PrORAM: Dynamic Prefetcher for Oblivious RAM

Xiangyao Yu† Syed Kamran Haider‡ Ling Ren† Christopher Fletcher† Albert Kwon†
Marten van Dijk‡ Srinivas Devadas†
† Massachusetts Institute of Technology ‡ University of Connecticut
{yxy, renling, cwfletch, kwonal, devadas}@mit.edu
{syed.haider, vandijk}@engr.uconn.edu

Abstract

Oblivious RAM (ORAM) is an established technique to hide the access pattern to an untrusted storage system. With ORAM, a curious adversary cannot tell what address the user is accessing when observing the bits moving between the user and the storage system. All existing ORAM schemes achieve obliviousness by adding redundancy to the storage system, i.e., each access is turned into multiple random accesses. Such redundancy incurs a large performance overhead.

Although traditional data prefetching techniques successfully hide memory latency in DRAM based systems, it turns out that they do not work well for ORAM because ORAM does not have enough memory bandwidth available for issuing prefetch requests. In this paper, we exploit ORAM locality by taking advantage of the ORAM internal structures. While it might seem apparent that obliviousness and locality are two contradictory concepts, we challenge this intuition by exploiting data locality in ORAM without sacrificing security. In particular, we propose a dynamic ORAM prefetching technique called PrORAM (Dynamic Prefetcher for ORAM) and comprehensively explore its design space. PrORAM detects data locality in programs at runtime, and exploits the locality without leaking any information on the access pattern.

Our simulation results show that with PrORAM, the performance of ORAM can be significantly improved. PrORAM achieves an average performance gain of 20% over the baseline ORAM for memory intensive benchmarks among Splash2 and 5.5% for SPEC06 workloads. The performance gain for YCSB and TPCC in DBMS benchmarks is 23.6% and 5% respectively. On average, PrORAM offers twice the performance gain than that offered by a static super block scheme.

ISCA '15, June 13-17, 2015, Portland, OR, USA

http://dx.doi.org/10.1145/2749469.2750413

1. Introduction

As cloud computing becomes more and more popular, privacy of users' sensitive data is a huge concern in computation outsourcing. One solution to this problem is to use *tamper-resistant hardware* and secure processors. In this setting, a user sends his/her encrypted data to the trusted hardware, inside which the data is decrypted and computed upon. The final results are encrypted and sent back to the user. The trusted hardware is assumed to be tamper-resistant, namely, an adversary is not able to look inside the chip to learn any information. Many such hardware platforms have been proposed, including Intel's TPM+TXT [15], which is based on TPM [1, 26, 34], eXecute Only Memory (XOM) [17, 18, 19] and Aegis [31, 32].

While an adversary cannot access the internal states inside the tamper-resistant hardware, information can still be leaked through main memory accesses. Although all the data stored in the external memory can be encrypted to hide the data values, the memory access pattern (i.e., address sequence) may leak information. Existing attacks ([39]) show that the control flow of a program can be learned by observing the main memory access patterns. This may leak the sensitive private data.

Completely preventing leakage from the memory access pattern requires the use of Oblivious RAM (ORAM). ORAMs were first proposed by Goldreich and Ostrovsky [11], and there has been significant follow-up work that has resulted in more and more efficient cryptographically-secure ORAM schemes [2, 7, 12, 13, 14, 22, 23, 27, 29, 30, 36]. The key idea which makes ORAM secure is to translate a single ORAM read/write into accesses to multiple randomized locations. As a result, the locations touched in each ORAM read/write would have exactly the same distribution and be indistinguishable to an adversary.

The cost of ORAM security is performance. Each ORAM access needs to touch multiple physical locations which incurs one to two orders of magnitude more bandwidth and latency when compared to a normal DRAM. Path ORAM [30], the most efficient and practical ORAM system for secure processors so far, still incurs at least $30\times$ more latency than a normal DRAM for a single access. This results in $2-10\times$ performance slowdown [10, 25].

Traditionally, *data prefetching* [24, 3] has been used to hide long memory access latency. Data prefetching uses the

memory access pattern from history to predict which data block will be accessed in the near future. The predicted block is prefetched from the memory before it is actually requested to hide the access latency.

Although it might seem that prefetching should be very effective with ORAM since ORAM has very high access latency, in reality prefetching does not work on ORAM when the program is memory bound. The main reason is that unlike DRAM, whose bottleneck is mainly memory latency, ORAM's bottleneck is in both latency and bandwidth. Prefetching only works when DRAM has extra bandwidth, therefore does not work well for ORAM (cf. Section 3.1).

In this paper, we enable ORAM prefetching by exploiting locality inside the ORAM itself, which is very different from traditional prefetching techniques. At first glance, exploiting data locality and obfuscation seem contradictory: on one hand, obfuscation requires that all data blocks are mapped to random locations in the storage system. On the other hand, locality requires that certain groups of data blocks can be efficiently accessed together. One might argue that ORAM is inherently poor in terms of locality. We challenge this intuition in this paper by exploiting data locality in ORAM without sacrificing provable security.

We propose a novel ORAM prefetcher called PrORAM (pronounced as *'Pro-RAM'*), which introduces a *dynamic super block* scheme. We demonstrate that it achieves the same level of security as normal Path ORAM, and comprehensively explore its design space. Our dynamic super block scheme detects data locality in programs at *runtime*, and exploits the locality without leaking information on the access pattern.

In particular, the paper makes the following contributions:

1. We study traditional data prefetching techniques in the context of ORAM, and observe that they do not work well for ORAM.
2. A dynamic super block scheme is proposed. The micro-architecture of the scheme is discussed in detail, and the design space is comprehensively explored.
3. Our simulation results show that PrORAM improves Path ORAM performance by 20.2% (upto 42.1%) over the baseline ORAM for memory bound Splash2 benchmarks, 5.5% for SPEC06 benchmarks, and 23.6% and 5% for YCSB and TPCC in DBMS benchmarks respectively. This is more than twice the performance gain offered by an existing static super block scheme.

The rest of the paper is organized as follows: Section 2 provides the necessary background of ORAM in general and Path ORAM in particular. Section 3 presents ORAM prefetch techniques and discusses a previously proposed scheme called static super block. A dynamic super block scheme is introduced in Section 4. The design space is explored, security is shown and hardware complexity is analyzed in detail. Section 5 evaluates different optimizations proposed in the paper. Related work is presented in Section 6 and we conclude the paper in Section 7.

2. Background

2.1. Oblivious RAM

ORAM ([11]) is a data storage primitive which hides the user's access pattern such that an adversary is not able to figure out what data the user is accessing by observing the address transferred from the user to the external untrusted storage (we assume DRAM in this paper). A user accesses a sequence of program addresses $A = (a_1, a_2, \ldots, a_n)$, which will be translated to a sequence of ORAM accesses $S = (s_1, s_2, \ldots, s_m)$, where a_i is the program address of the i^{th} access and the value of a_i should be hidden from the adversary. s_i is the physical address used to access the data storage engine. The value of s_i is exposed to the adversary. Given any two access sequences A_1 and A_2 of the same length, ORAM guarantees that the transformed access sequences S_1 and S_2 are computationally indistinguishable. In other words, the ORAM physical access pattern (S) is independent of the logical access pattern (A). Data stored in ORAMs should be encrypted using probabilistic encryption to conceal the data content and also hide which memory location, if any, is updated. With ORAM, an adversary should not be able to tell (a) whether a given ORAM access is a read or write, (b) which logical address in ORAM is accessed, or (c) what data is read from/written to that location.

In this paper, we focus on Path ORAM [30], which is currently the most efficient ORAM scheme for limited client (processor) storage, and, further, is appealing due to its simplicity.

2.2. Path ORAM

Path ORAM [30] has two main hardware components: the *binary tree storage* and the *ORAM controller* (cf. Figure 1).

Binary tree stores the data content of the ORAM and is implemented on DRAM. Each node in the tree is defined as a *bucket* which holds up to Z data blocks. Buckets with less than Z blocks are filled with *dummy blocks*. To be secure, all blocks (real or dummy) are encrypted and cannot be distinguished. The root of the tree is referred to as level 0, and the leafs as level L. Each leaf node has a unique leaf label s. The path from the root to leaf s is defined as path s. The binary tree can be observed by any adversary and is in this sense not trusted.

ORAM controller is a piece of trusted hardware that controls the tree structure. Besides necessary logic circuits, the ORAM controller contains two main structures, a *position map* and a *stash*. The *position map* is a lookup table that associates the program address of a data block (a) with a path in the ORAM tree (path s). The *stash* is a piece of memory that stores up to a small number of data blocks at a time.

At any time, each data block in Path ORAM is mapped (randomly) to some path s via the position map. Path ORAM maintains the following invariant: *if data block a is currently mapped to path s, then a must be stored either on path s, or in the stash* (see Figure 1). Path ORAM follows the following steps when a request on block a is issued by the processor.

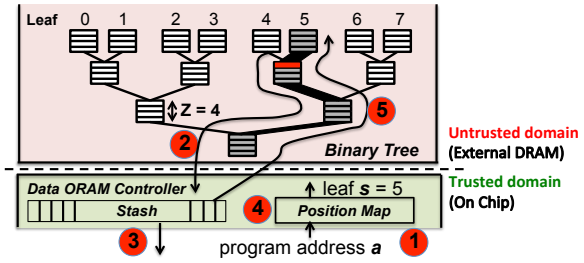

Figure 1: A Path ORAM for $L = 3$ levels. Path $s = 5$ is accessed.

1. Look up the position map with the block's program address a, yielding the corresponding leaf label s.
2. Read all the buckets on path s. Decrypt all blocks within the ORAM controller and add them to the stash if they are real (i.e., not dummy) blocks.
3. Return block a to the secure processor.
4. Assign a new random leaf s' to a (update the position map).
5. Encrypt and evict as many blocks as possible from the stash to path s. Fill any remaining space on path s with encrypted dummy blocks.

Step 4 is the key to Path ORAM's security. This guarantees that a random path will be accessed when block a is accessed later and this path is independent of any previously accessed random paths (*unlinkability*). As a result, each ORAM access is random and unlinkable regardless of the request pattern.

2.3. Recursive Path ORAM

In practice, the position map is usually too large to be stored in the trusted processor. Recursive ORAM has been proposed to solve this problem [27]. In a 2-level recursive Path ORAM, for instance, the original position map is stored in a second ORAM, and the second ORAM's position map is stored in the trusted processor (Figure 1(b)). The above trick can be repeated, i.e., adding more levels of ORAMs to further reduce the final position map size at the expense of increased latency. The recursive ORAM has a similar organization as OS page tables.

Unified ORAM [8] is an improved and state-of-the-art recursion technique. It leverages the fact that each block in a position map ORAM stores the leaf labels for multiple data blocks that are consecutive in the address space. Therefore, Unified ORAM caches position map ORAM blocks to exploit locality (similar to the TLB exploiting locality in page tables). To hide whether a position map access hits or misses in the cache, Unified ORAM stores both data and position map blocks in the same binary tree. In this paper, we use unified ORAM as our baseline ORAM design.

2.4. Background Eviction

In Steps 4 and 5 of the basic Path ORAM operation, the accessed data block is remapped from the old leaf s to a new random leaf s', making it likely to stay in the stash. In practice, this may cause blocks to accumulate in the stash and finally overflow. It has been proven that the stash overflow probability is negligible for $Z \geq 6$ [30]. For smaller Z, *background eviction* [25] has been proposed to prevent stash overflow.

The ORAM controller stops serving real requests and issues background evictions (*dummy accesses*) when the stash is full. A background eviction reads and writes a random path s_r in the binary tree, but does not remap any block. During the writing back phase (Step 5 in Section 2.2) of Path ORAM access, all blocks that are just read in can at least go back to their original places on s_r, so the stash occupancy cannot increase. In addition, the blocks that were originally in the stash are also likely to be written back to the tree (they may share a common bucket with s_r that is not full of blocks). Background eviction is proven secure in [25].

2.5. Timing Channel Protection

The original ORAM definition in [11] does not protect against timing attacks. Timing information includes when an ORAM access happens, the run time of the program, etc. For example, by observing that a burst of memory accesses happen, an adversary may be able to tell that a loop is being executed in the program. By counting the length of the burst, sensitive private information may be leaked.

In practice, periodic ORAM accesses are needed to protect the timing channel [10]. Following prior work, we use O_{int} as the public time interval between two consecutive ORAM accesses. ORAM timing behavior is completely determined by O_{int}. If there is no pending memory request when an ORAM access needs to happen due to periodicity, a dummy access will be issued (the same operation as background eviction). If one is willing to leak a few bits, timing channel protection schemes that allow for dynamically-changing O_{int} may be attractive [9], since they provide better performance. These schemes can be used with the techniques proposed in this paper if small data leakage is allowed.

2.6. Path ORAM Limitation

Clearly, Path ORAM is far less efficient compared to insecure DRAM. Under typical settings for secure processors (gigabytes of memory and 64- to 128-byte blocks), Path ORAM has a 20-30 level binary tree (note that adding one level doubles the capacity). In practice, Z is usually 3 or 4 [29, 25]. This indicates that for each ORAM access, about 60-120 blocks need to be read **and** written, in contrast to a single read **or** write operation in an insecure storage system. Since a single ORAM access saturates the available DRAM bandwidth, it brings no benefits to serve multiple ORAM requests in parallel.

Recursive/unified ORAMs introduce additional overheads of accessing multiple levels of ORAMs. This overhead hurts both performance and energy efficiency. In total, Path ORAM incurs roughly two orders of magnitude more bandwidth and one order of magnitude more latency than DRAM. This leads to up to an order of magnitude slowdown in a secure processor

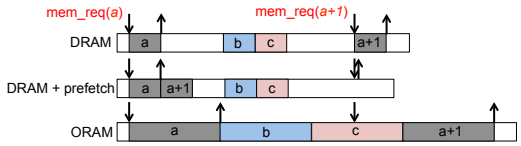

Figure 2: Data prefetching on DRAM and ORAM.

[25]. Although no study has looked into the energy overhead of ORAM, we expect that the hundreds of blocks transferred to and from Path ORAM binary tree will result in proportionally larger energy consumption.

3. ORAM Prefetch: Super Block

As access latency is the main bottleneck in ORAM, a natural solution that comes to mind is to apply latency hiding techniques to ORAM. In this section, data prefetching is studied in the context of Path ORAM. We will show in Section 5.2 that traditional prefetching techniques do not work for ORAM and therefore new techniques are required.

3.1. Traditional Data Prefetch

Figure 2 shows the basic idea of traditional data prefetching. When block a is accessed, if the prefetcher predicts that block $a+1$ will also be accessed in the near future, then block $a+1$ is prefetched from the DRAM. When the real request to $a+1$ arrives, the data is already in the cache and the DRAM does not need to be accessed again. Prefetching moves memory accesses out of the critical path of execution thus leading to overall speedup of the program.

When DRAM is replaced with ORAM, however, the situation changes. The much longer latency (more than 30×) and lower throughput (2 orders of magnitude) of ORAM leads to two effects. First, it is not useful to overlap multiple ORAM accesses, since a single ORAM access already fully utilizes the entire DRAM bandwidth (cf. Section 2.6). Second, for memory bound applications, ORAM requests line up in the ORAM controller and there is no idle time for prefetching. In other words, prefetching is likely to block normal requests and hurt performance.

In this section, a new prefetch technique specifically designed for ORAM is proposed. The technique is called *Super Block*.

3.2. General Idea of Super Block

The notion of *super block*, first proposed in [25], tries to exploit spatial locality in ORAM. In particular, it tries to load more than one block from the path in a single ORAM access. The blocks that are loaded together are called a *super block*. According to Section 2.2, this requires that all the blocks belonging to a super block be mapped to the same path.

In this paper, we only consider super blocks that consist of data blocks adjacent in program address space. Also, we only consider super blocks of size 2^k by merging blocks that differ only in the last k address bits. We define the size of a super

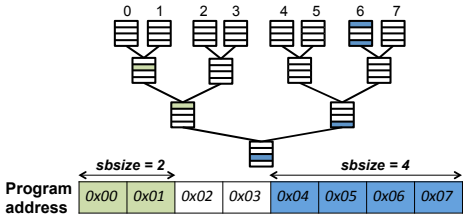

Figure 3: Super block construction. Blocks whose addresses are different only in the last k address bits can be merged into a super block of size $n = 2^k$. All blocks in a super block are mapped to the same path.

block as the number of data blocks in it, denoted as *sbsize*. For example, in Figure 3, block 0x00 and block 0x01 can be merged into a super block of size 2; Blocks from 0x04 to 0x07 can be merged into a super block of size 4. However, block 0x03 and 0x04 cannot be merged because their addresses are not properly aligned.

Whenever one block in a super block is accessed, all the blocks in that super block are loaded from the path and remapped to a same random path. The block of interest is returned to the processor and the other blocks are *prefetched* and put into the LLC (Last Level Cache). The idea is that the prefetched blocks may be accessed in the near future due to spatial data locality, which saves some expensive Path ORAM accesses.

The super block scheme maintains the invariant that blocks in the same super block are mapped to the same path in the binary tree (Figure 3). This guarantees that all the blocks belonging to the same super block can be found during a single ORAM access (they may or may not reside in the same bucket). It is important to note that although a super block is always read out as a unit, the blocks are not required to be written back to the binary tree at the same time. Rather, they can be written back separately and in any order, as long as they are mapped to the same path. This flexibility is useful in designing different super block algorithms.

3.3. Static Super Block

The above description of the super block scheme is very general and leaves many design decisions unspecified, for example, what size should a super block be, when and what blocks should be merged, etc.

Static super block has been proposed in previous work ([25]). In this scheme, every $n = 2^k$ data blocks consecutive in the program address space are merged into super blocks of size n. n is statically specified by the user before the program starts. n can be tuned for different applications or be the same for all applications. In the initialization stage of Path ORAM, blocks are merged into super blocks, each of which is forced to be mapped to the same path. During normal ORAM operations, a super block is accessed as a unit as described in Section 3.2.

3.3.1. Security Similar to the argument of background eviction (Section 2.4), super block schemes are secure as long as a super block access is indistinguishable from a normal ORAM access. Security of normal Path ORAM is maintained since an access to a super block loads and remaps ***all*** blocks in the super block. Thus, subsequent accesses to this super block or to different super blocks will always be touching independently random paths. An adversary is not able to tell the super block size or whether the static super block scheme is used at all.

3.3.2. Limitations Although the static super block scheme provides performance gain for programs with good locality, it has significant limitations which make it not practical:

First, it significantly hurts performance when the program has bad spatial locality (cf. Section 5). With these programs, prefetching always misses and pollutes the cache.

Second, it cannot adjust to different program behaviors in different phases. In practice, this leads to suboptimal performance for certain programs.

Third, it is the responsibility of programmers or the compiler to figure out whether the super block scheme should be used and the size of the super block.

4. Dynamic ORAM Prefetch: PrORAM

In this paper, we propose a dynamic ORAM prefetching scheme called *dynamic super block*, which is the foundation of PrORAM, to address the limitations of the static scheme and make super block scheme practical. The dynamic super block scheme has the following key differences from the static scheme:

1. Crucially, super block merging is determined at runtime. Only blocks that exhibit spatial locality will be merged into super blocks. Programmers or compilers are not involved in this process.
2. In determining whether blocks should be merged into a super block, the dynamic super block scheme also takes into account the *ORAM access rate*, *prefetch hit rate*, etc. For example, if the prefetch hit rate is too low, merging should be stopped.
3. Finally, when a super block stops showing locality, the super block is broken. This makes it possible to adjust to program phases.

The dynamic super block scheme does not merge blocks during Path ORAM initialization. In other words, all blocks have $sbsize = 1$ after initialization. Accessing a block b in ORAM involves the following steps:

1. Access the path s where b is mapped to (according to the position map) and return to the processor's LLC (Last Level Cache) all the blocks that constitute the super block.
2. Super blocks are merged or broken according to spatial locality information.
3. Update the spatial locality statistics based on whether the prefetched blocks are used or not.

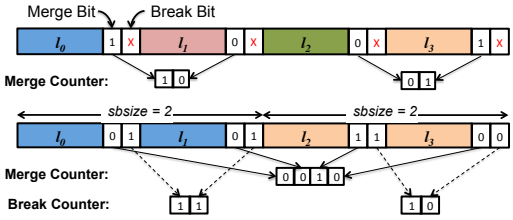

Figure 4: Hardware structure of *merge* and *break counter*. *Merge bits* from neighbor blocks form the their merge counter. Only super blocks have *break counters*.

The second and third steps are what make the dynamic super block scheme different from the static scheme. We propose a *per block counter-based* scheme to efficiently measure spatial locality to guide block merging/breaking.

4.1. Spatial Locality Counter

We first introduce the notion of a *neighbor block* to simplify the discussion. We call B' a neighbor block of block B, if they have the same size ($n = 2^k$) and can form a larger super block of size $2n = 2^{k+1}$. In Figure 3, block $0x02$ is a neighbor block of $0x03$ and super block $(0x00, 0x01)$ is a neighbor block of super block $(0x02, 0x03)$. However, super block $(0x02, 0x03)$ is not a neighbor block of super block $(0x04, 0x05)$. We restrict that only neighbor blocks can be merged into super blocks. Thus, as in the static super block scheme, only blocks with addresses differing in the last several bits can be merged into super blocks.

In order to decide what blocks should be merged, a *merge counter* and a *break counter* are introduced. Each pair of neighbor blocks have a *merge counter* indicating the spatial locality in them and to determine whether they should be merged or not (cf. Section 4.2). A *break counter* is associated with one super block to keep track of its spatial locality. A super block should be broken if it stops showing locality. The value of the counters will be updated based on the operations in Section 4.2 and Section 4.3.

Both counters are stored in the position map ORAM. Figure 4 shows the structure of a block in the position map ORAM (Pos-Map Block). Recall that each Pos-Map block contains the position map for multiple program addresses and stores the leaf labels for these addresses. A *merge bit* and a *break bit* are stored next to the leaf label in each position map. A counter is the concatenation of bits of the involved basic blocks. Once super blocks are merged or broken, the counters are reconstructed and the bits are reused for different super block sizes. This keeps the hardware overhead small.

Note that the maximum super block size is limited by the maximum number of position maps stored in a Pos-Map block. With our parameters, each Pos-Map block (128B) stores 32 position maps (25 bits each) of consecutive addresses along with their merge and break bits (i.e., $32 \times (25+2) = 864$ bits). A super block in our scheme only consists of the basic blocks which are consecutive in the address space, and its size is

always a power of 2. Therefore, position mapping of all the basic blocks of a super block always reside in the same Pos-Map block. When address a is accessed, the Pos-Map block containing a's mapping is loaded into the chip. The same Pos-Map block also contains the mappings of addresses near a (i.e., $\cdots, a-1, a+1, \cdots$) because they fit in the same Pos-Map block. As a result, whenever we access the position mapping of a [super]block, we also get the mapping for its neighbor blocks and its own sub-blocks along with their merge and break bits for free. Hence, there is no need for extra Pos-Map block accesses to load merge/break counter bits.

4.2. Merge Scheme

The merge operations are shown in Algorithm 1[1]. When a super block B of size n is returned to the LLC of the processor, the merge counter is first constructed. Since the position map for B and B' are stored in the same Pos-Map block, which has to be loaded into the ORAM controller before the data block can be accessed, the merge counter is completely reconstructed and can be operated on. The processor then detects whether all the n blocks in its *neighbor block* B' are also in the processor's cache. If so, we say B and B' have locality, and the merge counter of (B, B') is incremented. Otherwise, the merge counter will be decremented. If the merge counter reaches a *threshold*, B and B' are merged to a super block of size $2n$. How the *threshold* is determined will be discussed in Section 4.4.

Algorithm 1 Merge Algorithm

```
Super block B is loaded from ORAM to LLC
Merge counter is constructed for B and its neighbor B'
if all blocks in B' are in LLC then
    (B,B').merge_counter ++
    if (B,B').merge_counter ≥ threshold then
        Merge B and B' into (B,B')
    end if
else
    (B,B').merge_counter - -
end if
```

Merging block B and B' is achieved by changing the position map of B to the position map of B'. (Note that B' is already in the cache before merging) The changes are written to the Pos-Map block.

The Pos-Map blocks also keep track of the super block size. When the Pos-Map block is loaded, if the corresponding blocks in it are mapped to the same leaf label, the ORAM controller then treats these blocks as a super block.

Different from the static super block scheme discussed previously, Algorithm 1 dynamically exploits locality in the program. Blocks are merged only when they exhibit spatial locality, i.e., they are often present in the cache at the same time. After merging into super blocks, locality can be exploited since a single access now loads several useful data blocks.

[1]Incrementing a counter that is already the maximum value does not change the counter. Same for decrementing.

4.3. Break Scheme

Break operations may happen when super blocks are accessed in the ORAM. This is the time when all the blocks in a super block B are on-chip and the break counter of B can be fully reconstructed. Each data block in the ORAM or LLC is associated with a *prefetch bit* and a *hit bit*. The *prefetch bit* indicates whether a basic block was prefetched. The *hit bit* indicates whether the block's last prefetch was used.

Algorithm 2 Break Algorithm

```
In ORAM controller
Super block B = (B1,B2) is loaded from ORAM to LLC. The requested
block is in B1.
Reconstruct the break counter
for all basic block b in B coming from ORAM do
    if b.prefetch and not b.hit then
        B.break_counter - -
    else if b.prefetch and b.hit then
        B.break_counter ++
    end if
    b.prefetch = false
end for
if B.break_counter < threshold then
    break B into B1 and B2
    return B1 to LLC and write B2 back to ORAM
else
    for all basic block b in B2 do
        b.prefetch = true
        b.hit = false
    end for
end if

In Processor
when block b is accessed.
b.hit = true
```

Algorithm 2 specifies the super block breaking algorithm. Without loss of generality, we assume that the interesting block is located in the first half of $B = (B_1, B_2)$, which is B_1. The algorithm starts with updating the break counter with prefetch/hit information of previous accesses. The break counter is incremented by one for a prefetch hit and decremented by one for a prefetch miss.

If the resulting break counter is smaller than a *threshold*, super block B will be broken. Breaking of B is done by remapping B_1 and B_2 to independent leaf labels. And the half that does not have the requested block (B_2 in our case) is written back to ORAM. Note that the position map for both B_1 and B_2 are on chip at this time and can be modified.

Otherwise, the whole B will be returned to the LLC. In this case, each block in B_2 will have the *prefetch bit* set and *hit bit* reset indicating that the block is prefetched into the processor because B_2 is prefetched with respect to B_1 but has not been accessed yet. When a basic block with the *prefetch bit* set is accessed, a *prefetch hit* occurs and the *hit bit* is set. If a basic block has never been accessed since it was prefetched, the

block will be evicted to ORAM with the *hit bit* unset; this is deemed a *prefetch miss*. Both the *prefetch bit* and the *hit bit* will be read the next time the super block is loaded.

4.4. Counter Threshold

For both the *merge counter* and the *break counter*, merge and break operations are carried out when the value of the counter reaches a threshold. Properly determining the threshold value is important in achieving good system performance. We provide two algorithms to determine the threshold: *static thresholding* and *adaptive thresholding*.

4.4.1. Static Thresholding Static thresholding is very simple. The initial value of the merge counter is set to 0. Two neighbor blocks B_1 and B_2 of size $n = 2^k$ are merged when the value of their merge counter is higher or equal to $2n$ (note that this threshold fits in the merge counter which is $2n$ bits long). For block size of 1, 2 and 4 before merging, this corresponds to the threshold value of 2, 4 and 8, respectively. The threshold increases for larger block sizes because larger blocks incur more dummy accesses which may hurt performance.

Similarly, the initial value of break counter is $2n$ where n is the super block size. In our scheme, the threshold of break counter is 0, which is the minimal value of the break counter.

4.4.2. Adaptive Thresholding With static thresholding, blocks are merged whenever they exhibit enough data locality. However, even if all blocks have perfect spatial locality, if too many blocks are merged into large super blocks, system performance would still suffer due to the large number of background evictions required to ensure that the stash does not overflow (cf. Section 5.5.1). We propose to use *adaptive thresholding* to resolve this problem.

In particular, we propose to use the following equation to calculate the threshold.

$$threshold = C \times \frac{sbsize^2 \times eviction\_rate \times access\_rate}{prefetch\_hit\_rate} \quad (1)$$

$eviction\_rate$ is the number of background evictions divided by the total number of memory requests. $access\_rate$ is the percentage of time when the ORAM is busy. $prefetch\_hit\_rate$ is the percentage of hits out of all prefetched blocks. These numbers are collected within a time window and be updated periodically (every 1000 ORAM requests in this paper). Note that a larger threshold makes it harder to merge into super blocks.

The intuition behind the equation is fairly simple. As the threshold goes up, less blocks would be merged into super blocks, which reduces the number of background evictions. Take merging threshold as an example—when *sbsize* is large, we want to raise the threshold to be conservative [2] since larger *sbsize* incurs more background evictions. When *eviction_rate* and *access_rate* are high, we raise the threshold to prevent further increasing of background eviction. The *prefetch_hit_rate* is the opposite: we want to lower the threshold and merge more blocks when *prefetch_hit_rate* is high, which means block merging is accurate. The same arguments also hold for break threshold.

In practice, the merge threshold and break threshold are different by a small term to introduce some hysteresis into the system. In particular, $threshold_{Merge} = threshold + sbsize$ and $threshold_{Break} = threshold$. This prevents the case where a block keeps being merged and broken constantly.

Notice that the equation is not provably the optimal equation for the merge/break threshold, but it is simple and easy to implement in hardware. We leave the exploration of more complicated thresholding algorithms to future work.

4.5. Hardware Support

4.5.1. Storage The *hit bit* is stored with each data block in the ORAM and the LLC, since it is updated on an LLC hit and the corresponding position map block may not be on-chip. The *merge bit*, *break bit* and the *prefetch bit* are stored in the PosMap blocks. We assume 128-byte block size, so the storage overhead of dynamic super block is only 4 bits per block, less than 0.4%.

4.5.2. Computation For the merging scheme, when a block B is loaded into the LLC, we need to probe the LLC to check if the neighbor block B' exists in the cache (cf. Section 4.2). Only the tag array of the LLC needs to be accessed for this purpose. This can be done in parallel with the ORAM access and is not on the critical path. For the breaking scheme, the break counter is updated based on the prefetch bit and hit bit of each block in B, using simple comparison and arithmetic operations. When a block is accessed in LLC, the hit bit needs to be set. In summary, the scheme only involves several cache lookups and simple operations. They are cheap compared to the path read/write and data encryption/decryption in an ORAM access.

4.6. Security of Dynamic Super Block

The threat model under discussion is identical to prior ORAM work. We claim that ORAM with a dynamic super block scheme maintains the same level of security as a normal ORAM. In other words, adding dynamic super blocks to ORAM does not change the security guarantee of the original ORAM.

Following the security of the static super block scheme, accessing a super block of any size will look indistinguishable from accessing a normal data block because all the blocks in a super block are remapped at the same time. To demonstrate the security of dynamic super block, we only need to show the security of merging and breaking processes.

For merging, assume that block B_1 and B_2 (mapped to leaf s_1, s_2 respectively) are merged into a super block $B = (B_1, B_2)$. After merging, both blocks are mapped to a same leaf s which is a new independent random number, and is unlinkable to s_1, s_2 or any other previously accessed path. From the adversary's

[2] Experimental results show that $sbsize^2$ performs better than *sbsize*.

point of view, the leaves that are accessed in the ORAM are not linkable to each other. The adversary cannot figure out whether merging happens in an ORAM access at all.

Similarly, if block $B = (B_1, B_2)$ (mapped to s) breaks, the two halves B_1 and B_2 (mapped to s_1 and s_2) will be mapped to two independent random leaves. When one of B_1 and B_2 is accessed later, the leaf being accessed will be unlinkable to the leaf of the other half or s. This indicates that when or whether breaking happens cannot be learned by observing the ORAM access sequence.

To this point, the dynamic super block scheme does not leak any more information through the access pattern. However, one may argue that super block schemes leak *locality* information through timing channels. For example, merging blocks into super blocks reduces the total number of ORAM accesses which may be an indicator that the program has good spatial locality.

Although it is true that locality information may be learned through timing attacks, as said in Section 2.5, timing protection is not part of the original ORAM definition [11]. Very few ORAM designs in the literature considered timing attacks (i.e., when ORAM accesses happen or the total number of accesses) and ORAMs in general break under timing attacks. In order to protect the timing channel, periodic ORAM accesses need to be adopted ([10]), which can be easily added on top of ORAM with super blocks. We evaluate this design point in Section 5.6.

With periodic ORAM accesses to prevent timing channel leakage, the ORAM accesses are completely deterministic during the normal execution of a program and no leakage can happen. The only possible timing leakage is through the program's total execution time. As discussed by Fletcher *et al.* [9], the execution time of a program only leaks $\lg(T)$ bits information where T is the number of cycles the program takes. This is a very small leakage and applies to all types of ORAMs; adding a super block mechanism does not change it. Although it is true that enabling super blocks on a system may change the total runtime which tells the adversary some information about data locality, the same argument applies to other system components as well. For example, enabling vs. disabling branch prediction or the L3 cache leaks the branch behavior or memory footprint of the program. Super block is just one of these components and does not leak more information than other components.

To conclude, the dynamic super block scheme or any super block scheme in general maintains the same security level as conventional ORAMs. No extra leakage is introduced.

5. Evaluation

5.1. Methodology

Graphite [21] is used as the simulator in our experiments. Graphite simulates a tiled multi-core chip. The hardware configurations are listed in Table 1. We assume there is only one memory controller on the chip. While the insecure DRAM model can exploit bank-level parallelism and issue multiple memory requests at the same time (according to the Graphite DRAM model), all ORAM accesses are serialized (cf. Section 2.6).

Table 1: System Configuration.

Secure processor configuration	
Core model	1 GHz, in order core
L1 I/D Cache	32 KB, 4-way
Shared L2 cache	512 KB per tile, 8-way
Cacheline (block) size	128bytes
DRAM bandwidth	16 GB/s
conventional DRAM latency	100 cycles
Default ORAM configuration	
ORAM Capacity	8 GB
Number of ORAM hierarchies	4
ORAM basic block size	128 Bytes
Path ORAM latency	2364 cycles
Z	3
Max Super Block Size	2
Stash Size	100

The DRAM in Graphite is simply modeled by a flat latency. The 16 GB/s is calculated assuming a 1 GHz chip with 128 pins and pins are the bottleneck of the data transfer. Although this model neglects many DRAM internal structures, previous work ([25]) showed that using a more accurate model does not change the result much.

We use Splash-2 [37], SPEC06 [16] and a database management system (DBMS) application [38] to evaluate different ORAM prefetching techniques. For DBMS, we run two OLTP benchmarks: YCSB [5] and TPCC [33]. For most of the experiments, we will show both the speedup and the total memory accesses with respect to the baseline ORAM. The number of memory accesses is proportional to the energy consumption of the memory subsystem.

Three baseline designs are used for comparison: the insecure baseline using normal DRAM, the baseline Path ORAM without super blocks (*oram*) and the static super block scheme (*stat*). The default parameters for ORAM are shown in Table 1. Unless otherwise stated, all the experiments use these ORAM parameters.

We first evaluate a traditional stream prefetcher on both DRAM and Path ORAM (Section 5.2). Then, we will show the performance of different super block schemes with both synthetic and real benchmarks (Section 5.4). Different system parameters are explored in the sensitivity study section (Section 5.5). Finally, we evaluate the impact of having periodic ORAM accesses on these algorithms (Section 5.6).

5.2. Traditional Prefetching on Path ORAM

As discussed in Section 3.1, traditional prefetching does not help much in the context of ORAM. The reason behind this is the available memory bandwidth budget. Traditional DRAM prefetchers utilize the available bandwidth between useful accesses to issue prefetch requests. However, ORAM does not

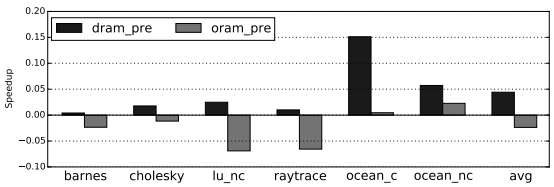

Figure 5: Traditional data prefetching on DRAM and ORAM

have that extra bandwidth available to issue prefetch requests. This conclusion is verified in Figure 5. Here, prefetching is added to both DRAM and ORAM based systems. Prefetching helps to improve performance on DRAM based systems. The ORAM, however, takes too much memory bandwidth and the memory subsystem is busy serving useful requests. Inserting prefetch requests will delay these useful requests. In the best case, if the prefetch hits and the prefetching is timely, the performance does not suffer. But if any prefetch misses, the performance decreases.

5.3. Synthetic Benchmark

In this section, a synthetic benchmark is used to study different aspects of super block schemes. $Z = 4$ is chosen here to make it easier to see the performance difference. The synthetic benchmark accesses an array with two patterns, *sequential* or *random*. For the *sequential* pattern, the part of the array is scanned sequentially, leading to good spatial locality. For the *random* pattern, the data is randomly accessed with no spatial locality.

5.3.1. Locality In Figure 6a, we sweep the percentage of data with locality. A benchmark with X% locality means that X% percentage of data are accessed sequentially and the rest are accessed randomly. Only the sequentially-accessed data has locality and can benefit from having super blocks.

As the figure shows, the static super block scheme only works when there is good spatial locality in the application and performs worse than the baseline ORAM if locality is bad. The dynamic super block scheme, on the other hand, always outperforms both the baseline ORAM and the static super block scheme. When there is no locality at all, the dynamic super block scheme has the same performance as the baseline ORAM. As the locality increases, performance also increases. Finally, the dynamic super block scheme has similar performance as the static scheme when there is 100% locality.

5.3.2. Phase Change The benchmark in Figure 6a only has static locality behavior, namely, the part of the data having locality always has locality throughout the whole execution. In practice, many programs have phase change behavior. Figure 6b models the phase change behavior where different parts of the data exhibit locality in different phases. Specifically, in the first phase, half of the data are accessed sequentially and the other half randomly. In the second phase, the first (second) half is randomly (sequentially) accessed. The pattern keeps switching in the following phases of the program.

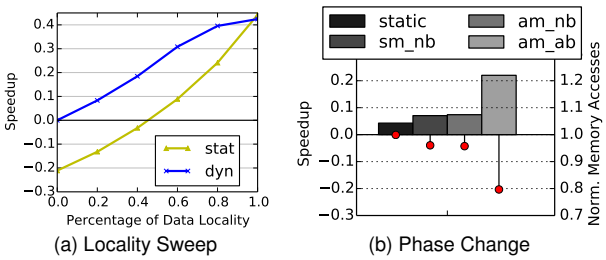

Figure 6: Different locality in the synthetic benchmark. Speedup is measured with respect to the baseline ORAM.

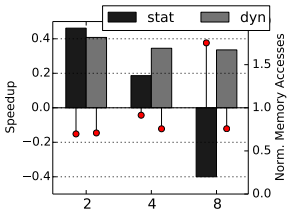

Figure 7: Sweep super block size of synthetic benchmark

In Figure 6b, *Sm*,*Am* stands for static and adaptive merging and *Nb*,*Ab* stands for no breaking and adaptive breaking respectively, and *static* represents the static super block scheme. In this case, it is obvious that breaking helps improve the performance. In the phases with locality, blocks will be merged to improve performance. And in the phases without locality, super blocks will be broken to prevent inaccurate prefetching. This helps to reduce the number of background evictions and improve prefetch hit rate.

5.3.3. Super Block Size Figure 7 sweeps the super block size (*sbsize*) in different super block schemes (for dynamic super block, *sbsize* is the maximum super block size). We run the synthetic benchmark which has 100% spatial locality.

Even with perfect locality, as *sbsize* increases, performance of the static super block scheme still degrades quickly due to excessive background evictions. On the other hand, the dynamic super block scheme will throttle merging of too large super blocks using the *adaptive thresholding* strategy introduced in Section 4.4. Once the background eviction rate is too high, super block merging is stopped and background eviction rate is kept low.

5.4. Real Benchmarks

Even though Path ORAM has inherent performance overhead over DRAM as explained in Section 2.6, it is important to note that this overhead is proportional to the memory intensiveness of the application. Memory intensive applications have significantly higher overhead than computation intensive applications. Figure 8a and Figure 8b show Splash2 and SPEC06 benchmarks respectively sorted in ascending order with respect to the overhead of baseline ORAM over DRAM. We consider all the benchmarks with less than $2\times$ overhead as *Computation Intensive* benchmarks (plotted over green background) and all those with more than $2\times$ overhead as *Memory Intensive* benchmarks (plotted over red background).

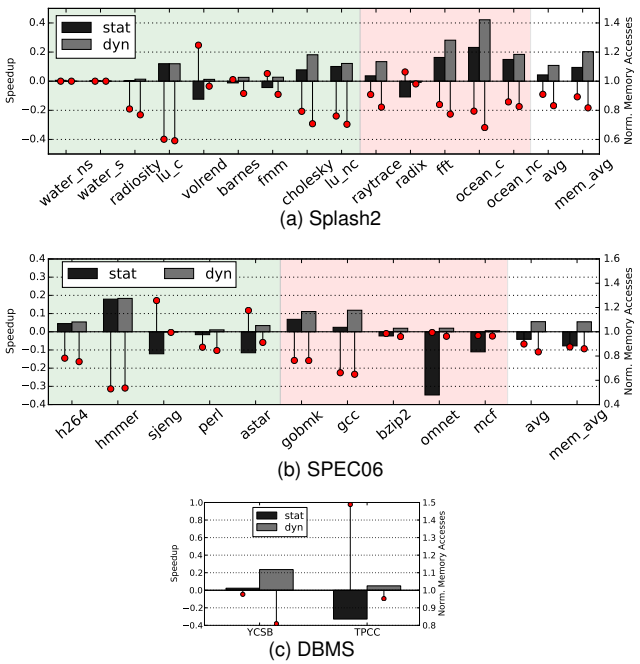

Figure 8: Speedup and normalized memory access count (with respect to baseline ORAM) of static and dynamic super block schemes on Splash2, SPEC06 and DBMS benchmarks.

As pointed out in Section 3.3, the static super block scheme is very sensitive to the nature of the application. I.e., it only shows performance gain for benchmarks that have good spatial locality (e.g., *ocean_contiguous*) which is clear from Figure 8. On some benchmarks (e.g., *volrend, radix, sjeng, astar, omnet, mcf, TPCC*), the static super block scheme gets worse performance than the baseline ORAM. This is either due to the fact that these benchmarks lack locality or that excessive background evictions hurt performance.

On the other hand, the dynamic super block scheme outperforms the baseline ORAM on all the benchmarks we evaluate here. On average, the performance gain of the dynamic super block scheme for memory intensive Splash2 benchmarks (*mem_avg*) over baseline ORAM is 20.2% whereas the overall average gain (*avg*) is 10.6% i.e., twice as much than what the static super block scheme offers. The overall average performance gain over the baseline ORAM for SPEC06 benchmarks is 5.5% whereas for DBMS, the performance gain over the baseline ORAM is 23.6% for YCSB and 5% for TPCC. The performance gain is most prominent for highly memory bound benchmarks. For *ocean_contiguous* for example, the performance gain is 42%. We also show the total number of ORAM accesses (normalized to the baseline ORAM) in Figure 8 (shown by red markers). This is proportional to the energy consumption of the memory subsystem. On average, the dynamic super block scheme saves 16.8% energy for Splash2, 16.6% for SPEC06 and 19.1% (4.8%) for YCSB (TPCC) over plain ORAM.

Figure 9 shows the prefetch miss rates of static and dynamic

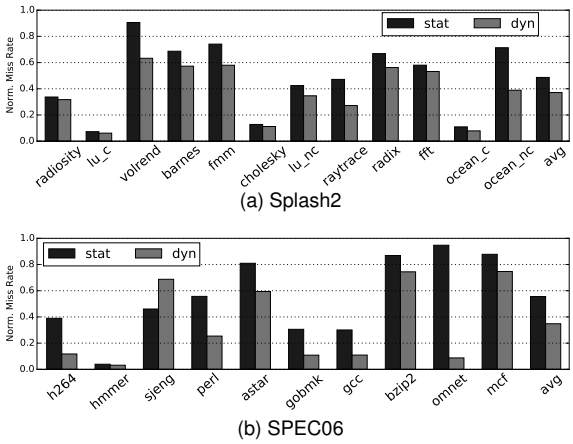

Figure 9: Miss rate for different Path ORAM schemes on Splash-2 and SPEC06 benchmarks.

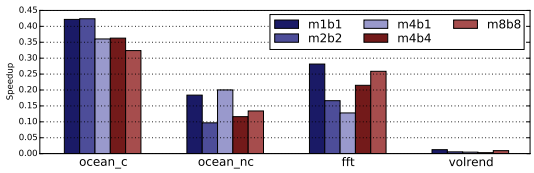

Figure 10: Sweep the coefficient in merging and breaking strategies.

super block schemes on Splash2 and SPEC06 benchmarks. *water-spatial* and *water-nsquared* are not shown here since they are too compute bound and do not access ORAM frequently. Since the static super block scheme prefetches all the neighbor blocks, the miss rate is very high for benchmarks that lack spatial locality (e.g., *volrend, omnet*). On average, the dynamic super block scheme lowers the overall prefetch miss rate of static super block from 48.6% to 37.1% for Splash2 benchmarks and from 55.5% to 34.8% for SPEC06 benchmarks.

5.5. Sensitivity Study

In this section, we will study how different parameters in the system affect the performance of super block schemes.

5.5.1. Merge/Break coefficient Equation 1 in Section 4.4.2 has a coefficient C unspecified. We now study how the coefficient affects the performance of super blocks.

Figure 10 sweeps the merge and break coefficients (C_{merge} and C_{break}). m*x*n*y* in the figure means that $C_{merge} = x$ and $C_{break} = y$.

For benchmarks with good spacial locality (e.g., *ocean_contiguous*, *ocean_non_contiguous*), smaller coefficient makes it easier for blocks to be merged into super blocks. As a result, merging happens earlier in the execution leading to better performance. For benchmarks with bad spacial locality (e.g., *volrend*), the coefficient actually does not have an effect on the performance, because merging does not happen regardless of the value of the coefficient.

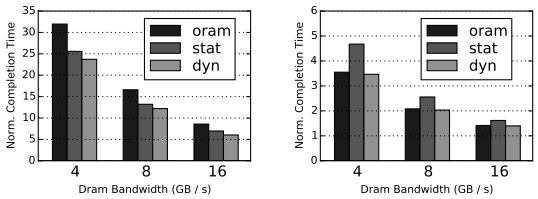

Figure 11: Sweep DRAM bandwidth.

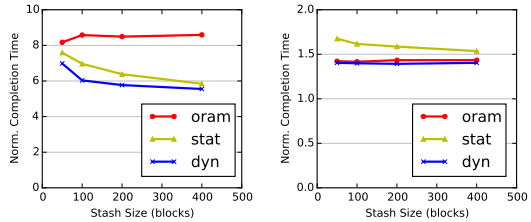

Figure 12: Sweep stash size.

For the rest of the paper, we use $C_{merge} = C_{break} = 1$ for our experiments.

5.5.2. DRAM Bandwidth The DRAM has a default bandwidth of 16 GB/s in our evaluations, which is calculated assuming that the pin bandwidth is the system bottleneck. In practice, however, this bandwidth might not be achievable. In Figure 11, we sweep different DRAM bandwidth values and the performance gain of the dynamic super block scheme is consistent across all configurations for memory intensive benchmarks (Figure 11a). This is because super blocks improve the memory efficiency in general by reducing the total number of ORAM accesses. This gain is orthogonal to the DRAM bandwidth.

For benchmarks with little or no locality (Figure 11b), the dynamic super block scheme does not have much gain over the baseline ORAM since merging does not take place. But both the dynamic super block scheme and the baseline ORAM have performance gain over the static super block scheme where blocks are blindly merged resulting in a large number of cache misses.

5.5.3. Stash Size As discussed in Section 2.2, the stash is an on-chip data structure which temporarily holds the blocks that cannot be evicted back to the binary tree. Whenever the stash becomes full, the ORAM does background eviction which introduces an extra ORAM access doing no real work. A larger stash is less likely to become full and thus reduces background eviction rate and improves performance.

In Figure 12, the stash size is swept for two different benchmarks, one with good spatial locality (*ocean_contiguous*) and the other with bad spatial locality (*volrend*). For both benchmarks, the performance of baseline ORAM does not change much when the stash size increases. This is because that the baseline ORAM already has a very low background eviction rate and enlarging the stash only gives marginal gain.

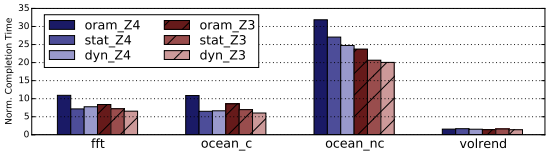

Figure 13: Different Z values.

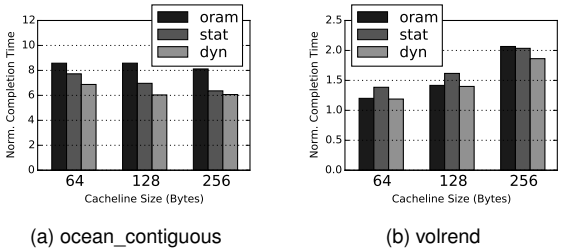

Figure 14: Sweep cacheline size.

For super block schemes, however, background eviction rate is high because multiple blocks may be added to the stash in each ORAM access. As a result, the performance increases as stash size becomes larger. For *ocean_contiguous*, both the dynamic and static super block schemes gain from larger stash size. For *volrend*, only the static super block scheme merges blocks into super blocks.

In general, dynamic super block scheme shows significant performance gain over ORAM even at small stash sizes in contrast to static super block.

5.5.4. Z Value Having larger Z increases the latency of each ORAM access since more data needs to be loaded. On the other hand, having smaller Z increases the background eviction rate. Previous work [25] showed that $Z = 3$ provides the best performance without super block. It is also the default parameter setting in this paper.

In Figure 13, both $Z = 3$ and $Z = 4$ are evaluated for baseline ORAM, static and dynamic super block schemes. $Z = 3$ achieves better performance than $Z = 4$ for the baseline ORAM, which corroborates previous results. The dynamic super block scheme has consistent performance gain for both Z values.

5.5.5. Cacheline Size The default cacheline size is 128 Bytes in this paper, in order to match the parameters in the previous work ([25]). Figure 14 shows the performance of different ORAM schemes at three different cacheline sizes (64, 128 and 256 Bytes). In general, the behaviors of dynamic and static super block schemes do not change.

5.6. Protecting Timing Channel

As pointed out in Section 4.6, the ORAM definition does not try to protect timing attacks. And an adversary may still learn lots of information from the timing of memory accesses. We need periodic ORAM accesses to protect the timing channel. Both dynamic and static super block schemes can be easily integrated with periodic ORAM accesses. Figure 15 shows the simulation results of periodic ORAM with static super blocks

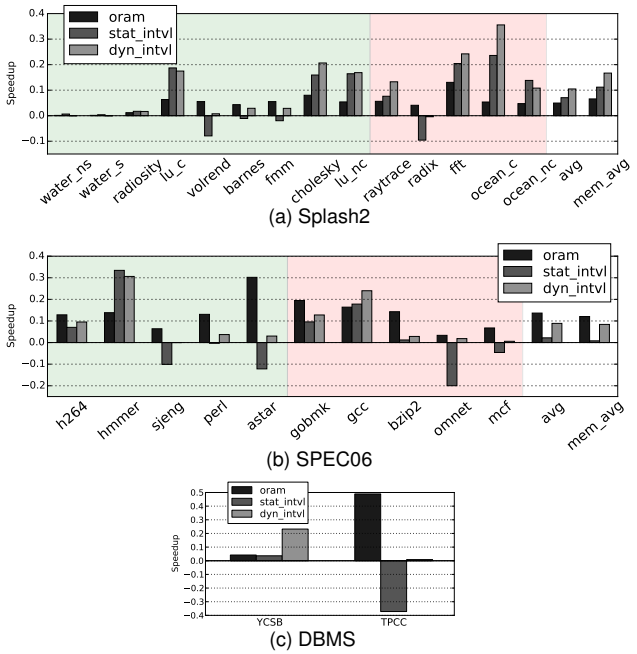

Figure 15: Periodic ORAM accesses. Speedup is respect to the baseline ORAM with periodic accesses. O_{int} = 100 cycles.

(*stat_intvl*) and dynamic super blocks (*dyn_intvl*) normalized to periodic ORAM. For comparison, non-periodic ORAM (*oram*) results are also shown. O_{int} is defined as the number of cycles between two consecutive ORAM accesses, which is chosen to be 100 cycles in this experiment.

Two observations can be made from Figure 15.

First, for most applications, adding periodicity in ORAM accesses does not significantly hurt performance. On average, only 3.6% additional performance degradation is incurred for Splash2. Part of the reason is that the O_{int} is chosen to be small in our evaluations thus ORAM bandwidth is almost maximized.

Second, dynamic super blocks provide performance gain regardless whether periodicity is used or not and therefore, once integrated with periodic ORAM, it can still provide significant speedup while preventing the information leakage over the timing channel.

Since the ORAM has a strictly periodic access pattern, the energy consumption of different ORAM schemes would be the same. However, this performance advantage of dynamic super blocks can be easily translated to energy advantage by setting O_{int} high, which slows down the system but makes it more energy efficient ([9]).

6. Related Work

This paper mainly focuses on applying data locality optimizations to *Oblivious RAM*. The most relevant previous works are ORAM optimization techniques and locality optimizations in the memory system.

6.1. ORAM Optimization

Previous work [25] has explored the Path ORAM design space and proposed a static super block scheme. We used this optimized Path ORAM as the baseline in this paper. We extend the static super block scheme to a dynamic super block scheme which is significantly more stable and has better performance.

Previous work has exploited the fact that ORAM operations can be easily parallelized and assigned to multiple trusted coprocessors [20]. The optimization techniques proposed in our paper are orthogonal and can be applied on top of [20].

While we used Path ORAM for discussion in this paper, the optimization techniques proposed are not constrained to Path ORAM. For example, other ORAM schemes (e.g., [27]) have similar binary tree structure to Path ORAM. After adding background eviction, these ORAM schemes can also benefit from using super blocks. In general, all ORAM schemes should be able to take advantage of super blocks as long as they have support for background eviction.

6.2. Exploiting Locality in Memory

In the architecture community, there has been lots of work exploiting data locality in programs. An important technique that has been widely used is data prefetch [28, 6, 4, 35], where the processor loads blocks that are likely to be accessed in the near future into the cache.

In this paper, we showed that traditional prefetching techniques do not work well in an ORAM context, and that super block schemes take advantage of ORAM internal structure to exploit locality.

Our paper makes the assumption that only the blocks consecutive in address space can be merged into super blocks. However, previous work in data prefetch [4] allows data striding in the address space to be prefetched. Merging striding blocks is also possible for the dynamic super block scheme. Such exploration is left for future work.

7. Conclusion

A novel ORAM prefetcher: *PrORAM* based on *dynamic super block* is proposed in this paper. The implementation details are discussed and the design space is comprehensively explored. We show that PrORAM introduces the first practical super block scheme and is much more stable than a static super block scheme over different benchmarks. On memory intensive Splash-2 and SPEC06 benchmarks, PrORAM improves over baseline ORAM by 20.2% and 5.5% in terms of completion time and reduces energy consumption by 16.8% and 16.6% respectively. For DBMS, the performance gain is 23.6% and 5% and energy reduction is 19.1% and 4.8% for YCBS and TPCC respectively.

Acknowledgments: This research was partially supported by QCRI under the QCRI-CSAIL partnership and by the National Science Foundation.

References

[1] W. Arbaugh, D. Farber, and J. Smith, "A Secure and Reliable Bootstrap Architecture," in *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, May 1997, pp. 65–71. [Online]. Available: citeseer.nj.nec.com/arbaugh97secure.html

[2] D. Boneh, D. Mazieres, and R. A. Popa, "Remote oblivious storage: Making oblivious RAM practical," Manuscript, http://dspace.mit.edu/bitstream/handle/1721.1/62006/MIT-CSAIL-TR-2011-018.pdf, 2011.

[3] T.-F. Chen and J.-L. Baer, "Effective hardware-based data prefetching for high-performance processors," *Computers, IEEE Transactions on*, vol. 44, no. 5, pp. 609–623, 1995.

[4] ——, "Effective hardware-based data prefetching for high-performance processors," *Computers, IEEE Transactions on*, vol. 44, no. 5, pp. 609–623, 1995.

[5] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *SoCC'10*, pp. 143–154.

[6] F. Dahlgren, M. Dubois, and P. Stenstrom, "Fixed and adaptive sequential prefetching in shared memory multiprocessors," in *Parallel Processing, 1993. ICPP 1993. International Conference on*, vol. 1. IEEE, 1993, pp. 56–63.

[7] I. Damgård, S. Meldgaard, and J. B. Nielsen, "Perfectly secure oblivious RAM without random oracles," in *TCC*, 2011.

[8] C. 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 *Proceedings of the* 20^{th} *Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[9] C. Fletcher, L. Ren, X. Yu, M. Van Dijk, O. Khan, and S. Devadas, "Suppressing the oblivious ram timing channel while making information leakage and program efficiency trade-offs," in *Proceedings of the Int'l Symposium On High Performance Computer Architecture*, 2014.

[10] C. Fletcher, M. van Dijk, and S. Devadas, "Secure Processor Architecture for Encrypted Computation on Untrusted Programs," in *Proceedings of the 7th ACM CCS Workshop on Scalable Trusted Computing; an extended version is located at http://csg.csail.mit.edu/pubs/memos/Memo508/memo508.pdf (Master's thesis)*, Oct. 2012, pp. 3–8.

[11] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," in *J. ACM*, 1996.

[12] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia, "Oblivious ram simulation with efficient worst-case access overhead," in *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, ser. CCSW '11. New York, NY, USA: ACM, 2011, pp. 95–100. [Online]. Available: http://doi.acm.org/10.1145/2046660.2046680

[13] ——, "Practical oblivious storage," in *Proceedings of the second ACM conference on Data and Application Security and Privacy*, ser. CODASPY '12. New York, NY, USA: ACM, 2012, pp. 13–24. [Online]. Available: http://doi.acm.org/10.1145/2133601.2133604

[14] ——, "Privacy-preserving group data access via stateless oblivious RAM simulation," in *SODA*, 2012.

[15] D. Grawrock, *The Intel Safer Computing Initiative: Building Blocks for Trusted Computing*. Intel Press, 2006.

[16] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[17] D. Lie, J. Mitchell, C. Thekkath, and M. Horwitz, "Specifying and verifying hardware for tamper-resistant software," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2003.

[18] D. Lie, C. Thekkath, and M. Horowitz, "Implementing an untrusted operating system on trusted hardware," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, 2003, pp. 178–192.

[19] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural Support for Copy and Tamper Resistant Software," in *Proceedings of the* 9^{th} *Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, November 2000, pp. 168–177.

[20] J. R. Lorch, J. W. Mickens, B. Parno, M. Raykova, and J. Schiffman, "Toward practical private access to data centers via parallel oram." *IACR Cryptology ePrint Archive*, vol. 2012, p. 133, 2012, informal publication. [Online]. Available: http://dblp.uni-trier.de/db/journals/iacr/iacr2012.html#LorchMPRS12

[21] J. E. Miller, H. Kasture, G. Kurian, C. G. III, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A Distributed Parallel Simulator for Multicores," in *HPCA*, 2010.

[22] R. Ostrovsky, "Efficient computation on oblivious rams," in *STOC*, 1990.

[23] R. Ostrovsky and V. Shoup, "Private information storage (extended abstract)," in *STOC*, 1997, pp. 294–303.

[24] S. Palacharla and R. E. Kessler, "Evaluating stream buffers as a secondary cache replacement," in *ACM SIGARCH Computer Architecture News*. IEEE Computer Society Press, 1994.

[25] L. Ren, X. Yu, C. Fletcher, M. van Dijk, and S. Devadas, "Design space exploration and optimization of path oblivious ram in secure processors," in *Proceedings of the Int'l Symposium on Computer Architecture*, June 2013, available at Cryptology ePrint Archive, Report 2013/76.

[26] L. F. G. Sarmenta, M. van Dijk, C. W. O'Donnell, J. Rhodes, and S. Devadas, "Virtual Monotonic Counters and Count-Limited Objects using a TPM without a Trusted OS," in *Proceedings of the 1st STC'06*, Nov. 2006.

[27] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li, "Oblivious ram with $o((\log n)^3)$ worst-case cost," in *Asiacrypt*, 2011, pp. 197–214.

[28] A. J. Smith, "Cache memories," *ACM Computing Surveys (CSUR)*, vol. 14, no. 3, pp. 473–530, 1982.

[29] E. Stefanov, E. Shi, and D. Song, "Towards practical oblivious RAM," in *NDSS*, 2012.

[30] E. Stefanov, M. van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path oram: An extremely simple oblivious ram protocol," in *Proceedings of the ACM Computer and Communication Security Conference*, 2013.

[31] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, " AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing," in *Proceedings of the* 17^{th} *ICS (MIT-CSAIL-CSG-Memo-474 is an updated version)*. New-York: ACM, June 2003. [Online]. Available: http://csg.csail.mit.edu/pubs/memos/Memo-474/Memo-474.pdf(revisedone)

[32] 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 *Proceedings of the* 32^{nd} *ISCA'05*. New-York: ACM, June 2005. [Online]. Available: http://csg.csail.mit.edu/pubs/memos/Memo-483/Memo-483.pdf

[33] The Transaction Processing Council, "TPC-C Benchmark (Revision 5.9.0)," http://www.tpc.org/tpcc/spec/tpcc_current.pdf, June 2007.

[34] Trusted Computing Group, "TCG Specification Architecture Overview Revision 1.2," http://www.trustedcomputinggroup.com/home, 2004.

[35] S. P. Vanderwiel and D. J. Lilja, "Data prefetch mechanisms," *ACM Computing Surveys (CSUR)*, vol. 32, no. 2, pp. 174–199, 2000.

[36] P. Williams and R. Sion, "Single round access privacy on outsourced storage," in *Proceedings of the 2012 ACM conference on Computer and communications security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 293–304. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382229

[37] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: characterization and methodological considerations," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995, pp. 24–36.

[38] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker, "Staring into the abyss: An evaluation of concurrency control with one thousand cores," *Proceedings of the VLDB Endowment*, vol. 8, no. 3, pp. 209–220, 2014.

[39] X. Zhuang, T. Zhang, and S. Pande, "HIDE: an infrastructure for efficiently protecting information leakage on the address bus," in *Proceedings of the 11th ASPLOS*, 2004.