Burst ORAM: Minimizing ORAM Response Times for Bursty Access Patterns

Jonathan Dautrich
University of California, Riverside

Emil Stefanov
University of California, Berkeley

Elaine Shi
University of Maryland, College Park

Abstract

We present Burst ORAM, the first oblivious cloud storage system to achieve both practical response times and low total bandwidth consumption for bursty workloads. For real-world workloads, Burst ORAM can attain response times that are nearly optimal and orders of magnitude lower than the best existing ORAM systems by reducing online bandwidth costs and aggressively rescheduling shuffling work to delay the bulk of the IO until idle periods.

We evaluate our design on an enterprise file system trace with about 7,500 clients over a 15 day period, comparing to an insecure baseline encrypted block store without ORAM. We show that when baseline response times are low, Burst ORAM response times are comparably low. In a 32TB ORAM with 50ms network latency and sufficient bandwidth capacity to ensure 90% of requests have baseline response times under 53ms, 90% of Burst ORAM requests have response times under 63ms, while requiring only 30 times the total bandwidth consumption of the insecure baseline. Similarly, with sufficient bandwidth to ensure 99.9% of requests have baseline responses under 70ms, 99.9% of Burst ORAM requests have response times under 76ms.

1 Introduction

Cloud computing allows customers to outsource the burden of data management and benefit from economy of scale, but privacy concerns hinder its growth [3]. Encryption alone is insufficient to ensure privacy in storage outsourcing applications, as information about the contents of encrypted records may still leak via data access patterns. Existing work has shown that access patterns on an encrypted email repository may leak sensitive keyword search queries [12], and that accesses to encrypted database tuples may reveal ordering information [5].

Oblivious RAM (ORAM), first proposed in a groundbreaking work by Goldreich and Ostrovsky [8, 9], is a cryptographic protocol that allows a client to provably hide access patterns from an untrusted storage server. Recently, the research community has focused on making ORAM schemes practical for real-world applications [7, 11, 21, 23–25, 27]. Unfortunately, even with recent improvements, ORAMs still incur substantial bandwidth and response time costs.

Many prior ORAM works focus on minimizing bandwidth consumption. Several recent works on cloud-based ORAMs achieve low bandwidth costs using a large amount of client-side storage [11, 23, 24]. Others rely on expensive primitives like PIR [17] or additional assumptions such as trusted hardware [15] or non-colluding servers [22] to reduce bandwidth costs.

To be practical, ORAM must also minimize response times observed by clients for each request. We propose Burst ORAM, a novel ORAM that dramatically reduces response times for realistic workloads with bursty characteristics. Burst ORAM is based on ObliviStore [23], the most bandwidth-efficient existing ORAM.

Burst ORAM uses novel techniques to minimize the *online* work of serving requests and delay *offline* block shuffling until idle periods. Under realistic bursty loads, Burst ORAM achieves orders of magnitude shorter response times than existing ORAMs, while retaining total bandwidth costs less than 50% higher than ObliviStore's.

During long bursts, Burst ORAM's behavior automatically and gracefully degrades to be similar to that of ObliviStore. Thus, even in a worst-case workload, Burst ORAM's response times and bandwidth costs are competitive with those of existing ORAMs.

We simulate Burst ORAM on a real-world corporate data access workload (7,500 clients and 15 days) to show that it can be used practically in a corporate cloud storage environment. We compare against an insecure baseline encrypted block store without ORAM and show that when baseline response times are low, Burst ORAM response times are also low. In a 32TB ORAM with 50ms network latency and sufficient bandwidth capacity to ensure 90% of requests have baseline response times un-

der 53ms, 90% of Burst ORAM requests have response times under 63ms. Similarly, with sufficient bandwidth to ensure 99.9% of requests have baseline responses under 70ms, 99.9% of Burst ORAM requests have response times under 76ms. Existing works exhibit response times on the order of seconds or higher, due to high bandwidth [11, 23, 25, 28] or computation [17] requirements. To our knowledge, our work is the first to evaluate ORAM response times on a realistic, bursty workload.

As in previous ORAM schemes, we do not seek to hide the timing of data requests. Thus, we assume request start times and durations are known. To ensure security, we do not allow the IO scheduler to make use of the data access sequence or other sensitive information. We analyze Burst ORAM security in Section 6.4.

1.1 Burst ORAM Contributions

Burst ORAM introduces several techniques for reducing response times and keeping bandwidth costs low that distinguish it from ObliviStore and other predecessors.

Novel scheduling policies. Burst ORAM prioritizes the *online* work that must be complete before requests are satisfied. If possible, our scheduler delays shuffle work until off-peak times. Delaying shuffle work consumes client-side storage, so if a burst is sufficiently long, client space will fill, forcing shuffling to resume. By this time, there are typically multiple shuffle jobs pending.

We use a greedy strategy to prioritize jobs that free the most client-side space per unit of shuffling bandwidth consumed. This strategy allows us to sustain lower response times for longer during an extended burst.

Reduced online bandwidth costs. We propose a new *XOR technique* that reduces the online bandwidth cost from $O(\log N)$ blocks per request in ObliviStore to nearly 1, where N is the outsourced block count. The XOR technique can also be applied to other ORAM implementations such as SR-ORAM [26] (see Appendix B).

Level caching. We propose a new technique for using additional available client space to store small levels from each partition. By caching these levels on the client, we are able to reduce total bandwidth cost substantially.

1.2 Related Work

Oblivious RAM was first proposed in a seminal work by Goldreich and Ostrovsky [9]. Since then, a fair amount of theoretic work has focused on improving its asymptotic performance [1, 4, 10, 11, 13, 18, 19, 21, 24, 27]. Recently, there has been much work designing and optimizing ORAM for cloud-based storage outsourcing settings, as noted below. Different ORAMs provide varying trade-offs between bandwidth cost, client/server storage, round complexity, and computation.

ORAM has been shown to be feasible for secure (co-) processor prototypes, which prevent information leakage due to physical tampering [6, 15, 16, 20]. Since on-chip trusted cache is expensive, such ORAM schemes need constant or logarithmic client-side storage, such as the binary-tree ORAM [21] and its variants [7, 17, 25].

In cloud-based ORAMs, the client typically has more space, capable of storing $O(\sqrt{N})$ blocks or a small amount of per-block metadata [10, 23, 24, 28] that can be used to reduce ORAM bandwidth costs. Burst ORAM also makes such client space assumptions.

Online and offline costs for ORAM were first made explicit by Boneh et al. [1] They propose a construction that has $O(1)$ online but $O(\sqrt{N})$ overall bandwidth cost. The recent Path-PIR work by Mayberry et al. [17] mixes ORAM and PIR to achieve $O(1)$ online bandwidth cost with an overall bandwidth cost of $O(\log^2 N)$ with constant client memory. Unfortunately, the PIR is still computationally expensive, so their scheme requires more than 40 seconds for a read from a 1TB database [17]. Burst ORAM has $O(1)$ online and $O(\log N)$ overall bandwidth cost, without the added overhead of PIR.

Other ORAMs that do not rely on trusted hardware or non-colluding servers have $\Omega(\log N)$ online bandwidth cost including works by Williams, Sion, et al. [27, 28]; by Goodrich, Mitzenmacher, Ohrimenko, and Tamassia [10, 11]; by Kushilevitz et al. [13]; and by Stefonov, Shi, et al. [21, 23–25]. Burst ORAM handles bursts much more effectively by reducing the online cost to nearly 1 block transfer per block request during a burst, greatly reducing response times.

2 Preliminaries

2.1 Bandwidth Costs

Bandwidth consumption is the primary cost in many modern ORAMs, so it is important to define how we measure its different aspects. Each block transferred between the client and server is a single unit of IO. We assume that blocks are large in practice (at least 1KB), so transferred meta-data (block IDs) have negligible size.

Definition 1 *The* bandwidth cost *of a storage scheme is given by the average number of blocks transferred in order to read or write a single block.*

We identify bandwidth costs by appending X to the number. A bandwidth cost of 2X indicates two blocks transferred per request, which is twice the cost of an unprotected scheme. We consider *online*, *offline*, *effective*, and *overall* IO and bandwidth costs, where each cost is given by the average amount of the corresponding type of IO.

Online IO consists of the block transfers needed before a request can be safely marked as satisfied, assuming the scheme starts with no pending IO. The *online bandwidth cost* of a storage scheme without ORAM is just 1X — the

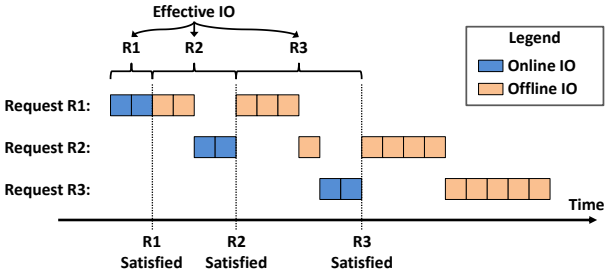

Figure 1: Simplified scheme with sequential IO and contrived capacity for delaying offline IO. 3 requests require same online (2), offline (5), and overall (7) IO. Online IO for R1 is handled immediately, so R1's effective IO is only 2. R2 waits for 2 units of offline IO from R1, so its effective IO is 4. R3 waits for the rest of R1's offline IO, plus one unit of R2's offline IO, so its effective IO is 6.

IO cost of downloading the desired block. In ORAM it may be higher, as additional blocks may be downloaded to hide the requested block's identity.

Offline IO consists of transfers needed to prepare for subsequent requests, but which may be performed after the request is satisfied. Without ORAM, the *offline bandwidth cost* is 0X. In ORAM it is generally higher, as additional *shuffle IO* is needed to obliviously permute blocks in order to guarantee privacy for future requests.

Overall IO / *bandwidth cost* is just the sum of the online and offline IO / bandwidth costs, respectively.

Effective IO consists of all online IO plus any pending offline IO from previous requests that must be issued before the next request can be satisfied. Without ORAM, effective IO and online IO are equal. In traditional ORAMs, offline IO is issued immediately after each request's online IO, so effective and overall IO are equal. In Burst ORAM, we delay some offline IO, reducing each request's effective IO as illustrated in Figure 1. Smaller effective costs mean less IO between requests, and ultimately shorter response times.

ORAM reads and writes are indistinguishable, so writes have the same bandwidth costs as reads.

2.2 Response Time

The *response time* of a block request (ORAM read/write operation) is defined as the lapse of wall-clock time between when the request is first issued by the client and when the client receives a response. The *minimum* response time is the time needed to perform all online IO. Response times increase when offline IO is needed between requests, increasing effective IO, or when requests are issued rapidly in a burst, delaying later requests.

2.3 ObliviStore ORAM

Burst ORAM builds on ObliviStore [23], so we give an overview of the scheme here. A full description of the ObliviStore system and its ORAM algorithm spans about 55 pages [23, 24], so we describe it at a high level, focusing only on components relevant to Burst ORAM.

Partitions and levels. ObliviStore stores N logical data blocks. Each block is encrypted using a standard symmetric key encryption scheme before it is stored on the server. Every time a block is uploaded by the client, it is re-encrypted using a new nonce to prevent linking.

ObliviStore securely splits blocks into $O(\sqrt{N})$ *partitions* of $O(\sqrt{N})$ blocks each. Each partition is an ORAM consisting of $O(\log N)$ *levels* with $2,4,8,\ldots,O(\sqrt{N})$ blocks each. Newly created levels are filled with half encrypted real blocks and half encrypted dummies, randomly permuted so that reals and dummies are indistinguishable to the server. Each level is occupied only half the time on average. The client has space to store $O(\sqrt{N})$ blocks and the locations of all N blocks.

Requests. When the client makes a block request, whether a read or write, the block must first be downloaded from the appropriate partition. To maintain obliviousness, ObliviStore must fetch one block from every non-empty level in the target partition ($O(\log N)$ blocks of *online* IO). Only one fetched block is real, and the rest are dummies, except in the case of early shuffle reads described below. Once a dummy is fetched, it is discarded, and new dummies are created later as needed. ObliviStore securely processes multiple requests in parallel, enabling full utilization of available bandwidth capacity.

Eviction. Once the real block is fetched, it is updated or returned to the client as necessary, then ranodmly assigned to a new partition p. The block is not immediately uploaded, but is scheduled for *eviction* to p and stored in a client-side *data cache*. An independent eviction process later obliviously evicts the block from the cache to p. The eviction triggers a write operation on p's ORAM, which creates or enlarges a *shuffling job* for p.

Shuffling Jobs. Each partition p has at most one pending *shuffle job*. A job consists of downloading up to $O(\sqrt{N})$ blocks from p, permuting them on the client with recent evictions and new dummies, and uploading. Shuffle jobs incur *offline* IO, and vary in *size* (amount of IO) from $O(1)$ to $O(\sqrt{N})$. Intuitively, to ensure that non-empty levels have at least one dummy left, we must re-shuffle a level once half its blocks have been removed. Larger levels need shuffling less often, so larger jobs occur less frequently, keeping offline bandwidth costs at $O(\log N)$.

Shuffle IO scheduling. A fixed amount of $O(\log N)$ shuffle IO is performed after each request to amortize the work required for large jobs. The IO for jobs from multi-

ple partitions may be executed in parallel: while waiting on reads for one partition, we may issue reads or writes for another. Jobs are started in the order they are created.

Early shuffle reads. *Early shuffle reads*, referred to as *early cache-ins* or *real cache-ins* in ObliviStore, occur when a request needs to fetch a block from a level, but at least half the level's original blocks have been removed. In this case, we cannot guarantee that any dummies remain. Thus, early shuffle reads must be treated as real blocks and stored separately by the client until they are returned to the server as part of a shuffle job. We call such reads *early shuffle reads* since the blocks would have eventually been read during a shuffle job. Early shuffle reads are infrequent, but made possible since ObliviStore performs requests while shuffling is in progress.

Level compression. ObliviStore uses a technique called *level compression* [24] to compress blocks uploaded during shuffling. It allows the client to upload k real and k dummy blocks using only k blocks of bandwidth without revealing which k are dummies. Level compression reduces only the offline (shuffling) bandwidth cost.

3 Overview of our Approach

Traditional ORAMs focus on reducing average and worst-case *overall bandwidth costs* (per-request overall IO). However, even the most bandwidth-efficient schemes [23, 24] suffer from a 20X–35X bandwidth cost.

In this paper, we instead focus on reducing *effective* IO by reducing online IO and delaying offline IO. We can then satisfy bursts of requests quickly, delaying most IO until idle periods. Figure 2 illustrates this concept.

Our approach allows many bursts to be satisfied with nearly a 1X effective bandwidth cost. That is, during the burst, we transfer just over one block for every block requested. After the burst we do extra IO to catch up on shuffling and prepare for future requests. Our approach maintains an overall bandwidth cost less than 50% higher than [23, 24] in practice (see Figure 12 in Section 7).

Bursts. Intuitively, a burst is a period of frequent block requests from the client preceded and followed by relatively idle periods. Many real-world workloads exhibit bursty patterns (e.g. [2, 14]). Often, bursts are not discrete events, such as when multiple network file system users are operating concurrently. Thus we handle bursts fluidly: the more requests issued at once, the more Burst ORAM tries to delay offline IO until idle periods.

Challenges. We are faced with two key challenges when building a burst-friendly ORAM system. The first is ensuring that we maintain security. A naive approach to reducing online IO may mark requests as satisfied before enough blocks are read from the server, leaking information about the requested block's identity.

The second challenge is ensuring that we maximally utilize client storage and available bandwidth while avoiding deadlock. An excessively aggressive strategy that delays too much IO may use so much client space that we run out of room to shuffle. It may also under-utilize available bandwidth, increasing response times. On the other hand, an overly conservative strategy may under-utilize client space or perform shuffling too early, delaying online IO and increasing response times.

Techniques and Outline. In Burst ORAM, we address these challenges by combining several novel techniques. In Section 4 we introduce our XOR technique for reducing online bandwidth cost to nearly 1X. We also describe our techniques for prioritizing online IO and delaying offline/shuffle IO until client memory is nearly full. In Section 5 we show how Burst ORAM prioritizes efficient shuffle jobs in order to delay the bulk of the shuffle IO even further, ensuring that we minimize effective IO during long bursts. We then introduce a technique for using available client space to cache small levels locally to reduce shuffle IO in both Burst ORAM and ObliviStore.

In Section 6 we discuss the system-level techniques used in Burst ORAM, and present its design in detail. In Section 7, we evaluate Burst ORAM's performance through micro-benchmarks and extensive simulations.

4 Prioritizing and Reducing Online IO

Existing ORAMs require high online and offline bandwidth costs to obscure access patterns. ObliviStore must fetch one block from every level in a partition (see Section 2.3), requiring $O(\log N)$ online IO per request. Figure 3 (left) illustrates this behavior. After each request, ObliviStore also requires $O(\log N)$ offline/shuffle IO. Since ObliviStore issues online and offline IO before satisfying the next request, its effective IO is high, leading to large response times during bursts. Other ORAMs work differently, such as Path ORAM [25] which organizes data as a tree, but still have high effective costs. We now show how Burst ORAM achieves lower effective bandwidth costs and response times than ObliviStore.

4.1 Prioritizing Online IO

One way we achieve low response times in Burst ORAM is by prioritizing online IO over shuffle IO. That is, we suppress shuffle IO during bursts, delaying it until idle periods. Requests are satisfied once online IO finishes,[1] so prioritizing online IO allows us to satisfy all requests before any shuffle IO starts, keeping response times low even for later requests. Figure 2 illustrates this behavior.

During the burst, we continue processing requests by fetching blocks from the server, but since shuffling is suppressed, no blocks are uploaded. Thus, we must resume shuffling once client storage fills. Section 5.2 dis-

[1] Each client write also incurs a read, so writes still incur online IO.

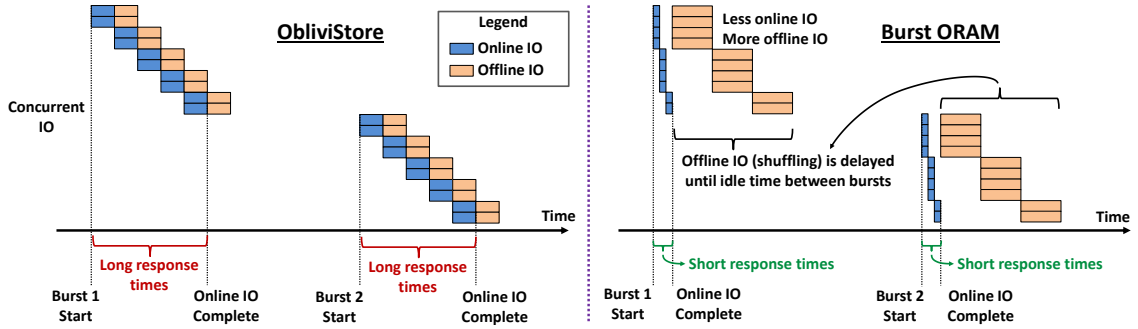

Figure 2: **Reducing response time.** Because Burst ORAM (right) does much less online IO than ObliviStore (left) and delays offline IO, it is able to respond to ORAM requests much faster. In this (overfly simplified) illustration, the bandwidth capacity is enough to transfer 4 blocks concurrently. Both ORAM systems do the same amount of IO.

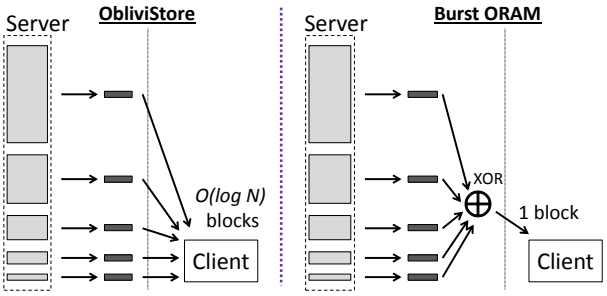

Figure 3: **Reducing online cost.** In ObliviStore (left) the online bandwidth cost is $O(\log N)$ blocks of IO on average. In Burst ORAM (right), we reduce online IO to only one block, improving handling of bursty traffic.

cusses how to delay shuffle IO even further. Section 6 details changes from the ObliviStore design required to avoid deadlock and fully utilize client space.

When available bandwidths are large and bursts are short, the response time saved by prioritizing online IO is limited, as most IO needed for the burst can be issued in parallel. However, when bandwidth is limited or bursts are long, the savings can be substantial. With shuffle IO delayed until idle times, online IO dominates the effective IO, becoming the bottleneck during bursts. Thus we can further reduce response times by reducing online IO.

4.2 XOR Technique: Reducing Online IO

We introduce a new mechanism called the *XOR technique* that allows the Burst ORAM server to combine the $O(\log N)$ blocks fetched during a request into a single block that is returned to the client (Figure 3 right), reducing the online bandwidth cost to $O(1)$.

If we fetched only the desired block, we would reveal its identity to the server. Instead, we XOR all the blocks together and return the result. Since there is at most one real block among the $O(\log N)$ returned, the client can locally reconstruct the dummy block values and XOR them with the returned block to recover the encrypted real block. XOR technique steps are shown in Figure 4.

4.2.1 XOR Technique Details

In Burst ORAM, as in ObliviStore, each request needs to retrieve a block from a single *partition*, which is a simplified hierarchical ORAM resembling those in [9]. The hierarchy contains $L \approx \frac{1}{2}\log_2 N$ levels with real-block capacities $1, 2, 4, \ldots, 2^{L-1}$ respectively.

To retrieve a requested block, the client must fetch exactly one block from each of the L levels. The XOR technique requires that the client be able to reconstruct dummy blocks, and that dummies remain indistinguishable from real blocks. We achieve this property by encrypting a real block b residing in partition p, level ℓ, and offset off as $\mathsf{AES}_{sk_{p,\ell}}(off||B)$. We encrypt a dummy block residing in partition p, level ℓ, and offset off as $\mathsf{AES}_{sk_{p,\ell}}(off)$. The key $sk_{p,\ell}$ is specific to partition p and level ℓ, and is randomized every time ℓ is rebuilt.

For simplicity, we start by considering the case without early shuffle reads. In this case, exactly one of the L blocks requested is the encryption of a real block, and the rest are encryptions of dummy blocks. The server XORs all L encrypted blocks together into a single block X_Q that it returns to the client. The client knows which blocks are dummies, and knows p, ℓ, off for each block, so it reconstructs all the encrypted dummy blocks and XORs them with X_Q to obtain the encrypted requested/real block.

4.2.2 Handling early shuffle reads

An early shuffle read occurs when we need to read from a level with no more than half its original blocks remaining. Since such early shuffle reads may be real blocks, they cannot be included in the XOR. Fortunately, the number of blocks in a level is public, so the server already knows which levels will cause early shuffle reads. Thus, the server simply returns early shuffle reads individually, then XORs the remaining blocks, leaking no information about the access sequence.

1. Client issues block requests to server, one per level
2. Server, to satisfy request
   (a) Retrieves and returns early shuffle reads
   (b) XORs remaining blocks together into single *combined* block and returns it
3. Client, while waiting for response
   (a) Regenerates encrypted dummy block for each non-early-shuffle-read
   (b) XORs all dummies to get *subtraction* block
4. Client receives combined block from server and XORs with subtraction block to get requested block
5. Client decrypts requested block

Figure 4: XOR Technique Steps

Since each early shuffle read block must be transferred individually, early shuffle reads increase online IO. Fortunately, early shuffle reads are rare, even while shuffling is suppressed during bursts, so the online bandwidth cost stays under 2X and near 1X in practice (see Figure 7).

4.2.3 Comparison with ObliviStore

ObliviStore uses *level compression* to reduce shuffle IO. When the client uploads a level to the server, it first compresses the level down to the combined size of the level's real blocks. Since half the blocks are dummies, half the upload shuffle IO is eliminated. For details on level compression and its security, see [24].

Unfortunately, Burst ORAM's XOR technique is incompatible with level compression due to discrepancies in the ways dummy blocks must be formed. The XOR technique requires that the client be able to reconstruct dummy blocks locally, so in Burst ORAM, each dummy's position determines its contents. In level compression, each level's dummy block contents are a function of the level's real block contents. Since the client cannot know the contents of all real blocks in the level, it cannot reconstruct the dummies locally.

Level compression and the XOR technique yield comparable overall IO reductions, though level compression performs slightly better. For example, the experiment in Figure 8 incurs roughly 23X and 26X overall bandwidth cost using level compression and XOR respectively. However, the XOR technique reduces *online* IO, while level compression reduces *offline* IO, so the XOR technique is more effective at reducing response times.

5 Scheduling and Reducing Shuffle IO

In Burst ORAM, once client space fills, we must start shuffling in order to return blocks to the server and continue the burst. If we are not careful about shuffle IO scheduling, we may immediately start doing large amounts of IO, dramatically increasing response times.

In this section, we show how Burst ORAM schedules shuffle IO so that jobs that free the most client space using the least shuffle IO are prioritized. Thus, at all times, Burst ORAM issues only the minimum amount of effective IO needed to continue the burst, keeping response times lower for longer. We also show how to reduce overall IO by locally caching the smallest levels from each partition. We start by defining *shuffle jobs*.

5.1 Shuffle Jobs

In Burst ORAM, as in ObliviStore, shuffle IO is divided into per-partition *shuffle jobs*. Each job represents the work needed to shuffle a partition p and upload blocks evicted to p. A shuffle job is defined by five entities:

- A partition p to which the job belongs
- Blocks evicted to but not yet returned to p
- Levels to read blocks from
- Levels to write blocks to
- Blocks already read from p (early shuffle reads)

Each shuffle job moves through three phases:

Creation Phase. We create a shuffle job for p when a block is evicted to p following a request. Every job starts out *inactive*, meaning we have not started work on it. If another block is evicted to p, we update the sets of eviction blocks and read/write levels in p's inactive job.

When Burst ORAM *activates* a job, it moves the job to the *Read Phase*, freezing the eviction blocks and read/write levels. Subsequent evictions to p will create a new *inactive* shuffle job. At any time, there is at most one active and one inactive shuffle job for each partition.

Read Phase. Once a shuffle job is activated, we begin fetching all blocks still on the server that need to be shuffled. That is, all previously unread blocks from all the job's read levels. Once all such blocks are fetched, they are *shuffled* with all blocks evicted to p and any early shuffle reads from the read levels. Shuffling consists of adding/removing dummies, pseudo-randomly permuting the blocks, and then re-encrypting each block. Once shuffling completes, we move the job to the *Write Phase*.

Write Phase. Once a job is shuffled we begin storing all shuffled blocks to the job's write levels on the server. Once all writes finish, the job is marked complete, and Burst ORAM is free to activate p's inactive job, if any.

5.2 Prioritizing Efficient Jobs

Since executing shuffle IO delays the online IO needed to satisfy requests, we can reduce response times by doing as little shuffling as is needed to free up space. The hope is that we can delay the bulk of the shuffling until an idle period, so that it does not interfere with pending requests.

By the time client space fills, there will be many partitions with inactive shuffle jobs. Since we can choose jobs in any order, we can minimize the up-front shuffling work by prioritizing the most *efficient* shuffle jobs: those

that free up the most client space per unit of shuffle IO. The space freed by completing a job for partition p is the number of blocks evicted to p plus the number of early shuffle reads from the job's read levels. Thus, we can define shuffle job efficiency as follows:

$$\text{Job Efficiency} = \frac{\#\text{ Evictions} + \#\text{ Early Shuffle Reads}}{\#\text{ Blocks to Read} + \#\text{ Blocks to Write}}$$

Job efficiencies vary substantially. Most jobs start with 1 eviction and 0 early shuffle reads, so their relative efficiencies are determined strictly by the sizes of the job's read and write levels. If the partition's bottom level is empty, no levels need be read, and only the bottom must be written, for an overall IO of 2 an an efficiency of 0.5. If instead the bottom 4 levels are occupied, all 4 levels must be read, and the 5th level written, for an total of roughly 15 reads and 32 writes, yielding a much lower efficiency of just over 0.02. Both jobs free equal amounts of space, but the higher-efficiency job uses less IO.

Since small levels are written more often than large ones, efficient jobs are common. Further, by delaying an unusually inefficient job, we give it time to accumulate more evictions. While such a job will also accumulate more IO, the added write levels are generally small, so the job's efficiency tends to improve with time. Thus, prioritizing efficient jobs reduces shuffle IO during the burst, thereby reducing response times.

Unlike Burst ORAM, ObliviStore does not use client space to delay shuffling, so there are fewer shuffle jobs to choose from at any one time. Thus, job scheduling is less important and jobs are chosen in creation order. Since ObliviStore is concerned with throughput, not response times, it has no incentive to prioritize efficient jobs.

5.3 Reducing Shuffle IO via Level Caching

Since small, efficient shuffle jobs are common, Burst ORAM spends a lot of time accessing small levels. If we use client space to locally cache the smallest levels of each partition, we can eliminate the shuffle IO associated with those levels entirely. Since levels are shuffled with a frequency inversely proportional to their size, each is responsible for roughly the same fraction of shuffle IO. Thus, we can greatly reduce shuffle IO by caching even a few levels from each partition. Further, since caching a level eliminates its early shuffle reads, which are common for small levels, caching can also reduce online IO.

We are therefore faced with a tradeoff between using client space to store requested blocks, which reduces response times for short bursts, and using it for local level caching, which reduces overall bandwidth cost.

5.3.1 Level Caching in Burst ORAM

In Burst ORAM, we take a conservative approach, and cache only as many levels as are guaranteed to fit in the worst case. More precisely, we identify the maximum number λ such that the client could store all real blocks from the smallest λ levels of every partition even if all were full simultaneously. We cache levels by only updating an inactive job when the number of evictions is such that all the job's write levels have index at least λ.

Since each level is only occupied half the time, caching λ levels consumes at most half of the client's space on average, leaving the rest for requested blocks. As we show experimentally in Section 7, level caching greatly reduces overall bandwidth cost, and can even reduce response times since it avoids early shuffle reads.

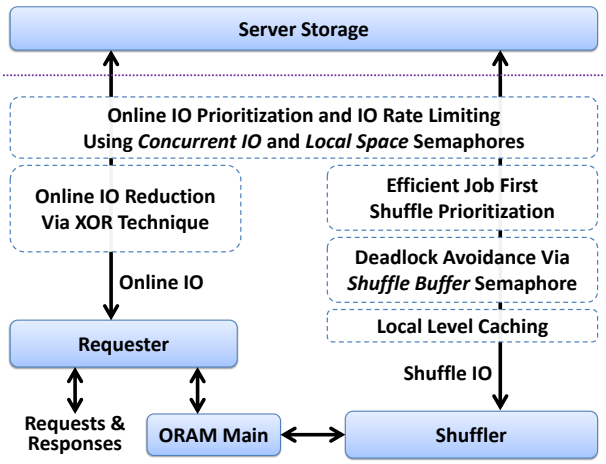

Figure 5: **Burst ORAM Architecture.** Solid boxes represent key system components, while dashed boxes represent functionality and the effects of the system on IO.

6 Detailed Burst ORAM Design

The Burst ORAM design is based on ObliviStore, but incorporates many fundamental functional and system-level changes. For example, Burst ORAM replaces or revises all the semaphores used in ObliviStore to achieve our distinct goal of online IO prioritization while maintaining security and avoiding deadlock. Burst ORAM also maximizes client space utilization, implements the XOR technique to reduce online IO, revises the shuffler to schedule efficient jobs first, and implements level caching to reduce overall IO.

6.1 Overall Architecture

Figure 5 presents the basic architecture of Burst ORAM, highlighting key components and functionality. Burst ORAM consists of two primary components, the online *Requester* and the offline *Shuffler*, which are controlled by the main event loop *ORAM Main*. Client-side memory allocation is shown in Figure 6.

ORAM Main accepts new block requests (reads and writes) from the client, and adds them to a *Request*

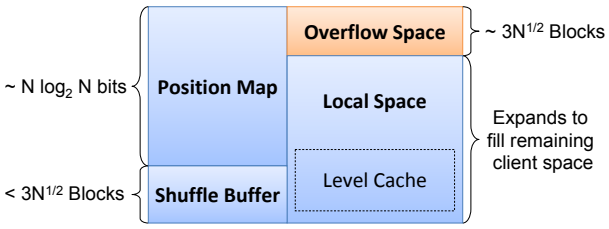

Figure 6: **Burst ORAM Client Space Allocation.** Fixed client space is reserved for the position map and shuffle buffer. A small amount of overflow space is needed for blocks assigned but not yet evicted (*data cache* in [24]). Remaining space is managed by *Local Space* and contains evictions, early shuffle reads, and the level cache.

Queue. On each iteration, ORAM Main tries advancing the Requester first, only advancing the Shuffler if the Requester needs no IO, thereby prioritizing online IO. The Requester and Shuffler use *semaphores* (Section 6.2) to regulate access to network bandwidth and client space.

The *Requester* reads each request from the Request Queue, identifies the desired block's partition, and fetches it along with any necessary dummies. To ensure oblivious behavior, the Requester must wait until all dummy blocks have been fetched before marking the request satisfied. All Requester IO is considered *online*.

The *Shuffler* re-encrypts blocks fetched by the Requester, shuffles them with other blocks, and returns them to the server. The Shuffler is responsible for managing shuffle jobs, including prioritizing efficient jobs and implementing level caching. All IO initiated by the shuffler is considered *offline* or *shuffle* IO.

6.2 Semaphores

Resources in Burst ORAM are managed via *semaphores*, as in ObliviStore. Semaphores are updated using only server-visible information, so ORAM can safely base its behavior on semaphores without revealing new information. Since Burst ORAM gives online IO strict priority over shuffle IO, our use of semaphores is substantially different than ObliviStore's, which tries to issue the same amount of IO after each request. ObliviStore uses four semaphores: *Shuffling Buffer*, *Early Cache-ins*, *Eviction*, and *Shuffling IO*. In Burst ORAM, we use three:

- *Shuffle Buffer* manages client space reserved for blocks from active shuffle jobs, and differs from ObliviStore's *Shuffling Buffer* only in initial value.
- *Local Space* manages all remaining space, combining ObliviStore's *Early Cache-in* and *Eviction* semaphores.
- *Concurrent IO* manages concurrent block transfers based on network link capacity, preventing the Shuffler from starving the Requester. It differs fundamentally from ObliviStore's *Shuffling IO* semaphore, which manages per-request shuffle IO.

***Shuffle Buffer* semaphore.** *Shuffle Buffer* gives the number of blocks that may be added to the client's shuffle buffer. We initialize it to double the maximum partition size (under $2.4\sqrt{N}$ total for $N > 2^{10}$), to ensure that the shuffle buffer is large enough to store at least two in-progress shuffle jobs. When *Shuffle Buffer* reaches 0, the Shuffler may not issue additional reads.

***Local Space* semaphore.** *Local Space* gives the number of blocks that may still be stored in remaining client space (space not reserved for the position map or shuffle buffer). If *Local Space* is 0, the Requester may not fetch more blocks. Blocks fetched by the Requester count toward *Local Space* until their partition's shuffle job is activated and they are absorbed into *Shuffle Buffer*. Once a block moves from *Local Space* to *Shuffle Buffer*, it is considered *free* from the client, and more requests may be issued. The more client space, the higher *Local Space*'s initial value, and the better our burst performance.

***Concurrent IO* semaphore.** *Concurrent IO* is initialized to the network link's block capacity. Queuing a block transfer decrements *Concurrent IO*, and completing a transfer increments *Concurrent IO*. The Shuffler may only initiate a transfer if *Concurrent IO* > 0. However, the Requester may continue to initiate transfers and decrement *Concurrent IO* even if it is negative. This mechanism ensures that no new shuffle IO starts while there is sufficient online IO to fully utilize the link. If no online IO starts, *Concurrent IO* eventually becomes positive, and shuffle IO resumes, ensuring full utilization.

6.3 Detailed System Behavior

We now describe the interaction between ORAM Main, the Requester, the Shuffler, and the semaphores in detail. Accompanying pseudocode can be found in Appendix A.

ORAM Main (Algorithm 1). Incoming read and write requests are asynchronously added to the Request Queue. During each iteration, ORAM Main first tries to advance the Requester, which attempts to satisfy the next request from the Request Queue. If the queue is empty, or *Local Space* too low, ORAM Main advances the Shuffler instead. This mechanism suppresses new shuffle IO during a new burst of requests until the Requester has fetched as many blocks as possible.

For each request, we evict v blocks to randomly chosen partitions, where v is the *eviction rate*, set to 1.3 as in ObliviStore [23]. When evicting, if the Requester has previously assigned a block to be evicted to partition p, then we evict that block. If there are no assigned blocks, then to maintain obliviousness we evict a new dummy block instead. Eviction does not send a block to the server immediately. It merely informs the Shuffler that

the block is ready to be shuffled into p.

Requester (Algorithm 2). To service a request, the Requester first identifies the partition and level containing the desired block. It then determines which levels require early shuffle reads, and which need only standard reads. If *Local Space* is large enough to accommodate the retrieved blocks, the requester issues an asynchronous request for the necessary blocks Else, control returns to ORAM Main, giving the Shuffler a chance to free space.

The server asynchronously returns the early shuffle read blocks and a single *combined* block obtained from all standard-read blocks using the XOR technique (Section 4). The Requester extracts the desired block from the combined block or from an early shuffle read block, then updates the block (write) or returns it to the client (read). The Requester then assigns the desired block for eviction to a randomly chosen partition.

Shuffler (Algorithm 3). The Shuffler may only proceed if *Concurrent IO* > 0. Otherwise, there is pending online IO, which takes priority over shuffle IO, so control returns to ORAM Main without any shuffling.

The Shuffler places shuffle jobs into three queues based on phase. The *New Job Queue* holds inactive jobs, prioritized by efficiency. The *Read Job Queue* holds active jobs for which some reads have been issued, but not all reads are complete. The *Write Job Queue* holds active jobs for which all reads, not writes, are complete.

If all reads have been issued for all jobs in the *Read Job Queue*, the Shuffler *activates* the most efficient job from the *New Job Queue*, if any. *Activating* a job moves it to the *Read Job Queue* and freezes its read/write levels, preventing it from being updated by subsequent evictions. It also moves the job's eviction and early shuffle read blocks from *Local Space* to *Shuffle Buffer*, freeing up *Local Space* to handle online requests. By ensuring that all reads for all active jobs are issued before activating new jobs, we avoid hastily activating inefficient jobs.

The Shuffler then tries to decrement *Shuffle Buffer* to determine whether a shuffle read may be issued. If so, the Shuffler asynchronously fetches a block for a job in the *Read Job Queue*. If not, the Shuffler asynchronously writes a block from a job in the *Write Job Queue* instead. Unlike reads, writes do not require *Shuffle Buffer* space, so they can always be issued. The Shuffler prioritizes reads since they are critical prerequisites to activating new jobs and freeing up *Local Space*. The equally costly writes can be delayed until *Shuffle Buffer* space runs out.

Once all reads for a job complete, the job is *shuffled*: dummy blocks are added as needed, then all are permuted and re-encrypted. We then move the job to the *Write Job Queue*. When all writes finish, we mark the job complete and remove it from the *Write Job Queue*.

6.4 Burst ORAM Security

We assume the server knows *public* information such as the values of each semaphore and the start and end times of each request. The server also knows the level configuration of each partition and the size and phase of each shuffle job, including which encrypted blocks have been read from and written to the server. We must prevent the server from learning the contents of any encrypted block, or anything about which plaintext block is being requested. Thus, the server may not know the location of a given plaintext block, or even the prior location of any previously requested encrypted block.

All of Burst ORAM's publicly visible actions are, or appear to the server to be, independent of the client's sensitive data access sequence. Since Burst ORAM treats the server as a simple block store, the publicly visible actions consist entirely of deciding when to transfer which blocks. Intuitively, we must show that each action taken by Burst ORAM is either deterministic and dependent only on public information, or appears random to the server. Equivalently, we must be able to generate a sequence of encrypted block transfers that appears indistinguishable from the actions of Burst ORAM using only public information. We now show how each Burst ORAM component meets these criteria.

ORAM Main & Client Security. ORAM Main (Algorithm 1) chooses whether to advance the Requester or the Shuffler, and depends on the size of the request queue and the *Local Space* semaphore. Since the number of pending requests and the semaphores are public, ORAM Main is deterministic and based only on public information. For each eviction, the choice of partition is made randomly, and exactly one block will always be evicted. Thus, every action in Algorithm 1 is either truly random or based on public information, and is trivial to simulate.

Requester Security. The Requester (Algorithm 2) must first identify the partition containing a desired block. Since the block was assigned to the partition randomly and this is the first time it is being retrieved since it was assigned, the choice of partition appears random to the server. Within each partition, the requester deterministically retrieves one block from each occupied level. The choice from each level appears random, since blocks were randomly permuted when the level was created.

The Requester singles out early shuffle reads and returns them individually. The identity of levels that return early shuffle reads is public, since it depends on the number of blocks in the level. The remaining blocks are deterministically combined using XOR into a single returned block. Finally, the request is marked satisfied only after all blocks have been returned, so request completion time depends only on public information.

The Requester's behavior can be simulated using only

public information by randomly choosing a partition and randomly selecting one block from each occupied level. Blocks from levels with at most half their original blocks remaining should be returned individually, and all others combined using XOR and returned. Once all blocks have been returned, the request is marked satisfied.

Shuffler Security. As in ObliviStore, Shuffler (Algorithm 3) operations depend on public semaphores. Job efficiency, which we use for prioritizing jobs, depends on the number of blocks to be read and written to perform shuffling, as well as the number of early shuffle reads and blocks already *evicted* (not *assigned*). The identity of early shuffle read levels and the number of evictions is public. Further, the number of reads and writes depends only on the partition's level configuration. Thus, job efficiency and job order depend only on public information. Since the Shuffler's actions are either truly random (e.g. permuting blocks) or depend only on public information (i.e. semaphores), it is trivial to simulate.

Client Space. Since fetched blocks are assigned randomly to partitions, but evicted using an independent process, the number of blocks awaiting eviction may grow. The precise number of such blocks may leak information about where blocks were assigned, so it must be kept secret, and the client must allocate a fixed amount of space dedicated to storing such blocks (see *Overflow Space* in Figure 6). ObliviStore [23] relies on a probabilistic bound on overflow space provided in [24]. Since Burst ORAM uses ObliviStore's assignment and eviction processes, the bound holds for Burst ORAM as well. Level caching uses space controlled by the *Local Space* semaphore, so it depends only on public information.

7 Evaluation

We ran simulations comparing response times and bandwidth costs of Burst ORAM with ObliviStore and an insecure baseline, using real and synthetic workloads.

7.1 Methodology

7.1.1 Baselines

We compare Burst ORAM and its variants against two baselines. The first is the ObliviStore ORAM described in [23], including its level compression optimization. For fairness, we allow ObliviStore to use extra client space to locally cache the smallest levels in each partition. The second baseline is an insecure scheme without ORAM in which blocks are encrypted, but access patterns are not hidden. It transfers exactly one block per request.

We evaluate Burst ORAM against ObliviStore since ObliviStore is the most bandwidth-efficient existing ORAM scheme. Other schemes require less client storage [25], but incur higher bandwidth costs, and thus would yield higher response times. We did not include results from Path-PIR [17] because it requires substantially larger block sizes to be efficient, and its response times are dominated by the orthogonal consideration of PIR computation. Path-PIR reports response times in the 40–50 second range for comparably-sized databases.

7.1.2 Metrics

We evaluate Burst ORAM and our baselines using *response time* and *bandwidth cost* as metrics (see Section 2). We measure average, maximum, and p-percentile response times for various p. A p-percentile response time of t seconds indicates that p percent of the requests were satisfied with response times under t seconds.

We explicitly measure online, effective, and overall bandwidth costs. In the insecure baseline, all are 1X, so response times are minimal. However, if a burst has high enough frequency to saturate available bandwidth, requests may still pile up, yielding large response times.

7.1.3 Workloads

We use three workloads. The first consists of an endless burst of requests all issued at once, and compares changes in bandwidth costs of each scheme as a function of burst length. The second consists of two identical bursts with equally-spaced requests, separated by an idle period. It shows how response times change in each scheme before and after the idle period.

The third workload is based on the NetApp Dataset [2, 14], a corporate workload containing file system accesses from over 5000 corporate clients and 2500 engineering clients during 100 days. The file system uses 22TB of its 31TB of available space. More details about the workload are provided in the work by Leung et al. [14].

Our NetApp workload uses a 15 day period (Sept. 25 through Oct. 9) during which corporate and engineering clients were active. Requested chunk sizes range from a few bits to 64KB, with most at least 4KB [14]. Thus, we chose a 4KB block size. In total, 312GB of data were requested using $8.8 \cdot 10^7$ 4KB queries.

We configure the NetApp workload ORAM with a 32TB capacity, and allow 100GB of client space, for a usable storage increase of 328 times. For Burst ORAM and ObliviStore, at least 33GB is consumed by the position map, and only 64GB is used for local block storage. The total block count is $N = 2^{33}$. Blocks are divided into $\lfloor 2^{17}/3 \rfloor$ partitions to maximize server space utilization, each with an upper-bound partition size of 2^{18} blocks.

7.2 Simulator

We evaluated Burst ORAM's bandwidth costs and response times using a detailed simulator written in Java. The simulator creates an asynchronous event for each block to be transferred. We calculate the transfer's expected end time from the network latency, the network bandwidth, and the number of pending transfers.

Our simulator also measures results for ObliviStore and the insecure baseline. In all schemes, block requests are time-stamped as soon as they arrive, and serviced as soon as possible. Requests pile up indefinitely if they arrive more frequently than the scheme can handle them.

Burst ORAM's behavior is driven by semaphores and appears data-independent to the server. Each request reads from a partition that appears to be chosen uniformly at random, so bandwidth costs and response times depend only on request arrival times, not on requested block IDs or contents. Thus, the simulator need only store counters representing the number of remaining blocks in each level of each partition, and can avoid storing block IDs and contents explicitly.

Since the simulator need not represent blocks individually, it does not measure the costs of encryption, looking up block IDs, or performing disk reads for blocks. Thus, measured bandwidth costs and response times depend entirely on network latency, bandwidth capacity, request arrival times, and the scheme itself.

7.2.1 Extrapolating Results to Real-World Settings

Burst ORAM can achieve near-optimal performance for realistic bursty traffic patterns. In particular, in many real-life cases bandwidth is overprovisioned to ensure near-optimal response time under bursts – for the insecure baseline. However, in between bursts, most of the bandwidth is not utilized. Burst ORAM's idea is leveraging the available bandwidth in between bursts to ensure near-optimal response time during bursts.

Our simulation applies mainly to scenarios where the client-server bandwidth is the primary bandwidth bottleneck (i.e., client-server bandwidth is the narrowest pipe in the system), which is likely to be the case in a real-life outsourced storage scenario, such as a corporate enterprise outsourcing its storage to a cloud provider. While the simulation assumes that there is a single server, in practice, the server-side architecture could be more complicated and involve multiple servers interacting with each other. But as long as server-server bandwidth is not the bottleneck, our simulation results would be applicable. Similarly, we assume that the server's disk bandwidth is not a bottleneck. This is likely the case if fast Solid State Drives (SSD) are employed. For example, assuming 4KB blocks and only one such array of SSDs with a 100μs random 4KB read latency, our single-array throughput limits us to satisfying 10,000 requests per second. In contrast, even a 1Gbps network connection lets us satisfy only 32,000 requests per second. Thus, with even six such arrays ($3\log_2 N$ SSDs total), assigning roughly $\sqrt{N}/6$ partitions to each array, we can expect the client-server network to be the bottleneck.

Other than bandwidth, another factor is inherent system latencies, e.g., network round-trip times, or inherent disk latencies. Under the same overall bandwidth configuration, increased latency is unlikely to affect the near-optimality of Burst ORAM– while they would increase Burst ORAM's total response times, we would expect a comparable increase in response times for the insecure baseline.

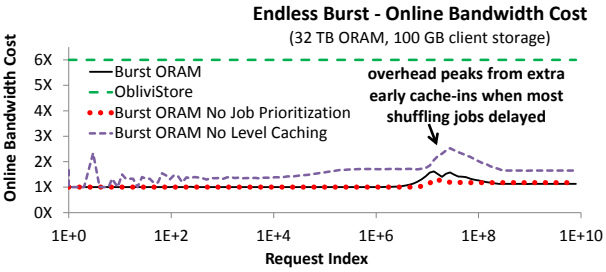

Figure 7: Online bandwidth costs as a burst lengthens. Burst ORAM maintains low online cost regardless of burst length, unlike ObliviStore.

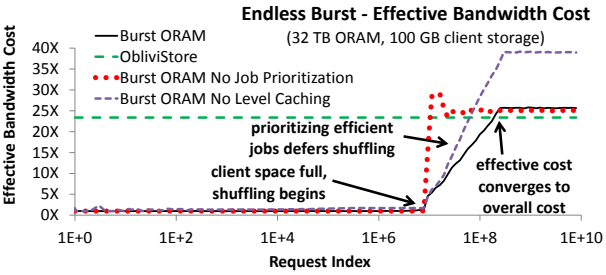

Figure 8: Effective bandwidth costs as burst grows. Burst ORAM handles most bursts with ∼1X effective cost. Effective costs converge to overall costs for long bursts.

7.3 Endless Burst Experiments

For the endless burst experiments, we use a 32TB ORAM with $N = 2^{33}$ 4KB blocks and 100GB client space. We issue 2^{33} requests at once, then start satisfying requests in order using each scheme. We record the bandwidth costs of each request, averaged over requests with similar indexes and over three trials. Figures 7 and 8 show online and effective costs, respectively. The insecure baseline is not shown, since its online, effective, and overall bandwidth costs are all 1.

Figure 7 shows that Burst ORAM maintains 5X–6X lower online cost than ObliviStore for bursts of all lengths. When Burst ORAM starts to delay shuffling, it incurs more early shuffle reads, increasing online cost, but stays well under 2X on average. Burst ORAM effective costs can be near 1X because writes associated with requests are not performed until blocks are shuffled.

Burst ORAM defers shuffling, so its effective cost stays close to its online cost until client space fills, while ObliviStore starts shuffling immediately, so its effective

cost stays constant (Figure 8). Thus, response times for short bursts will be substantially lower in Burst ORAM than in ObliviStore.

Eventually, client space fills completely, and even Burst ORAM must shuffle continuously to keep up with incoming requests. This behavior is seen at the far right of Figure 8, where each scheme's effective cost converges to its overall cost. Burst ORAM's XOR technique results in slightly higher overall cost than ObliviStore's level compression, so Burst ORAM is slightly less efficient for very long bursts. Without local level caching, Burst ORAM spends much more time shuffling the smallest levels, yielding the poor performance of *Burst ORAM No Level Caching*.

If shuffle jobs are started in arbitrary order, as for *Burst ORAM No Prioritization*, the amount of shuffling per request quickly increases, pushing effective cost toward overall cost. However, by prioritizing efficient shuffle jobs as in *Burst ORAM* proper, more shuffling can be deferred, keeping effective costs lower for longer, and maintaining shorter response times.

7.4 Two-Burst Experiments

Our Two-Burst experiments show how each scheme responds to idle time between bursts. We show that Burst ORAM uses the idle time effectively, freeing up as much client space as possible. The longer the gap between bursts, the longer Burst ORAM maintains low effective costs during Burst 2.

Figure 9 shows response times during two closely-spaced bursts, each of $\sim 2^{27}$ requests spread evenly over 72 seconds. The ORAM holds $N = 2^{25}$ blocks, and the client has space for 2^{18} blocks. Since we must also store early shuffle reads and reserve space for the shuffle buffer, the client space is not quite enough to accommodate a single burst entirely. We simulate a 100Mbps network connection with 50ms latency.

All ORAMs start with low response times during Burst 1. ObliviStore response times quickly increase due to fixed shuffle work between successive requests. Burst ORAMs delay shuffle work, so response times stay low until client space fills. Without level caching, additional early shuffle reads cause early shuffling and thus premature spikes in response times.

When Burst 1 ends, the ORAMs continue working, satisfying pending requests and catching up on shuffling during the idle period. Longer idle times allow more shuffling and lower response times at the start of Burst 2. None of the ORAMs have time to fully catch up, so response times increase sooner during Burst 2. ObliviStore cannot even satisfy all Burst 1 requests before Burst 2 starts, so response times start high on Burst 2. Burst ORAM does satisfy all Burst 1 requests, so it uses freed client space to efficiently handle early Burst 2 requests.

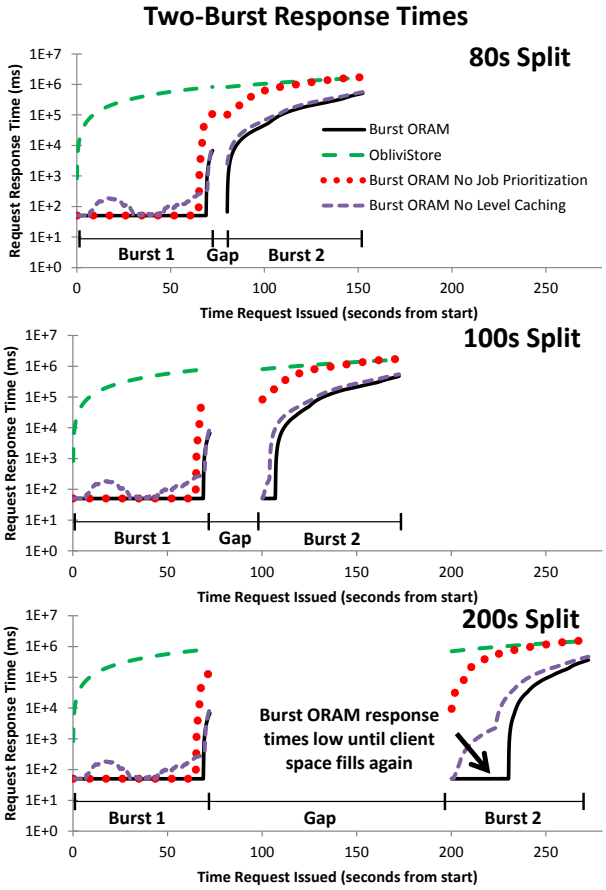

Figure 9: Response times during two same-size bursts of just over 2^{17} requests spread evenly over 72 seconds. Client has space for at most 2^{18} blocks. No level caching causes early spikes due to extra early shuffle reads.

Clearly, Burst ORAM performs better with shuffle prioritization, as it allows more shuffling to be delayed to the idle period, satisfying more requests quickly in both bursts. Burst ORAM also does better with local level caching. Without level caching, we start with more available client space, but the extra server levels yield more early shuffle reads to store, filling client space sooner.

7.5 NetApp Workload Experiments

The NetApp experiments show how each scheme performs on a realistic, bursty workload. Burst ORAM exploits the bursty request patterns, minimizing online IO and delaying shuffle IO to achieve near-optimal response times far lower than ObliviStore's. Level caching keeps Burst ORAM's overall bandwidth costs low.

Figure 10 shows 99.9-percentile response times for several schemes running the 15-day NetApp workload for varying bandwidths. All experiments assume a 50ms network latency. For most bandwidths, Burst ORAM response times are orders of magnitude lower than those

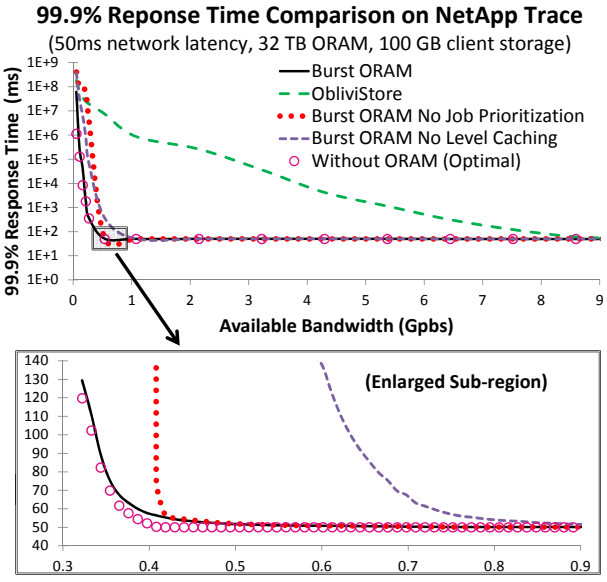

Figure 10: (Top) Burst ORAM achieves short response times in bandwidth-constrained settings. Since ObliviStore has high effective cost, it requires more available client-server bandwidth to achieve short response times. (Bottom) Burst ORAM response times are comparable to those of the insecure (without ORAM) scheme.

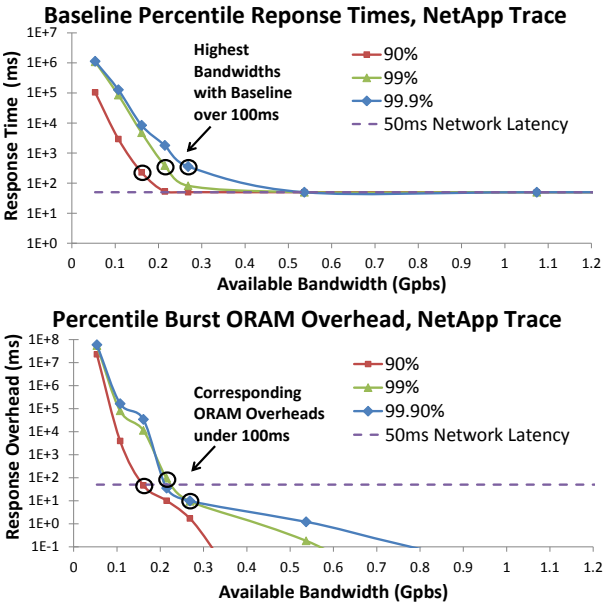

Figure 11: (Top) Insecure baseline (no ORAM) p-percentile response times for various p. (Bottom) Overhead (difference) between insecure baseline and Burst ORAM's p-percentile response times. Marked nodes show that when baseline p-percentile response times are $<$ 100ms, Burst ORAM overhead is also $<$ 100ms.

of ObliviStore and comparable to those of the insecure baseline. Shuffle prioritization and level caching noticeably reduce response times for bandwidths under 1Gbps.

Figure 11 compares p-percentile response times for p values of 90%, 99%, and 99.9%. It gives absolute p-percentile response times for the insecure baseline, and differences between the insecure baseline and Burst ORAM p-percentile response times (Burst ORAM *overhead*). When baseline response times are low, Burst ORAM response times are also low across multiple p.

The NetApp dataset descriptions [2, 14] do not specify the total available network bandwidth, but since it was likely sufficient to allow decent performance, we expect from Figure 10 that it was at least between 200Mbps and 400Mbps. Figure 12 compares the overall bandwidth costs incurred by each scheme running the NetApp workload at 400Mbps. Costs for other bandwidths are similar. Burst ORAM clearly achieves an online cost several times lower than ObliviStore's.

Level caching reduces Burst ORAM's overall cost from 42X to 29X. Burst ORAM's higher cost is due to a combination of factors needed to achieve short response times. First, Burst ORAM uses the XOR technique, which is less efficient overall than ObliviStore's mutually exclusive level compression. Second, Burst ORAM handles smaller jobs first. Such jobs are more efficient in the short-term, but since they frequently write blocks

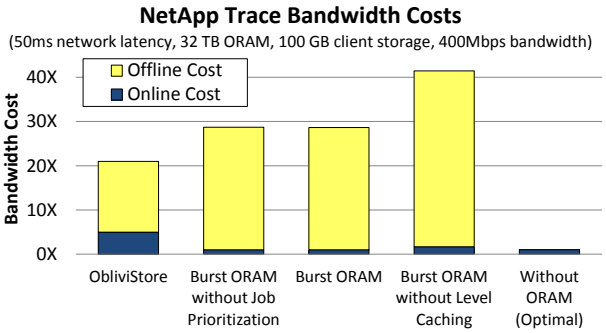

Figure 12: To achieve shorter response times, Burst ORAM incurs higher overall bandwidth cost than ObliviStore, most of which is consumed during idle periods. Level caching keeps bandwidth costs in check. Job prioritization does not affect overall cost, but does reduce effective costs and response times (Figures 8, 10).

to small levels, they create more future shuffle work. In ObliviStore, such jobs are often delayed during a large job, so fewer levels are created, reducing overall cost.

8 Conclusion

We have presented Burst ORAM, a novel Oblivious RAM scheme based on ObliviStore and tuned for practical response times on bursty workloads. We presented a

novel ORAM architecture for prioritizing online IO, and introduced the XOR technique for reducing online IO. We also introduced a novel scheduling mechanism for delaying shuffle IO, and described a level caching mechanism that uses extra client space to reduce overall IO.

We simulated Burst ORAM on a real-world workload and showed that it incurs low online and effective bandwidth costs during bursts. Burst ORAM achieved near-optimal response times that were orders of magnitude lower than existing ORAM schemes.

Acknowledgements.. This work was supported in part by grant N00014-07-C-0311 from ONR, the National Physical Science Consortium Graduate Fellowship; by NSF under grant number CNS-1314857, a Sloan Research Fellowship, a Google Faculty Research Award; by the NSF Graduate Research Fellowship under Grant No. DGE-0946797, a DoD National Defense Science and Engineering Graduate Fellowship, an Intel award through the ISTC for Secure Computing, and a grant from Amazon Web Services.

References

[1] BONEH, D., MAZIERES, D., AND POPA, R. A. Remote oblivious storage: Making oblivious RAM practical. Manuscript, http://dspace.mit.edu/bitstream/handle/1721.1/62006/MIT-CSAIL-TR-2011-018.pdf, 2011.

[2] CHEN, Y., SRINIVASAN, K., GOODSON, G., AND KATZ, R. Design implications for enterprise storage systems via multi-dimensional trace analysis. In *Proc. ACM SOSP* (2011).

[3] CHOW, R., GOLLE, P., JAKOBSSON, M., SHI, E., STADDON, J., MASUOKA, R., AND MOLINA, J. Controlling data in the cloud: outsourcing computation without outsourcing control. In *Proc. ACM CCSW* (2009), pp. 85–90.

[4] DAMGÅRD, I., MELDGAARD, S., AND NIELSEN, J. B. Perfectly secure oblivious RAM without random oracles. In *TCC* (2011), pp. 144–163.

[5] DAUTRICH, J., AND RAVISHANKAR, C. Compromising privacy in precise query protocols. In *Proc. EDBT* (2013).

[6] FLETCHER, C., VAN DIJK, M., AND DEVADAS, S. Secure Processor Architecture for Encrypted Computation on Untrusted Programs. In *Proc. ACM CCS Workshop on Scalable Trusted Computing* (2012), pp. 3–8.

[7] GENTRY, C., GOLDMAN, K., HALEVI, S., JULTA, C., RAYKOVA, M., AND WICHS, D. Optimizing ORAM and using it efficiently for secure computation. In *PETS* (2013).

[8] GOLDREICH, O. Towards a theory of software protection and simulation by oblivious rams. In *STOC* (1987).

[9] GOLDREICH, O., AND OSTROVSKY, R. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM) 43*, 3 (1996), 431–473.

[10] GOODRICH, M., AND MITZENMACHER, M. Privacy-preserving access of outsourced data via oblivious RAM simulation. *Automata, Languages and Programming* (2011), 576–587.

[11] GOODRICH, M. T., MITZENMACHER, M., OHRIMENKO, O., AND TAMASSIA, R. Privacy-preserving group data access via stateless oblivious RAM simulation. In *Proc. SODA* (2012), SIAM, pp. 157–167.

[12] ISLAM, M., KUZU, M., AND KANTARCIOGLU, M. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *Proc. NDSS* (2012).

[13] KUSHILEVITZ, E., LU, S., AND OSTROVSKY, R. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *Proc. SODA* (2012), SIAM, pp. 143–156.

[14] LEUNG, A. W., PASUPATHY, S., GOODSON, G., AND MILLER, E. L. Measurement and analysis of large-scale network file system workloads. In *Proc. USENIX ATC* (2008), USENIX Association, pp. 213–226.

[15] LORCH, J. R., PARNO, B., MICKENS, J. W., RAYKOVA, M., AND SCHIFFMAN, J. Shroud: Ensuring private access to large-scale data in the data center. *FAST* (2013), 199–213.

[16] MAAS, M., LOVE, E., STEFANOV, E., TIWARI, M., SHI, E., ASANOVIC, K., KUBIATOWICZ, J., AND SONG, D. PHANTOM: Practical oblivious computation in a secure processor. In *ACM CCS* (2013).

[17] MAYBERRY, T., BLASS, E.-O., AND CHAN, A. H. Efficient private file retrieval by combining ORAM and PIR. In *NDSS* (2014).

[18] OSTROVSKY, R., AND SHOUP, V. Private information storage (extended abstract). In *STOC* (1997), pp. 294–303.

[19] PINKAS, B., AND REINMAN, T. Oblivious RAM revisited. In *CRYPTO* (2010).

[20] REN, L., YU, X., FLETCHER, C. W., VAN DIJK, M., AND DEVADAS, S. Design space exploration and optimization of path oblivious RAM in secure processors. In *Proc. ISCA*. 2013.

[21] SHI, E., CHAN, H., STEFANOV, E., AND LI, M. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *Proc. ASIACRYPT* (2011).

[22] STEFANOV, E., AND SHI, E. Multi-Cloud Oblivious Storage. In *CCS* (2013).

[23] STEFANOV, E., AND SHI, E. ObliviStore: High performance oblivious cloud storage. In *IEEE Symposium on Security and Privacy* (2013).

[24] STEFANOV, E., SHI, E., AND SONG, D. Towards practical oblivious RAM. NDSS, 2012.

[25] STEFANOV, E., VAN DIJK, M., SHI, E., FLETCHER, C., REN, L., YU, X., AND DEVADAS, S. Path ORAM: An extremely simple oblivious RAM protocol. In *ACM CCS* (2013).

[26] WILLIAMS, P., AND SION, R. Sr-oram: Single round-trip oblivious ram. *ACNS, industrial track* (2012), 19–33.

[27] WILLIAMS, P., SION, R., AND CARBUNAR, B. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *Proc. ACM CCS* (2008), pp. 139–148.

[28] WILLIAMS, P., SION, R., AND TOMESCU, A. PrivateFS: A parallel oblivious file system. In *CCS* (2012).

A Pseudocode

Algorithms 1–4 give pseudocode for Burst ORAM, using the notation summarized in Table 1. The algorithms are described in detail in Section 6, but we clarify some of the code and notation below.

The efficiency of shuffle job J_p is given by:

$$E_{J_p} = \frac{V_{J_p} + A_{J_p}}{R_{J_p} + W_{J_p}}. \quad (1)$$

C_p represents the state of partition p at the time p's last shuffle job completed, and determines the current set of occupied levels in p. V_p represents the number of blocks that have been evicted to p, since p's last shuffle job completed. $C_p + V_p$ determines which levels would be occupied if p were to be completely shuffled.

V_{J_p} represents the number of evicted blocks that will be shuffled into p by J_p. Thus, C_p and V_{J_p} together determine J_p's read and write levels.

If J_p is inactive, it is updated whenever V_p changes, setting $V_{J_p} \leftarrow V_p$ (Algorithm 1, Line 25). However, we

Table 1: Algorithm Notation

v	Eviction rate: blocks evicted per request
λ	Number of levels cached locally
p	A partition
V_p	# blocks evicted to p since p's last shuffle end
C_p	p's state after last shuffle (shuffled evictions)
b	Block ID
$D(b)$	Plaintext contents of b
$E(b)$	Encrypted contents of b
$S(b)$	Server address/ID of b
$P(b)$	Partition containing b, or random if none
$L(b)$	Level containing b, or $\perp$ if none
Q	IDs of standard-read blocks to fetch
C	IDs of early shuffle read blocks to fetch
X_Q	Combined block (XOR of all blocks in Q)
X'_Q	Subtraction block (XOR of dummies in Q)
J_p	Shuffle job for p
V_{J_p}	Number of evicted blocks J_p will shuffle
E_{J_p}	Efficiency of J_p
A_{J_p}	Number of early shuffle reads for J_p
R_{J_p}	Total blocks remaining to be read for J_p
W_{J_p}	Total blocks to write for J_p
NJQ	New Job Queue
RJQ	Read Job Queue
WJQ	Write Job Queue

Algorithm 1 Pseudocode for Client and ORAM Main

```
1:  function ClientRead(b)
2:      Append b to RequestQueue
3:      On RequestCallback(D(b)), return D(b)
4:  procedure Write(b, d)
5:      Append b to RequestQueue
6:      On RequestCallback(D(b)), write d to D(b)
7:  procedure ORAM Main
8:      RequestMade ← false
9:      if RequestQueue ≠ ∅ then
10:         b ← Peek(RequestQueue)
11:         if Fetch(b) then                ▷ Request Issued
12:             RequestMade ← true
13:             Pop(RequestQueue)
14:             MakeEvictions
15:     if RequestMade = false then
16:         TryShuffleWork
17: procedure MakeEvictions
18:     PendingEvictions = PendingEvictions + v
19:     while PendingEvictions ≥ 1 do
20:         p ← random partition
21:         Evict new dummy or assigned real block to p
22:         V_p = V_p + 1
23:         if shuffling p only writes levels ≥ λ then
24:             J_p ← p's inactive job ▷ Create if needed
25:             V_{J_p} ← V_p
26:             if p has no active job then
27:                 NJQ = NJQ ∪ J_p
28:         PendingEvictions = PendingEvictions − 1
```

implement level caching by skipping those updates to J_p that would cause J_p to write to levels with indexes less than λ (Algorithm 1, Line 23). Once J_p is active, V_{J_p} is no longer updated. When J_p completes, p's state is updated to reflect the blocks shuffled in by J_p, setting $C_p \leftarrow C_p + V_{J_p}$ (Algorithm 3, Line 37).

If p has no inactive shuffle job, the job is created following the first eviction to p that would allow updating (Algorithm 1, Line 24). If p has no active job, the inactive job moves to the *New Job Queue* (NJQ) as soon as the job is created (Algorithm 1, Line 27), where it stays until the job is activated. If p does have an active shuffle job, the inactive job is not added to NJQ until the active job completes (Algorithm 3, Line 38).

Thus, NJQ contains only inactive shuffle jobs for those partitions with no active job, ensuring that any job in NJQ may be activated. NJQ is a priority queue serving the most efficient jobs first. Job efficiency may change while the job is in NJQ, since V_{J_p} can still be updated.

B Reducing Online Costs of SR-ORAM

We now briefly describe how SR-ORAM [26] can benefit from our XOR technique. Like ObliviStore, SR-ORAM requires only a single round-trip to satisfy a request, and has online bandwidth cost $O(\log N)$. SR-ORAM uses an encrypted Bloom filter to let the server obliviously check whether each level contains the requested block. The server retrieves the requested block from its level, and client-selected dummies from all others. Since at most one block is real, the server can XOR all the blocks together and return a single combined block.

One difference in SR-ORAM is that the client *does not* know a priori which level contains the requested block. Thus, SR-ORAM must be modified to include the level index of each retrieved block in its response. To allow the client to easily reconstruct dummies, we must also change SR-ORAM to generate the contents of each dummy block as in Burst ORAM. Since the client knows the indexes of the dummy blocks it requested from each level, it can infer the real block's level from the server's response. The client then reconstructs the all dummy block contents and XORs them with the returned block to obtain the requested block, as in Burst ORAM.

SR-ORAM is a synchronous protocol, so it has no notion equivalent to early shuffle reads. Thus, the XOR technique reduces SR-ORAM's online bandwidth cost from $O(\log N)$ to 1. The reduction in overall

Algorithm 2 Pseudocode for Requester

```
1:  function FETCH(b)
2:      P(b), L(b) ← position map lookup on b
3:      Q = ∅, C = ∅
4:      for level ℓ ∈ P(b) do
5:          if ℓ is non-empty then
6:              b_ℓ ← b if ℓ = L(b)
7:              b_ℓ ← ID of next dummy in ℓ if ℓ ≠ L(b)
8:              if ℓ more than half full then
9:                  Q ← Q ∪ S(b_ℓ)            ▷ Standard read
10:             else
11:                 C ← C ∪ S(b_ℓ)            ▷ Early shuffle read
12:     Ret ← |C| + MAX(|Q|, 1)               ▷ # blocks to return
13:     if Not TRYDEC(Local Space, Ret) then
14:         return false                      ▷ Not enough space for blocks
15:     DEC(Concurrent IO, Ret)
16:     Issue asynch. request for (C, Q) to server
17:     When done, server calls:
18:         FETCHCALLBACK(E(C), XOR of E(Q))
19:     return true
20: procedure FETCHCALLBACK({E(c_i)}, X_Q)
21:     INC(Concurrent IO, 1)
22:     if b ∈ Q then
23:         X'_Q ← ⊕{E(q_i) | S(q_i) ∈ Q, q_i ≠ b}
24:                ▷ Subtraction block, computed locally
25:         E(b) ← X_Q ⊕ X'_Q
26:     if b ∈ C then
27:         E(b) ← E(c_i) where c_i = b
28:     D(b) ← decrypt E(b)
29:     Assign b for eviction to random partition
30:     REQUESTCALLBACK(D(b))
```

cost is negligible, as SR-ORAM has an offline cost $O(\log^2 N \log\log N)$. SR-ORAM contains only one hierarchy of $O(\log N)$ levels, so XOR incurs only $O(\log N)$ extra storage cost for the level-specific keys, fitting into SR-ORAM's logarithmic client storage.

Algorithm 3 Pseudocode for Shuffler

```
1:  procedure TRYSHUFFLEWORK
2:      if Not TRYDEC(Concurrent IO, 1) then
3:          return
4:      ReadIssued, WriteIssued ← false
5:      if All reads for jobs in RJQ issued then
6:          TRYACTIVATE                       ▷ Try to add job to RJQ
7:      if J_p ∈ RJQ has not issued read b_R then
8:          if TRYDEC(Shuffle Buffer, 1) then
9:              Issue asynch. request for S(b_R)
10:             When done: READCALLBACK(E(b_R))
11:             ReadIssued ← true
12:     if !ReadIssued and J_p ∈ WJQ has write b_W then
13:         Write E(b_W) to server
14:         When done, call WRITECALLBACK(S(b_W))
15:         WriteIssued ← true
16:     if Not ReadIssued and Not WriteIssued then
17:         INC(Concurrent IO, 1)             ▷ No shuffle work
18: procedure TRYACTIVATE
19:     if NJQ ≠ ∅ then
20:         J_p ← PEEK(NJQ)                   ▷ Most efficient job
21:         if TRYDEC(Shuffle Buffer, V_{J_p} + A_{J_p}) then
22:             Mark J_p active               ▷ V_{J_p} frozen
23:             INC(Local Space, V_{J_p} + A_{J_p})
24:             Move J_p from NJQ to RJQ
25: procedure READCALLBACK(E(b_R))
26:     INC(Concurrent IO, 1)
27:     Decrypt E(b_R), place D(b_R) in Shuffle Buffer
28:     if all reads in J_p have finished then
29:         Create dummy blocks to get W_{J_p} blocks total
30:         Permute and re-encrypt the blocks
31:         Move J_p from RJQ to WJQ
32: procedure WRITECALLBACK(S(b_W))
33:     INC(Concurrent IO, 1)
34:     if all writes in J_p have finished then
35:         Mark J_p complete
36:         Remove J_p from WJQ
37:         Update C_p ← C_p + V_{J_p}, V_p ← V_p − V_{J_p}
38:         Add p's inactive job, if any, to NJQ
```

Algorithm 4 Pseudocode for semaphores

```
1: procedure DEC(Semaphore, Quantity)
2:     Semaphore ← Semaphore − Quantity
3: procedure INC(Semaphore, Quantity)
4:     Semaphore ← Semaphore + Quantity
5: function TRYDEC(Semaphore, Quantity)
6:     if Semaphore < Quantity then return false
7:     DEC(Semaphore, Quantity); return true
```