Towards Practical Oblivious RAM

Emil Stefanov
UC Berkeley
emil@cs.berkeley.edu

Elaine Shi
UC Berkeley/PARC
elaines@cs.berkeley.edu

Dawn Song
UC Berkeley
dawnsong@cs.berkeley.edu

Abstract

We take an important step forward in making Oblivious RAM (O-RAM) practical. We propose an O-RAM construction achieving an amortized overhead of $20 \sim 35X$ (for an O-RAM roughly 1 terabyte in size), about 63 times faster than the best existing scheme. On the theoretic front, we propose a fundamentally novel technique for constructing Oblivious RAMs: specifically, we partition a bigger O-RAM into smaller O-RAMs, and employ a background eviction technique to obliviously evict blocks from the client-side cache into a randomly assigned server-side partition. This novel technique is the key to achieving the gains in practical performance.

1 Introduction

As cloud computing gains momentum, an increasing amount of data is outsourced to cloud storage, and data privacy has become an important concern for many businesses and individuals alike. Encryption alone may not suffice for ensuring data privacy, as data access patterns can leak a considerable amount of information about the data as well. Pinkas *et al.* gave an example [13]: if a sequence of data access requests q_1, q_2, q_3 is always followed by a stock exchange operation, the server can gain sensitive information even when the data is encrypted.

Oblivious RAM (or O-RAM) [3, 4, 11], first investigated by Goldreich and Ostrovsky, is a primitive intended for hiding storage access patterns. The problem was initially studied in the context of software protection, i.e., hiding a program's memory access patterns to prevent reverse engineering.

With the trend of cloud computing, O-RAM also has important applications in privacy-preserving storage outsourcing applications. In this paper, we consider the setting where a client wishes to store N blocks each of size B bytes at an untrusted server.

The community's interest in O-RAM has recently rekindled, partly due to its potential high impact in privacy-preserving storage outsourcing applications. One of the best schemes known to date is a novel construction recently proposed by Goodrich and Mitzenmacher [5]. Specifically, let N denotes the total storage capacity of the O-RAM in terms of the number of blocks. The Goodrich-Mitzenmacher construction achieves $O((\log N)^2)$ amortized cost when parametrized with $O(1)$ client-side storage; or it achieves $O(\log N)$ amortized cost when parametrized with $O(N^a)$ client-side storage where $0 < a < 1$. In this context, an amortized cost of $f(N)$ means that each data request will generate $f(N)$ read or write operations on the server.

Despite elegant asymptotic guarantees, the practical performance of existing O-RAM constructions are still unsatisfactory. As shown in Table 1, one of the most practical schemes known to date is the construction by Goodrich and Mitzenmacher [5] when parametrized with N^a $(a < 1)$ client-side storage. This scheme has more than $1,400X$ overhead compared to non-oblivious storage under reasonable parametrization, which is prohibitive in practice. In summary, although it has been nearly two decades since O-RAM was first invented, so far, it has mostly remained a theoretical concept.

1.1 Results and Contributions

Our main goal is to make Oblivious RAM practical for cloud outsourcing applications.

Practical construction. We propose an O-RAM construction geared towards optimal practical performance. The practical construction achieves an amortized overhead of **20 $\sim$ 35X** (Tables 1 and 2), about **63 times faster** than the best known construction. In addition, this practical construction also achieves *sub-linear* $O(\log N)$ *worst-case cost*, and *constant round-trip latency* per operation. Although our practical construction requires asymptotically linear amount of client-side storage, the constant is so small (0.01% to 0.3% of the O-RAM capacity) that in realistic settings, the amount of client-side storage is comparable to $\sqrt{N}$.

Theoretical construction. By applying recursion to the practical construction, we can reduce the client-side

Scheme	Amortized Cost	Worst-case Cost	Client Storage	Server Storage	Practical Performance
Goldreich-Ostrovsky [4]	$O((\log N)^3)$	$\Omega(N)$	O(1)	$O(N \log N)$	$> 120,000$X
Pinkas-Reinman [13]	$O((\log N)^2)$	$O(N \log N)$	$O(1)$	$8N$	$60,000 \sim 80,000$X
Goodrich-Mitzenmacher [5]	$O(\log N)$	$O(N \log N)$	$O(N^a)$ $(0 < a < 1)$	$8N$	$> 1,400$X
This paper:					
Practical, Non-Concurrent	$O(\log N)$	$< 3\sqrt{N} + o(\sqrt{N})$	cN (c very small)	$< 4N + o(N)$	$20 \sim 35$X
Practical, Concurrent	$O(\log N)$	$O(\log N)$	cN (c very small)	$< 4N + o(N)$	$20 \sim 35$X
Theoretic, Non-Concurrent	$O((\log N)^2)$	$O(\sqrt{N})$	$O(\sqrt{N})$	$O(N)$	—
Theoretic, Concurrent	$O((\log N)^2)$	$O((\log N)^2)$	$O(\sqrt{N})$	$O(N)$	—

Table 1: Our contributions. The practical performance is the number of client-server operations per O-RAM operation for typical realistic parameters, e.g., when the server stores terabytes of data, and the client has several hundred megabytes to gigabytes of local storage, and $N \geq 2^{20}$. For our theoretic constructions, the same asymptotic bounds also work for the more general case where client-side storage is N^a for some constant $0 < a < 1$.

O-RAM Capacity	# Blocks	Block Size	Client Storage	Server Storage	$\frac{\text{Client Storage}}{\text{O-RAM Capacity}}$	Practical Performance
64 GB	2^{20}	64 KB	204 MB	205 GB	0.297%	22.5X
256 GB	2^{22}	64 KB	415 MB	819 GB	0.151%	24.1X
1 TB	2^{24}	64 KB	858 MB	3.2 TB	0.078%	25.9X
16 TB	2^{28}	64 KB	4.2 GB	51 TB	0.024%	29.5X
256 TB	2^{32}	64 KB	31 GB	819 TB	0.011%	32.7X
1024 TB	2^{34}	64 KB	101 GB	3072 TB	0.009%	34.4X

Table 2: Suggested parametrizations of our practical construction. The practical performance is the number of client-server operations per O-RAM operation as measured by our simulation experiments.

storage to a sublinear amount, and obtain a novel construction of theoretic interest, achieving $O((\log N)^2)$ amortized and *worst-case* cost, and requires $O(\sqrt{N})$ client-side storage, and $O(N)$ server-side storage. Note that in the $O((\log N)^2)$ asymptotic notation, one of the $\log N$ factors stems from the the depth of the recursion; and in realistic settings (see Table 2), the depth of the recursion is typically 2 or 3.

Table 1 summarizes our contributions in the context of related work. Table 2 provides suggested parametrizations for our practical construction.

1.2 Main Technique: Partitioning

We propose a novel *partitioning* technique, which is the key to achieving the claimed theoretical bounds as well as major practical savings. The basic idea is to partition a single O-RAM of size N blocks into P different O-RAMs of size roughly $\frac{N}{P}$ blocks each. This allows us to break down a bigger O-RAM into multiple smaller O-RAMs.

The idea of partitioning is motivated by the fact that the major source of overhead in existing O-RAM constructions arises from an expensive *remote oblivious sorting* protocol performed between the client and the server. Because the oblivious sorting protocol can take up to $O(N)$ time, existing O-RAM schemes require $\Omega(N)$ time in the worst-case or have unreasonable $O(\sqrt{N})$ amortized cost.

We partition the O-RAM into roughly $P = \sqrt{N}$ partitions, each having $\sqrt{N}$ blocks approximately. This way, the client can use $\sqrt{N}$ blocks of storage to sort/reshuffle the data blocks locally, and then simply transfer the reshuffled data blocks to the server. This not only circumvents the need for the expensive oblivious sorting protocol, but also allows us to achieve $O(\sqrt{N})$ worst-case cost. Furthermore, by allowing reshuffling to happen concurrently with reads, we can further reduce the worst-case cost of the practical construction to $O(\log N)$.

While the idea of partitioning is attractive, it also brings along an important challenge in terms of security. Partitioning creates an extra channel through which the data access pattern can potentially be inferred by observing the sequence of partitions accessed. Therefore, we must take care to ensure that *the sequence of partitions accessed does not leak information about the identities of blocks being accessed*. Specifically, our construction ensures that the sequence of partitions accessed appears pseudo-random to an untrusted server.

It is worth noting that Ostrovsky and Shoup [12] also came up with a technique to spread the reshuffling work of the hierarchical solution [4] over time, thereby achieving poly-logarithmic worst-case cost. However, our technique of achieving poly-logarithmic worst-case cost is fundamentally from Ostrovsky and Shoup [12].

Moreover, our partitioning and background eviction techniques are also key to the practical performance gain that we can achieve.

1.3 Related Work

Oblivious RAM was first investigated by Goldreich and Ostrovsky [3, 4, 11]. Since their original work, several seminal improvements have been proposed [5, 13, 15, 16]. These approaches mainly fall into two broad categories: constructions that use $O(1)$ client-side storage, and constructions that use $O(N^a)$ client-side storage where $0 < a < 1$.

Williams and Sion [15] propose an O-RAM construction that requires $O(\sqrt{N})$ client-side storage, and achieves an amortized cost of $O((\log N)^2)$. Williams *et al.* propose another construction that uses $O(\sqrt{N})$ client-side storage, and achieves $O(\log N \log\log N)$ amortized cost [16]; however, researchers have expressed concerns over the assumptions used in their original analysis [5, 13]. A corrected analysis of this construction can be found in an appendix in a recent work by Pinkas and Reinman [13].

Pinkas and Reinman [13] discovered an O-RAM construction that achieves $O((\log N)^2)$ overhead with $O(1)$ client-side storage. However, some researchers have observed a security flaw of the Pinkas-Reinman construction, due to the fact that the lookups can reveal, with considerable probability, whether the client is searching for blocks that exist in the hash table [5]. The authors of that paper will fix this issue in a future journal version. While Table 1 shows the overhead of the Pinkas-Reinman scheme as is, the overhead of the scheme is likely to increase after fixing this security flaw.

In an elegant work by Goodrich and Mitzenmacher [5], they proposed a novel O-RAM construction which achieves $O((\log N)^2)$ amortized cost with $O(1)$ client-side storage; or $O(\log N)$ amortized cost with $O(N^a)$ client-side storage where $0 < a < 1$. The Goodrich-Mitzenmacher construction achieves the best asymptotic performance among all known constructions. However, their practical performance is still prohibitive. For example, with $O(\sqrt{N})$ client-side storage, their amortized cost is $> 1,400$X from a very conservative estimate. In reality, their overhead could be higher.

In an independent and concurrent work by Boneh, Mazieres, and Popa [2], they propose a construction that can support up to $O(\sqrt{N})$ reads while shuffling (using $O(\sqrt{N})$ client-side storage). The scheme achieves $O(\log N)$ online cost, and $O(\sqrt{N})$ amortized cost.

Almost all prior constructions have $\Omega(N)$ worst-case cost, except the seminal work by Ostrovsky and Shoup [12], in which they demonstrate how to spread the reshuffling operations of the hierarchical construction [4] across time to achieve poly-logarithmic worst-case cost. While the aforementioned concurrent work by Boneh *et al.* [2] alleviates this problem by separating the cost into an online part for reading and writing data, and an offline part for reshuffling, they do so at an increased amortized cost of $O(\sqrt{N})$ (when their scheme is configured with $O(\sqrt{N})$ client-side storage). In addition, if $\Omega(\sqrt{N})$ consecutive requests take place within a small time window (e.g., during peak usage times), their scheme can still block on a reshuffling operation of $\Omega(N)$ cost.

Concurrent and subsequent work. In concurrent/subsequent work, Goodrich *et al.* [6] invented an O-RAM scheme achieving $O((\log N)^2)$ worst-case cost with $O(1)$ memory; and and Kushilevitz *et al.* [9] invented a scheme with $O(\frac{(\log N)^2}{\log\log N})$ worst-case cost. Goodrich *et al.* also came up with a stateless Oblivious RAM [7] scheme, with $O(\log N)$ amortized cost and $O(N^a)$ $(0 < a < 1)$ client-side transient (as opposed to permanent) buffers. Due to larger constants in their constructions, our construction is two to three orders of magnitude more efficient in realistic settings.

2 Problem Definition

As shown in Figure 1, we consider a client that wishes to store data at a remote untrusted server while preserving its privacy. While traditional encryption schemes can provide confidentiality, they do not hide the data access pattern which can reveal very sensitive information to the untrusted server. We assume that the server is untrusted, and the client is trusted, including the client's CPU and memory hierarchy (including RAM and disk).

The goal of O-RAM is to completely hide the data access pattern (which blocks were read/written) from the server. In other words, each data read or write request will generate a completely random sequence of data accesses from the server's perspective.

Notations. We assume that data is fetched and stored in atomic units, referred to as *blocks*, of size B bytes each. For example, a typical value for B for cloud storage is 64 KB to 256 KB. Throughout the paper, we use the notation N to denote total number of data blocks that the O-RAM can support, also referred to as the capacity of the O-RAM.

Practical considerations. One of our goals is to design a *practical* O-RAM scheme in realistic settings. We observe that *bandwidth is much more costly than computation and storage* in real-world scenarios. For example, typical off-the-shelf PCs and laptops today have gigabytes of RAM, and several hundred gigabytes of disk

storage. When deploying O-RAM in a realistic setting, it is very likely that the bottleneck is network bandwidth and latency.

As a result, our practical O-RAM construction leverages available client-side storage as a working buffer, and this allows us to drastically optimize the bandwidth consumption between the server and the client. As a typical scenario, we assume that the client wishes to store *terabytes* of data on the remote server, and the client has *megabytes* to *gigabytes* of storage (in the form of RAM or disk). We wish to design a scheme in which the client can maximally leverage its local storage to reduce the overhead of O-RAM.

Security definitions. We adopt the standard security definition for O-RAMs. Intuitively, the security definition requires that the server learns nothing about the access pattern. In other words, no information should be leaked about: 1) which data is being accessed; 2) how old it is (when it was last accessed); 3) whether the same data is being accessed (linkability); 4) access pattern (sequential, random, etc); or 5) whether the access is a read or a write. Like previous work, our O-RAM constructions do not consider information leakage through the timing channel, such as when or how frequently the client makes data requests.

Definition 1 (Security definition)**.** *Let* $\vec{y} := ((\mathsf{op}_1, \mathsf{u}_1, \mathsf{data}_1), (\mathsf{op}_2, \mathsf{u}_2, \mathsf{data}_2), ..., (\mathsf{op}_M, \mathsf{u}_M, \mathsf{data}_M))$ *denote a data request sequence of length M, where each* op_i *denotes a* $\mathsf{read}(\mathsf{u}_i)$ *or a* $\mathsf{write}(\mathsf{u}_i, \mathsf{data})$ *operation. Specifically,* u_i *denotes the identifier of the block being read or written, and* data_i *denotes the data being written. Let* $A(\vec{y})$ *denote the (possibly randomized) sequence of accesses to the remote storage given the sequence of data requests* $\vec{y}$*. An O-RAM construction is said to be secure if for any two data request sequences* $\vec{y}$ *and* $\vec{z}$ *of the same length, their access patterns* $A(\vec{y})$ *and* $A(\vec{z})$ *are computationally indistinguishable by anyone but the client.*

3 The Partitioning Framework

In this section, we describe our main technique, *partitioning*, as a framework. At a high level, the goal of partitioning is to subdivide the O-RAM into much smaller partitions, so that the operations performed on the partitions can be handled much more efficiently than if the O-RAM was not partitioned.

The main challenge of partitioning the O-RAM is to ensure that the sequence of partitions accessed during the lifetime of the O-RAM appears random to the untrusted server while keeping the client-side storage small. In this way, no information about data access pattern is revealed.

3.1 Server Storage

We divide the server's storage into P fully functional *partition O-RAM's*, each containing N/P blocks on average. For now, we can think of each partition O-RAM as a black box, exporting a read and a write operation, while hiding the access patterns within that partition.

At any point of time, each block is randomly assigned to any of the P partitions. Whenever a block is accessed, the block is logically removed from its current partition (although a stale copy of the block may remain), and logically assigned to a fresh random partition selected from all P partitions. Thus, the client needs to keep track of which partition each block is associated with at any point of time, as specified in Section 3.2.

The maximum amount of blocks that an O-RAM can contain is referred to as the *capacity* of the O-RAM. In our partitioning framework, blocks are randomly assigned to partitions, so the capacity of an O-RAM partition has to be slightly more than N/P blocks to accommodate the variance of assignments. Due to the standard balls and bins analysis, for $P = \sqrt{N}$, each partition needs to have capacity $\sqrt{N} + o(\sqrt{N})$ to have a sufficiently small failure probability $\frac{1}{\mathrm{poly}(N)}$.

3.2 Client Storage

The client storage is divided into the following components.

Data cache with P slots. The data cache is a cache for temporarily storing data blocks fetched from the server. There are exactly P cache slots, equal to the number of partitions on the server. Logically, the P cache slots can be thought of an extension to the server-side partitions. Each slot can store 0, 1, or multiple blocks. In the full version of this paper [1], we prove that each cache slot will have a constant number of data blocks in expectation, and that the total number of data blocks in all cache slots will be bounded by $O(P)$ with high probability . In both our theoretic and practical constructions, we will let $P = \sqrt{N}$. In this case, the client's data cache capacity is $O(\sqrt{N})$.

Position map. As mentioned earlier, the client needs to keep track of which partition (or cache slot) each block resides in. The position map serves exactly this purpose. We use the notation $\mathsf{position}[\mathsf{u}]$ for the partition number where block u currently resides. In our practical construction described in Section 4, the position map is extended to also contain the exact location (level number and index within the level) of block u within its current partition.

Intuitively, each block's position (i.e., partition number) requires about $c \log N$ bits to describe. In our prac-

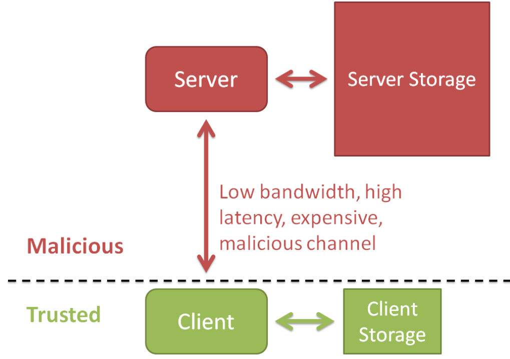

Figure 1: Oblivious RAM system architecture.

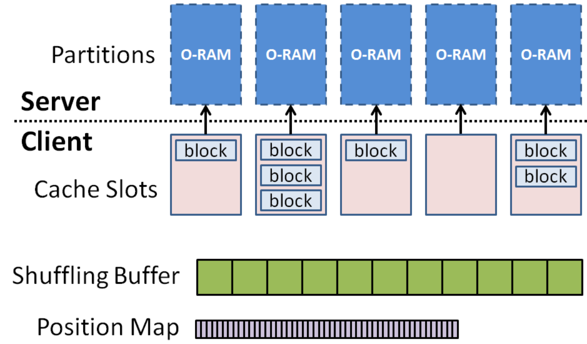

Figure 2: The partitioning framework.

tical construction, $c \leq 1.1$, since the practical construction also stores the block's exact location inside a partition. Hence, the position map requires at most $cN \log N$ bits of storage, or $\frac{c}{8} N \log N$ bytes, which is $\frac{cN \log N}{8 \cdot B}$ blocks. Since in practice the block size $B > \frac{c}{8} \log N$, the size of the position map is a constant fraction of the original capacity of the O-RAM (with a very small constant).

Shuffling buffer. The shuffling buffer is used for the shuffling operation when two or more levels inside a partition O-RAM need to be merged into the next level. For this paper, we assume that the shuffling buffer has size $O(\sqrt{N})$.

Miscellaneous. Finally, we need some client-side storage to store miscellaneous states and information, such as cryptographic keys for authentication, encryption, and pseudo-random permutations.

3.3 Intuition

In our construction, *regardless of whether a block is found in the client's data cache, the client always performs a read and a write operation to the server upon every data request* – with a dummy read operation in case of a cache hit. Otherwise, the server might be able to infer the age of the blocks being accessed. Therefore, the client data cache is required for security rather than for efficiency.

In some sense, the data cache acts like a holding buffer. When the client fetches a block from the server, it cannot immediately write the block back to a some partition, since this would result in linkability attacks the next the this block is read. Instead, the fetched block is associated with a fresh randomly chosen cache slot, but the block resides in the client data cache until a background eviction process writes it back to the server partition corresponding to its cache slot.

Another crucial observation is that the eviction process should not reveal which client cache slots are filled and which are not, as this can lead to linkability attacks as well. To achieve this, we use an eviction process that is independent of the load of each cache slot. Therefore we sometimes have to write back a dummy block to the server. For example, one possible eviction algorithm is to sequentially scan the cache slots at a fixed rate and evict a block from it or evict a dummy block if the cache slot is empty.

To aid the understanding of the partitioning framework, it helps to think of each client cache slot i as an extension of the i-th server partition. At any point of time, a data block is associated with a random partition (or slot), and the client has a position map to keep track of the location of each block. If a block is associated with partition (or slot) $i \in [P]$, it means that an up-to-date version of a block currently resides in partition i (or cache slot i). However, it is possible that other partitions (or even the same partition) may still carry a stale version of block i, which will be removed during a future reshuffling operation.

Every time a read or write operation is performed on a block, the block is re-assigned to a partition (or slot) *selected independently at random from all P partitions (or slots)*. This ensures that two operations on the same block cannot be linked to each other.

3.4 Setup

When the construction is initialized, we first assign each block to an independently random partition. Since initially all blocks are zeroed, their values are implicit and we don't write them to the server. The position map stores an additional bit per block to indicate if it has never been accessed and is hence zeroed. In the practical construction, this bit can be implicitly calculated from other metadata that the client stores. Additionally, the data cache is initially empty.

3.5 Partition O-RAM Semantics and Notations

Before we present the main operations of our partitioning-based O-RAM, we first need to define the operations supported by each partition O-RAM.

Recall that each partition is a fully functional O-RAM by itself. To understand our partitioning framework, it helps to think of each partition as a blackbox O-RAM. For example, for each partition, we can plug in the Goodrich-Mitzenmacher O-RAM [5] (with either $O(1)$ or $O(\sqrt{N})$ client-side storage) or our own partition O-RAM construction described in Section 4.

We make a few small assumptions about the partition O-RAM, and use slightly different semantics to refer to the partition O-RAM operations than existing work. Existing O-RAM constructions [3, 5, 13] always perform both a read and a write operation upon any data access request. For the purpose of the partitioning framework, it helps to separate the reads from the writes. In particular, we require that a ReadPartition operation "logically remove" the fetched block from the corresponding partition. Many existing constructions [3,5,13] can be easily modified to support this operation, simply by writing back a dummy block to the first level of the hierarchy after reading.

Formally, we think of each partition O-RAM as a blackbox O-RAM exporting two operations, ReadPartition and WritePartition, as explained below.

- $\mathsf{ReadPartition}(p, \mathsf{u})$ reads a block identified by its unique identifier $\mathsf{u} \in \{\perp, 1, 2, \ldots, N-1\}$ from partition p. In case $\mathsf{u} = \perp$, the read operation is called a dummy read. We assume that the ReadPartition operation will *logically* remove the fetched block from the corresponding partition.
- $\mathsf{WritePartition}(p, \mathsf{u}, \mathsf{data})$ writes back a block identified by its unique identifier $\mathsf{u} \in \{\perp, 1, 2, \ldots, N-1\}$ to partition p. In case $\mathsf{u} = \perp$, the write operation is called a dummy write. The parameter data denotes the block's data.

Remark 1 (About the dummy block identifier $\perp$)**.** *The dummy block identifier $\perp$ represents a meaningless data block. It is used as a substitute for a real block when the client does not want the server to know that there is no real block for some operation.*

Remark 2 (Block identifier space)**.** *Another weak assumption we make is that each partition O-RAM needs to support non-contiguous block identifiers. In other words, the block identifiers need not be a number within $[1, N]$, where N is the O-RAM capacity. Most existing schemes [3, 5, 13] satisfy this property.*

3.6 Reading a Block

Let $\mathsf{read}(\mathsf{u})$ denote a read operation for a block identified by u. The client looks it up in the position map, and finds out which partition block u is associated with. Suppose that block u is associated with partition p. The client then performs the following steps:

Step 1: Read a block from partition p.

- If block u is found in cache slot p, the client performs a dummy read from partition p of the server, i.e., call $\mathsf{ReadPartition}(p, \perp)$ where $\perp$ denotes a reading a dummy block.
- Otherwise, the client reads block u from partition p of the server by calling $\mathsf{ReadPartition}(p, \mathsf{u})$.

Step 2: Place block u that was fetched in Step 1 into the client's cache, and update the position map.

- Pick a fresh random slot number s, and place block u into cache slot s. This means that block u is scheduled to be evicted to partition s in the future, unless another $\mathsf{read}(\mathsf{u})$ preempts the eviction of this block.
- Update the position map, and associate block u with partition s. In this way, the next $\mathsf{read}(\mathsf{u})$ will cause partition s to be read and written.

Afterwards, a background eviction takes place as described in Section 3.8.

3.7 Writing a Block

Let $\mathsf{write}(\mathsf{u}, \mathsf{data}^*)$ denote writing data^* to the block identified by u. This operation is implemented as a $\mathsf{read}(\mathsf{u})$ operation with the following exception: when block u is placed in the cache during the $\mathsf{read}(\mathsf{u})$ operation, its data is set to data^*.

Observation 1. *Each* read *or* write *operation will cause an independent, random partition to be accessed.*

Proof. (*sketch.*) Consider each client cache slot as an extension of the corresponding server partition. Every time a block u is read or written, it is placed into a fresh random cache slot s, i.e., associated with partition s. Note that every time s is chosen at random, and independent of operations to the O-RAM. The next time block u is accessed, regardless of whether block u has been evicted from the cache slot before this access, the corresponding partition s is read and written. As the value of the random variable s has not been revealed to the server before this, from the server's perspective s is independently and uniformly at random. □

3.8 Background Eviction

To prevent the client data cache from building up, blocks need to be evicted to the server at some point.

```
Access(op, u, data*):
 1: r ← UniformRandom(1 . . . P)
 2: p ← position[u], position[u] ← r
 3: if block u is in slot[p] then
 4:     data ← slot[p].read_and_del(u)
 5:     ReadPartition(p, ⊥)
 6: else
 7:     data ← ReadPartition(p, u)
 8: end if
 9: if op = write then
10:     data ← data*
11: end if
12: slot[r] ← slot[r] ∪ {(u, data)}
13: Call Evict(p)    /*Optional eviction piggy-backed on normal data access requests. Can improve performance by a constant factor.*/
14: Call SequentialEvict(ν) or RandomEvict(ν)
15: return data
```

Figure 3: Algorithm for data access. Read or write a data block identified by u. If op $=$ read, the input parameter data* $=$ None, and the Access operation returns the newly fetched block. If op $=$ write, the Access operation writes the specified data* to the block identified by u, and returns the old value of the block u.

There are two eviction processes:

1. *Piggy-backed evictions* are those that take place on regular O-RAM read or write operations (see Line 13 of Figure 3). Basically, if the data access request operates on a block currently associated with partition p, we can piggy-back a write-back to partition p at that time. The piggy-backed evictions are optional, but their existence can improve performance by a constant factor.

2. *Background evictions* take place at a rate proportional to the data access rate (see Line 14 of Figure 3). The background evictions are completely independent of the data access requests, and therefore can be equivalently thought of as taking place in a separate background thread. Our construction uses an eviction rate of $\nu > 0$, meaning that in expectation, ν number of background evictions are attempted with every data access request. Below are two potential algorithms for background eviction:

   (a) Sequentially scan the cache slots at a fixed rate ν (see the SequentialEvict algorithm in Figure 4);

   (b) At a fixed rate ν, randomly select a slot from all P slots to evict from. (a modified version of the random eviction algorithm is presented as RandomEvict in Figure 4);

Our eviction algorithm is designed to deal with two main challenges:

- **Bounding the cache size.** To avoid the client's data cache from building up indefinitely, the above two eviction processes combined evict blocks at least as fast as blocks are placed into the cache. The actual size of the client's data cache depends on the choice of the background eviction rate ν. We choose $\nu > 0$ to be a constant factor of the actual data request rate. For our practical construction, in Section 5.1 we empirically demonstrate the relationship of ν and the cache size. In the full version of this paper [1], we prove that our background eviction algorithm results in a cache size of $O(P)$.

- **Privacy.** It is important to ensure that the background eviction process does not reveal whether a cache slot is filled or the number of blocks in a slot. For this reason, if an empty slot is selected for eviction, a dummy block is evicted to hide the fact that the cache slot does not contain any real blocks.

Observation 2. *By design, the background eviction process generates a partition access sequence independent of the data access pattern.*

Lemma 1 (Partition access sequence reveals nothing about the data request sequence.)**.** *Let $\vec{y}$ denote a data request sequence. Let $f(\vec{y})$ denote the sequence of partition numbers accessed given data request sequence $\vec{y}$. Then, for any two data request sequences of the same length $\vec{y}$ and $\vec{z}$, $f(\vec{y})$ and $f(\vec{z})$ are identically distributed. In other words, the sequence of partition numbers accessed during the life-time of the O-RAM does not leak any information about the data access pattern.*

```
Evict(p):
1: if len(slot[p]) = 0 then
2:     WritePartition(p, ⊥, None)
3: else
4:     (u, data) ← slot[cnt].pop()
5:     WritePartition(cnt, u, data)
6: end if
```

```
RandomEvict(ν):
1: for i = 1 to ν do    //Assume integer ν
2:     r ← UniformRandom(1 . . . P)
3:     Evict(r)
4: end for
```

```
SequentialEvict(ν):
1: num ← D(ν)        //Pick the number of blocks to evict according to distribution D
2: for i = 1 to num do
3:     cnt ← cnt + 1      //cnt is a global counter for the sequential scan
4:     Evict(cnt)
5: end for
```

Figure 4: Background evicition algorithms with eviction rate ν. Here we provide two candidate eviction algorithms SequentialEvict and RandomEvict. SequentialEvict determines the number of blocks to evict num based on a prescribed distribution $\mathcal{D}(\nu)$ and sequentially scans num slots to evict from. RandomEvict samples $\nu \in \mathbb{N}$ random slots (with replacement) to evict from. **In both SequentialEvict and RandomEvict, if a slot selected for eviction is empty, evict a dummy block for the sake of security.**

Proof. The sequence of partition numbers are generated in two ways 1) the regular read or write operations, and 2) the background eviction processes. Due to Observations 1 and 2, both of the above processes generate a sequence of partition numbers completely independent of the data access pattern. □

Theorem 1. *Suppose that each partition uses a secure O-RAM construction, then the new O-RAM construction obtained by applying the partitioning framework over P partition O-RAMs is also secure.*

Proof. Straightforward conclusion from Lemma 1 and the security of the partition O-RAM. □

3.9 Algorithm Pseudo-code

Figures 3 and 4 describe in formal pseudo-code our O-RAM operations based on the partitioning framework. For ease of presentation, in Figure 3, we unify read and write operations into an $\mathsf{Access}(\mathsf{op}, \mathsf{u}, \mathsf{data}^*)$ operation.

4 Practical Construction

In this section, we apply the partitioning techniques mentioned in the previous section to obtain a practical construction with an amortized cost of $20 \sim 35$X overhead under typical settings, about 63 times faster than the best known construction. The client storage is typically 0.01% to 0.3% of the O-RAM capacity. The worst-case cost of this construction is $O(\sqrt{N})$, and we will later show how to allow concurrent shuffling and reads to reduce the worst-case cost to $O(\log N)$ (Section 6).

While the practical construction require cN client-side storage, the constant c is so small that our cN is smaller than or comparable to $\sqrt{N}$ for typical storage sizes ranging from gigabytes to terabytes. For the sake of theoretic interest, in Appendix A, we show how to recursively apply our O-RAM construction to part of the client-side storage, and reduce the client-side storage to $O(\sqrt{N})$, while incurring only a logarithmic factor in the amortized cost.

4.1 Overview

Our practical construction uses the partitioning framework (Section 3). For the partitions, we use our own highly optimized O-RAM construction resembling the Pinkas-Reinman O-RAM at a very high level [13].

Choice of parameters. In this practical construction, we choose $P = \sqrt{N}$ partitions, each with $\sqrt{N}$ blocks on average. We use the SequentialEvict algorithm as the background eviction algorithm. Every time SequentialEvict is invoked with an eviction rate of ν, it decides the number of blocks to evict numbased on a bounded geometric distribution with mean ν, i.e., let c be a constant representing the maximum number of evictions per data access operation, then $\Pr[\mathsf{num} = k] \propto p^k$ for $0 \le k \le c$, and $\Pr[\mathsf{num} = k] = 0$ for $k > c$. Here is $0 < p < 1$ is a probability dependent on ν and c.

As mentioned earlier, the piggybacked evictions enable practical savings up to a constant factor, so we use

```
ReadPartition(p, u):
1:  L ← number of levels
2:  for ℓ = 0, 1, ..., L − 1   (in parallel)  do
3:      if level ℓ of partition p is not filled then
4:          continue    // skip empty levels
5:      end if
6:      if block u is in partition p, level ℓ then
7:          i = position[u].index
8:      else
9:          i = nextDummy[p, ℓ]
10:         nextDummy[p, ℓ] ← nextDummy[p, ℓ] + 1
11:     end if
12:     i' = PRP (K[p, ℓ], i)
13:     Fetch from the server the block in partition p, level ℓ, and offset i'.
14:     Decrypt the block with the key K[p, ℓ].
15: end for
```

Figure 5: The ReadPartition operation of our practical construction that reads the block with id u from partition p.

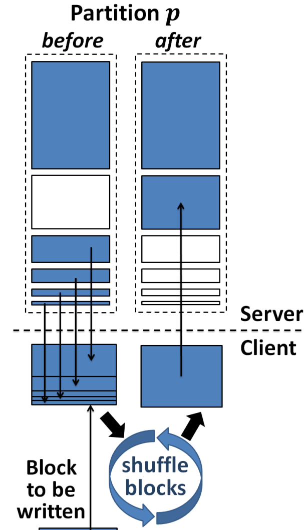

Figure 6: WritePartition leads to the shuffling of consecutively filled levels into the first empty level.

piggybacked evictions in this construction.

Optimized partition O-RAM construction. While any existing O-RAM construction satisfying the modified ReadPartition and WritePartition semantics can be used as a partition O-RAM, we propose our own highly optimized partition O-RAM. Our partition O-RAM construction resembles the Pinkas-Reinman O-RAM at a very high level [13], but with several optimizations to gain practical savings. The practical savings come from at least three sources, in comparison with the Pinkas-Reinman construction:

- *Local sorting.* Due our partitioning framework, each partition is now of size $O(\sqrt{N})$ blocks. This allows us to use a client shuffling buffer of size $O(\sqrt{N})$ blocks to reshuffle the partition locally, thereby eliminating the need for extremely expensive oblivious sorting procedures during a reshuffling operation. This is our most significant source of saving in comparison with all other existing schemes.
- *No Cuckoo hashing.* Second, since we use a position map to save the locations of all blocks, we no longer need Cuckoo hashing, thereby saving a 2X factor for lookups.
- *Compressed data transfer during reshuffling.* Third, during the reshuffling operation, the client only reads blocks from each level that have not been previously read. Also, when the client writes back a set of shuffled blocks to the server (at least half of which are dummy blocks), it uses a compression algorithm to compress the shuffling buffer down to half its size. These two optimizations save about another 2X factor.
- *Latency reduction.* In the practical construction, the client saves a position map which records the locations of each block on the server. This allows the client to query the $O(\log N)$ levels in each partition in a single round-trip, thereby reducing the latency to $O(1)$.

4.2 Partition Layout

As mentioned earlier, we choose $P = \sqrt{N}$ partitions for the practical construction. Each partition consists of $L = \log_2(\sqrt{N}) + 1 = \frac{1}{2}\log_2 N + 1$ levels, indexed by $0, 1, \ldots, \frac{1}{2}\log_2 N$ respectively. Except for the top level, each level ℓ has $2 \cdot 2^\ell$ blocks, among which at most half are real blocks, and the rest (at least half) are dummy blocks.

The top level where $\ell = \frac{1}{2}\log_2 N$ has $2 \cdot 2^\ell + \epsilon = 2\sqrt{N} + \epsilon$ blocks, where the surplus ϵ is due to the fact that some partition may have more blocks than others when the blocks are assigned in a random fashion to the partitions. Due to a standard balls and bins argument [14], each partition's maximum size (including real and dummy blocks) should be $4\sqrt{N} + o(\sqrt{N})$ such that the failure probability $\frac{1}{\text{poly}(N)}$. In Appendix 5.3, we empirically demonstrate that in practice, the maximum number of real blocks in each partition is not more than $1.15\sqrt{N}$ for $N \geq 20$, hence the partition capacity is no more than $4 \cdot 1.15\sqrt{N} = 4.6\sqrt{N}$ blocks, and the total server storage is no more than $4.6N$ blocks. In Appendix B.2, we propose an optimization to reduce the server storage to less than $3.2N$ blocks.

At any given time, a partition on the server might have some of its levels *filled* with blocks and others *unfilled*. The top partition is always filled. Also, a data block can be located in any partition, any filled level, and any offset within the level. In the practical construction, we extend the position map of the partition framework to also keep track of the level number and offset of each block.

From the perspective of the server, all blocks within a level are pseudo-randomly arranged. Because the blocks are encrypted, the server cannot even tell which blocks are real and which ones are dummy. We use keyed pseudo-random permutation (PRP) function for permuting blocks within a level in our construction. When the context is clear, we omit the range or the PRP function in the pseudo-code.

4.3 Setup

The initial set of filled levels that contain the blocks depends on the partition number p. In order to better amortize the reshuffling costs of our scheme, the client randomly chooses which levels of each partition will be initially filled (with the restriction that the top level is always filled). Note that there are 2^{L-1} such possible *fillings* of a partition where L is the number of levels in the partition. The client notifies the server which levels are filled but does not write the actual blocks to the server because the blocks are initially zeroed and their values can be calculated implicitly by storing one bit for each level of each partition. This bit indicates if the entire level has never been reshuffled and is hence zeroed.

4.4 Reading from a Partition

The ReadPartition operation reads the block with id u from partition p as described in Figure 5. If $\mathsf{u} = \perp$, then the ReadPartition operation is a dummy read and a dummy block is read from each filled level. If $\mathsf{u} \neq \perp$, block u is read from the level that contains it, and a dummy block is read from from each of the other filled levels. Note that all of the fetches from the server are performed in parallel and hence this operation has single round trip latency unlike existing schemes [3, 4, 11] which take $\Omega(\log N)$ round trips.

4.5 Writing to a Partition

Each write to a partition is essentially a reshuffling operation performed on consecutively filled levels in a partition. Therefore, we sometimes use the terms "write" and "shuffling" interchangeably. First, unread blocks from consecutively filled levels of the partition are read from the server into the client's shuffling buffer. Then, the client permutes the shuffling buffer according to a pseudo-random permutation (PRP) function. Finally, the client uploads its shuffling buffer into the first unfilled level and marks all of the levels below it as unfilled. The detailed pseudo-code for the WritePartition operation is given in Figure 7.

There is an exception when all levels of a partition are filled. In that case, the reshuffling operation is performed on all levels, but at the end, the top level (which was already filled) is overwritten with the contents of the shuffling buffer and the remaining levels are marked as unfilled. Note that the shuffling buffer is never bigger than the top level because only *unread real (not dummy)* blocks are placed into the shuffling buffer before it is padded with dummy blocks. Since the top level is big enough to contain all of the real items inside a partition, it can hold the entire shuffling buffer.

During a reshuffling operation, the client uses the pseudo-random permutation PRP to determine the offset of all blocks (real and dummy) within a level on the server. Every time blocks are shuffled and written into the next level, the client generates a fresh random PRP key $K[p, \ell]$ so that blocks end up at random offsets every time that level is constructed. The client remembers the keys for all levels of all partitions in its local cache.

Reading levels during shuffling. When the client reads a partition's levels into the shuffling buffer (Line 5 of Figure 7), it reads exactly 2^ℓ previously unread blocks. Unread blocks are those that were written during a WritePartition operation when the level was last constructed, but have not been read by a ReadPartition operation since then. The client only needs to read the unread blocks because the read blocks were already logically removed from the partition when they were read. There is a further restriction that among those 2^ℓ blocks must be all of the *unread real (non-dummy)* blocks. Since a level contains up to 2^ℓ real blocks, and there are always at least 2^ℓ unread blocks in a level, this is always possible.

The client can compute which blocks have been read/unread for each level. It does this by first fetching from the server a small amount of metadata for the level that contains the list of all blocks (read and unread) that were in the level when it was last filled. Then the client looks up each of those blocks in the position map to determine if the most recent version of that block is still in this level. Hence, the client can obtain the list of unread real blocks. The offsets of the of unread dummy blocks can be easily obtained by repeatedly applying the PRP function to nextDummy and incrementing nextDummy. Note that for security, the offsets of the 2^ℓ unread blocks must be first computed and then the blocks must be read in order of their offset (or some other order independent

WritePartition($p, \mathsf{u}^*, \mathsf{data}^*$):

1: *// Read consecutively filled levels into the client's shuffling buffer denoted* sbuffer*.*
2: $\ell_0 \leftarrow$ last consecutively filled level
3: **for** $\ell = 0$ to ℓ_0 **do**
4: Fetch the metadata (list of block ID's) for level ℓ in partition p. Decrypt with key $K[p, \ell]$.
5: Fetch exactly 2^ℓ previously unread blocks from level ℓ into sbuffer such that all unread real blocks are among them. Decrypt everything with the key $K[p, \ell]$. Ignore dummy blocks when they arrive.
6: Mark level ℓ in partition p as *unfilled*.
7: **end for**
8: $\ell = \min(\ell_0, L-1)$ *// Don't spill above the top level.*
9: Add the $(\mathsf{u}^*, \mathsf{data}^*)$ to sbuffer.
10: $k \leftarrow$ number of real blocks in sbuffer.
11: **for** $i = 1$ to k **do**
12: Let $(\mathsf{u}, \mathsf{data}) = \mathsf{sbuffer}[i]$
13: $\mathsf{position}[u] \leftarrow \{p, \ell, i\}$ *// update position map*
14: **end for**
15: $K[p, \ell] \leftarrow_R \mathcal{K}$ *// generate fresh key for level ℓ in partition p*
16: Pad the shuffling buffer with dummy blocks up to length $2 \cdot 2^\ell$. *// The first k blocks are real and the rest are dummy.*
17: Permute sbuffer with $\mathsf{PRP}_\ell(K[p, \ell], \cdot)$. A block originally at index i in the shuffling buffer is now located at offset i' in the shuffling buffer, where $i' = \mathsf{PRP}_\ell(K[p, \ell], i)$
18: Write the shuffling buffer into level ℓ in partition p on the server, encrypted with key $K[p, \ell]$.
19: Write the metadata (list of block ID's) of level ℓ in partition p to the server, encrypted with key $K[p, \ell]$.
20: Mark level ℓ in partition p as *filled*.
21: $\mathsf{nextDummy}[p, \ell] \leftarrow k + 1$ *// initialize counter to first dummy block.*

Notations:

$K[p, \ell]$	Secret key for partition p, level ℓ. (PRP or symmetric encryption key depending on context)
$\mathsf{nextDummy}[p, \ell]$	Index of next unread dummy block for partition p, level ℓ.
$\{p, \ell, i\} \leftarrow \mathsf{position}[u]$	Position information for block u (partition p, level ℓ, index i within the level).

Figure 7: The WritePartition operation of our practical construction that writes block u^* with data^* to partition p.

of which blocks are real/dummy).

4.6 Security

Our practical construction has the following security guarantees: **obliviousness (privacy), confidentiality, and authentication (with freshness)**.

Theorem 2. *The practical construction is oblivious according to Definition 1.*

Proof. The proof is presented in the full version of the paper [1]. □

Theorem 3. *The practical construction provides confidentiality and authentication (with freshness) of all data stored on the server.*

Proof. The proof is presented in the full version of the paper [1]. □

It should be noted that although our partition O-RAM construction resembles the Pinkas-Reinman construction, it does not have the security flaw discovered by Goodrich and Mitzenmacher [5] because it does not use Cuckoo hash tables.

5 Experimental Results

For our experiments, we implemented a simulator of our construction. Each read/write operation is simulated and the simulator keeps track of exactly where each block is located, the amount of client side storage used, and the total bytes transferred for all communication between the client and server. We also implemented a simulator for the best previously known O-RAM scheme for comparison.

For each parametrization of our O-RAM construction, we simulated exactly $3N$ read/write operations. For example, for each O-RAM instances with $N = 2^{28}$ blocks, we simulated about 800 million operations. We used a round-robin access pattern which maximizes the size of the client's data cache of our construction by maximizing the probability of a cache miss. Therefore our results always show the worst case cache size. Also, because our construction is oblivious, our amortized cost

measurements are independent of the simulated access pattern. We used the level compression, server storage reduction, and piggy-backed eviction optimizations described in Appendix B.

5.1 Client Storage and Bandwidth

In this experiment, we measure the performance overhead (or *bandwidth overhead*) of our O-RAM. An O-RAM scheme with a bandwidth overhead of w performs w times the data transfer as an unsecured remote storage protocol. In the experiments we ignore the metadata needed to store and fetch blocks because in practice it is much smaller than the block size. For example, we may have 256 KB blocks, but the metadata will be only a few bytes.

In our scheme, the bandwidth overhead depends on the background eviction rate, and the background eviction rate determines the client's cache size. The client is free to choose its cache size by using the appropriate eviction rate. Figure 8 shows the correlation between the background eviction and cache size as measured in our simulation.

Once the client chooses its cache size it has determined the total amount of client storage. As previously mentioned, our scheme requires $O(\sqrt{N}B)$ bytes of client storage plus an extra cNB bytes of client storage for the position map with a very small constant. For most practical values of N and B, the position map is much smaller than the remaining $O(\sqrt{N}B)$ bytes of client storage, so the client storage approximately scales like $O(\sqrt{N}B)$ bytes. We therefore express the total client storage as $k\sqrt{N}B$ bytes. Then, we ask the question: How does the client's choice of k affect the bandwidth overhead of our entire O-RAM construction? Figure 9 shows this trade-off between the total client storage and the bandwidth overhead.

5.2 Comparison with Previous Work

To the best of our knowledge, the most practical existing O-RAM construction was developed by Goodrich *et al.* [5]. It works by constructing a hierarchy of Cuckoo hash tables via Map-Reduce jobs and an efficient sorting algorithm which utilizes N^a ($a < 1$) blocks of client-side storage. We implemented a simulator that estimates a lower bound on the performance of their construction. Then we compared it to the simulation of our construction.

To be fair, we parametrized both our and their construction to use the exact same amount of client storage: $4\sqrt{N}B$ bytes. The client storage includes all of our client data structures, including our position map (stored uncompressed). We parametrized both constructions for exactly 1 TB O-RAM capacity (meaning that each construction could store a total of 1 TB of blocks). We varied the number of blocks from $N = 2^{16}$ to $N = 2^{24}$. Since the O-RAM size was fixed to 1 TB, the blocks size varied between $B = 2^{24}$ bytes and $B = 2^{16}$ bytes. Table 11 shows the results. As it can be clearly seen, our construction uses *63 to 66 times less bandwidth* than the best previously known scheme for the exact same parameters.

5.3 Partition Capacity

Finally, we examine the effects of splitting up the O-RAM into partitions. Recall that in our practical construction with N blocks, we have split up the server storage into $\sqrt{N}$ partitions each containing about $\sqrt{N}$ blocks. Since the blocks are placed into partitions uniformly *randomly* rather than uniformly, a partition might end up with slightly more or less than $\sqrt{N}$ blocks. For security reasons, we want to hide from the server how many blocks are in each partition at any given time, so a partition must be large enough to contain (with high probability) the maximum number of blocks that could end up in a single partition.

Figure 10 shows how many times more blocks a partition contains than the expected number: $\sqrt{N}$. Note that as the size of the O-RAM grows, the maximum size of a partition approaches its expected size. In fact, one can formally show that the maximum number of real data blocks in each partition over time is $\sqrt{N} + o(\sqrt{N})$ [14]. Hence, for large enough N, the partition capacity is less than 5% larger than $\sqrt{N}$ blocks.

6 Reducing the Worst-Case Cost With Concurrency

The constructions described thus far have a worst-case cost $O(\sqrt{N})$ because a WritePartition operation sometimes causes a reshuffling of $O(\sqrt{N})$ blocks. We reduce the worst-case cost by spreading out expensive WritePartition operations of $O(\sqrt{N})$ cost over a long period of time, and at each time step performing $O(\log N)$ work.

To achieve this, we allow reads and writes (i.e., reshuffling) to a partition to happen concurrently. This way, an operation does not have to wait for previous long-running operations to complete before executing. We introduce an *amortizer* which keeps track of which partitions need to be reshuffled, and schedules $O(\log N)$ work (or $O((\log N)^2)$ for the theoretic recursive construction) per time step. There is a slight storage cost of allowing these operations to be done in parallel, but we will later show that concurrency does not increase the

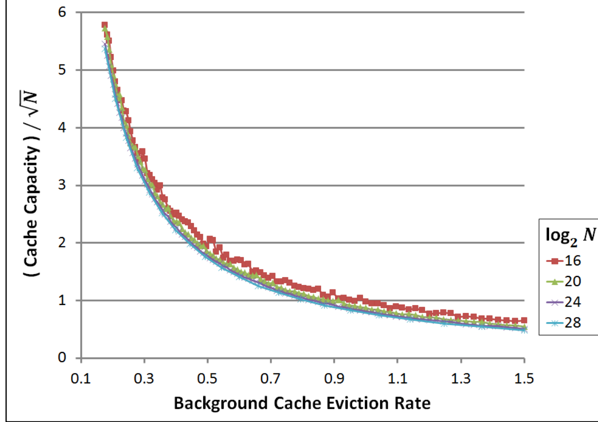

Figure 8: Background Eviction Rate vs. Cache Capacity. The x-axis is the eviction rate, defined as the ratio of background evictions over real data requests. For example, an eviction rate of 1 suggests an equal rate of data requests and background evictions. The y-axis is the quantity $(\text{cache capacity})/\sqrt{N}$, where cache capacity is the maximum number of data blocks in the cache over the course of time.

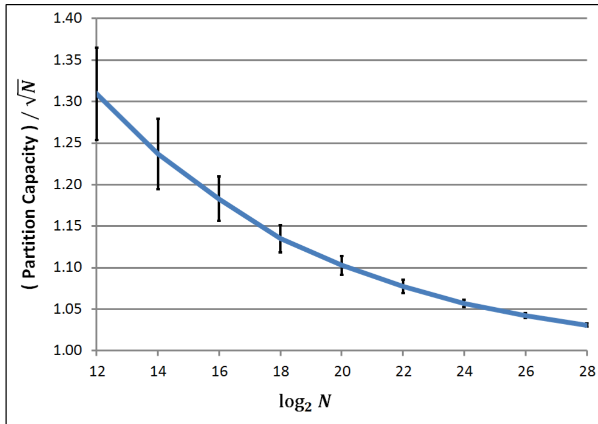

Figure 10: Partition capacity. The y-axis is the quantity $(\text{partition capacity})/\sqrt{N}$, where partition capacity is the maximum number of real data blocks that the partition must be able to hold, i.e., the maximum number of real data blocks inside a partition over time.

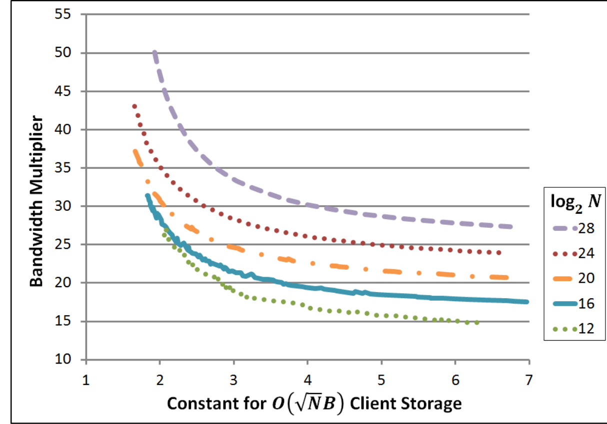

Figure 9: Trade-off between Client Storage and Bandwidth. The plot shows what bandwidth overhead a client can achieve by using exactly $k\sqrt{NB}$ bytes of client storage for different values of k (horizontal axis). The client storage includes the cache, sorting buffer, and an uncompressed position map. A block size of 256 KB was assumed.

# Blocks	Block Size	Practical Performance	
		Ours	Best Known [5]
2^{16}	16 MB	18.4X	$>$ 1165X
2^{18}	4 MB	19.9X	$>$ 1288X
2^{20}	1 MB	21.5X	$>$ 1408X
2^{22}	256 KB	23.2X	$>$ 1529X
2^{24}	64 KB	25.0X	$>$ 1651X

Figure 11: Comparison between our construction and the best known previous O-RAM construction. A 1 TB O-RAM is considered with both constructions using exactly $4\sqrt{NB}$ client storage. The practical performance is the number of client-server operations per O-RAM operation. Our construction has a *63 to 66 times better performance* than the best previously known scheme for the exact same parameters.

asymptotic storage and amortized costs of our constructions.

By performing operations concurrently, we decrease the worst-case cost of the practical construction from $O(\sqrt{N})$ to $O(\log N)$ and we reduce the worst-case cost of the recursive construction from $O(\sqrt{N})$ to $O((\log N)^2)$. Our concurrent constructions preserve the same amortized cost as their non-concurrent counterparts; however, in the concurrent constructions, the *worst-case cost is the same as the amortized cost*. Furthermore, in the concurrent practical construction, the latency is $O(1)$ just like the non-concurrent practical construction, as each data request requires only a single round-trip to complete.

6.1 Overview

We reduce the worst case cost of our constructions by inserting an Amortizer component into our system which explicitly amortizes ReadPartition and WritePartition operations as described in Figure 12. Specifically, the Amortizer schedules a ReadPartition operation as a ConcurrentReadPartition operation, so the read can occur while shuffling. A ReadPartition al-

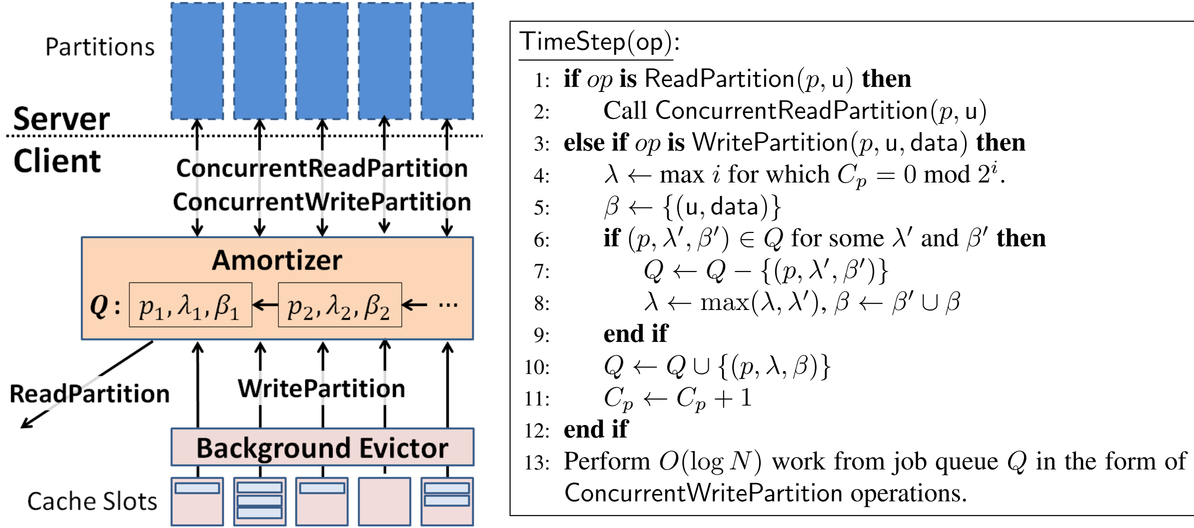

Figure 12: The Amortizer component helps reduce the worst case costs of our constructions. It is inserted between the background eviction process and the server-side partitions as shown on the left. The component executes one operation per time step as defined on the right.

ways finishes in $O(\log N)$ time. Upon a WritePartition operation (which invokes the shuffling of a partition), the Amortizer creates a new shuffling "job", and appends it to a queue Q of jobs. The Amortizer schedules $O(\log N)$ amount of work to be done per *time step* for jobs in the shuffling queue.

If reads are taking place concurrently with shuffling, special care needs to be taken to avoid leakages through the access pattern. This will be explained in the detailed scheme description below.

Terminology. To aid understanding, it helps to define the following terminology.

- **Job.** A job (p, λ, β) denotes a reshuffling of levels $0, \ldots, \lambda$ of partition p and then writing the blocks in β to partition p on the server.
- **Job Queue** Q**.** The *job queue* Q is a FIFO list of jobs. It is also possible to remove jobs that are not necessarily at the head of the queue for the purpose of merging them with other jobs, however jobs are always added at the tail.
- **Partition counter.** Let $C_p \in \mathbb{Z}_s$ denote a counter for partition p, where s is the maximum capacity of partition p. All operations on C_p are modulus s.
- **Work.** The *work* of an operation is measured in terms of the number of blocks that it reads and writes to partitions on the server.

Handling ReadPartition **operations.** The amortizer performs a ReadPartition(p, u) operation as a ConcurrentReadPartition(p, u) operation as defined in Sections 6.1.1 for the practical and recursive constructions respectively. If block u is cached by a previous ConcurrentReadPartition operation, then it is instead read from β where $(p, \lambda, \beta) \in Q$ for some λ and β.

Handling WritePartition **operations.** The amortizer component handles a WritePartition operation by adding it to the job queue Q. The job is later dequeued in some time step and processed (possibly across multiple time steps). If the queue already has a job involving the same partition, the existing job is merged with the new job for the current WritePartition operation. Specifically, if one job requires shuffling levels $0, \ldots, \lambda$ and the other job requires shuffling levels $0, \ldots, \lambda'$, we merge the two jobs into a job that requires shuffling levels $0, \ldots, \max(\lambda, \lambda')$. We also merge the blocks to be written by both jobs.

Processing jobs from the job queue. For each time step, the reshuffling component perform $w \log N$ work for a predetermined constant w such that $w \log N$ is greater than or equal to the amortized cost of the construction. Part of that work may be consumed by a ConcurrentReadPartition operation executing at the beginning of the time step as described in Figure 12. The remaining work is performed in the form of jobs obtained from Q.

Definition 2 (Processing a job)**.** *A job* (p, λ, β) *is performed as a* ConcurrentWritePartition(p, λ, β) *operation that reshuffles levels* $0, \ldots, \lambda$ *of partition* p *and writes the blocks in* β *to partition* p*. The* ConcurrentWritePartition *operation is described in Sections 6.1.2 for the practical and recursive constructions respectively. Additionally, every block read and*

written to the server is counted to calculate the amount of work performed as the job is running. A job may be paused after having completed part of its work.

Jobs are always dequeued from the head of Q. At any point only a single job called the *current job* is being processed unless the queue is empty (then there are no jobs to process). Each job starts after the previous job has completed, hence multiple jobs are never processed at the same time.

If the current job does not consume all of the remaining work of the time step, the the next job in Q becomes the current job, and so on. the current job is *paused* when the total amount of work performed in the time step is exactly $w \log N$. In the next time step, the current job is resumed from where it was paused.

We now explain how to perform ConcurrentReadPartition and ConcurrentWritePartition operations in the practical construction to achieve an $O(\log N)$ worst-case cost with high probability.

6.1.1 Concurrent Reads

The client performs the ConcurrentReadPartition(p, u) operation by reading 0 or 1 blocks from each filled level ℓ of partition p on the server as follows:

- If level ℓ in partition p contains block u, then
  - Read block u from level ℓ in partition p on the server like in the ReadPartition operation.
- If level ℓ in partition p does not contain block u and this level is not being reshuffled, then
  - Read the next dummy block from level ℓ in partition p on the server like in the ReadPartition operation.
- If level ℓ in partition p does not contain block u and this level is being reshuffled, then
  - Recall that when level ℓ is being reshuffled, 2^ℓ previously unread blocks are chosen to be read. Let S be the identifiers of that set of blocks for level ℓ in partition p.
  - Let $S' \subseteq S$ be the IDs of blocks in S that were not read by a previous ConcurrentReadPartition operation after the level started being reshuffled. The client keeps track of S' for each level by first setting it to S when a ConcurrentWritePartition operation begins and then removing u from S' after every ConcurrentReadPartition(p, u) operation.
  - If S' is not empty, the client reads a random block in S' from the server.
  - If S' is empty, then the client doesn't read anything from level ℓ in partition p. Revealing this fact to the server does not affect the security of the construction because the server already knows that the client has the entire level stored in its reshuffling buffer.

When $\mathsf{u} = \perp$, block u is treated as not being contained in any level. Due to concurrency, it is possible that a level of a partition needs to be read during reshuffling. In that case, blocks may be read directly from the client's shuffling buffer containing the level.

6.1.2 Concurrent Writes

A ConcurrentWritePartition(p, λ, β) operation is performed like the non-concurrent WritePartition operation described in Figure 7, except for three differences.

The first difference is that the client does not shuffle based on the last consecutively filled level. Instead it shuffles the levels $0, ..., \lambda$ which may include a few more levels than the WritePartition operation would shuffle.

The second difference is that at Line 9 of Figure 7, the client adds all of the blocks in β to the buffer.

The third difference is at Line 4 of Figure 7. In the non-concurrent construction, client fetches the list of 2^ℓ blocks ID's in a level that is about to be reshuffled. It then uses this list to determine which blocks have already been read as described in Section 4.5). Because 2^ℓ is $O(\sqrt{N})$ fetching this metadata in the non-concurrent construction takes $O(\sqrt{N})$ work in the worst case.

To ensure the worst case cost of the concurrent construction is $O(\log N)$, the metadata is stored as a bit array by the client. This bit array indicates which real blocks in that level have already been read. The client also knows which dummy blocks have been read because it already stores the nextDummy counter and it can apply the PRP function for all dummy blocks between $k + 1$ and nextDummy where k is the number of real blocks in a level. Observe that the client only needs to store a single bit for each real block on the server. Hence this only increases the client storage by $2N + \epsilon\sqrt{N}$ bits, which is significantly smaller than the size of index structure that the client already stores.

Theorem 4 (Practical concurrent construction)**.** *With $1 - \frac{1}{poly(N)}$ probability, the concurrent practical construction described above has $O(\log N)$ worst-case and amortized cost, and requires cN client-side storage with a very small c, and $O(N)$ server-side storage.*

The formal proof of the above theorem is in the full version of the paper [1].

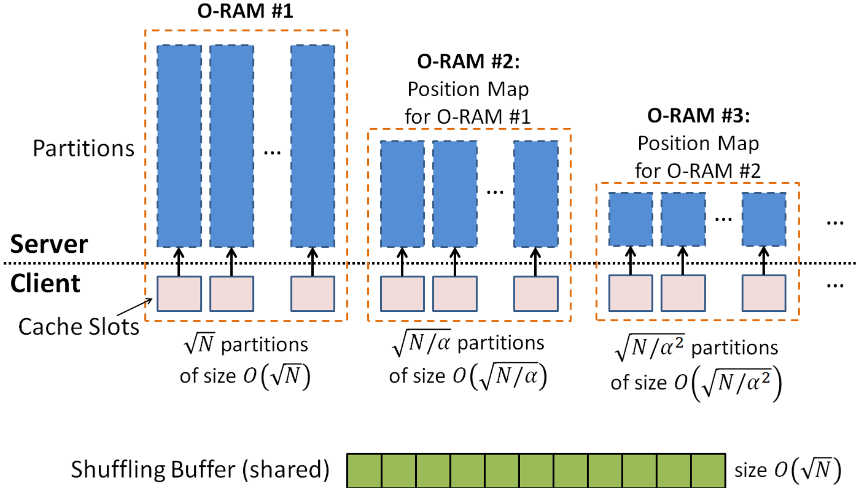

Figure 13: The recursive construction.

Appendices

A Recursive Construction

The practical (non-current and concurrent) constructions described so far are geared towards optimal practical performance. However, they are arguably not ideal in terms of asymptotic performance, since they require a linear fraction of client-side storage for storing a position map of linear size.

For theoretic interest, we describe in this section how to recursively apply our O-RAM constructions to store the position map on the server, thereby obtaining O-RAM constructions with $O(\sqrt{N})$ client-side storage, while incurring only a logarithmic factor in terms of amortized and worst-case cost.

We first describe the recursive, non-concurrent construction which achieves $O((\log N)^2)$ amortized cost, and $O(\sqrt{N})$ worst-case cost. We then describe how to apply the concurrency techniques to further reduce the worst-case cost to $O((\log N)^2)$, such that the worst-case cost and amortized cost will be the same.

A.1 Recursive Non-Concurrent Construction

Intuition. Instead of storing the linearly sized position map locally, the client stores it in a separate O-RAM on the server. Furthermore, the O-RAM for the position map is guaranteed to be a constant factor smaller than the original O-RAM. In other words, each level of recursion reduces the O-RAM capacity by a constant factor. After a logarithmic number of recursions, the size of the position map stored on the client is reduced to $O(1)$. The total size of all data caches is $O(\sqrt{N})$, hence the construction uses $O(\sqrt{N})$ client storage.

For the recursive construction, we employ the Goodrich-Mitzenmacher O-RAM scheme as the partition O-RAM. Specifically, we employ their O-RAM scheme which for an O-RAM of capacity N, achieves $O(\log N)$ amortized cost and $O(N)$ worst-case cost, while using $O(\sqrt{N})$ client-side storage, and $O(N)$ server-side storage.

Definition 3 (O-RAM$_{GM}$)**.** *Let O-RAM$_{GM}$ denote the Goodrich-Mitzenmacher O-RAM scheme [5]: for an O-RAM of capacity N, the O-RAM$_{GM}$ scheme achieves $O(\log N)$ amortized cost, and $O(N)$ worst-case cost, while using $O(\sqrt{N})$ client-side storage, and $O(N)$ server-side storage.*

Definition 4 (O-RAM$_{base}$)**.** *Let O-RAM$_{base}$ denote the O-RAM scheme derived through the partitioning framework with the following parameterizations: (1) we set $P = \sqrt{N}$ denote the number of partitions, where each partition has approximately $\sqrt{N}$ blocks, and (2) we use the O-RAM$_{GM}$ construction as the partition O-RAM.*

Notice that in O-RAM$_{base}$, the client requires a data cache of size $O(\sqrt{N})$ and a position map of size less than $\frac{2N \log N}{B}$ blocks. If we assume that the data block size $B > 2 \log N$, then the client needs to store at most $\frac{2N \log N}{B} + \sqrt{N} \log N = \frac{N}{\alpha} + o(N)$ blocks of data, where the compression rate $\alpha = \frac{B}{2 \log N} > 1$. To reduce the client-side storage, we can recursively apply the O-RAM construction to store the position map on the server side.

Definition 5 (Recursive construction: O-RAM*)**.** *Let O-RAM* denote a recursive O-RAM scheme constructed*

as below. In O-RAM$_{base}$, the client needs to store a position map of size $cN (c < 1)$. Now, instead of storing the position map locally on the client, store it in a recursive O-RAM on the server side. The pseudocode of the O-RAM scheme would be otherwise be the same as in Figure 3 except that Line 2 is modified to the the following recursive O-RAM lookup and update operation:*

The position position[u] *is stored in block* u$/\alpha$ *of the smaller O-RAM. The client looks up this block, updates the corresponding entry* position[u] *with the new value r, and writes the new block back. Note that the read and update can be achieved in* **a single O-RAM operation** *to the smaller O-RAM.*

Theorem 5 (Recursive O-RAM construction)**.** *Suppose that the block size $B > 2 \log N$, and that the number of data accesses $M < N^k$ for some $k = O(\frac{\sqrt{N}}{\log N})$. Our recursive O-RAM construction achieves $O((\log N)^2)$ amortized cost, $O(\sqrt{N})$ worst-case cost, and requires $O(N)$ server-side storage, and $O(\sqrt{N})$ client-side storage.*

The proof of the above theorem is presented in in the full version of the paper [1].

A.2 Recursive Concurrent Construction

Using similar concurrency techniques as in Section 6, we can further reduce the worst-case cost of the recursive construction to $O((\log N)^2)$. Recall that the recursive construction differs from the practical construction in two ways: (1) it uses the O-RAM$_{GM}$ (Goodrich-Mitzenmacher [5]) scheme as the partition O-RAM and (2) it recurses on its position map. We explain how to to perform concurrent operations in the O-RAM$_{GM}$ scheme to reduce the worst case cost of the base construction to $O(\log N)$ with high probability. Then when the recursion is applied, the recursive construction achieves a worst case cost of $O((\log N)^2)$ with high probability.

A.2.1 Concurrent Reads

As concurrency allows reshuffles to be queued for later, it is possible that a level ℓ is read more than 2^ℓ times in between reshufflings. The O-RAM$_{GM}$ scheme imposes a restriction that at most 2^ℓ blocks can be read from a level before it must be reshuffled by using a set of 2^ℓ dummy blocks. We observe that it is possible to perform a *dummy read* operation instead of using a dummy block and performing a *normal read* on it. This essentially eliminates the use of dummy blocks. Note that the same idea was suggested in the work by Goodrich *et al.* [8].

A dummy read operation ConcurrentReadPartition$(p, \perp)$ is performed by reading two random blocks within a level instead of applying a Cuckoo hash on a element from small domain. Observe that the Cuckoo hashes for real read operations output uniformly random block positions. Because the blocks read by dummy read operation are also chosen from a uniformly random distribution, dummy reads are indistinguishable from real reads.

This observation allows the client to securely perform a ConcurrentReadPartition(p, u) operation as follows. For each level (from smaller to larger) of partition p, as usual the client performs a Cuckoo hash of the block identifier u to determine which two blocks to read within the level. Once the block is found in some level, the client performs dummy reads on subsequent levels. The client always first checks the local storage to see if the block is in the job queue. If so, then the client performs dummy reads of all levels.

In summary, instead of reading specific dummy blocks (which can be exhausted since there are only 2^ℓ dummy blocks in level ℓ), the client performs dummy reads by choosing two random positions in the level.

A.2.2 Concurrent Writes

In the concurrent recursive construction, the ConcurrentWritePartition(p, λ, β) operation is performed by first reshuffling levels $0, ..., \lambda$ along with the blocks in β using the oblivious shuffling protocol of the O-RAM$_{GM}$ scheme. After the reshuffling has completed, the updated Cuckoo hash tables have been formed.

Theorem 6 (Recursive concurrent O-RAM)**.** *Assume that the block size $B > 2 \log N$. With $1 - \frac{1}{poly(N)}$ probability, the concurrent recursive construction described above has $O((\log N)^2)$ worst-case and amortized cost, and requires $O(\sqrt{N})$ client-side storage, and $O(N)$ server-side storage.*

The formal proof of the above theorem will be included in the full version of the paper [1].

B Optimizations and Extensions

B.1 Compressing the Position Map

The position map is *highly* compressible under realistic workloads due to the natural sequentiality of data accesses. Overall we can compress the position map to about 0.255 bytes per block. Hence its compressed size is $0.225N$ bytes. Even for an extremely large, 1024 TB O-RAM with $N = 2^{32}$ blocks, the position map will be under 1 GB in size. We now explain how to compress the position map.

Compressing partition numbers. In [10], Oprea *et al.* showed that real-world file systems induce almost entirely sequential access patterns. They used this observation to compress a data structure that stores a counter of how many times each block has been accessed. Their experimental results on real-world traces show that the compressed data structure stores about 0.13 bytes per block. Every time a block is accessed, their data structure stores a unique value (specifically, a counter) for that block. In our construction, instead of placing newly read blocks in a random cache slot, we can place them in a pseudo-random cache slot determined by the block id and counter. Specifically, when block i is accessed for the j'th time (i.e., it's counter is j), it is placed in cache slot $\mathsf{PRF}(i.j)$. $\mathsf{PRF}(\cdot)$ is a pseudo-random function that outputs a cache slot (or partition) number.

Compressing level numbers. Recall that each partition contains L levels such that level ℓ contains at most 2^ℓ real blocks. We can represent the level number of each block by using only 1 bit per block on average, regardless of the number of levels. This can be easily shown by computing the entropy of the level number as follows. If all levels are filled, each block has probability $2^{\ell-L}$ of being in level ℓ. Then the entropy of the level number is

$$-\sum_{\ell=0}^{L-1} 2^{\ell-L}\log_2(2^{\ell-L}) = \sum_{i=1}^{L} i\cdot 2^{-i} < 1$$

If not all levels in a partition are filled, then the entropy is even less, but for the sake of simplicity let's still use 1 bit to represent a level number within that partition. Note that since the top level is slightly larger (it contains ϵ extra blocks), the entropy might be slightly larger than 1 bit, but only by a negligible amount.

Using the compressed position map. These two compression tricks allow us to compress our position map immensely. On average, we can use 0.13 bytes for the partition number and 0.125 bytes (1 bit) for the level number, for a total of 0.255 bytes per block.

Once we have located a block's partition p and level ℓ, retrieving it is easy. When each level is constructed, each real block can be assigned a fresh alias $\mathsf{PRF}(K[p,\ell], \text{“real-alias”}, \mathsf{u})$ where u is the ID of the block and PRF is a pseudo-random function. Each dummy block can be assigned the alias $\mathsf{PRF}(K[p,\ell], \text{“dummy-alias”}, i)$ where i is the index of the dummy block in partition p, level ℓ. Then during retrieval, the client fetches blocks from the server by their alias.

B.2 Reducing Server Storage

Each partition's capacity is $\sqrt{N}+\epsilon$ blocks, where the surplus ϵ is due to the fact that some partitions may have more blocks than others when the blocks are assigned in a random fashion to the partitions. A partition has levels $\ell = 0,\ldots,\log_2\sqrt{N}$. Each level contains $2\cdot 2^\ell$ blocks (real and dummy blocks), except for the top level that contains 2ϵ additional blocks. Then, the maximum size of a partition on the server is $4\sqrt{N}+2\epsilon$ blocks. Therefore, the maximum server storage is $4N + 2\epsilon\sqrt{N}$.

However, the maximum amount of server storage required is less than that, due to several reasons:

1. Levels of partitions are sometimes not filled. It is extremely unlikely that at some point in time, all levels of all partitions are simultaneously filled.
2. As soon as a block is read from a level, it can be deleted by the server because its value is no longer needed.

In our simulation experiments, we calculated that the server never needs to store more than $3.2N$ blocks at any point in time. Hence, in practice, the server storage can be regarded as being less than $3.2N$ blocks.

B.3 Compressing Levels During Uploading

In Line 18 of the $\mathsf{WritePartition}$ algorithm in Figure 7, the client needs to write back up to $2\sqrt{N}+o(\sqrt{N})$ blocks to the server, at least half of which are dummy blocks. Since the values of the dummy blocks are irrelevant (since the server cannot differentiate between real and dummy blocks), it is possible to use a matrix compression algorithm to save a 2X factor in terms of bandwidth.

Suppose that the client wishes to transfer $2k$ blocks $b := (b_1, b_2, \ldots b_{2k})$. Let $S \subseteq \{1,2,\ldots,2k\}$ denote the offsets of the real blocks, let b_S denote the vector of real blocks. We consider the case when exactly k of the blocks are real, i.e., b_S is of length k (if less than k blocks are real, simply select some dummy blocks to fill in). To achieve the 2X compression, the server and client share a Vandermonde matrix $M_{2k\times k}$ during an initial setup phase. Now to transfer the blocks b, the client solves the linear equation:

$$M_S \cdot x = b_S$$

where M_S denotes the matrix formed by rows of M indexed by the set S, and b_S denote the vector B indexed by the set S, i.e., the list of real blocks.

The client can simply transfer x (length k) to the server in place of b (length $2k$). The server decompresses it by computing $y \leftarrow Mx$, and it is not hard to see that $y_S = b_S$. The server is unable to distinguish which blocks are real and which ones are dummy, since the Vandermonde matrix ensures that any subset of k values of y are a predetermined linear combination of the remaining k values of y.

Acknowledgments

We would like to thank Hubert Chan, Yinian Qi, and Alina Oprea for insightful feedback, helpful discussions, and proofreading.

This material is partially supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGE-0946797, the National Science Foundation under Grants No. CCF-0424422, 0311808, 0832943, 0448452, 0842694, 0627511, 0842695, 0808617, 0831501 CT-L, by the Air Force Office of Scientific Research under MURI Award No. FA9550-09-1-0539, by the Air Force Research Laboratory under grant No. P010071555, by the Office of Naval Research under MURI Grant No. N000140911081, by the MURI program under AFOSR Grant No. FA9550-08-1-0352, and by a grant from the Amazon Web Services in Education program. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies.

References

[1] Towards practical oblivious ram. Techinical Report, http://arxiv.org/abs/1106.3652, 2011.

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

[3] O. Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *STOC*, 1987.

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

[5] M. T. Goodrich and M. Mitzenmacher. Privacy-preserving access of outsourced data via oblivious ram simulation. *CoRR*, abs/1007.1259, 2010.

[6] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Oblivious ram simulation with efficient worst-case access overhead. In *CCSW*, 2011.

[7] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Privacy-preserving group data access via stateless oblivious ram simulation. In *SODA*, 2012.

[8] M. T. Goodrich, O. Ohrimenko, M. Mitzenmacher, and R. Tamassia. Privacy-preserving group data access via stateless oblivious ram simulation. http://arxiv.org/abs/1105.4125, 2011.

[9] E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in)security of hash-based oblivious ram and a new balancing scheme. In *SODA*, 2012.

[10] A. Oprea and M. K. Reiter. Integrity checking in cryptographic file systems with constant trusted storage. In *USENIX Security*, 2007.

[11] R. Ostrovsky. Efficient computation on oblivious rams. In *STOC*, 1990.

[12] R. Ostrovsky and V. Shoup. Private information storage (extended abstract). In *STOC*, pages 294–303, 1997.

[13] B. Pinkas and T. Reinman. Oblivious ram revisited. In *CRYPTO*, 2010.

[14] M. Raab and A. Steger. Balls into bins – a simple and tight analysis, 1998.

[15] P. Williams and R. Sion. Usable PIR. In *NDSS*, 2008.

[16] P. Williams, R. Sion, and B. Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *CCS*, 2008.