not compromise security guarantee either. W-Path alters the timing of the write operations, but not the contents. The encrypted data blocks are eventually written back to the memory, the same as the baseline.

In addition, our proposed schemes do not increase stash overflow probability. The read buffer and write buffer ensure the stash remains the same as the baseline.

IV. The Experimental Methodology

To evaluate the effectiveness of CP-ORAM, we used USIMM, a cycle accurate memory system simulator [3], to simulate the proposed schemes and compare them to the state-of-the-art. Table II summarizes the baseline server configuration on which we co-ran two applications — one S-App and one NS-App. The DRAM memory follows JEDEC DDR3-1600 specification. We adopted the default values in the specification that are strictly enforced in USIMM.

In the baseline setting, each application has its own 4GB memory space. The address mapping follows the order of "`row:bank:column:rank:channel:offset`" such that Path ORAM maximizes its row buffer hit and both applications fully utilize all channels, as shown in [19].

TABLE II
Baseline System Configuration

Parameter	Value
Processor	Dual-core, 3.2GHz
Processor ROB size	128
Processor retire width	4
Processor fetch width	4
Last Level Cache	512 KB per core
Memory bus speed	800MHz
Memory channels	4
Ranks per channel	1
Banks per rank	8
Rows per bank	16384
Columns (cache lines)/row	128
Memory Space	4GB + 4GB

We chose a set of memory intensive benchmarks from PARSEC suite, commercial, SPEC and BioBench, as they were used in MSC [1]. Each benchmark is simulated for 5 billion instructions, and 500 million representative instructions were selected with a methodology similar to Simpoint [1]. We constructed the workloads for evaluation as follows: each workload consists of one S-App and one NS-App that are of the same program: S-App version adopts encryption and Path ORAM protection while NS-App version does not. Their visible physical address sequences are completely different. We also show other combinations in the detailed result analysis. Table III describes the benchmark programs. The MPKI

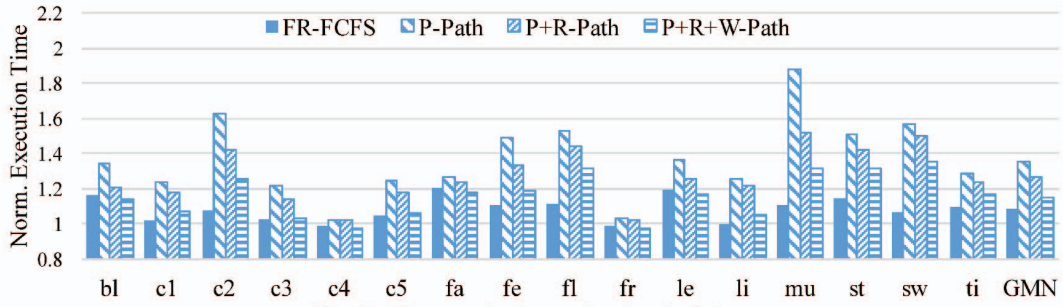


Fig. 13. The normalized execution time for S-App.

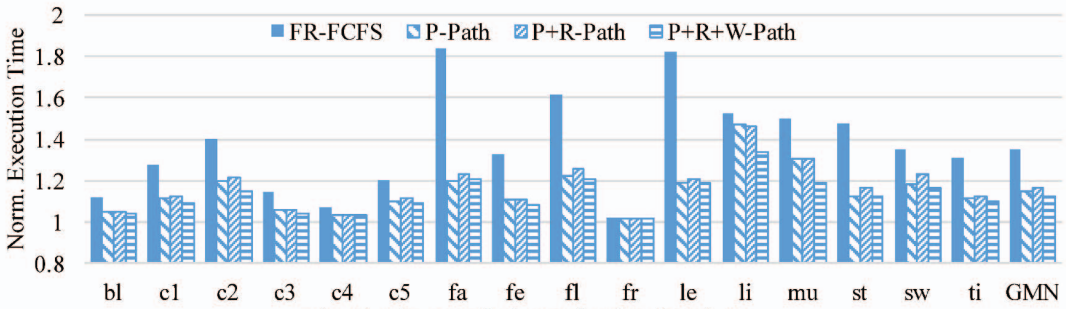


Fig. 14. The normalized execution time for NS-App.

(memory access per kilo instructions) is listed in parenthesis. We use the first two letters of each program to indicate the workload.

TABLE III
SIMULATED BENCHMARK PROGRAMS

Suite	Workloads
PARSEC	black (4.2), face (26.8), ferret (8.0), fluid (17.5), freq (4.5), stream (12.9), swapt (10.9)
COMM.	comm1 (7.3), comm2 (12.6), comm3 (4.2), comm4 (3.7), comm5 (4.5)
SPEC	leslie (23.1), libq (12.0)
BIOBENCH	mummer (24.0), tigr (6.7)

V. THE RESULTS

In the experiments, we evaluated the following schemes:

- `Baseline`. This is the baseline for comparison purpose. We collected S-App and NS-App performance, respectively, in their solo execution mode.
- `FR-FCFS`. This is the co-run case that adopts FR-FCFS memory scheduling algorithm. Each workload consists of S-App version and NS-App version of the same program.
- `P-Path`. This is the co-run case that adopts P-Path scheduling enhancement. The threshold `th` used in P-Path is set to 50% by default.
- `P+R-Path`. This the co-run case that adopts P-Path and R-Path scheduling enhancements.
- `P+R+W-Path`. It adopts all three cooperative Path-ORAM scheduling enhancement, i.e., this is CP-ORAM design that we elaborated in the paper.

A. Performance Analysis

We first compared the effectiveness of different CP-ORAM schemes with the results summarized in Figure 13 and Figure 14 for S-App and NS-App, respectively. The results are normalized to the solo execution `Baseline`. From the figure, we found that FR-FCFS leads to large performance degradation to both S-App and NS-App — on average, 8.3% and 35.5%, as shown in the motivation section. S-App and NS-App may suffer up to 20% and 84% degradation, respectively.

`P-Path`, while reducing the degradation from 35.5% to 15% for NS-App, significantly increases the degradation for S-App, i.e., from 8.3% to 35.2% on average. Our R-Path and W-Path schemes target at improving S-App performance. For S-App, `P+R-Path` reduces the performance loss to 26.3% on average while `P+R+W-Path` further reduces it to 15.4% on average. That is, CP-ORAM achieves around 20% performance improvement over the FR-FCFS scheduling for S-App.

In general, the R-Path and W-Path schemes have little impact on NS-App. When R-Path promotes read operations from the next ORAM access, the memory bandwidth utilization is improved, which slightly hurts NS-App. We observed around 1% extra degradation. When W-Path defers write operations to future ORAM accesses and thus move to write buffers, the channel is less busier, which helps NS-App. We observed an average of 2.4% reduction. In summary, CP-ORAM reduces the the performance degradation from 15% to 12.6% for NS-App on average.

B. P-Path Pre-Allocation Threshold

By default, `P-Path` pre-allocates resources for half of NS-App row buffer misses, i.e., `th`=50%. We studied the performance impact with different threshold values and summarized the results in Figure 15. The x-axis is the pre-allocation threshold. In general, larger threshold values results in larger

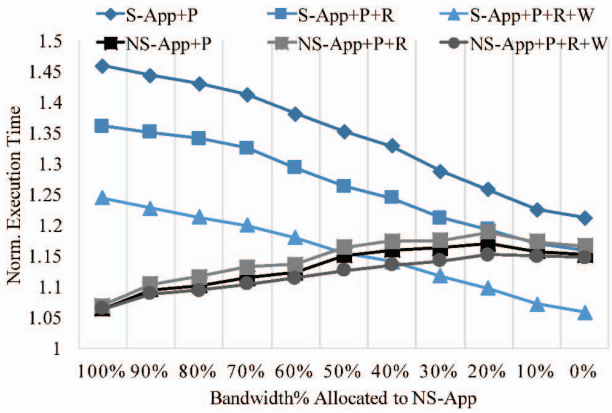


Fig. 15. The effectiveness of CP-ORAM with different threshold values.

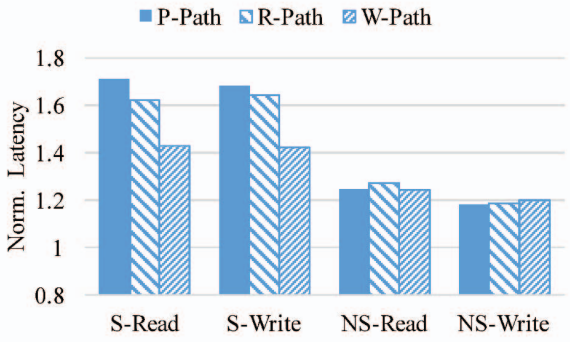


Fig. 16. Comparing memory access latency under different schemes.

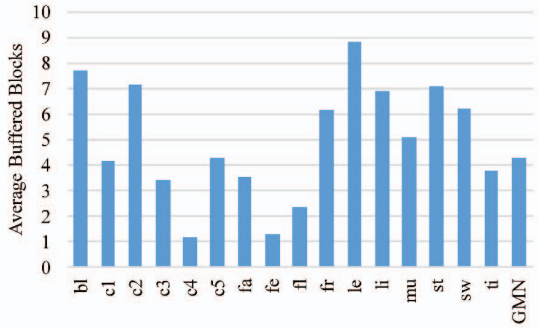


Fig. 17. The average number of blocks per channel in read buffer.

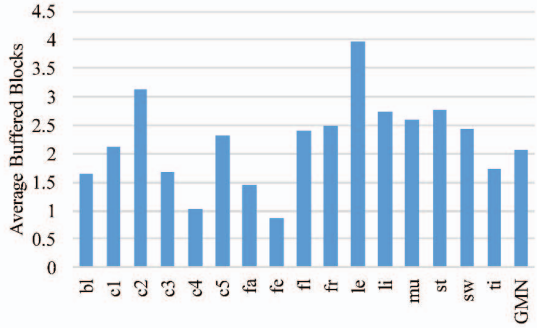


Fig. 18. The average number of blocks per channel in write buffer.

performance degradation to S-App and smaller degradation to NS-App.

Also from the figure, R-Path and W-Path improve S-App performance significantly, but has little impact on NS-App. This is because improved bandwidth utilization benefits S-App the most. The maximal gain (sum of percentage improvements from both S-App and NS-App) occurs at `th`=80%, in which the workload achieves 21.6% sum in total.

C. Memory Access Latency

Figure 16 reports the average read and write latencies for S-App and NS-App under different schemes. All results are normalized to the solo-run. From the figure, R-Path shows larger reduction on read latency than that on write latency. This is because R-Path prefetches read operations and thus shortens the read phase in general. For S-App, W-Path shows more reduction on both read and write latencies over R-Path. This is because W-Path shortens the current write phase such that pending requests are serviced early.

For NS-App, both R-App and W-App have little impact — the difference across three schemes is less than 3% for read latency and 1% for write latency.

D. Buffer Usage

We then analyzed the effectiveness of the read buffer and the write buffer. Figure 17 reports the average number of blocks prefetched per channel. From the figure, R-Path prefetches more than 4 blocks per channel on average, with the maximum being around 9 for `leslie`. `comm4` and `ferret` prefetch around 1 block per channel per access, leading to negligible performance improvement in Figure 13. We found that the more imbalanced channel use the NS-App brings to the system, the more opportunities R-Path has to prefetch next read operations.

For W-Path, Figure 18 reports the average number of blocks in write buffer per channel. We observed the similar trend — with more blocks deferred, W-Path gains better improvements.

E. Sensitivity to Core Number

At last, we studied the effectiveness of CP-ORAM design with more than two cores. We compared two settings using four cores: one co-runs one S-App and three NS-App applications; the other co-runs three S-App and one NS-App applications, referred to as 1S3NS and 3S1NS, respectively. Figure 19 and 20 report the geometric mean of normalized performance results from 16 workloads. P-Path adopts `th`=50%. 1S1NS refer to default setting that was used in previous experiments.

From the figure, we observed larger performance degradation when there are more applications. Given that one S-App uses almost all memory bandwidth, 3S1NS has extreme intensity and introduces close to 3× performance degradation to S-App. On the other hand, 1S3NS has more bandwidth demand

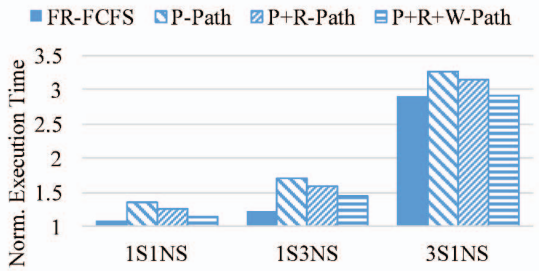


Fig. 19. Comparing S-App performance with more co-running applications.

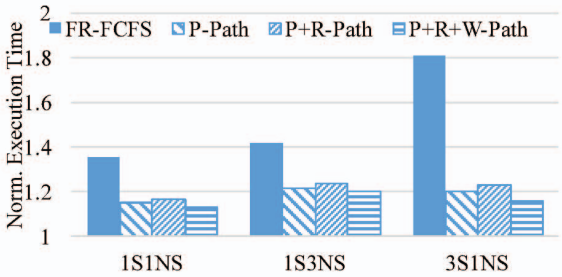


Fig. 20. Comparing NS-App performance with more co-running applications.

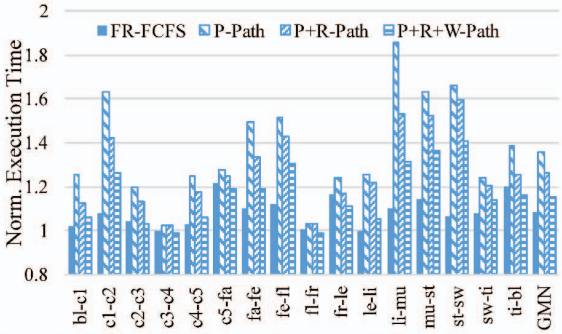


Fig. 21. Comparing S-App performance with mixed co-running applications.

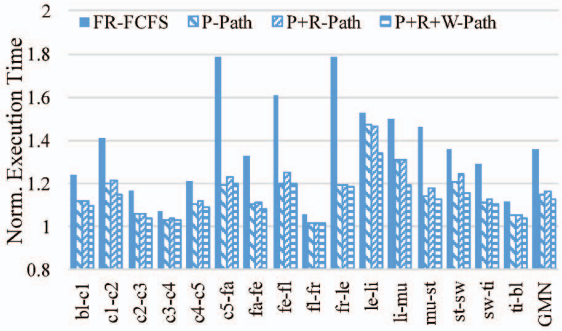


Fig. 22. Comparing NS-App performance with mixed co-running applications.

than 1S1NS but much less than 3S1NS — we observed slightly larger degradation than 1S1NS but much less than 3S1NS.

In all cases, our cooperative scheduling schemes are robust. `P-Path` reduces the NS-App degradation to around 20% degradation while `P+R+W-Path` reduces more than 20% slowdown over `P-Path` for S-App.

F. More Co-run Examples

In this section, we present more co-run examples results. The first study is to co-run different S-App and NS-App. Figure 21 and 22 shows the normalized execution of this co-run scenario. For x-axis label, the first application is secure and the second one is non-secure. We still set the threshold as 50% in this case. We found similar performance compared to the main results in SectionV-A. Overall, `P+R+W-Path` reduce S-App 19.8% execution time compared to `R-Path` only, and is only a slight increase of 7.1% over FR-FCFS. The overall technique also reduce NS-App execution time of 22.9% compared to FR-FCFS.

We also studied co-run effect of multiple copies of application on the same memory channel configuration. Figure 23 shows the normalized execution time across all workloads when there are multiple copies of S-App and NS-App running. It can be observed that co-run 2 and 4 S-App will cause 1.78x and 3.46x more execution time, while co-run 2 and 4 NS-App will only cause 5.6% and 13.8% more time.

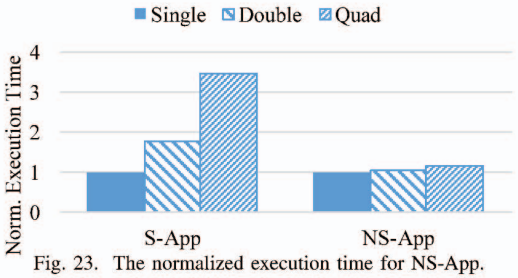


Fig. 23. The normalized execution time for NS-App.

VI. Related Work

Secure processors. The XOM processor is one of the first secure processor designs [12]. XOM-based processors focus on protecting data secrecy [24], [29], [25], [16]. Later studies defended potential information leakage from address and command buses [32]. Studies showed that completely defending information leakage demands ORAM [9]. Path ORAM [23] is a simple and practical ORAM protocol that received wide adoption. The first hardware implementation of Path ORAM was Phantom [15] based on FPGA.

Memory security. When multiple threads co-run on chip-multiprocessors (CMPs), information may be leaked through timing channels [27], [6] and side channels [14], [13]. Wang *et al.* [27] designed a queuing structure per security domain and allocated timing slots to different domains to eliminate timing channels. Ferraiuolo *et al.* [6] improved memory scheduling performance by matching the system security demands. Recent studies on side channel attacks focus on the last level cache (LLC). Liu *et al.* [14] showed that an adversary can attack cross-core, cross-VM side channel and leak keys as well as secret data accesses. Liu *et al.* [13] defended LLC side channel attacks using a performance optimization Cache Allocation Technology (CAT) that was recently introduced in commodity

Intel processors.

Memory scheduling. Several memory scheduling algorithms have been proposed to achieve fair sharing of the memory bandwidth. Mutlu *et al.* [17] proposed to achieve fair schedule between streaming and random access applications in DRAM system. Mutlu *et al.* [18] improved fairness in memory scheduling using batching. Craeynest *et al.* [5] proposed equal-time scheduling and equal-progress scheduling to adjust the amount of resource that each thread receives. However, these schemes did not consider the extreme biased co-run of S-App and NS-App, and cannot address the new challenges in this paper.

VII. Conclusion

In this paper, we proposed CP-ORAM, a cooperative Path ORAM design to address the severe interference when secure and non-secure applications co-run in server settings. We analyzed the memory scheduling in details from which we identified the root causes of the large performance degradation to S-App and NS-App applications. We then proposed three schemes, P-Path, R-Path, and W-Path, to improve memory bandwidth sharing. Our experimental results showed that, CP-ORAM achieves 20% performance improvement on average over the baseline with the 4-channel settings.

Acknowledgment

We thank all the anonymous reviewers for their valuable comments and suggestions. This research is supported in part by US NSF CCF-1422331,CCF-1535755,CCF-1617071.

References

[1] 2012 memory scheduling championship (msc). http://www.cs.utah.edu/~rajeev/jwac12/. Accessed: 2016-07-01.

[2] A. Arasu and R. Kaushik, "Oblivious query processing", In *arXiv preprint arXiv:1312.4012*, 2013.

[3] 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", In *University of Utah, TR*, 2012.

[4] S. Chen, R. Wang, X. Wang, and K. Zhang, "Side-channel leaks in web applications: A reality today, a challenge tomorrow", In *IEEE Symposium on Security and Privacy*, 2010.

[5] K. V. Craeynest, S. Akram, W. Heirman, A. Jaleel, and L. Eeckhout, "Fairness-aware scheduling on single-ISA heterogeneous multi-cores", In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2013.

[6] A. Ferraiuolo, Y. Wang, D. Zhang, A. C. Myers, and G. E. Suh, "Lattice priority scheduling: Low-overhead timing-channel protection for a shared memory controller", In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[7] C. W. Fletcher, M. van Dijk, and S. Devadas, "A secure processor architecture for encrypted computation on untrusted programs", In *ACM workshop on Scalable trusted computing*, 2012.

[8] 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 *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[9] O. Goldreich, "Towards a theory of software protection and simulation by oblivious rams", In *ACM Symposium on Theory of Computing (STOC)*, 1987.

[10] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams", *Journal of the ACM (JACM)*, 1996.

[11] ISO. Iso/iec 11889-1:2009. Technical Report, International Organization for Standardization, 2013.

[12] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software", In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.

[13] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B Lee, "Catalyst: Defeating last-level cache side channel attacks in cloud computing", In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[14] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B Lee, "Last-level cache side-channel attacks are practical", In *IEEE Symposium on Security and Privacy*, 2015.

[15] 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 *ACM SIGSAC Conference on Computer & Communications Security (CCS)*, 2013.

[16] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution", In *International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.

[17] O. Mutlu and T. Moscibroda, "Stall-time fair memory access scheduling for chip multiprocessors", In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2007.

[18] O. Mutlu and T. Moscibroda, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems", In *International Symposium on Computer Architecture (ISCA)*, 2008.

[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 *ACM International Symposium on Computer Architecture (ISCA)*, 2013.

[20] S. Rixner, "Memory controller optimizations for web servers", In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2004.

[21] S. Rixner, W. J Dally, U. J Kapasi, P. Mattson, and J. D Owens. "Memory access scheduling," In *International Symposium on Computer Architecture (ISCA)*, 2000.

[22] E. Stefanov and E. Shi, "Oblivistore: High performance oblivious cloud storage", In *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[23] 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 *ACM SIGSAC Conference on Computer & Communications Security (CCS)*, 2013.

[24] G.E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "Efficient Memory Integrity Verification and Encryption for Secure Processors", In *the 36th International Symposium on Microarchitecture (MICRO)*, 2003.

[25] G. E. Suh, C. W. O'Donnell, I. Sachdev, and S. Devadas, "Design and Implementation of the AEGIS Single-Chip Secure Processor Using Physical Random Functions", In *International Symposium on Computer Architecture (ISCA)*, 2005.

[26] Nexus Technology. Ma5100/4100 series memory analyzer. http://www.nexustechnology.com/.

[27] Y. Wang, A. Ferraiuolo, and G. E. Suh, "Timing channel protection for a shared memory controller", In *2014 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[28] P. Williams, R. Sion, and A. Tomescu, "Privatefs: A parallel oblivious file system", In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[29] J. Yang, Y. Zhang, and L. Gao, "Fast Secure Processor for Inhibiting Software Piracy and Tampering", In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2003.

[30] X. Yu, S. K. Haider, L. Ren, C. Fletcher, A. Kwon, M. v. Dijk, and S. Devadas, "Proram: dynamic prefetcher for oblivious ram", In *International Symposium on Computer Architecture (ISCA)*, 2015.

[31] 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 *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015.

[32] X. Zhuang, T. Zhang, and S. Pande, "Hide: an infrastructure for efficiently protecting information leakage on the address bus", In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2004.