Enhancing Oblivious RAM Performance Using Dynamic Prefetching

Xiangyao Yu[†], Ling Ren[†], Christopher Fletcher[†], Albert Kwon[†], Marten van Dijk[‡], Srinivas Devadas[†]

[†] Massachusetts Institute of Technology – {yxy, renling, cwfletch, devadas}@mit.edu
[‡] University of Connecticut – {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 data 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.

Though traditional data prefetching techniques successfully hide memory latency in DRAM based systems, it turns out that they do not work well for ORAM. In this paper, we exploit ORAM locality by taking advantage of the ORAM internal structures. Though it might seem apparent that obliviousness and locality are two contradictory concepts, we challenge this intuition by exploiting data locality in ORAM without sacrificing provable security. In particular, we propose an ORAM prefetching technique called dynamic super block scheme and comprehensively explore its design space. The dynamic super block scheme detects data locality in the program's working set at runtime, and exploits the locality in a data-independent way.

Our simulation results show that with dynamic super block scheme, ORAM performance without super blocks can be significantly improved. After adding timing protection to ORAM, the average performance gain is 25.5% (up to 49.4%) over the baseline ORAM and 16.6% (up to 30.1%) over the best ORAM prefetching technique proposed previously.

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 [12], which is based on TPM [30, 1, 23], eXecute Only Memory (XOM) [13, 14, 15] and Aegis [28, 29].

While an adversary cannot access the internal states inside the tamper-resistant hardware, information can still be leaked through side channels, e.g., memory I/O channel. 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 ([34]) show that control flow of a program can be learned by observing main memory access patterns. This can lead to leakage of 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 [8], and there has been significant follow-up work that has resulted in more and more efficient cryptographically-secure ORAM schemes [19, 18, 6, 2, 11, 32, 9, 10, 24, 26, 27]. 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. A recently proposed ORAM design, Path ORAM [27], is so far the most efficient and practical ORAM system for secure processors. However, Path ORAM still incurs at least $30\times$ more latency than a normal DRAM for a single access. This results in $2-10\times$ performance slowdown as shown in previous work [22, 7]. High access latency is the main obstacle of Path ORAM performance.

Traditionally, *data prefetching* [20, 3] has been used to hide long memory access latency. *Data prefetching* uses the memory access pattern from history to guess which data block will be accessed in the near future. The predicted block is loaded from memory before it is actually requested to hide memory access latency.

Though 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 memory latency and bandwidth. Prefetching only works when DRAM has extra bandwidth, therefore does not work well for ORAM (cf. Section 3).

In this paper, we try to enable ORAM prefetching by exploiting locality inside the ORAM itself, which is totally different from traditional prefetching techniques. However, 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, since accesses to adjacent addresses must be made indistinguishable and each access should incur the same amount of redundancy. We challenge this intuition by exploiting data locality in ORAM without sacrificing provable security.

We propose a novel dynamic super block scheme, 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 the program's working set at *runtime*, and exploits the locality in a data-independent way.

In particular, the paper makes the following contributions:

1. Traditional data prefetching techniques are studied in the context of ORAM. An observation is made that directly applying data prefetching does not work for ORAM.
2. A dynamic super block scheme is proposed, and the design space is comprehensively explored. The micro-architecture of dynamic super block scheme is discussed in detail.

   Our simulation results show that with timing protection, dynamic super scheme improves Path ORAM performance by 25.5%.

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 studies how traditional prefetching techniques work on ORAM systems. Section 4 presents ORAM prefetch techniques in general and discusses a previously proposed scheme called super block. Dynamic super blocks are introduced in Section 5. The design space is explored, security is shown and hardware complexity is analyzed in detail. Section 6 presents our evaluation methodology and Section 7 evaluates different optimizations proposed in the paper. Related work is presented in Section 8 and we conclude the paper in Section 9.

2. Background

We provide background for Oblivious RAM in Section 2.1 and details of Path ORAM in Section 2.2.

2.1. Oblivious RAM

Oblivious RAM (ORAM) was first proposed and investigated in [8]. ORAM is a primitive for data storage while hiding the access patterns to it such that an adversary will not be able to figure out what data a user is trying to access. In general, a user may access the sequence of program addresses $A = (a_1, a_2, ..., a_n)$, which will be translated to a sequence of oblivious ORAM accesses $S = (s_1, s_2, ..., s_m)$. a_i is the program address of the data block i and the value of a_i is what

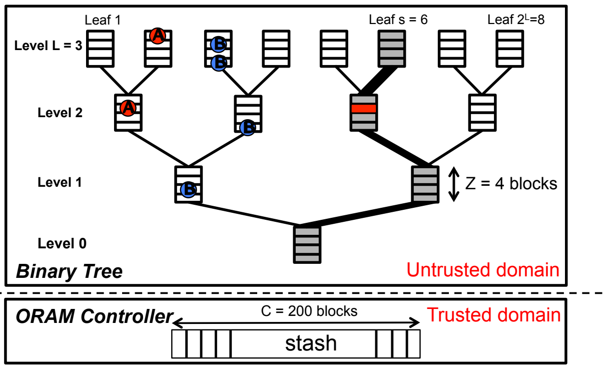


Figure 1: A Path ORAM for $L = 3$ levels. Path $s = 6$ is accessed.

we are trying to hide. s_i is the physical address used to access ORAM and the value of s_i is exposed to adversaries. 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 address sequence (A). Data 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 [27], which is currently the most efficient ORAM scheme, and is appealing to secure processors due to its simplicity.

2.2. Path ORAM

Path ORAM [27] has two main hardware components: *binary tree storage* and *ORAM controller* (cf. Figure 1). We briefly introduce the functionality of both.

Binary tree can be implemented based on any storage system (we choose DRAM in this paper) and is used to store the protected data. Each node in the tree is defined as a *bucket* and can hold up to Z data blocks. Buckets that have less than Z data 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 logical address of a data block (a) with a leaf in the ORAM tree (leaf s). The *stash* is a memory that stores up to a small number of data blocks at a time.

At any time, each data block (with logical address a) in the Path ORAM is mapped (randomly) to some leaf s via the position map. Path ORAM maintains the following invariant: *if data block a is currently mapped to leaf s, then a must be stored either (i) on the path from the root to leaf s, or (ii) in the stash* (see Figure 1). The steps to access a block a in Path ORAM are as follows:

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

The path read and write operation (Step 2 and Step 5, respectively) should be done in a data-independent way (e.g., always from the root to the leaf).

Step 4 is the key to Path ORAM's security, where a block is randomly remapped to a new leaf whenever it is accessed. This guarantees that a random path is read and written on every access regardless of the requested address sequence.

2.3. Hierarchical Path ORAM

In practice, the position map is usually too large to be stored in the trusted processor. In literature, hierarchical Path ORAM has been proposed to solve this problem [24, 7, 22].

In a 2-level hierarchical 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. 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.

Each block in a position map ORAM stores the leaf labels for multiple blocks that are consecutive in the address space. In other words, we can find positions of several blocks in a single access to position map ORAM, although only one of them is of interest. Later we will show that this fact is used to enable locality optimizations in Path ORAM (cf. Section 5).

2.4. Background Eviction

In Steps 4 and 5 of the Path ORAM operation, the accessed data block is remapped from the old leaf s to a new random leaf s', so it can only be written back to the common sub-path shared by path s and s' (in order to maintain the Path ORAM invariant). If these buckets are completely filled with real data blocks, the remapped block cannot be written back to the binary tree and must stay in the stash. In practice, this may cause blocks to accumulate in the stash and finally overflow. We say Path ORAM fails if its stash overflows. [27] proves that the stash overflow probability is negligible for $Z \geq 6$. [22] proposed a *background eviction* scheme based on the notion of dummy access to prevent stash overflow even with small bucket sizes (e.g., $Z \leq 4$).

ORAM controller stops serving real requests and issues background evictions (dummy accesses), when the available slots in the stash is less than the number of blocks on one path (otherwise, there is a chance that the next access will overflow the stash). 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 [22].

2.5. Timing Channel Protection

Though ORAM makes the memory access sequence indistinguishable from each other, the original ORAM definition in [8] does not protect timing attacks and virtually all ORAMs break under timing attacks. The timing informations include when an ORAM access happens and 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, we need to add periodic access on top of ORAM in order to protect the timing channel ([7]). This means that ORAM accesses will happen periodically. We define O_{int} as the time interval between two consecutive ORAM accesses. O_{int} is a constant number thus ORAM timing behavior will be fixed and public. If no memory request exists when an ORAM access needs to happen due to periodicity, a dummy access will be issued. A dummy access will just access a random path in Path ORAM similar to a background eviction.

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 [26, 22]. 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. In order to minimize ORAM latency, the entire DRAM bandwidth needs to be used for a single ORAM access. Thus the system cannot serve multiple ORAM requests simultaneously.

Hierarchical ORAMs introduce additional overheads of accessing multiple level 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

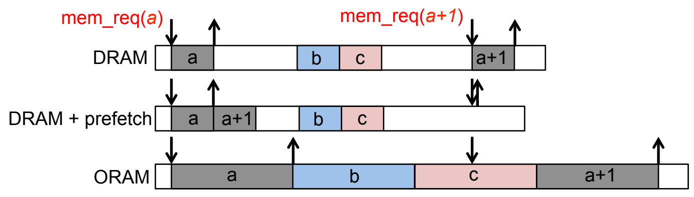
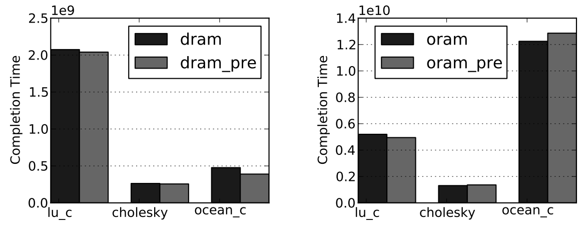


Figure 2: Data prefetching on DRAM and ORAM.



(a) DRAM Prefetch (b) ORAM Prefetch

Figure 3: Traditional data prefetching on DRAM and ORAM.

to up to an order of magnitude slowdown in a secure processor [22]. Though 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 proportional energy consumption.

3. Traditional Data Prefetch

As access latency is the main bottleneck in ORAM, a natural solution that comes to mind is to apply traditional latency hiding techniques to ORAM. *Prefetching* is a mature latency hiding technique used for decades by architects. In this section, we will study how hardware prefetching performs with Path ORAM.

3.1. Stream Prefetching

There are two main classes of prefetching techniques, *software pretching* and *hardware prefetching*. We only study the latter in this paper, but our conclusion also applies to software prefetching.

Figure 2 shows the basic idea of data prefetching. When block a is accessed, if the prefetcher somehow figures out that block $a+1$ will also be used in the near future, then block $a+1$ will be loaded earlier from the DRAM. When the real request to $a+1$ arrives, the data can be immediately returned back to the pipeline without paying an extra access latency. Prefetching moves memory accesses out of the critical path of execution thus leading to overall speedup of the program.

Specifically, we study the classic prefetcher from [20], which is based on stream buffers. Though this prefetcher is simple in terms of hardware structure, it represents the common features of many hardware prefetchers (e.g., prefetcher in Xeon Phi [21]), and thus suffices our study.

In particular, the prefetcher contains 16 stream buffers and 16 history buffers (filters). One particular stream buffer fetches the next data block in a stream if the stream pattern is detected. Prefetches do not happen for all blocks. The filters detect which blocks have spatial locality based on access patterns in history and only assign a stream buffer when locality is detected.

When a memory request arrives, if the requested address matches an entry in the stream buffer, the data can be immediately returned back to the Last Level Cache (LLC) without paying the DRAM latency (and the stream buffer will start to prefetch the next data block). If the prefetch hit rate is high (which is true with good filtering algorithms) and the prefetching is timely, hardware prefetching buys a lot of performance gain for programs.

3.2. Prefetching with ORAM

When DRAM is replaced with an ORAM, however, the situation changes. As shown in Figure 2, ORAM latency is much higher than DRAM latency (more than 30×). More importantly, ORAM consumes much higher bandwidth than DRAM (2 orders of magnitude more). This leads to two effects. First, multiple ORAM requests would not overlap with each other. In order to hide ORAM access latency, a single ORAM already fully utilizes the entire DRAM bandwidth (cf. Section 2.6) and cannot share with a second access. Second, for memory bound applications, ORAM requests line up in the ORAM controller and there is no free time slot. As a result, the ORAM is serving normal memory requests all the time. If a prefetch request is served, it will block normal requests and may hurt performance.

Figure 3 shows the performance of using prefetch on both DRAM and ORAM systems. Prefetching helps to improve performance in DRAM systems, especially for memory bound benchmarks like *ocean_contiguous*. For the ORAM system, however, it does not work if the application is too memory bound. If the application is not too memory bound, we may see some performance gain like in the case of *lu_contiguous*. But this gain is generally small and disappears in many other applications (cf. Section 7).

4. ORAM Prefetch: Super Block

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

4.1. General Idea

The notion of *super block*, first proposed in [22], tries to exploit spatial locality in ORAM. In particular, it tries to load multiple blocks that have exhibited spatial locality 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 leaf thus reside on the same path. For example, in Figure 1, blocks marked as *A* form a super block of size 2 and blocks marked with B form a super block of size 4.

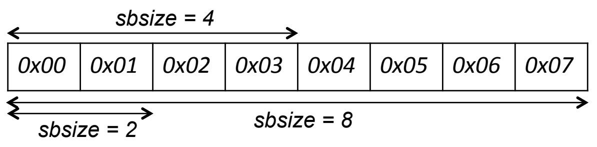


Figure 4: Super block construction. Blocks whose addresses are different only in the last k address bits can be merged into a super block of size $n = 2^k$.

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 block as the number of data blocks in it, denoted as *sbsize*. For example, in Figure 4 block 0x02 and block 0x03 can be merged into a super block of size 2 and blocks from 0x00 to 0x03 can be merged into a super block of size 4. But we cannot merge block 0x03 and 0x04 into a super block of size 2 because their addresses differ in the second least significant bit.

Whenever one block in a *super block* is accessed, all the blocks in that *super block* are read out from the path and remapped to a new random leaf. The block of interest is returned to the processor and the other blocks are *prefetched* and put into LLC (Last Level Cache). The hope is that the prefetched blocks will be accessed in the near future due to 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 1). This guarantees that all the blocks belonging to the same super block can be found during a single ORAM access.[1] It is important to note that though 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.

4.2. Static Super Block

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

[22] proposed a naïve and static scheme, called *static super block*. In this scheme, every $n = 2^k$ consecutive data blocks are merged into super blocks of size n. n is statically specified by the user before the programs start to run. 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 leaf. During normal ORAM operations, a super block is accessed as a unit as described in Section 4.1. This scheme is very simple and requires minimal hardware change to basic Path ORAM.

[1]Note that they do not need to reside in the same bucket.

4.2.1. 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 achieved since each access will read and write a random path which is not linkable to another access. This property is still true for static super block. Accesses to different super blocks will be touching independent random paths. Accesses to blocks in the same super block will also touch independent paths since each access will load and remap ***all*** blocks in the super block. Thus, an adversary is not able to tell the super block size or whether static super block scheme is used at all.

4.2.2. Limitations The static super block scheme discussed above has significant limitations which make it not practical:

First, static super blocks are constructed in advance and cannot be resized during program execution. If blocks that have no locality are merged into super blocks, they will hurt performance since the cache will be polluted and the number of background evictions may increase. This argument will be verified by experiments in Section 7.3.1.

Second, it is very hard for an average programmer to learn the locality behavior of a program to decide whether static super block scheme should be used or not. In order to figure this out, the programmer needs to run a test program either on the real machine or on a simulator. Both solutions require significant effort. This limits the applicability of static super block scheme.

5. Dynamic ORAM Prefetch

This paper proposes a dynamic ORAM prefetch scheme called *dynamic super block* to resolve the limitations of the static scheme in Section 4.2. The dynamic super block scheme has the following key differences from the static scheme:

1. Crucially, super block merging is determined at runtime based on the spatial locality exhibited in the blocks. Only blocks with locality are merged into super blocks. Programmers 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.

As in the static super block scheme, only blocks consecutive in address space are candidates for merging into super blocks.

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 all the

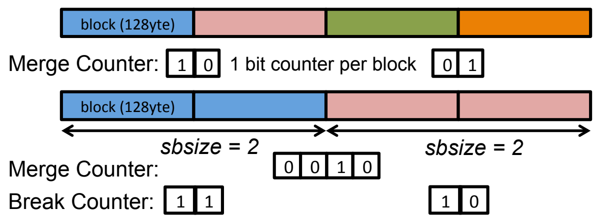


Figure 5: Hardware structure of *merge* and *break counter*.

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 the access pattern of loaded blocks.

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

5.1. Spatial Locality Counter

We now introduce the notion of a *neighbor block* to simplify the discussion. We call B' a neighbor block, relative to another block B, if they have the same size ($n = 2^k$) and the base address of B' is adjacent to B and they constitute a larger super block of size $2n = 2^{k+1}$. In Figure 4, block $0x02$ is a neighbor block of $0x03$ and super block $(0x00, 0x01)$ is a neighbor block of super block $(0x02, 0x03)$. We restrict that only neighbor blocks can be merged into super blocks.

In order to decide what blocks should be merged into super blocks, a *merge counter* and a *break counter* are stored next to the position map block in the position map ORAM. Both the block and the counters are encrypted and are not leaked to outside observers/adversaries. A *merge counter* is associated with two neighbor blocks to discover locality in them and determine whether they should be merged or not (cf. Section 5.2). 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 5.2 and Section 5.3.

To keep the hardware cost as low as possible, both the *merge counter* and the *break counter* are 1-bit per basic block in ORAM, as shown in Figure 5. The counter for a super block is the concatenation of counters of comprising basic blocks. Once super blocks are merged or broken, the old counter values are discarded. In this way, we can reuse the counters for different super block sizes and keep the hardware overhead small.

5.2. Merge Scheme

The merge operations are shown in Algorithm 1[2]. When a

Algorithm 1 Merge Algorithm

```
Super block $B$ is loaded from ORAM to LLC
**if** $B$'s neighbor block $B'$ is in LLC **then**
    $(B,B')$.merge_counter ++
    **if** $(B,B')$.merge_counter $\geq$ threshold **then**
        Merge $B$ and $B'$ into $(B,B')$
    **end if**
**else**
    $(B,B')$.merge_counter - -
**end if**
```

super block B of size n is returned to the LLC of the processor, the processor first 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. Note that B and B' share the same merge counter, which is stored in the position map ORAM and is loaded to the chip together with the blocks. 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 5.4

Merging of blocks is achieved by mapping both B and B' to the same leaf in the data ORAM through changing the position map of B to the position map of B' (Note that B' is already in the cache before merging). If hierarchical Path ORAMs are used, the position map ORAMs need to be updated. However, updating the position map ORAMs does not require an extra ORAM access. This is because the position map of both block B and B' will be loaded to the chip before the data blocks and be present during the merging operations (cf. Section 2.3).

The position map ORAMs are also used to keep track of super block size. When the position 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 and are often present in the cache at the same time. After merging into super blocks, locality can be exploited as a single access will now load several useful data blocks.

5.3. Break Scheme

The break operation is trickier than the merge operation. According to the super block definition, the basic blocks comprising B may not be in the cache together all the time. Some basic blocks may have already been evicted back to the Path ORAM while some others stay in the cache (cf. Section 4.1). Since each bit in the break counter is associated with a basic block in B, the whole break counter cannot be found if part of B is not in the cache and the break counter cannot be properly updated.

Fortunately, all basic blocks of B will be in the LLC when B is loaded from the ORAM. This is the time when the break

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

counter of B can be fully reconstructed and the locality information can be updated. To enable this, any basic block in ORAM or cache is associated with a *prefetch bit* and a *hit bit*. The *prefetch bit* indicates whether a basic block was prefetched or not. The *hit bit* indicates whether the block's last access was a prefetch hit or not. Both bits are stored together with the data block (both in ORAM and cache).

Algorithm 2 Break Algorithm

```
In ORAM controller
Super block $B = (B_1, B_2)$ is loaded from ORAM to LLC.
The requested block is in $B_1$.
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 $B_1$ and $B_2$
    return $B_1$ to LLC and write $B_2$ back to ORAM
else
    for all basic block $b$ in $B_2$ 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, otherwise the whole B will be returned to LLC.

Breaking of B is done by remapping B_1 and B_2 to different leaf labels. And the half that does not have the requested block (B_2 in our case) is written back to ORAM.

If B is returned to LLC, each block in B_2 will have the *prefetch bit* set and *hit bit* reset indicating that the block is prefetched into the processor [3] but has not been accessed yet. When a basic block is accessed by the pipeline with the *prefetch bit* set, 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.

[3] Blocks in B_1 do not have *prefetch bit* set. The intuition is that B_2 is prefetched with respect to B_1 and the locality inside B_1 does not indicate the locality between B_1 and B_2.

5.4. Counter Threshold

For both *merge counter* and *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*.

5.4.1. Static Thresholding Static thresholding is very simple. The initial value of 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 set to be $2n$ where n is the super block size. In our scheme, the threshold of break counter is set to be 0, which is the minimal value of the break counter.

5.4.2. Adaptive Thresholding With static thresholding, blocks would be 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 the stash does not overflow (c.f. Section 7.3.3). We propose to use *adaptive thresholding* to resolve this problem.

In particular, we propose to use the following equation to calculate 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 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 [4] since large *sbsize* incurs lots of 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. Experiments show that performance is not sensitive to the coefficient C. For the rest of the paper $C = 1$ will be assumed.

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

5.5. Hardware Support

In general, dynamic super block incurs very small storage and computation overhead, as analyzed in this section.

5.5.1. Storage For the merging scheme, the only hardware structure added is the 1-bit *merge counter* per block. In our secure processor configuration (Section 6), each block is 64 bytes. So, the 1-bit counter incurs less than 0.2% storage overhead for the ORAM.

For the breaking scheme, 3 bits need to be added to each basic block: 1 bit for the *break counter*, the *predict bit* and the *hit bit* respectively. The *break counter* only needs to be stored in the ORAM while the other two bits need to be stored in both ORAM and cache. This incurs less than 0.6% storage overhead for the ORAM and less than 0.4% for the cache.

If both merging and breaking mechanisms are used, the overall storage overhead would be less than 0.8%, which is still very small.

5.5.2. Computation For the merging scheme, when a block B is loaded to the LLC, we need to check if the neighbor block B' exists in the cache (cf. Section 5.2). In our configuration, this requires that the LLC be probed for B'. Only the tag array of a cache needs to be accessed for presence test. Note that the presence test would not be in the critical path of the ORAM access since it can be done while the path is read in, which may take hundreds to thousands of cycles.

For the breaking scheme, more computation is involved. After super block B is loaded into LLC, the break counter of B should be reconstructed. This can be done by accessing all the basic blocks in B and collecting the break counter bits. Then, the break counter is updated based on the *prefetch bit* and *hit bit* of each block in B, which requires minimal computation. Again, all the computation can be largely overlapped with the path load/store process. Finally, when a block is accessed, the *prefetch bit* and *hit bit* may be updated accordingly. But this computation is minimal and can be overlapped with the cache access.

In summary, the extra computation required for the dynamic super block scheme is very small. All this computation is not on the critical path thus will not add overhead to the performance. In terms of power, only several cache lookups and simple logic operations are added to the chip. This is negligible compared to the amount of power spent in path read/write and data encryption/decryption in an ORAM access.

[4]Experimental results show that *sbsize*^{2} performs better than *sbsize*.

5.6. Security of Dynamic Super Block

The threat model under discussion is identical to prior ORAM work. We claim that ORAM with 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 sacrifice any security.

Following the security of 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. Dynamic super block scheme is secure for fixed super block sizes. In order to demonstrate the security of the whole scheme, we only need to shown 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 $B = (B_1, B_2)$. Counters are encrypted and stored in position map ORAM. The counter update and merging operation themselves happen inside the tamper-resistant processor, and do not leak information. After merging, both blocks are mapped to a same leaf s which is a new random number and indistinguishable from s_1 and s_2. From the adversary's point of view, the leaves that are accessed in the ORAM are not linkable to each other. Thus the adversary is not able to figure out which ORAM access involves a merging or whether merging happens 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 random independent leaves. When one of the sub-blocks is accessed later, the leaf being accessed will be indistinguishable and unlinkable to the leaf of the other half or to leaf s. This indicates that when or whether breaking happens cannot be learned by observing ORAM access sequence.

To this point, dynamic super block scheme does not leak any more information through access patterns. 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.

Though 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 [8]. Virtually no literature on ORAM considers timing attacks (i.e., when ORAM accesses happen or the total number of accesses) and all ORAMs break under timing attacks. In order to protect timing channel, periodic ORAM accesses need to be adopted ([7]), which can be easily added on top of ORAM with super blocks. We evaluate this design point in Section 7.4.

To conclude, the dynamic super block scheme or super

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	64 bytes
Main Memory configuration	
ORAM Capacity	8 GB
Number of ORAM hierarchies	5
Data ORAM basic block size	64 Bytes
Position map ORAM basic block size	32 Bytes
DRAM bandwidth	20 GB/s
Path ORAM latency	1728 cycles
conventional DRAM latency	50 cycles

block scheme in general maintains the same security level as conventional ORAMs. No extra leakage is introduced.

6. Methodology

6.1. Processor and ORAM Model

Graphite [17] is used as the simulator in our experiments. Graphite simulates a tiled multicore chip. The hardware configurations are listed in Table 1. We assume there is only one memory controller on the chip. While insecure DRAM model can exploit the bank level parallelism and issue multiple memory requests at the same time (according to Graphite DRAM model), all ORAM accesses must be serialized (cf. Section 2.6).

We will use Splash-2 [33] benchmarks to evaluate different ORAM prefetching techniques. All the applications are run to completion.

6.2. Metrics

Two main metrics are used to measure different Path ORAM schemes: *Performance Speedup* and *I/O traffic*. *Performance* is measured as the number of CPU cycles required to run the program to completion.

I/O traffic here is defined as the total number of bytes transferred between Path ORAM and the secure processor. This metric is also a measure of energy efficiency. The dominant component of power dissipation in Path ORAM comes from DRAM accesses and encryption/decryption operations. Both are proportional to ORAM I/O traffic. Thus I/O traffic is a good indication of total system energy consumption since ORAM power usually dominates in a secure processor setting.

7. Evaluation

We evaluate different ORAM prefetch techniques in this section. Specifically, we will first evaluate the performance of a traditional stream prefetcher on both DRAM and Path ORAM (Section 7.1). Then we will show the performance of different super block schemes with Splash-2 benchmark (Section 7.2). Different variations of the dynamic super block scheme will

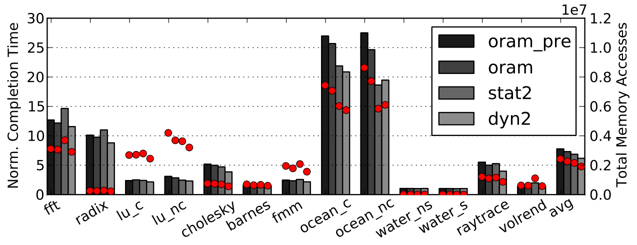


Figure 6: Completion time (normalized to DRAM) and total memory accesses (red dots) of different prefetching techniques.

then be explored in the sensitivity study section (Section 7.3). Finally, we evaluate the impact of having periodic ORAM accesses on these algorithms (Section 7.4).

Three baseline designs are used for comparison: the insecure baseline using normal DRAM, baseline Path ORAM without super block (*oram*) and static super block scheme (*stat*).

Figure 6 shows the performance of different prefetching techniques on ORAM over Splash-2 benchmarks. The bars show the benchmark completion time normalized to the insecure baseline which uses DRAM as the main memory. The red dots show the total number of memory accesses which indicates the power consumption of the system. Unless otherwise stated, in the rest of this section, dynamic super block scheme would mean *adaptive merging* and *adaptive breaking* as defined in Section 5.4. We also assume *sbsize* $= 2$ for all super block schemes in this figure.

7.1. Traditional Prefetching on Path ORAM

As discussed in Section 3, traditional prefetching does not help much in the context of ORAM. This conclusion is verified in Figure 6. For most of the benchmarks, prefetching makes performance even worse than a basic ORAM design without prefetching. For an ORAM system, the memory is serving normal memory requests all the time and there is no space to fit in a prefetch memory access. Inserting in a prefetch request at best does not make performance worse. But most of the time, prefetch requests will block normal accesses thus hurting performance.

7.2. Splash-2 Benchmarks

As pointed out in Section 4.2, 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*). On some benchmarks(e.g. fft, volrend), static super block scheme actually gets worse performance than 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, dynamic super block scheme outperforms the baseline ORAM on all the benchmarks we evaluate here. On average, the overall performance gain is 18% com-

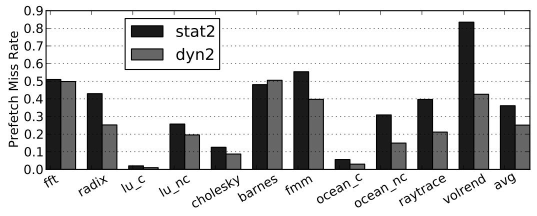


Figure 7: Miss rate for different Path ORAM schemes on Splash-2 benchmarks.

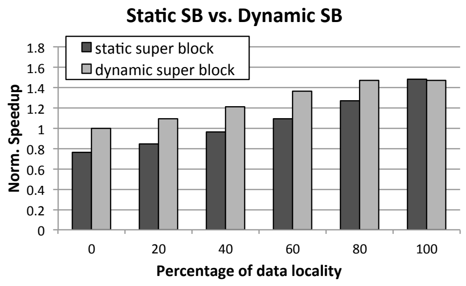


Figure 8: Sweep the level of locality with $sbsize = 2$. Speedup normalized to baseline ORAM.

pared to the baseline ORAM and 12% compared to static super block scheme. The performance gain is most prominent for memory bound benchmarks. For *cholesky* for example, the performance gain is 29%. At the same time, the number of ORAM accesses is reduced by 18.4% and 12.7% compared to baseline ORAM and static super block scheme respectively. The reduction of ORAM accesses can be interpreted as the energy reduction of the system.

Figure 7 shows the prefetch miss rate of static and dynamic super block schemes. *water-spatial* and *water-nsquared* are not shown here since they are too compute bound and 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). On average, dynamic super block lowers the miss rate over all benchmarks from 36.1% to 25.1%.

7.3. Sensitivity Study

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

7.3.1. Locality Figure 8 compares dynamic vs. static super block schemes with different level of data locality. We use a synthetic benchmark to enable locality sweep. The result confirms our understanding of the static super block scheme: it only works when there is good spatial locality in the application and performs worse than baseline ORAM if locality is bad. Dynamic super block scheme, on the other hand, always outperforms both baseline ORAM and static super block scheme.

7.3.2. Super Block Size The discussion in Section 5 is general to any super block size. But we have only evaluated $sbsize = 2$ to this point. Figure 9 sweeps the size of the super block

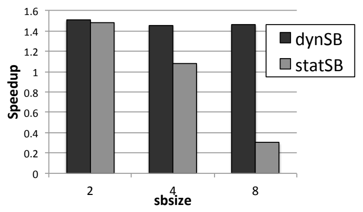
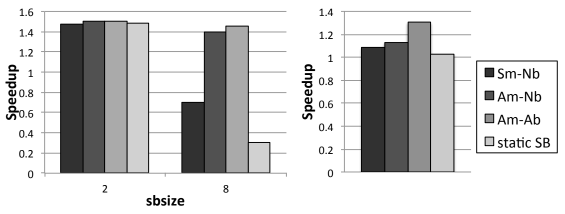


Figure 9: Sweep super block size.



(a) Perfect spatial locality (b) Phase change benchmark

Figure 10: Merging and breaking strategies.

size (*sbsize*) in different super block schemes (for dynamic super block, *sbsize* is the maximum super block size). In this experiment, we run a synthetic benchmark which has 100% spatial locality.

As *sbsize* increases, performance of the static super block scheme 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 5.4. Once the background eviction rate is too high, super block merging is stopped and background eviction rate is kept low.

7.3.3. Merge/Break strategy As discussed in Section 5.4, there are a class of different merge/break strategies in dynamic super block scheme. Specifically, we have 2 merging strategies: *static thresholding* and *adaptive thresholding*, and 3 breaking strategies: *no break*, *static thresholding* and *adaptive thresholding*.

We will first compare *static merging* with *adaptive merging* and then show the effect of block breaking on top of that.

In Figure 10, Sm, Am stands for static and adaptive merging and Nb, Ab stands for no breaking and adaptive breaking respectively. We use a synthetic benchmark with perfect locality. In Figure 10 (a), the effect of different merging strategies is studied. When $sbsize = 2$, all strategies have very good performance including static super block as the locality is perfect. When $sbsize = 8$, the benefit of dynamic merging vs. static merging starts to show up. Since the static strategy merges blocks whenever they show locality, it may merge too many blocks and creates too many background evictions hurting performance. Adaptive merging will throttle the merging process when background eviction rate is too high thus solving this problem.

In Figure 10 (b), we evaluate the same merge/break strategies on a synthetic benchmark that has phase change behavior. In certain phases of the program, the data would be accessed

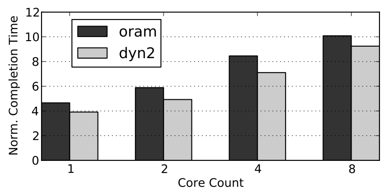


Figure 11: Sweep the number of cores per ORAM controller. (normalized to DRAM)

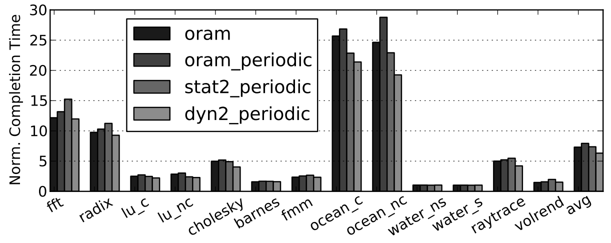


Figure 12: Periodic ORAM accesses. O_{int} = 100 cycles.

sequentially (with locality) while in other phases the data would be accessed randomly (without locality). In this case, block 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 can reduce the number of background evictions and improve prefetch hit rate.

7.3.4. Core Count In Figure 11, we run baseline ORAM and dynamic super block scheme on Splash-2 benchmarks with different number of cores in the system. For clarity, only the average completion time (normalized to DRAM) is shown in this figure.

The number of cores in the system directly affects the pressure at the ORAM controller. In general, the more cores running simultaneously, the shorter total runtime will be. But the total number of ORAM accesses does not decrease that much, which leads to more contention at the ORAM controller. However, the performance gain of dynamic super block scheme over baseline ORAM remains consistent. This indicates that the performance of dynamic super block is stable over a large range of memory boundedness.

7.4. Protecting Timing Channel

As pointed out in Section 5.6, the ORAM definition does not try to protect timing attacks. And an adversary may still learn lots of information by observing the timing of memory accesses. In order to achieve better security in practice, we also need to have periodic ORAM access in order to protect the timing channel. Both dynamic and static super block schemes can be easily integrated with periodic ORAM accesses. The simulation results are shown in Figure 12. O_{int} is defined the number of cycles between 2 consecutive ORAM accesses, which is chosen to be 100 cycles in this experiment.

Two insights can be derived from this Figure.

First, adding periodicity in ORAM accesses does not significantly hurt performance. On average, only 8.4% of performance degradation is incurred by forcing accesses to be periodic. Part of the reason is that the O_{int} is chosen to be very small in our evaluations thus ORAM bandwidth is almost maximized.

Second and more important, the performance gain of the dynamic super block scheme becomes even higher after periodicity is added. The speedup over baseline Path ORAM becomes 25.5% and the speedup over the static super block scheme becomes 16.6%. The speedup over baseline ORAM can be as much as 49.4% on memory bound benchmarks (*ocean_non_contiguous*). Periodic ORAM accesses introduce the concept of *dummy access* into the system. These *dummy accesses* help to flush blocks in the stash back to ORAM similar to background evictions. The dynamic super block scheme has much higher background eviction rate than the baseline ORAM so dummy accesses is more helpful. This explains the extra performance gain.

Since the ORAM has strict periodic access pattern, the power consumption of different ORAM schemes would be the same. However, the performance advatage of dynamic super block can be easily transalated to power advantage by setting O_{int} high. Which slows down the system but makes it more power efficient.

8. 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.

8.1. ORAM Optimization

Previous work [22] has explored the Path ORAM design space and proposed a static super block scheme. We are using their optimized Path ORAM as the baseline in our work. We extend the static super block scheme to a dynamic super block scheme which is significantly more stable and has better performance.

[16] has exploited the fact that ORAM operations can be easily parallelized and be assigned to multiple trusted coprocessors. The optimization techniques proposed in this paper are orthogonal to [16] and can also be applied to their setting to have similar performance gain.

Though we used Path ORAM for discussion in this paper, the optimization techniques proposed are not constrained to Path ORAM. For example, [24] has a similar binary tree structure as Path ORAM. After adding background eviction to [24], their ORAM scheme 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. Also, all hierarchical ORAM schemes can benefit from having smaller Z in position map ORAMs.

8.2. 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 [25, 5, 4, 31], 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 ORAM context. And super block scheme takes advantage of the 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.

9. Conclusion

A novel ORAM prefetching scheme: *dynamic super block* is proposed for the first time in this paper. The implementation details are discussed and the design space is comprehensively explored. We show that dynamic super block scheme is much more stable than static super blocks over different benchmarks and program behaviors. On Splash-2 benchmarks, the proposed techniques improve a system with ORAM by 18% in terms of completion time and reduces energy consumption by 18.4%. After adding timing protection to the ORAM, the overall performance gain is 25.5% over the baseline ORAM and 16.6% over the best static super block scheme previously proposed.

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.

[2] D. Boneh, D. Mazieres, and R. A. Popa, "Remote oblivious storage: Making oblivious RAM practical," Manuscript, , 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] 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.

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

[7] 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*, Oct. 2012, pp. 3–8.

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

[9] 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.

[10] ——, "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.

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

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

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

[20] 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.

[21] R. Rahman, *Intel® Xeon Phi*^{TM} *Coprocessor Architecture and Tools: The Guide for Application Developers*. Apress, 2013.

[22] 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 2012/76.

[23] 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.

[24] 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.

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

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

[27] 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.

[28] 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.

[29] 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.

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

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

[32] 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.

[33] 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.

[34] 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.