Circuit OPRAM: A (Somewhat) Tight Oblivious Parallel RAM

T-H. Hubert Chan* Elaine Shi†

Abstract

An Oblivious Parallel RAM (OPRAM) provides a general method to simulate any Parallel RAM (PRAM) program, such that the resulting memory access patterns leak nothing about secret inputs. OPRAM was originally proposed by Boyle et al. as the natural parallel counterpart of Oblivious RAM (ORAM), which was shown to have broad applications, e.g., in cloud outsourcing, secure processor design, and secure multi-party computation. Since parallelism is common in modern computing architectures such as multi-core processors or cluster computing, OPRAM is naturally a powerful and desirable building block as much as its sequential counterpart ORAM is.

Although prior work (in particular, Circuit ORAM) has shown how to construct ORAM schemes that are asymptotically tight under certain parameter ranges, the performance of known OPRAM schemes still do not match their sequential counterparts, leaving open two main questions: (i) whether one can construct an OPRAM scheme whose performance is competitive relative to known sequential counterparts; and (ii) whether one can construct OPRAM schemes that are asymptotically optimal.

Our paper answers the first question positively and gives partial answers to the second question. Specifically, we construct Circuit OPRAM, a new OPRAM scheme that makes the following contributions:

1. We achieve asymptotical improvement in terms of both total work and parallel runtime in comparison with known OPRAM schemes. More specifically, we improve the total work by a logarithmic factor and parallel runtime by poly-logarithmic factors.
2. We show that under sufficiently large block sizes, we can construct an OPRAM scheme that is asymptotically optimal both in total work and parallel runtime when the number of CPUs is not too small.
3. Our construction features a combination of novel techniques that enable us to (i) have asymptotically tight stochastic processes that do not suffer from worst-case and average-case discrepancies; and (ii) temporally reallocate work over offline and online stages of the algorithm such that we can achieve small parallel runtime overall. These techniques can be of independent interest in the design of of parallel oblivious algorithms in general.

*The University of Hong Kong. Email: hubert@cs.hku.hk
†Cornell University. Email: elaine@cs.cornell.edu

Contents

1 Introduction **1**
1.1 Our Results and Contributions . . . 1
1.2 Technical Highlights . . . 3
1.3 Organization . . . 4

2 Technical Roadmap **5**
2.1 Background: Circuit ORAM . . . 5
2.2 Simplifying Assumptions and Notations . . . 6
2.3 Pre-Warmup: The CLT OPRAM Scheme . . . 6
2.4 Warmup: Achieve $O(\log^2 N)$ Blowup in Total Work and Parallel Runtime . . . 8
2.5 Achieve Small Parallel Runtime: The Challenges . . . 11
2.6 From Flat Pool to Semi-Structured Group Stashes . . . 12
2.7 Routing Fetched Position Identifiers to the Next Recursion Level . . . 13
2.8 Putting it Altogether . . . 16
2.9 Extensions . . . 17
2.10 Related Work . . . 18

3 Preliminaries **19**
3.1 Parallel Random-Access Machines . . . 19
3.2 Oblivious Parallel Random-Access Machines . . . 20
3.3 Building Blocks . . . 21

4 Overview of OPRAM Construction **22**
4.1 Notations . . . 22
4.2 Data Structures . . . 23
4.3 Overview of One Simulated PRAM Step . . . 23

5 Details of the Maintain Phase **25**
5.1 Removing Fetched Blocks from Subtrees and Group Stashes . . . 26
5.2 Convert Group Stashes to Pool . . . 28
5.3 Evictions . . . 28
5.4 Cleanup: Reconstructing the Group Stashes . . . 30
5.5 Performance of the Maintain Phase . . . 32
5.6 Analysis of Stash Usage . . . 32

6 Details of the Fetch Phase **35**
6.1 Basic Building Blocks Related to Permutations . . . 35
6.2 Oblivious Compression (Dummy Removal) . . . 37
6.3 Offline Phase: Conflict Resolution and Preparation . . . 39
6.4 Online Phase: Routing Position Identifiers to the Next Recursion Level . . . 40
6.5 Performance of the Fetch Phase . . . 42

7 Performance and Security of Circuit OPRAM **42**
7.1 Performance . . . 42
7.2 Security . . . 43

8 Extensions and Optimizations **43**
8.1 Results for Large Block Sizes . . . 43
8.2 Additional Extensions . . . 45

A Optimizing Parallel Runtime for Path Eviction **48**

1 Introduction

Oblivious RAM (ORAM), initially proposed by Goldreich and Ostrovsky [16, 17], is a powerful primitive that allows oblivious accesses to sensitive data, such that access patterns during the computation reveal no secret information. Since its original proposal [17], ORAM has been shown to be promising in various application settings including secure processors [10, 12, 13, 24, 28], cloud outsourced storage [18, 31, 32, 38] and secure multi-party computation [14, 15, 19, 21, 23, 35].

Although ORAM is broadly useful, it is inherently sequential and does not support parallelism. On the other hand, parallelism is universal in modern architectures such as cloud platforms and multi-core processors. Motivated by this apparent discrepancy, in a recent seminal work [3], Boyle et al. extended the ORAM notion to the parallel setting. Specifically, they define Oblivious Parallel RAM (OPRAM), and demonstrated that any PRAM program can be simulated obliviously while incurring roughly $O(\log^4 N)$ blowup in both total work and parallel runtime, where N is the total memory size. The result by Boyle et al. [3] was later improved by Chen et al. [6], who showed a logarithmic factor improvement, attaining $O(\log^3 N)$ blowup in both total work and parallel runtime.

Before this work, we still know of no OPRAM algorithm whose performance can "match" the state-of-the-art sequential counterparts (e.g., Circuit ORAM [34], Path ORAM [33], or the KLO ORAM [22]). In particular, state-of-the-art ORAM schemes achieve $O(\log^2 N/\log\log N)$ [22] runtime blowup for general block sizes (i.e., assuming that each memory block can store its own address). For large enough block sizes, the best known scheme Circuit ORAM [34] achieves $O(\alpha \log N)$ runtime blowup[1] for any super-constant $\alpha = \omega(1)$. Since Goldreich and Ostrovsky proved a $\Omega(\log N)$ ORAM lower bound [16, 17], Circuit ORAM additionally demonstrates the existence of a (somewhat) tight ORAM scheme in the sequential setting [34]. Therefore, two natural questions arise.

- *Can we construct an OPRAM scheme whose asymptotical performance matches the best known sequential counterpart?*
- *Can we show a tight OPRAM scheme?*

1.1 Our Results and Contributions

We now describe our contributions and how to interpret our results in light of prior work and known lower bounds.

Asymptotically more efficient OPRAM. We construct a novel OPRAM scheme called Circuit OPRAM. Circuit OPRAM shows that any PRAM that runs in parallel time T and performs a total work of W can be simulated by an equivalent OPRAM that runs in parallel time $T \cdot O(\log N \log\log N)$, performs $W \cdot O(\alpha \log^2 N)$ total work, and requires that each CPU store only $O(1)$ blocks. The above result makes no assumption about the block size except that the block must be able to store its own memory address[2], and holds even if the PRAM uses a varying number of CPUs in different time steps. Since Circuit OPRAM is based on the tree-based ORAM framework [30], we achieve *statistical* security, i.e., security against computationally unbounded adversaries.

Circuit OPRAM achieves significant performance improvement relative to the prior state-of-the-art [6]. Specifically, in comparison with the prior best known OPRAM scheme [6], we achieve a

[1] In this paper, when we say that a scheme achieves $O(\alpha f(N))$ cost, we mean that for any $\alpha = \omega(1)$, there exists a scheme that achieves $O(\alpha f(N))$ cost.

[2] Note that all previous ORAM/OPRAM works, including those based on hierarchical ORAMs [16,17,22], implicitly assume that the block must be able to at least store its own memory address; or else all previous ORAM/OPRAM schemes will incur an extra multiplicative $\log N$ factor in cost.

Scheme	Total work blowup	Para. runtime blowup	Varying CPUs
Prior work			
BCP [3]	$O(\alpha \log^4 N)$	$O(\alpha \log^4 N)$	$\surd$
CLT [6][†]	$O(\alpha \log^3 N \log\log N)$	$O(\alpha \log^3 N \log\log N)$	$\times$
This paper			
Circuit OPRAM	$O(\alpha \log^2 N)$	$O(\log N \log\log N)$	$\surd$
	$O(\log^{2+\epsilon} N)$	$O(\log N)$	$\surd$

Table 1: Our contributions. Here we state asymptotic bounds assuming $O(1)$ blocks of CPU cache, general block sizes, and that the failure probability is at most $\frac{1}{N^{\Theta(\alpha)}}$, where N denotes the total number of memory blocks and $\alpha = \omega(1)$.

[†]: CLT [6] describes their result assuming $O(\log^2 N)$ CPU cache. For $O(1)$ CPU cache, their bounds would incur an extra $O(\log\log N)$ factor (since we will need oblivious sort to implement the eviction algorithm [35]). We state the bounds for $O(1)$ CPU cache here for a fair comparison.

multiplicative $O(\log N)$ improvement in terms of total work, and a $O(\log^2 N)$ factor improvement in parallel runtime. Further, Chen et al. [6] assumes that the PRAM must use a (somewhat) fixed number of CPUs each time step — otherwise, their scheme can incur an extra (super-) linear in m blowup where m is the maximum number of CPUs employed by the PRAM. By contrast, we can handle varying number of CPUs efficiently while preserving all asymptotic overheads.

Table 1 summarizes our contributions relative to prior works. Most of the table is self-explanatory. We remark that Chen et al. [6] considers CPU-to-memory communication and CPU-to-CPU communication as two separate metrics, but we consider the more general PRAM model where CPU-to-CPU communication is implemented by shared memory reads and writes. Therefore, we coalesce the two metrics into one (i.e., the maximum of the two metrics).

Theorem 1 (OPRAM for general block sizes). *For any $\alpha = \omega(1)$ and block size $B = \Omega(\log N)$, there exists an OPRAM scheme that has failure probability at most $\frac{1}{N^{\Theta(\alpha)}}$ and achieves $O(\alpha \log^2 N)$ total work blowup and $O(\log N \log\log N)$ parallel runtime blowup.*

Furthermore, our scheme can accomodate the case where each PRAM step involves a different number of CPUs, whereas the scheme by Chen et al. [6] can incur up to $\Omega(m)$ loss in total work when the PRAM has a varying number of CPUs across time.

Tightness under large block sizes. Table 1 shows only the results for general block sizes. We show that for sufficiently large block sizes $B \geq N^\epsilon$ and for an arbitrarily small constant $\epsilon > 0$, Circuit OPRAM achieves $O(\alpha \log N)$ total work blowup and $O(\log N)$ parallel runtime blowup. This suggests the asymptotical optimality of Circuit OPRAM under certain parameter ranges:

- *Optimality in total work.* Goldreich and Ostrovsky showed that any ORAM scheme must incur $\Omega(\log N)$ total work blowup — and their bound is directly applicable to the parallel setting. In this sense, Circuit OPRAM is asymptotically optimal in terms of total work for large enough blocks.
- *Optimality in parallel runtime.* Chan et al. [5] recently showed an $\Omega(\log m)$ lower bound for any OPRAM's parallel runtime blowup, where m is the number of CPUs and N is the total number of memory blocks. In light of this, Circuit OPRAM is asymptotically optimal for large enough blocks and when $m \geq N^\epsilon$ for any constant $\epsilon > 0$. With some (non-trivial) additional work, we construct another variant called Circuit OPRAM* that achieves $O(\log N \log\log N)$ total work blowup and $O(\log m + \log\log N)$ parallle runtime — this variant suffers from a slight $\log\log N$ loss in total work, but is optimal in parallel runtime for a broader range, that is, $m = \Omega(\log N)$.

We remark that achieving $O(\log N)$ parallel runtime is trivial if no requirement is imposed on the total work blowup — one can simply leverage $\Theta(N)$ CPUs, one for each memory location, to obliviously route data to the request CPUs. However, the total work blowup is $\omega(\frac{N}{m})$ in this case.

Theorem 2 (OPRAM for large block sizes)**.** *For any constant $\epsilon > 0$ and block size $B \geq N^\epsilon$, the following holds.*

- *For any $\alpha = \omega(1)$, there exists an OPRAM scheme that has failure probability at most $\frac{1}{N^{\Theta(\alpha)}}$ and achieves $O(\alpha \log N)$ total work blowup and $O(\log N)$ parallel runtime blowup.*
- *There exists an OPRAM scheme that has failure probability negligible in N and achieves $O(\log N \log\log N)$ total work blowup and $O(\log m + \log\log N)$ parallel runtime blowup.*

1.2 Technical Highlights

We introduce several novel techniques to achieve these asymptotic improvements.

Improving total work: avoid worst-case and average-case discrepancy. We observe that earlier OPRAM constructions often suffer from a worst-case and average-case discrepancy. We abuse the terminology slightly here: by *worst case* of a random variable, we mean an upper bound derived from some tail inequality beyond which the probability is negligible. When a secret random variable's worst-case value is asymptotically worse than its expectation, previous schemes had to pad the random variable to the worst-case value, since disclosing its true value would result in information leakage. For example, in Chen et al. [6], in expectation only $O(1)$ blocks are assigned to each subtree; but in the worst case there can be as many as $\Theta(\alpha \log N)$ assuming a negligible in N failure probability. To hide the number of blocks assigned to each subtree, Chen et al. [6] had to make $\Theta(\alpha \log N)$ evictions per subtree.

Our Circuit OPRAM also relies on the disjoint subtrees approach similar to Chen et al. [6]. We solve the aforementioned worst-case and average-case discrepancy by making only $O(1)$ evictions per subtree per access. This, however, introduces overflowing blocks. We have to handle the overflowing blocks in non-trivial manners to allow the read phase of the algorithm to be efficient (in terms of parallel runtime). Specifically, we introduce a new data structure called group stashes, where each group stash is shared by every $\Theta(\alpha \log N)$ consecutive subtrees. The design of the group stashes again features the core idea of avoiding worst-case and average-case discrepancies. Specifically, having every $\Theta(\alpha \log N)$ subtrees share a stash allows us to ensure that each group stash's worst-case occupancy is only $O(1)$ factor larger than its average occupancy; and therefore padding to the worst case would only introduce a constant factor penalty.

Achieving small parallel runtime: online/offline routing. Achieving small parallel runtime turns out to be rather non-trivial. In particular, since we adopt the standard approach of recursive tree-based ORAMs [30], the fetch phase must be conducted sequentially over $\Theta(\log N)$ recursion levels. In particular, metadata fetched must be routed to the $O(m)$ fetch CPUs at the next recursion level, before the next level can begin its fetch.

To achieve $O(\log N \log\log N)$ parallel runtime overall, this means that we can afford only $O(\log\log N)$ parallel runtime per (metadata) recursion level during the online fetch phase. This requirement immediately rules out approaches that involve global oblivious sorting (over many elements) during the online fetch phase — recall that oblivious sorting is a standard building block in designing (parallel) oblivious algorithms; but to sort m elements in parallel would immediately incur $\Theta(\log m)$ parallel runtime which is too expensive for our purpose, e.g., when $m = \sqrt{N}$.

One of our core ideas in achieving small parallel runtime is through temporal reallocation of work. During an offline phase, we perform preparation work to set up auxiliary data structures that will facilitate the online phase. Since the offline preparation work can be parallelized across all

recursion levels, we can afford to rely on oblivious-sorting-based techniques during the offline stage. In this way, we obliviously construct the "routing permutation" between any two adjacent levels during the offline phase, such that the online phase simply needs to apply this routing permutation such that the next recursion level can receive the metadata fetched at the previous level. Applying a pre-determined permutation to an array can be trivially parallelized and takes only a single parallel step. Interestingly, as we explain in Sections 2.7 and 6, for the offline phase to construct the level-to-level routing permutations, it basically has to "emulate" the online fetch phase but leaving the fetched metadata fields blank.

Achieving small parallel runtime: localized work. We also encounter another difficulty: as we explain later in Sections 2.7 and 6, the fetched metadata must be compressed obliviously first (by removing a subset of the dummy entries) before being routed to the next recursion level. Compression can be done naïvely through oblivious sorting. However, as explained earlier, during the online stage, we cannot afford to perform global oblivious sorting over all the $\Theta(m)$ entries of fetched metadata at a recursion level. Instead, we devise a new algorithm that localizes the work by performing compression over every $\Theta(\alpha \log N)$-sized group. Provided that the entries have been randomly permuted, the ratio of real and dummy elements within each group is close to the overall ratio over all elements. Since we now perform compression over much smaller groups, we can afford to perform oblivious sorting within each small group and incur only $O(\log \log N)$ parallel runtime.

Additional techniques. Our final scheme is rather sophisticated and involves numerous additional techniques and building blocks. For example, we devise a new algorithm for performing parallel removal of fetched blocks from the OPRAM's data structure. We devise a lazy adaptation technique for adjusting our OPRAM's number of subtrees if the number of PRAM CPUs varies over time. To achieve asymptotically optimal parallel runtime for large blocks, we devise non-trivial techniques for parallelizing Circuit ORAM's eviction algorithm.

1.3 Organization

Our final construction is rather sophisticated, and we use the following structure to break down the complexity in our presentation.

Section 2 contains an overview of our scheme described in the order of our thought process: we start by reviewing Chen et al. [6] scheme and observing why their scheme is inefficient. Since our final construction has many inter-related components, we first describe a much simpler (but nonetheless non-trivial) warmup scheme (Section 2.4) that improves the total work by a logarithmic factor in comparison with Chen et al. [6], but does not guarantee small parallel runtime. Using the warmup scheme as a stepping stone, we then introduce non-trivial techniques to asymptotically improve the parallel runtime of the online stage (Section 2.7), by performing appropriate preparation work in the offline stage. For clarity, Section 2 makes a few simplifying assumptions, e.g., the number of PRAM CPUs is fixed.

Then, Sections 4, 5, and 6 spell out the details of the full scheme and proofs, and generalize the description to allow the number of PRAM CPUs to vary. We suggest that the reader read Section 2 first to understand the high-level intuition before looking for the details of concrete building blocks and subroutines in Sections 4, 5, and 6.

Besides those presented earlier in this section, additional related work is provided in Section 2.10.

2 Technical Roadmap

2.1 Background: Circuit ORAM

We review tree-based ORAMs [7, 30, 33, 34] originally proposed by Shi et al. [30]. We specifically focus on describing the Circuit ORAM algorithm [34] which we build upon.

We assume that memory is divided into atomic units called blocks. We first focus on describing the *non-recursive version*, in which the CPU stores in its local cache a *position map* henceforth denoted as posmap that gives information on the position for every block.

Data structures. The memory is organized in the form of a binary tree, where every tree node is a *bucket* with a capacity of $O(1)$ blocks. Buckets hold blocks, where each block is either dummy or real. Throughout the paper, we use the notation N to denote the total number of blocks. Without loss of generality, we assume that $N = 2^L$ is a power of two. The ORAM binary tree thus has height L.

Besides the buckets, there is also a *stash* in memory that holds overflowing blocks. The stash is of size $O(\alpha \log N)$, where $\alpha = \omega(1)$ is a parameter related to the failure probability. Just like buckets, the stash may contain both real and dummy blocks. Henceforth, for convenience, we will often *treat the stash as part of the root bucket.*

Main path invariant. The main invariant of tree-based ORAMs is that every block is assigned to the path from the root to a randomly chosen leaf node. Hence, the path for each block is indicated by the *leaf identifier* or the *position identifier*, which is stored in the aforementioned position map posmap. A block with virtual address i must reside on the path indicated by posmap$[i]$.

Operations. We describe the procedures for reading or writing a block at virtual address i.

- *Read and remove.* To read a block at virtual address i, the CPU looks up its assigned path indicated by posmap$[i]$, and reads this entire path. If the requested block is found at some location on the path, the CPU writes a dummy block back into the location. Otherwise, the CPU simply writes the original block back. In both cases, the block written back is re-encrypted such that the adversary cannot observe which block is removed.

- *Remap.* Once a block at virtual address i is fetched, it is immediately assigned to a new path. To do this, a fresh random path identifier is chosen and posmap$[i]$ is modified accordingly. The block fetched is then written to the last location in the stash (the last location is guaranteed to be empty except with negligible probability at the end of each access). If this is a write operation, the block's contents may be updated prior to writing it back to the stash.

- *Evict.* Two paths (particularly, one to the left of the root and one to the right of the root) are chosen for eviction according to an appropriate data independent criterion. Specifically, we recommend using the reverse lexicographical order approach initially proposed by Gentry et al. [14] and later adopted in other schemes such as Circuit ORAM [34].

  For each path chosen (that includes the stash), the CPU performs an eviction procedure along this path. On a high level, eviction is a maintenance operation that seeks to move blocks along tree paths towards the leaves — and importantly, in a way that respects the aforementioned path invariant. The purpose of eviction is to avoid overflow at any bucket.

  Specifically in Circuit ORAM, this eviction operation involves making two metadata scans of the eviction path followed by a single data block scan.

A useful property of Circuit ORAM's eviction algorithm. For the majority of this paper

with the exception of Section A, the reader needs not know the details of the eviction algorithm. However, we point out a useful observation regarding Circuit ORAM's eviction algorithm.

Fact 1 (Circuit ORAM eviction)**.** *Suppose Circuit ORAM's eviction algorithm is run once on some path denoted* $\mathsf{path}[0..L]$*, where by convention we use* $\mathsf{path}[0]$ *to denote the root (together with the stash) and* $\mathsf{path}[L]$ *is the leaf in the path. Then, for every height* $i \in \{1, \ldots, L\}$*, it holds that at most one block moves from* $\mathsf{path}[0..i-1]$ *to* $\mathsf{path}[i..L]$*. Further, if a block did move from* $\mathsf{path}[0..i-1]$ *to* $\mathsf{path}[i..L]$*, then it must be the block that can be evicted the deepest along the eviction path (and if more than one such block exists, an arbitrary choice could be made).*

Recursion. So far, we have assumed that the CPU can store the entire position map posmap in its local cache. This assumption can be removed using a standard recursion technique [30]. Specifically, instead of storing the position map in the CPU's cache, we store it in a smaller ORAM in memory — and we repeat this process until the position map is of constant size.

As long as each block can store at least two position identifiers, each level of the recursion will reduce the size of the ORAM by a constant factor. Therefore, there are at most $O(\log N)$ levels of recursion. Several tree-based ORAM schemes also describe additional tricks in parametrizing the recursion for larger block sizes [33, 34]. We will not describe these tricks in detail here, but later in Section 8 we will recast these tricks in our OPRAM context and describe further optimizations for large block sizes.

Circuit ORAM performance. For general block sizes, Circuit ORAM achieves $O(\alpha \log N)$ blowup (in terms of bandwidth and the number of accesses) in the non-recursive version, and $O(\alpha \log^2 N)$ blowup across all levels of recursion. The CPU needs to hold only $O(1)$ blocks at any point in time.

2.2 Simplifying Assumptions and Notations

To aid understanding, let us first make a few simplifying assumptions — all these assumptions are not needed in our final scheme, we make them here only for ease of exposition.

- Every PRAM step contains m access requests, where m is a power of 2. For the time being, we consider that m is fixed. We will also assume that all these m access requests ask for *distinct* logical addresses — if not, as Boyle et al. pointed out [3], we can easily adopt a conflict resolution algorithm to suppress and replace duplicate requests with dummy ones, and after data has been fetched, oblivious multicast techniques can send fetched data to all request CPUs. Boyle et al. [3] showed that such conflict resolution and multicast can be implemented obliviously with total $O(m \log m)$ work and $O(\log m)$ parallel runtime.
- Initially, it will help the reader to think of the special case where $m = \sqrt{N}$ (or $m = N^\epsilon$ for any constant $0 < \epsilon < 1$). Note that in this case, $O(\log m)$ and $O(\log N)$ are the same. In some sense, only intermediate values of m are interesting. When m is as large as $\Theta(N)$, we can simply have N CPUs perform *oblivious routing* [3] to serve m simultaneous requests. On the other hand, when m is as small as $O(1)$, the OPRAM basically degenerates to the sequential case of ORAM.

  In our formal scheme description later, we will express our bounds for general m and N.

2.3 Pre-Warmup: The CLT OPRAM Scheme

We outline the approach of Chen et al. [6] which achieves $O(\log^3 N)$ blowup in both total work and parallel runtime. We describe a variant of their scheme [6] that uses Circuit ORAM instead of Path ORAM, but in a way that captures all the core ideas of Chen et al. [6].

Suppose we start with Circuit ORAM [34], the state-of-the-art tree-based ORAM. Circuit ORAM is sequential, i.e., supports only one access at a time — but we now would like to support m simultaneous accesses.

Challenge for parallel accesses: write conflicts. A strawman idea for constructing OPRAM is to have m CPUs perform m ORAM access operations simultaneously. Reads are easy to handle, since the m CPUs can read m paths simultaneously. The difficulty is due to write conflicts, which arise from the need for m CPUs to 1) each remove a block from its bucket if it is the requested one; and 2) to perform eviction after the reads. In particular, observe that the paths accessed by the m CPUs overlap, and therefore it may be possible that two or more CPUs will be writing the same location at the same time. It is obvious that if such write conflicts are resolved arbitrarily where an arbitrary CPU wins, we will not be able to maintain even correctness.

Subtree partitioning to reduce write contention. Chen et al.'s core idea is to remove buckets from smaller heights of the Circuit ORAM tree, and start at a height with m buckets. In this way, we can view the Circuit ORAM tree as m disjoint subtrees — write contentions can only occur inside each subtree but not across different subtrees.

Now since there are m CPUs in the original PRAM, each batch contains m memory access requests — without loss of generality, we will assume that all of these m requests are distinct — had it not been the case, it is easy to apply the conflict resolution algorithm of Boyle at al. [3] to achieve distinctness.

Each of these m requests will look for its block in a random subtree independently. By the Chernoff bound, each subtree receives $O(\alpha \log m)$ requests with all but negligible probability. Chen et al.'s algorithm proceeds as follows, where performance metrics are for the non-recursive version.

1. *Fetch.* A designated CPU per subtree performs the read phase of these $O(\alpha \log m)$ requests *sequentially*, which involves reading $O(\alpha \log m)$ paths in the tree. Since each path is $O(\log N)$ in length, this step incurs $O(\alpha m \log m \log N)$ total work and $O(\alpha \log m \log N)$ parallel runtime.
2. *Route.* Obliviously route the fetch results to the requesting CPUs. This step incurs $O(m \log m)$ total work and $O(\log m)$ depth.
3. *Remap.* Assign each fetched block to a random new subtree and a random leaf within that subtree. Similarly, each subtree receives $\mu = O(\alpha \log m)$ blocks with extremely high probability. Now, adopt an oblivious routing procedure to route exactly μ blocks back to each subtree, such that each tree receives blocks destined for itself together with padded dummy blocks. This step incurs incurs $O(\alpha m \log^2 m)$ total work and $O(\log m)$ parallel runtime.
4. *Evict.* Each subtree CPU *sequentially* performs $\mu = O(\alpha \log m)$ evictions for its own subtree. This step incurs $O(\alpha m \log m \log N)$ total work and $O(\alpha \log m \log N)$ parallel runtime.

Note that to make the above scheme work, Chen et al. [6] must assume that each subtree CPU additionally stores an $O(\alpha \log N)$-sized stash that holds all overflowing blocks that are destined for the particular subtree — later in our scheme, we will get rid of this CPU cache, such that each CPU only needs $O(1)$ blocks of transient storage and does not need any permanent storage.

Recursive version. The above performance metrics assumed that all CPUs get to store, read, and update a shared position map for free. To remove this assumption, we can employ the standard recursion technique of the tree-based ORAM framework [30] to store this position map. Assuming that each block has size at least $\Omega(\log N)$ bits, there can be up to $\log N$ levels of recursion.

Therefore, assuming $m = \sqrt{N}$, the total work across all levels of recursion is $O(\alpha m \log^3 N)$, and the parallel runtime is $O(\alpha \log^3 N)$ for Chen et al. [6].

2.4 Warmup: Achieve $O(\log^2 N)$ Blowup in Total Work and Parallel Runtime

As a warmup exercise, we describe a basic scheme that achieves $O(\log^2 N)$ blowup in total work and parallel runtime. Since this is not our final construction, we will not formalize this warmup scheme later.

We stress that even this warmup scheme already achieves asymptotic savings in comparison with the state-of-the-art OPRAM: as mentioned, the best existing scheme Chen et al. [6] achieves $O(\log^3 N)$ blowup in both total work and depth; and thus our warmup scheme saves a logarithmic factor in both metrics. To achieve this, we introduce a few new techniques on top of Chen et al. [6] — although the warmup scheme is not the most technically sophisticated part of our work, these new techniques are nonetheless non-trivial and interesting in their own right.

A pool and $2m$ subtrees: reduce write contention by partitioning. Following the approach of Chen et al. [6], we reduce write contention by partitioning the Circuit ORAM into $2m$ subtrees[3]. However, on top of Chen et al. [6], we additionally introduce the notion of a pool, a data structure that we will later utilize to amortize evictions across time[4].

We restructure a standard Circuit ORAM tree in the following manner. First, we consider a height with $2m$ buckets, which gives us $2m$ disjoint subtrees. All buckets from smaller heights, including the Circuit ORAM's stash, contain at most $O(m + \alpha \log N)$ blocks— we will simply store these $O(m + \alpha \log N)$ blocks in an unstructured fashion in memory, henceforth referred to as a *pool*.

Fetch. Whenever there are m CPUs requesting (distinct) blocks, we now need to look up the blocks in two places:

- *Subtree lookup:* Each of the m requests will go to a random subtree. Clearly each subtree gets $O(1)$ number of requests in expectation. Further, by a simple Chernoff bound, each subtree gets $O(\alpha \log N)$ requests with all but negligible probability in N. Note that this already serves to reduce write contention, since each subtree now gets $O(\alpha \log N)$ contention, as opposed to m, had had we used m disjoint subtrees. The best existing scheme by Chen et al. [6] stops here and does not further parallelize within each subtree but we will. It is easy to observe that reading up to $O(\alpha \log N)$ paths can easily be parallelized by assigning each path to a designated path-CPU. This way, in $O(\log N)$ steps, each path-CPU fetches the requested block (or dummy) from its own path — for the time being, we do not need to parallelize reads over a path, but we will later.

  All fetched blocks are merged into the central pool. Notice that at this moment, the pool size has grown by a constant factor, but later in a cleanup step, we will compress the pool back to its original size. Also, at this moment, we have not removed the requested blocks from the subtrees yet, and we will remove them later in the maintain phase.

- *Pool lookup:* At this moment, all requested blocks must be in the pool. We now rely on oblivious routing to route blocks back to each requesting CPU. More specifically, we assign $O(m)$ auxiliary CPUs, one to each block in the pool. Now, using blivious routing (described in Section 2.2), the m request CPUs can each get back the item requested, or output $\perp$ if not found. This results in $O(m \log m)$ total work and $O(\log m)$ parallel runtime.

Maintain. In the maintain phase, we must 1) remove all blocks fetched from the paths read; and 2) perform eviction on each subtree.

- *Efficient simultaneous removals.* After reading each subtree, we need to remove up to $\mu := O(\alpha \log N)$ blocks that are fetched. Such removal operations can lead to write contention when

[3] Although we choose $2m$ for concreteness, any $c \cdot m$ for a constant $c > 1$ would work.

[4] In our final scheme, this (flat) pool will be replaced with (semi-structured) group stashes — an ingredient (among several others) necessary to achieve small parallel runtime.

done in parallel: since the paths read by different CPUs overlap, up to $\mu := O(\alpha \log N)$ CPUs may try to write to the same location in the subtree.

Therefore, we propose a new oblivious parallel algorithm for efficient simultaneous removal. Our algorithm allows removal of the m fetched blocks across all trees in $O(m \log N)$ total work and $O(\log m)$ parallel runtime.

We defer the detailed description of this simultaneous removal algorithm to Section 5.1.

- *Lazy evictions.* Simultaneous evictions are even trickier to handle than simultaenous removals. Recall that after the reads and the simultaneous removals, each block fetched is now assigned to a new random subtree (and a random new leaf). Similarly due to a simple Chernoff bound, we can show that with all but negligible probability, each subtree receives at most $\mu := O(\alpha \log N)$ blocks to be evicted into that subtree. Notice that to retain obliviousness, the number of blocks getting reassigned to any subtree must be kept secret.

  A naive approach therefore is to perform $\mu := O(\alpha \log N)$ evictions per subtree per access. Unfortunately, this clearly incurs an extra $\log N$ factor in total work. In particular, observe that in expectation each subtree gets reassigned only $O(1)$ number of blocks from the current batch fetched, but we would be making every subtree suffer from an extra $\alpha \log N$ if we made μ evictions per subtree.

  Our solution is lazy eviction. Instead of making $\mu := O(\alpha \log N)$ evictions per access, we amortize evictions over time: with each access, we stipulate that each subtree gets to evict exactly *once*, i.e., on exactly one eviction path chosen according to some data independent criterion. Any block assigned to the subtree that cannot be evicted or simply overflows from the subtree is entered into the central pool. As we mention later, it is not hard to see that the pool size can be upper bounded by $O(m)$ with all but negligible probability.

- *Efficient pool-to-subtree routing.* At this moment, it helps to think of the pool as the union of all subtrees' stashes, where each subtree's stash is the union of overflowing blocks destined for the subtree. Suppose an eviction CPU is in charge of evicting on a particular path in subtree i. A naive approach would require the CPU to extract the stash for subtree i from the pool.

  Recall that with Circuit ORAM, each subtree's stash is $O(1)$ in expectation but more than $\Omega(\alpha \log N)$ in the rare case. To retain obliviousness, we cannot reveal how many blocks in the pool are destined for each subtree. Therefore, a naive approach is to rely on oblivious routing, such that we start out with the pool of $O(m)$ blocks, and at the end of the routing, every subtree's eviction CPU obtains $O(\alpha \log N)$ possibly dummy blocks. Unfortunately, this naive approach incurs an extra $O(\alpha \log N)$ blowup due to an average-case rare-case discrepancy — effectively we are padding to the worst-case for obliviousness, and we thus incur the rare-case cost rather than the average-case.

  Fortunately, this need not be the case. Here we leverage a special property of the Circuit ORAM algorithm: due to Fact 1, each eviction on a path will move at most one block — particularly the *deepest* block with respect to the eviction path — from the stash into the tree. Since we perform $O(1)$ evictions per subtree per access, it suffices to route $O(1)$ blocks from the pool to each subtree rather than routing the entire subtree's stash which can be as large as $O(\alpha \log N)$. We can achieve such routing in $O(m \log m)$ total work and $O(\log m)$ parallel runtime by oblivious sorting. The routing algorithm will need to use the Circuit ORAM's *deepest* criterion for selecting the blocks for each subtree.

- *Pool cleanup.* At the end of the maintain phase, we must compress the pool back to $c \cdot m$ blocks for some fixed constant c. Otherwise, the pool will keep growing as we place more fetched blocks in them. Such compression can easily be achieved through oblivious sorting.

To make this work, we need to prove a concentration bound that the pool size will be bounded by $O(m)$ with all but negligible probability — despite lazy eviction and possibly more blocks overflowing into the pool as a result. Since each of the $2m$ subtrees in expectation gets back only $\frac{1}{2}$ block with each batch of accesses, as long as we perform at least one eviction per subtree per access,it is not hard to show that at any time, the pool size is bounded by $O(m)$ except with negligible probability.

Later in our paper, we will also form semi-structured group stashes from this flat pool for reasons we shall explain. Therefore, rather than proving the above bound on the pool, our formal sections will instead bound the group stashes.

Notable differences from Chen et al. [6]. In summary, even our warmup scheme asymptotically improves the state-of-the-art OPRAM [6] through several non-trivial techniques. Specifically, the key new ideas are the following:

1. We avoid suffering from the average-case rare-case discrepancy by amortizing evictions across time. To enable this we need a couple techniques, including introducing an $O(m)$-sized pool to hold overflowing blocks; and leveraging a special property of the Circuit ORAM algorithm as mentioned above.

   As a result, we perform only $O(1)$ evictions per subtree per request, and we only need to oblivious route $O(1)$ blocks to each subtree for eviction. In comparison, Chen et al. [6] obliviously routes $O(\alpha \log m)$ blocks to each subtree and performs $O(\alpha \log m)$ evictions per subtree. At the moment, we can simply think that each eviction along a path is performed sequentially by a single CPU as described in [34]. However, in Appendix A, we shall see that each path eviction can be parallelized to reduce the asymptotic parallel runtime when the block size is large.

2. While Chen et al. [6] performs up to $\mu := O(\alpha \log m)$ read and remove operations within each subtree sequentially, we parallelize the operations within each subtree. To do this, we need to design a novel simultaneous removal algorithm to avoid write conflicts (see Section 5).

Performance. When we count all $O(\log N)$ recursion levels, the above warmup scheme achieves $O(\log^2 N)$ blowup in both total work and parallel runtime.

Bound on pool occupancy. To obtain a bound on the pool occupancy at the end of each batch of accesses, we need to prove the following. First, to analyze the stochastic process, we imagine that there is an imaginery big ORAM tree where levels $\log_2(m)$ or above correspond to our OPRAM's subtrees, and all smaller levels correspond to the pool. Our stochastic process is very similar to that of Circuit ORAM, but with the following difference. In Circuit ORAM, we always read and remove a block, and then perform exactly one eviction. Here, we perform m read and remove operations and then m evictions. We henceforth refer to the latter stochastic process Batched Circuit ORAM. Modulo the parallelism, the stochastic process of the warmup OPRAM scheme is identical to that of Batched Circuit ORAM. We then show that Circuit ORAM stochastically dominates Batched Circuit ORAM in terms of the number of blocks in the smaller $\log_2(m)$ levels of the tree (see Section 5.6). We can then bound the pool size of the warmup scheme by bounding the total number of blocks in the smaller $\log_2(m)$ levels of a Circuit ORAM tree. It is not hard to see that the pool occupancy does not exceed $O(m) + \alpha \log N$ except with negligible probability.

Remark 1. We remark that when the block size is large enough, i.e., $B = N^\epsilon$, the warmup scheme already achieves asymptotic optimality in terms of both total work and parallel runtime when $m = N^\epsilon$, where m denotes the number of PRAM CPUs and N denotes the total number of memory blocks. In the remainder, we consider a model where the OPRAM is allowed to consume more CPUs than the original PRAM. We show that in this case, the parallel runtime can be further reduced asymptotically.

2.5 Achieve Small Parallel Runtime: The Challenges

Previous works on OPRAM assume that the OPRAM must consume the same number of CPUs as the PRAM, and they aim to use a small number of OPRAM steps to simulate each PRAM step. In this case, the parallel runtime and total work are the same.

We consider a new OPRAM model where the OPRAM can consume more CPUs than the PRAM. We show that in this case, the parallel runtime blowup of the OPRAM can be further reduced. from $O(\log^2 N)$ to $O(\log N \log\log N)$ while preserving total work. Achieving this turns out to be highly non-trivial. We also remark that the $\Omega(\log m)$ parallel runtime lower bound by Chan et al. is for this more permissive model, i.e., even when the OPRAM can consume more CPUs than the PRAM, there is still an $\Omega(\log m)$ lower bound in parallel runtime. It turns out that reducing the parallel runtime incurs significant challenges.

To distinguish between different levels of recursion, we refer to the highest-level OPRAM holding the N original blocks as data-OPRAM. For each $1 \leq i < \log_2 N - 1$, we refer to the level-i ORPAM as $\mathsf{pos\text{-}OPRAM}_i$, which holds 2^i blocks.

In Section 2.4, we saw that each level of the recursion is a non-recursive OPRAM that has a fetch phase and a maintain phase. The fetch phase is inherently sequential across all recursion levels: one must look up the leaf identifier of the next level before proceeding to read the next recursion level. On the other hand, the maintain phase is trivially parallelizable across all recursion levels: after m blocks are fetched, the data-OPRAM and all pos-OPRAMs can perform the write-back process simultaneously. Therefore, over all recursion levels, the maintain phase's parallel runtime does not blow up due to recursion, but the fetch phase's parallel runtime must be multiplied by an $O(\log N)$ factor that is the depth of the recursion.

Goals. Henceforth we will focus on how to reduce the parallel runtime of the fetch phase of each non-recursive OPRAM. Our goals include the following:

1. For each pos-OPRAM, we would like to achieve $O(\log\log N)$ parallel runtime for the fetch phase.

2. However, we allow the (highest-level) data-OPRAM's fetch phase to have $O(\log\log N + \log m)$ parallel runtime — the data-OPRAM is special because requests must be routed to the request CPUs at the end, and this routing step will incur $O(\log m)$ parallel runtime.

3. We would like to achieve the reduction in parallel runtime while maintaining the same asymptotic total work.

Dissecting the challenges. We now dissect our warmup scheme described in Section 2.4, particularly focusing on what it takes to reduce the parallel runtime of the fetch phase. The fetch phase of the warmup scheme involves three major steps (all other operations such as simultaneous removal of fetched blocks, routing from pool to subtrees, and evictions are non-blocking and are considered part of the maintain phase):

- *Subtree lookup (easy):* First, the warmup scheme involves reading m tree paths. This step can be quite trivially parallelized achieving $O(\log\log N)$ parallel runtime and without blowing up total work. First, we we assign $\log N$ CPUs per path to read each physical location on all paths simultaneously. Then we run an instance of the parallel oblivious select algorithm (see Section 3.3) per path to aggregate each path's fetch result to a designated CPU.

- *Pool lookup (hard):* In the warmup scheme, the fetch phase also involves looking for the m requested blocks in the pool. This may be achieved with oblivious routing, which unfortunately takes $\Theta(\log m) = \Theta(\log N)$ parallel runtime (when $m = \sqrt{N}$ for instance). At this moment, it is not straightforward how to reduce the parallel runtime incurred in this step — in fact, as

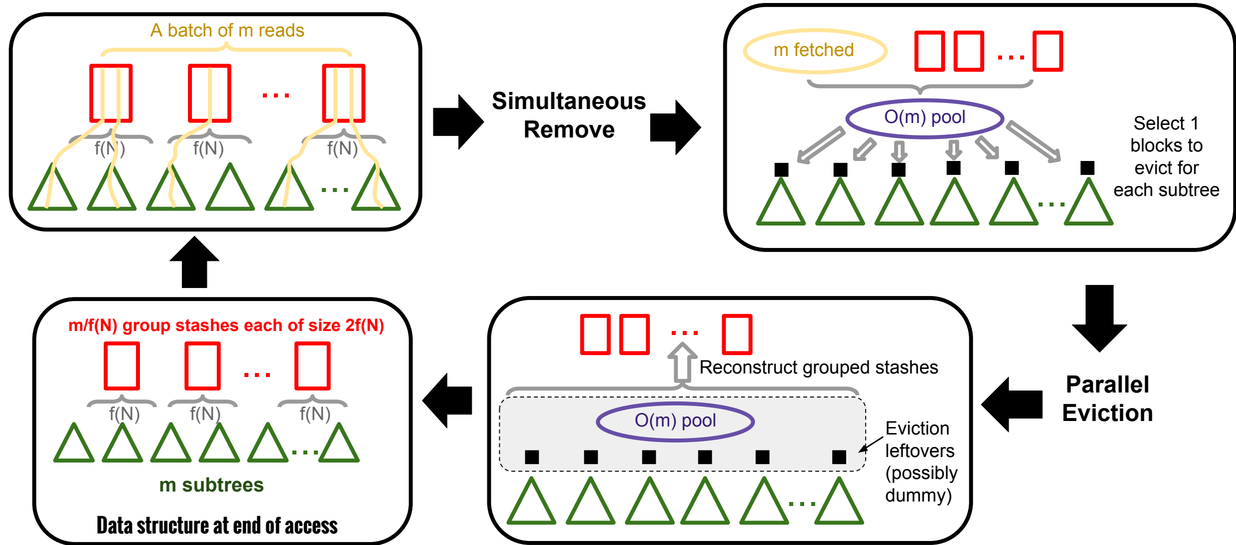

Figure 1: Circuit OPRAM data structure and operations. $f(N) := \alpha \log N$ for some $\alpha = \omega(1)$.

we describe later, new techniques are required, involving replacing the (flat) pool with (semi-structured) group stashes instead.

- *Route position identifiers fetched in one level to the next level's fetch CPUs (hard):* Another notable difficulty is how to route the fetched position identifiers in one recursion level to the next level's fetch CPUs, who need to know which tree paths to look for the blocks. For simplicity, let us imagine that in the d-level of recursion, a $\mathsf{pos\text{-}OPRAM}_d$ block with address $\mathsf{addr} \in \{0,1\}^d$ stores two position identifiers for the $(d+1)$-level addresses $(\mathsf{addr}||0)$ and $(\mathsf{addr}||1) \in \{0,1\}^{d+1}$. At the next level of recursion, there are 1 or 2 fetch CPUs waiting for the corresponding leaf identifiers. Routing fetched results to the next level's fetch CPUs must be done obliviously, e.g., without revealing whether both $(\mathsf{addr}||0)$ and $(\mathsf{addr}||1)$ are consumed by the next level of recursion, or only one of them is (and if so, which one).

  Again, a naive approach is to rely on oblivius routing, but again this would incur $\Theta(\log m)$ parallel runtime which is too expensive for our purpose.

In the next two subsections, we will describe how to address each of the above challenges identified. We will start by describing how to avoid using oblivious routing for the pool reads. Next, we will describe how to route fetched results to the next recursion level's fetch CPUs in small parallel runtime.

2.6 From Flat Pool to Semi-Structured Group Stashes

Two extremes. As mentioned earlier, reading the (structured) tree paths requires $O(\log \log N)$ parallel runtime whereas fetching from the flat pool takes $\Theta(\log m)$ parallel steps due to oblivious routing. Previously, we have described two approaches at two ends of the spectrum:

1. The OPRAM scheme by Chen et al. [6] routes *all* fetched blocks back to each subtree during the offline phase. Thus, for every access, their scheme requires routing $O(\alpha \log N)$ blocks to each subtree and performing $O(\alpha \log N)$ evictions per subtree. Each subtree has a stash of $O(\alpha \log N)$

in size. Therefore, this approach is inefficient during the maintain phase but the fetch phase can be done in $O(\log \log N)$ parallel time for each recursion level.

2. By contrast, our warmup scheme (Section 2.4) routes only 1 block back to each subtree and performs 1 eviction per subtree for each access. However, our warmup scheme requires an $O(m)$ flat pool to store overflowing blocks. In comparison, our warmup scheme is more efficient than Chen et al. [6] in the maintain phase by a logarithmic factor, but unfortunately the introduction of the flat pool causes reads in the fetch phase to take $\Theta(\log m)$ parallel time.

Our approach: seek middle ground. Our approach seeks a middle ground between these two extremes. At the end of the maintain phase, we convert the pool to $\frac{2m}{\alpha \log N}$ group stashes where each group stash is the shared stash of $\alpha \log N$ subtrees. Each group stash obtains $\alpha \log N$ blocks in expectation, and $2\alpha \log N$ blocks with all but $\mathsf{negl}(N)$ probability, because $\alpha = \omega(1)$. Notably, except with negligible probability , the group stash's occupancy is only a constant factor larger than is mean. Therefore, constructing the group stashes from the flat pool could be achieved by obliviously routing $2\alpha \log N$ possibly dummy blocks to each group stash — and this incurs $O(m \log m)$ total work and $O(\log m)$ parallel time. The advantage to incur $O(\log m)$ parallel time in the maintain phase (as opposed to the fetch phase) is that the maintain phases across different levels of recursion can be performed in parallel, whereas the fetch phases of different levels have to be performed sequentially.

Observe that in the fetch phase of the next step, the group stashes are ready to be used. At the beginning of the maintain phase, we just take the union of the group stashes and the fetched blocks and treat it as a flat pool again.

Another way to view our scheme is the following: at the end of the maintain phase, we perform some preparation work to restore some structure on the $O(m)$ flat pool containing overflowing blocks. With such preparation, during the batch of reads in the next fetch phase, each subtree can view the corresponding group stash as part of its root bucket — we can therefore leverage the aforementioned oblivious aggregation algorithm to perform reads in $O(\log \log N)$ parallel steps.

At this moment, the most significant missing piece is how one can efficiently route the fetched position identifiers to the next recursion level, which we shall explain in the next subsection.

2.7 Routing Fetched Position Identifiers to the Next Recursion Level

Recall that the problem is the following: m fetch CPUs at recursion level d have fetched position identifiers to be sent to recursion level $d + 1$. We would like to route these position identifiers to the corresponding fetch CPUs at level $d + 1$. A naive approach is to rely on oblivious routing — but this would incur $\Theta(\log m)$ parallel steps. Improving this in small parallel runtime turns out to be rather challenging.

Notations. To understand our solution, it helps to introduce a few notations. We will henceforth assume a fan-out of 2, i.e., each block holds exactly two position identifiers.

Let $D := \log_2 N$ denote the total number of recursion levels. Let $\mathsf{addr}^{\langle D \rangle} \in \{0, 1\}^D$ denote a virtual address. We will use the notation $\mathsf{addr}^{\langle d \rangle}$ to denote a truncated d-bit address at recursion level d. If we access a block with virtual address $\mathsf{addr}^{\langle d \rangle}$ in recursion level d, we can obtain the position identifiers of the next-level addresses $(\mathsf{addr}||0)^{\langle d \rangle}$ and $(\mathsf{addr}||1)^{\langle d \rangle}$.

Suppose that at the beginning of each PRAM step, a preparation procedure is performed in parallel across all recursion levels. This preparation procedure examines the batch of m virtual addresses requested (again, without loss of generality, we assume that conflict resolution has already been done and that the m virtual addresses are distinct). Then, at each recursion level d, the preparation procedure writes down the following *instruction array* (at this moment we simply claim

that such preparation can be done efficiently and we defer the detailed algorithm to Section 6.3):

$$\underline{\text{Incomplete instruction array:}}\ \ \mathsf{Instr}^{\langle d\rangle} := \left\{(\mathsf{addr}^{\langle d\rangle}, \mathsf{flag}, _) \mid \mathsf{dummy}\right\}_{2m}$$

Specifically, each entry of the instruction array is either a dummy element (denoted dummy) or is a real instruction of the form $(\mathsf{addr}^{\langle d\rangle}, \mathsf{flag}, _)$ where $\mathsf{addr}^{\langle d\rangle}$ is a virtual address at level d, and flag is a 2-bit indicator that indicates whether each child of the address $\mathsf{addr}^{\langle d\rangle}$ (i.e., the level-$(d+1)$ addresses $(\mathsf{addr}||0)^{\langle d+1\rangle}$ and $(\mathsf{addr}||1)^{\langle d+1\rangle}$) is requested at the next level. At this moment, each instruction is *incomplete*, and the wildcard $_$ is an empty space left to later receive the position identifier for $\mathsf{addr}^{\langle d\rangle}$ — only when the position identifier is received, can a fetch CPU actually execute the instruction. Finally, the subscript $2m$ denotes the length of the array. Even though there are at most m requests per recursion level, for reasons explained later, we need to pad the instruction array with dummy elements to size $2m$. To ensure obliviousness, the entries of the array are randomly permuted obliviously.

Now, each of the $2m$ fetch CPU at recursion level d reads a block that contains two next-level position identifiers. We call the result of the read a *fetched array* whose size is $4m$ (note the factor-2 blowup in size)[5]:

$$\underline{\text{Fetched position identifiers:}}\ \ \mathsf{Fetched}^{\langle d\rangle} := \left\{(\mathsf{addr}^{\langle d+1\rangle}, \mathsf{pos}) \mid \mathsf{dummy}\right\}_{4m},$$

where pos denotes the current position identifier for the virtual address $\mathsf{addr}^{\langle d+1\rangle}$ at recursion level $d+1$. Since a complete instruction contains a flag that indicates whether each of the next-level child address is needed, we assume that the $\mathsf{Fetched}^{\langle d\rangle}$ array has suppressed all level-$(d+1)$ addresses not requested and replaced them with dummy.

Problem statement. We now have to solve the following challenge: given the following arrays

$$\mathsf{Fetched}^{\langle d\rangle} := \left\{(\mathsf{addr}^{\langle d+1\rangle}, \mathsf{pos}) \mid \mathsf{dummy}\right\}_{4m} \quad \text{and} \quad \mathsf{Instr}^{\langle d+1\rangle} := \left\{(\mathsf{addr}^{\langle d+1\rangle}, \mathsf{flag}, _) \mid \mathsf{dummy}\right\}_{2m}$$

perform routing to "complete" the instructions at recursion level $d+1$ where each incomplete instruction at recursion level $d+1$ receives the correct position identifier, thus forming a *completed* instruction array denoted $\mathsf{CInstr}^{\langle d+1\rangle}$:

$$\underline{\text{Completed instruction array:}}\ \ \mathsf{CInstr}^{\langle d+1\rangle} := \left\{(\mathsf{addr}^{\langle d+1\rangle}, \mathsf{flag},\ \mathsf{pos}) \mid \mathsf{dummy}\right\}_{2m}$$

As mentioned earlier, we would like to achieve this in a small number of (online) parallel steps.

Overview of our approach. Our approach involves two steps:

1. *Localized compression.* Compress the $4m$-sized $\mathsf{Fetched}^{\langle d\rangle}$ array back to size $2m$, suppressing dummy elements. We will henceforth refer to this compressed array as $\mathsf{Send}^{\langle d\rangle}$ (whose type declaration is below):

   $$\underline{\text{Compressed fetched position identifiers:}}\ \ \mathsf{Send}^{\langle d\rangle} := \left\{(\mathsf{addr}^{\langle d+1\rangle}, \mathsf{pos}) \mid \mathsf{dummy}\right\}_{2m}$$

   The challenge here is that we cannot resort to a global oblivious sort for compression. Instead, we propose an algorithm that localizes the compression work to each smaller group of elements.

[5] In the real construction detailed in Section 6, each entry of the fetched array actually carries the current position identifier pos, as well as the new (random) position identifier pos' of the corresponding block. We omit this here for clarity of exposition.

2. *Use an offline/online paradigm for routing.* Observe that at each recursion level, a read cannot be executed until the required position identifiers are received from the previous level.

   Instead of performing all the work associated with one level before starting the next level, we observe that in each level, part of the work can be performed without the knowledge of the position identifiers from the previous level. We refer to this part of the work as the *offline* phase, which can be performed in parallel across all levels. We shall see that in the offline phase, for each level d, we pre-compute a permutation that will align the arrays $\mathsf{Send}^{\langle d \rangle}$ and $\mathsf{Instr}^{\langle d+1 \rangle}$ with respect to the level-$(d+1)$ addresses.

   In contrast, the *online* phase refers to work that can be performed only when the position identifiers are known, and hence is performed sequentially level by level. After the reads are performed in level d to form the $\mathsf{Send}^{\langle d \rangle}$ array, we can make use of the pre-computed permutation in the offline phase to "send" the required position identifiers to the correct entries in $\mathsf{Instr}^{\langle d+1 \rangle}$ using only $O(1)$ parallel runtime.

To accomplish either of the above would require non-trivial techniques as we explain below.

Localized oblivious compression. Suppose we have an array of $4m$ elements of which at most m are real. How can we obliviously compress the array down to $2m$ elements such that all real elements are preserved (i.e., $2m$ dummy elements are removed)? A naive idea is to rely on oblivious sorting over these $4m$ elements, but this would incur $\Theta(\log m)$ parallel runtime and would be too expensive.

Our key idea is to "localize" the compression. Suppose that these $4m$ elements have been randomly permuted. Then, every consecutive $2\alpha \log N$ group of elements can only have $\alpha \log N$ real elements with all but negligible probability[6], where $\alpha = \omega(1)$ is any super-constant function. Since this is true, we can perform *localized* oblivious compression, by performing oblivious sorting (where real elements have higher priority) on each non-overlapping group of $2\alpha \log N$ consecutive elements and discarding half of the elements in each group with the lowest priorities. This incurs only $O(\log \log N)$ parallel runtime instead of $O(\log N)$.

Offline/online routing. After the oblivious compression phase, we obtain a send array of the form $\mathsf{Send}^{\langle d \rangle} := \left\{(\mathsf{addr}^{\langle d+1 \rangle}, \mathsf{pos}) \mid \mathsf{dummy}\right\}_{2m}$. This send array contains the fetched position identifiers of the next recursion level. The next level's instruction array $\mathsf{Instr}^{\langle d \rangle} := \left\{(\mathsf{addr}^{\langle d \rangle}, \mathsf{flag}, _) \mid \mathsf{dummy}\right\}_{2m}$ must now receive these position identifiers such that the instructions can be completed. Again, we would like to achive this without performing oblivious routing in the online phase.

Our idea is the following: what if, in the offline phase, it is possible to figure out the permutation to align the arrays $\mathsf{Send}^{\langle d \rangle}$ and $\mathsf{Instr}^{\langle d+1 \rangle}$ with respect the level-$(d+1)$ addresses? In this way, the online phase can simply apply the permutation in a single parallel step to align the two arrays, after which "receiving" the position identifiers becomes a simple coordinate-wise copy. Below we describe how to carry out this idea in a way that guarantees obliviousness. The offline phase performs the following:

1. *Oblivious random permutation of incomplete instruction arrays.* First, in the offline preparation phase, after generating the incomplete instruction array $\mathsf{Instr}^{\langle d+1 \rangle}$ at each recursion level d, we permute this array at random obliviously, i.e., without revealing the random permutation.

2. *Offline emulation of the online phase.* Next, the offline preparation phase will emulate the online execution but without filling in the actually fetched contents. Specifically, it will emulate the

[6] Recall that we need to pad each recursion level to $2m$ instructions. The reason is the following: for the oblivious compression to cut half of the elements, the fraction of real elements in the array must be a constant smaller than $\frac{1}{2}$. Here we set this fraction to be $\frac{1}{4}$.

$\overline{\mathsf{Fetched}}^{\langle d \rangle}$ array (while leaving the fetched position identifiers blank). Then, it will emulate the oblivious compression which results in the send array $\overline{\mathsf{Send}}^{\langle d \rangle}$. We use the notational convention $\overline{\mathsf{arr}}$ to denote the offline, emulated version of arr — specifically, $\overline{\mathsf{arr}}$ is the same as arr except that it leaves the fetched position identifiers and contents blank.

3. *Construction of the routing permutation.* Finally, the offline phase will figure out the permutation to apply to $\overline{\mathsf{Send}}^{\langle d \rangle}$ such that it is aligned with $\mathsf{Instr}^{\langle d+1 \rangle}$. This step can be achieved through a sequence of oblivious sorts as we describe in detail in Section 6.3.

It is not hard to see that the above offline operations can be parallelized across multiple recursion levels. As a result, the offline phase can be completed in $O(\log m)$ parallel runtime, and $O(\alpha m \log m \log N)$ total work — we defer a more detailed explanation to Section 6.3.

Given the above offline preparation, we quickly recap the online phase for clarity:

1. Let d denote the current recursion level. Assign $2m$ fetch CPUs, each of which read one completed instruction in $\mathsf{CInstr}^{\langle d \rangle}$. The completed instruction specifies which path to read and what address to look for. Each fetch CPU recruits $\log N$ auxiliary CPUs to read its path and aggregate the result, which contains 2 position identifiers for the next level.

   The result of this step is the fetched array $\mathsf{Fetch}^{\langle d \rangle}$ of size $4m$.

2. Apply the oblivious compression algorithm to compress $\mathsf{Fetch}^{\langle d \rangle}$ down to size-$2m$ array $\mathsf{Send}^{\langle d \rangle}$.

3. In a single parallel step, apply the routing permutation that the offline phase has pre-computed to align the arrays $\mathsf{Send}^{\langle d \rangle}$ and $\mathsf{Instr}^{\langle d+1 \rangle}$ in order to construct the completed instruction array $\mathsf{CInstr}^{\langle d+1 \rangle}$ in the next level.

It is not hard to see that the online phase incurs only $O(\log \log N)$ parallel steps, since both the parallel path read and the localized oblivious compression requires only $O(\log \log N)$ parallel steps.

Obliviousness. It is important to note why the above approach preserves obliviousness. The key observation is that each $\mathsf{Instr}^{\langle d+1 \rangle}$ array is independently and randomly permuted in the offline phase (without revealing the permutation). Therefore, every permutation from $\mathsf{Send}^{\langle d \rangle}$ to $\mathsf{Instr}^{\langle d+1 \rangle}$ that is revealed to the adversary is an independent, random permutation. Therefore, the adversary could have simulated these permutations itself without knowing the data contents.

2.8 Putting it Altogether

It is now a good time to give an overview of the resulting scheme (see Figure 1). Below we describe what happens in a pos-OPRAM level. For the (highest-level) data-OPRAM, the operations are almost the same, except that instead of routing position identifiers to the next level's fetch CPUs, the data-OPRAM would instead route fetched blocks to the requesting CPUs. For each of the following steps, we indicate whether it is performed level-by-level sequentially or in parallel across different recursion levels of pos-OPRAM.

- *All levels in parallel: offline preparation.* Given a batch of memory requests (assume distinct addresses), perform offline preparation at all recursion levels. As mentioned in Section 2.7, the offline preparation is necessary to allow the online fetch phase to have small parallel runtime.

- *Level-by-level: fetch.* Given $2m$ fetch instructions at a recursion level among which at most m are real[7], in parallel perform $2m$ reads, each of which involves reading blocks from 1) a path in the corresponding subtree; and 2) the subtree's corresponding group stash. The parallel reads can be accomplished in $O(\log \log N)$ time for an appropriately small $\alpha = \omega(1)$.

- *Level-by-level: route position identifiers to next level (Section 2.7).* Each of the $2m$ fetches returns a block. Without loss of generality, suppose that each block contains 2 position identifiers (for the next level). Among these resulting $4m$ position identifiers, at most m of them are needed by the next recursion level (recall that the next recursion level will also have $2m$ fetch instructions, of which at most m are real). Now, we leverage a special procedure to route position identifiers to the next level's fetch CPUs in $O(1)$ parallel steps — see Section 2.7.

- *All levels in parallel: simultaneous removal.* Perform a parallel remove procedure to remove all fetched blocks from their respective paths.

- *All levels in parallel: eviction.* Now, the fetched m blocks and the original group stashes are merged into a flat pool. From this pool, obliviously route exactly one (possibly dummy) block to each subtree for eviction, where the block is selected using Circuit ORAM's "deepest" criterion, where deepest is defined with respect to the current eviction path. Now, perform exactly one eviction per subtree on a path chosen according to the Circuit ORAM algorithm.

- *All levels in parallel: reconstruct group stashes and compress.* After eviction, each subtree has one (possibly dummy) block leftover. All eviction leftovers are merged into the pool. Now, we leverage oblivious sorting to reconstruct group stashes from the pool. Although not made explicit, we in fact also perform a compression operation along with the reconstruction of the group stashes: notice that the pool has grown due to the addition of the m fetched blocks. However, the subtree evictions should in expectation have evicted $O(m)$ blocks back into the subtrees. Therefore, the total occupancy of the pool (or the group stashes) will not keep increasing.

2.9 Extensions

Varying number of CPUs. Our overview earlier assumes that the original PRAM always has the same number of CPUs in every time step, i.e., all batches of memory requests have the same size. We can further extend our scheme for the case when the number of PRAM CPUs varies over time. Below we briefly describe the idea while leaving details to Sections 5 and 6. Without loss of generality, henceforth we assume that in every time step, the number of requests in a batch m is always a power of 2 — if not, we can simply round it up to the nearest power of 2 incurring only $O(1)$ penalty in performance.

Suppose that the OPRAM scheme currently maintains $\widehat{m}$ subtrees, but the incoming batch has $m > \frac{\widehat{m}}{2}$ number of requests. In this case, we will immediately adjust the number of subtrees to $\widehat{m} := 2m$. This can be done simply by merging more heights of the tree into the flat pool (and at the end of the maintain phase, the flat pool will be converted to the group stashes).

The more difficult case is when the incoming batch contains less than $m < \frac{\widehat{m}}{2}$ requests. In this case, we need to decrease the number of subtrees $\widehat{m}$. However, instead of decreasing $\widehat{m}$ immediately to $2m$, we use a *lazy* strategy: every time we halve the value of $\widehat{m}$ — and to achieve this we only need to reconstruct one extra height of the big ORAM tree, which can be achieved through oblivious sorting in $O(\widehat{m} \log \widehat{m})$ total work and $O(\log \widehat{m})$ parallel time. In comparison, had we immediately

[7] Although at most m fetch instructions are real, we pad the instruction array to $2m$ with dummy instructions — the need arises due to the algorithms in Section 2.7 for routing fetched position identifiers to the next recursion level.

changed $\widehat{m}$ to $2m$, it would have been too expensive to reconstruct up to $\log N$ heights of the ORAM tree (e.g., when m suddenly drops to 1).

Results for large block sizes. When the block size is sufficiently large, we show the following results:

- Circuit OPRAM achieves $O(\alpha \log N)$ total work blowup, and $O(\log N)$ parallel runtime. In light of Goldreich and Ostrovsky's $\Omega(\log N)$ lower bound [16, 17], Circuit OPRAM is therefore asymptotically optimal in terms of total work
- We introduce a variant of the scheme called Circuit OPRAM* that achieves $O(\log N \log\log N)$ total work blowup, and $O(\log m + \log\log N)$ parallel runtime. In light of the $\Omega(\log m)$ parallel runtime lower bound by Chan et al. [5], Circuit OPRAM* is asymptotically optimal in terms of parallel runtime when $m = \Omega(\log N)$. We stress that to construct this variant requires non-trivial techniques for constructing a parallelized counterpart of Circuit ORAM's eviction algorithm [34].

We defer the technical details and additional extensions and parametrization tricks to Section 8.

2.10 Related Work

In the introduction, we compared with some of the closely related works earlier. In this section, we describe additional related work.

Oblivious RAM (ORAM). Oblivious RAM (ORAM) was initially proposed by Goldreich and Ostrovsky [16, 17] who showed that any RAM program can be simulated obliviously incurring only $O(\alpha \log^3 N)$ runtime blowup, while achieving a security failure probability that is negligible in N. Numerous subsequent works [9, 18, 22, 26, 27, 30, 33, 34, 34–38] improved Goldreich and Ostrovsky's seminal result in different application settings including cloud outsourcing, secure processor, and secure multi-party computation.

Most of these schemes follow one of two frameworks: the hierarchical framework, originally proposed by Goldreich and Ostrovsky [16, 17], or the tree-based framework proposed by Shi et al. [30]. To date, some of the (asymptotically) best schemes include the following: 1) Kushilevitz et al. [22] showed a *computationally secure* ORAM scheme with $O(\log^2 N/\log\log N)$ runtime blowup for general block sizes; and 2) Wang et al. construct Circuit ORAM [34], a *statistically secure* ORAM that achieves $O(\alpha \log^2 N)$ runtime blowup for general block sizes and $O(\alpha \log N)$ runtime blowup for large enough blocks. Further, using a standard trick initially proposed by Fletcher et al. [11] and later improved in the Circuit ORAM paper [34], we can modify Circuit ORAM into a computationally secure variant by relying on a PRF, and achieve $O(\alpha \log^2 N/\log\log N)$ runtime blowup where α can be any super-constant function — this makes Circuit ORAM (almost) competitive to Kushilevitz et al. [22] even for small block sizes. Unfortunately there does not seem to be a straightforward way to adapt this trick to the parallel setting and make it compatible with Circuit OPRAM's level-by-level position map routing algorithm. Therefore we leave it as an open question whether there is an OPRAM scheme that achieves $O(\log^2 N/\log\log N)$ total work blowup and parallel runtime blowup competitive to Circuit OPRAM.

On the lower bound side, Goldreich and Ostrovsky [16, 17] demonstrated that any ORAM scheme (with constant CPU cache) must incur at least $\Omega(\log N)$ runtime blowup. This well-known lower bound was recently shown to be tight (under certain parameter ranges) by the authors of Circuit ORAM [34], who showed a matching upper bound for sufficiently large block sizes. We note that while the Goldreich and Ostrovsky lower bound is quite general, it models each block as being opaque — recently, an elegant result by Boyle and Naor [4] discussed the possibility of proving a lower bound without this restriction. Specifically, they showed that proving a lower bound without the block opaqueness restriction is as hard as showing a superlinear lower bound on the sizes of

certain sorting circuits. Further, the Goldreich-Ostrovsky lower bound is also known not to hold when the memory (i.e., ORAM server) is capable of performing computation [2, 9].

Oblivious Parallel RAM (OPRAM). Given that many modern computing architectures support parallelism, it is natural to extend ORAM to the parallel setting. As mentioned earlier, Boyle et al. [3] were the first to formulate the OPRAM problem, and they constructed a scheme that achieves $O(\alpha \log^4 N)$ blowup both in terms of total work and parallel runtime. Their result was later improved by Chen et al. [6] who were able to achieve $O(\alpha \log^3 N)$ blowup both in terms of total work and parallel runtime under $O(\log^2 N)$ blocks of CPU cache. There results can easily be recast to the $O(1)$ CPU cache setting by applying a standard trick that leverages oblivious sorting to perform eviction [34, 35]. We note that Chen et al. [6] actually considered CPU-memory communication and inter-CPU communication as two separate metrics, and their scheme achieves $O(\alpha \log^2 N \log \log N)$ CPU-memory communication blowup, but $O(\alpha \log^3 N)$ inter-CPU communication blowup. In this paper, we consider the more general PRAM model where all inter-CPU communication is implemented through CPU-memory communication. In this case, the two metrics coalesce into one (i.e., the maximum of the two).

Besides OPRAM schemes in the standard setting, Dachman-Soled et al. [8] considered a variation of the problem (which they refer to as "Oblivious Network RAM") where each memory bank is assumed to be oblivious within itself, and the adversary can only observe which bank a request goes to. Additionally, Nayak et al. [25] show that for parallel computing models that are more restrictive than the generic PRAM (e.g., the popular GraphLab and MapReduce models), there exist efficient parallel oblivious algorithms that asymptotically outperform known generic OPRAM. Some of the algorithmic techniques employed by Nayak et al. [25] are similar in nature to those of Boyle et al. [3].

3 Preliminaries

3.1 Parallel Random-Access Machines

A *parallel random-access machine* (PRAM) consists of a set of CPUs and a shared memory denoted mem indexed by the address space $[N] := \{1, 2, \ldots, N\}$. In this paper, we refer to each memory word also as a *block*, and we use B to denote the bit-length of each block.

We support a more general PRAM model where the number of CPUs in each time step may vary. Specifically, in each step $t \in [T]$, we use m_t to denote the number of CPUs. In each step, each CPU executes a next instruction circuit denoted Π, updates its CPU state; and further, CPUs interact with memory through request instructions $\vec{I}^{(t)} := (I_i^{(t)} : i \in [m_t])$. Specifically, at time step t, CPU i's instruction is of the form $I_i^{(t)} := (\mathsf{op}, \mathsf{addr}, \mathsf{data})$, where the operation is $\mathsf{op} \in \{\mathsf{read}, \mathsf{write}\}$ performed on the virtual memory block with address addr and block value $\mathsf{data} \in \{0,1\}^B \cup \{\bot\}$. If $\mathsf{op} = \mathsf{read}$, then we have $\mathsf{data} = \bot$ and the CPU issuing the instruction should receive the content of block $\mathsf{mem}[\mathsf{addr}]$ at the initial state of step t. If $\mathsf{op} = \mathsf{write}$, then we have $\mathsf{data} \neq \bot$; in this case, the CPU still receives the initial state of $\mathsf{mem}[\mathsf{addr}]$ in this step, and at the end of step t, the content of virtual memory $\mathsf{mem}[\mathsf{addr}]$ should be updated to data.

Write conflict resolution. By definition, multiple read operations can be executed concurrently with other operations even if they visit the same address. However, if multiple concurrent write operations visit the same address, a conflict resolution rule will be necessary for our PRAM be well-defined. In this paper, we assume the following:

- The original PRAM supports concurrent reads and concurrent writes (CRCW) with an arbitary, parametrizable rule for write conflict resultion. In other words, there exists some priority rule

to determine which write operation takes effect if there are multiple concurrent writes in some time step t.

- The compiled, oblivious PRAM (defined below) is a "concurrent read, exclusive write" PRAM (CREW). In other words, the design of our OPRAM construction must ensure that there are no concurrent writes at any time.

We note that a CRCW-PRAM with a parametrizable conflict resolution rule is among the most powerful CRCW-PRAM model, whereas CREW is a much weaker model. Our results are stronger if we allow the underlying PRAM to be more powerful but the our compiled OPRAM uses a weaker PRAM model. For a detailed explanation on how stronger PRAM models can emulate weaker ones, we refer the reader to the work by Hagerup [20].

CPU-to-CPU communication. In the remainder of the paper, we sometimes describe our algorithms using CPU-to-CPU communication. For our OPRAM algorithm to be oblivious, the inter-CPU communication pattern must be oblivious too. We stress that such inter-CPU communication can be emulated using shared memory reads and writes. Therefore, when we express our performance metrics, we assume that all inter-CPU communication is implemented with shared memory reads and writes. In this sense, our performance metrics already account for any inter-CPU communication, and there is no need to have separate metrics that characterize inter-CPU communication. In contrast, Chen et al. [6] defines separate metrics for inter-CPU communication.

Additional assumptions and notations. Henceforth, we assume that each CPU can only store $O(1)$ memory blocks. Further, we assume for simplicity that the runtime of the PRAM, the number of CPUs activited in each time step and which CPUs are activited in each time step are *fixed* a priori and *publicly known* parameters. Therefore, we can consider a PRAM to be a tuple

$$\mathsf{PRAM} := (\Pi, N, T,\ (S_t : t \in [T])),$$

where Π denotes the next instruction circuit, N denotes the total memory size (in terms of number of blocks), T denotes the PRAM's total runtime, and S_t denotes the set of CPUs to be activated in each time step $t \in [T]$, where $m_t := |S_t|$.

Without loss of generality, we assume that $N \geq m_t$ for all t. Otherwise, if some $m_t > N$, we can adopt a trivial parallel oblivious algorithm (through a combination of conflict resolution and oblivious multicast) to serve the batch of m_t requests in $O(m_t \log m_t)$ total work and $O(\log m_t)$ parallel time.

Remark 2. In fact, for the orginal PRAM, we only need to assume that the number of CPUs invoked per time step is publicly known, but not which CPUs — the latter can depend on the secret input. Specificially, we make a slight modification to our OPRAM algorithm: after serving each batch of requests, each CPU writes down its internal state and an invocation bit indicating whether it thinks it needs to be invoked in the next PRAM step. Our OPRAM can now obliviously sort this array of CPU internal states, such that the ones that want to be invoked in the next PRAM step appear before those that do not want to be invoked. In this way, if the t-th PRAM step requires m_t CPUs, in the corresponding OPRAM, the first m_t CPUs can always simulate the PRAM's request CPUs.

3.2 Oblivious Parallel Random-Access Machines

Randomized PRAM. A *randomized PRAM* is a special PRAM where the CPUs are allowed to generate private, random numbers. For simplicity, we assume that a randomized PRAM has

a priori known, deterministic runtime, and that the CPU activation pattern in each time step is also fixed a priori and publicly known.

Statistical closeness. We use the notation $\{X_N\} \overset{\epsilon(N)}{\approx} \{Y_N\}$ to mean that the statistical distance between the probability distributions $\{X_N\}$ and $\{Y_N\}$ is smaller than $\epsilon(N)$.

Oblivious PRAM (OPRAM). A randomized PRAM parametrized with total memory size N is said to be statistically *oblivious*, iff there exists a negligible function $\epsilon(\cdot)$ such that for any inputs $x_0, x_1 \in \{0,1\}^*$,

$$\mathsf{Addresses}(\mathsf{PRAM}, x_0) \overset{\epsilon(N)}{\approx} \mathsf{Addresses}(\mathsf{PRAM}, x_1),$$

where $\mathsf{Addresses}(\mathsf{PRAM}, x)$ denotes the joint distribution of memory accesses made by PRAM upon input x. More specifically, for each time step $t \in [T]$, $\mathsf{Addresses}(\mathsf{PRAM}, x)$ includes the memory addresses requested by the set of active CPUs S_t in time step t along with their CPU identifiers, as well as whether each memory request is a read or write operation. Henceforth we often use the notation OPRAM to denote a PRAM that satisfies statistical obliviousness.

Although it is also possible to define a more relaxed, computational notion of security for OPRAM, in this paper we will achieve the stronger notion of stastitical security. Further, following the convention of most other ORAM and OPRAM works [16,17,22,33,34], we will require that the access patterns for any two inputs have statistical distance that is negligible in the parameter N, that is, the PRAM's total memory size.

Oblivious simulation and performance measures. We say that a given OPRAM *simulates* a PRAM if for every input $x \in \{0,1\}^*$, $\mathsf{OPRAM}(x) = \mathsf{PRAM}(x)$, i.e., OPRAM and PRAM output the same outcome on any input x.

Suppose that an OPRAM simulates a certain PRAM, where

$$\mathsf{PRAM} := (\Pi, N, T,\ (S_t : t \in [T])), \quad \mathsf{OPRAM} := (\Pi', N', T',\ (S'_t : t \in [T']))$$

We define the following performance measures that characterize the overhead incurred by such oblivious simulation.

(a) *Parallel runtime blowup.* The ratio $\frac{T'}{T}$ is said to be the parallel runtime blowup of the oblivious simulation.

(b) *Total work blowup.* Let $m_t := |S_t|$ for any $t \in [T]$ and $m'_t := |S'_t|$ for any $t \in [T']$. The ratio $\frac{\sum_{t\in[T']} m'_t}{\sum_{t\in[T]} m_t}$ is said to be the total work blowup of the oblivious simulation.

3.3 Building Blocks

We now describe a set of standard building blocks that we use.

Oblivious sort. Parallel oblivious sort solves the following problem. The input is an array denoted arr containing n elements and a total ordering over all elements. The output is a sorted array arr' that is constructed obliviously. Parallel oblivious sorting can be achieved in a straightforward way through sorting networks [1], consuming $O(n \log n)$ total work and $O(\log n)$ parallel steps.

Oblivious conflict resolution. Oblivious conflict resolution solves the following problem: given a list of memory requests of the form $\mathsf{In} := \{(\mathsf{op}_i, \mathsf{addr}_i, \mathsf{data}_i)\}_{i\in[m]}$, output a new list of requests denoted Out also of length m, such that the following holds:

- Every non-dummy entry in Out appears in In;

- Every address addr that appears in In appears exactly once in Out. Further, if multiple entries in In have the address addr, the following priority rule is applied to select an entry: 1) writes are preferred over reads; and 2) if there are multiple writes, a parametrizable function $\mathsf{priority}$ is used to select an entry.

We will use the standard parallel oblivious conflict resolution algorithm described by Boyle et al. [3], which can accomplish the above in $O(m \log m)$ total work and $O(\log m)$ parallel steps. More specifically, Boyle et al.'s conflict resolution algorithm relies on a constant number of oblivious sorts and oblivious aggregation.

Oblivious routing. Oblivious routing solves the following problem. Suppose n source CPUs each holds a data block with a distinct key (or a dummy block). Further, n destination CPUs each requests a data block identified by its key. An oblivious routing algorithm routes the requested data block to the destination in an oblivious manner. Boyle et al. [3] showed that through a combination of oblivious sorts and oblivious aggregation, oblivious routing can be achieved in $O(n \log n)$ total work and $O(\log n)$ parallel runtime.

Oblivious select. Parallel oblivious select solves the following problem: given an array containing n values, $v_1, v_2, \ldots, v_n$, compute the minimum value $\min(v_1, \ldots, v_n)$. Parallel oblivious select can be achieved through an aggregation tree: imagine a binary tree where the leaves represent the initial values. We assign one CPU to each node in the tree. Now for each height ℓ going from the leaves to the root, in parallel, each internal node at height ℓ computes the minimum of its two children. Clearly, oblivious select can be attained in $O(n)$ total work and $O(\log n)$ parallel steps.

In a straightforward fashion, oblivious select can also be applied to the following variant problem: suppose that each of n CPUs holds a value, and that only one CPU is holding a real value s (but which one is not known), and all others hold the dummy value $\perp$. At the end of the algorithm, we would like a designated fetch CPU to obtain the real value s.

4 Overview of OPRAM Construction

4.1 Notations

Number of active CPUs in PRAM. In the PRAM model, the number m_t of active CPUs in each step can change — for convenience we assume that it is a power of 2 (if not, we can always pad it up to the nearest power of 2 incurring only a constant factor blowup in costs). The OPRAM internally keeps a parameter $\widehat{m}_t$, which is the number of disjoint subtrees at the end of serving the t-th batch of memory requests. The OPRAM will maintain the invariant $\widehat{m}_t \geq 2m_t$ where m_t denotes the size of the t-th batch of memory requests for the original PRAM (i.e., the PRAM's number of CPUs in the t-th step). Henceforth, without risk of ambiguity, we will write m and $\widehat{m}$ omitting the subscript t.

In general, since the PRAM can have a varying number of CPUs in each time step, our OPRAM algorithm will (lazily) adjust its internal parameter $\widehat{m}$ accordingly to closely track the real m. Our algorithm will be described for this more general scenario, but to aid understanding, the reader may assume that $\widehat{m} = 2m$ and m stays constant during the first read.

Parameter α related to failure probability. Throughout the description, we use $\alpha = \omega(1)$ to denote an appropriately small super constant parameter such that the failure probability is at most $\frac{1}{N^{\Theta(\alpha)}}$, i.e., negligible in N.

4.2 Data Structures

Subtrees. On a high level, our OPRAM maintains a binary tree structure as in Circuit ORAM [34]. We refer the reader to Section 2.1 for a review of the Circuit ORAM algorithm. However, instead of having a complete tree, our OPRAM scheme truncates the tree at height $\ell := \log_2 \widehat{m}$ containing $\widehat{m}$ buckets. In this way, we can view the tree data structure as $\widehat{m}$ disjoint subtrees.

In the Circuit ORAM algorithm, all buckets with heights smaller than ℓ contain at most $O(\widehat{m} + \alpha \log N)$ blocks. In our OPRAM scheme, these blocks are treated as overflowing blocks, and they are held in an overflowing data structure called group stashes as described below.

Group stashes. Instead of having one stash per subtree like in Chen et al. [6], every $\min\{\widehat{m}, \alpha \log_2 N\}$ subtrees share a *group stash*. In this way, when $\widehat{m}$ is sufficiently large, the worst-case occupancy of a group stash is only a constant factor larger than the mean — in comparison, in Chen et al. [6], there is a logarithmic gap between the worst-case and average occupancy of each subtree's stash, causing them to suffer from an extra logarithmic factor in performance.

All group stashes together hold a total of $O(\widehat{m} + \alpha \log N)$ blocks. We use K to denote the capacity of a group stash. When $\widehat{m} < \alpha \log_2 N$, there is only one stash, and its capacity is $K = \Theta(\alpha \log N)$; when $\widehat{m} \geq \alpha \log_2 N$, each stash has capacity $K = \Theta(\alpha \log N)$.

Position map. Our OPRAM scheme views each such stash as the parent of the corresponding subtrees. For ease of description, we still think there is an imaginary root that is the parent of all the stashes.

Now, just like in Circuit ORAM (see Section 2.1), each address $\mathsf{addr} \in [N]$ is associated with a random path, and the path is identified by a leaf node. We use a position map $\mathsf{posmap}[\mathsf{addr}]$ to store the position identifier for address addr. Recall that the *path invariant* states that if a block with address addr is stored in the tree, then it must reside in one of the buckets on the path from the (imaginery) root to the leaf $\mathsf{posmap}[\mathsf{addr}]$. When block addr is accessed (via read or write), its position $\mathsf{posmap}[\mathsf{addr}]$ will be updated to a new leaf chosen uniformly and independently at random. As in previous works [30, 33, 34], the position map is stored in an OPRAM recursively. We use the notation pos-OPRAMs to denote all recursion levels for storing the position map, and we use data-OPRAM to denote the top recursion level for storing data blocks.

Request CPUs and auxiliary CPUs. The t-th batch contains m_t memory requests. We assume that in the OPRAM, there are m_t *request CPUs* that are responsible for receiving the outcome of the memory requests, and then emulate the original PRAM's next instruction circuit based on the fetched data.

Besides the request CPUs, our OPRAM algorithm will need to recruit the help of *auxiliary* CPUs when serving each batch of memory requests. We will give meaningful names to these auxiliary CPUs, such as fetch CPUs, stash CPUs, which indicate their functions.

4.3 Overview of One Simulated PRAM Step

To serve each batch of memory requests, a set of CPUs interact with memory in two synchronized phases: in the *fetch* phase, the request CPUs receive the contents of the requested blocks; in the second *maintain* phase, the CPUs collaborate to maintain the data structure to be ready for the next PRAM step.

The description below can be regarded as an expanded version of Section 2.8. In particular, we now spell out what happens if m_t varies over time.

Fetch phase. At the beginning of this phase, each request CPU sends its instruction to the corresponding fetch CPU. The fetch CPUs run the following synchronized steps.

(i) *Offline preparation: all recursion levels in parallel.* First, for each recursion level d in parallel, write down all request addresses truncated to level d. Now, run an instance of the oblivious conflict resolution algorithm to suppress duplicates of the requests at level d.

Now, given a list of m (possibly dummy) memory requests with conflicts resolved, an offline preparation phase is conducted simultaneously across all recursion levels. As a result, each recursion level d generates an "incomplete instruction array" denoted $\mathsf{Instr}^{\langle d \rangle}$ of length $2m$ (of which at most m instructions are real) as well as additional auxiliary data to later facilitate the routing of position identifiers to the next recursion level. Later, when the incomplete instruction array receives the position identifiers from the previous level of recursion, it becomes complete. The specifics of the offline preparation algorithm are described in Sections 2.7 and 6.

(ii) *Fetch and route: level by level.* For each recursion level, we now start with a list of $2m$ (possibly dummy) completed instructions. Now,

- *Parallel read.* $2m$ fetch CPUs each obtain one instruction, and each instruction (if not dummy) specifies a path to read and a block to look for. Now, each fetch CPU further recruits the help of $K := O(\alpha \log N)$ auxiliary CPUs; and the auxiliary CPUs in parallel read all locations on the assigned path (containing the path in the corresponding subtree, as well as the group stash) looking for the requested block. This step can be completed in $O(K) = O(\alpha \log N)$ total work per path and a single parallel step.
- *Gather.* Using the oblivious select algorithm described in Section 3.3, the auxiliary CPUs for each path jointly route the fetched block back to the corresponding fetch CPU. This can be achieved in $O(K) = O(\alpha \log N)$ total work per path and $O(\log K) = O(\log \log N)$ parallel runtime.
- *Route.* For a pos-OPRAM level, assume that each block contains two position identifiers. Then, a total of $4m$ position identifiers have been fetched, among which at most m must be consumed by the next recursion level.

  We now rely on an efficient online routing algorithm to route the m position identifiers to the next recursion level, and in the process we complete the instruction arrays of the next recursion level. We stress that this step contains some of the most non-trivial techniques we devise in this paper. In particular, we show that given appropriate offline preparation, the online routing can be completed in $O(\log \log N)$ parallel runtime. We defer the details of this algorithm to Section 6.

(iii) *Oblivious multicast: once per batch of requests.* Finally, when the data-OPRAM has fetched all requested blocks, we rely on a standard oblivious routing algorithm (see Section 3.3) to route the resulting blocks to the request CPUs. This step takes $O(\log m)$ parallel time and $O(m \log m)$ work.

Maintain phase. All of the following steps are performed *in parallel across all recursion levels*:

(i) *Removing accessed blocks from group stashes and subtrees.* The fetch CPUs collaboratively remove the accessed blocks from the group stashes and the subtrees in an oblivious manner. We devise an efficient algorithm (see Section 5.1) to perform such removals in parallel. In our algorithm, each fetch CPU recruits $O(K) = O(\alpha \log N)$ assistant CPUs, and the removal procedure can be accomplished with $O(\log m)$ parallel runtime and $O(mK) = O(\alpha m \log N)$ total work.

(ii) *Treating all the group stashes as a single pool.* After the accessed blocks are removed, all the group stashes are treated as a single pool without any particular structure. If $\widehat{m} < 2m$, i.e., $\widehat{m}$ is too small, we then set $\widehat{m} := 2m$ and include all buckets with heights less than $\log_2 \widehat{m}$ in the pool. Henceforth we may assume that $\widehat{m} \geq 2m$. The case when $\widehat{m}$ is too large will be dealt with later.

(iii) *Passing updated blocks to the pool.* Each fetch CPU updates its fetched block and tags the block with its new position identifier (recall that the position identifier was chosen by the previous recursion level and passed down through the level-to-level routing). The updated blocks are merged into the pool. The pool temporarily increases its capacity to hold these extra blocks, but the extra memory will be released at the end of the maintain phase when we reconstruct group stashes from the pool.

(iv) *Selection of eviction candidates.* Following the deterministic, reverse-lexicographical order eviction strategy of Circuit ORAM [34], we choose the next $2m$ eviction paths (pretending that all subtrees are part of the same big ORAM tree). The $2m$ eviction paths go through $2m$ subtrees, henceforth referred to as the *evicting subtrees*. Note that if m stays constant throughout, then it always holds that $\widehat{m} = 2m$, and all subtrees would then be evicting subtrees.

    We devise an "oblivious eviction candidate selection" algorithm in Section 5.3 that leverages oblivious sorting to select one candidate block for each of the evicting subtrees. For large $\widehat{m}$, this step takes $O(\log \widehat{m})$ parallel runtime and $O(\widehat{m} \log \widehat{m})$ total work.

(v) *Eviction into substrees.* In parallel, for each evicting subtree, the eviction algorithm of Circuit ORAM [34] is performed for the candidate block the subtree has received. The straightforward strategy takes $O(\log N)$ parallel runtime and $O(m \log N)$ total work over all the $2m$ subtrees.

    After the eviction algorithm completes, if the candidate block fails to be evicted into the subtree, it will be returned to the pool; otherwise if the candidate block successfully evicts into the subtree, a dummy block is returned to the pool.

(vi) *Adjusting the number of subtrees.* If $\widehat{m} > 2m$, i.e., the number of subtrees is too large, we will halve the number of subtrees by letting $\widehat{m} := \frac{\widehat{m}}{2}$. This means that we must reconstruct one more height of the big Circuit ORAM. Let Z be the bucket size of the ORAM tree. To reconstruct a height of size $\widehat{m}$, we must reconstruct $\widehat{m}$ buckets each of size Z. This can be achieved by repeating the eviction candidate selection algorithm Z number of times (see Section 5.4).

    Note that when $\widehat{m} > 2m$, we do not immediately adjust the value of $\widehat{m}$ to $2m$, since if $\widehat{m}$ is much larger than $2m$, this would require us to reconstruct multiple heights of big Circuit ORAM tree in one shot, and the cost of this can be too expensive. We therefore adopt a lazy adaptation algorithm that only halves $\widehat{m}$ in each step should it ever be too large.

(vii) *Cleanup: reconstructing the group stashes.* Finally, we convert the flat pool back to $\frac{\widehat{m}}{\alpha \log_2 N}$ number of group stashes. To achieve this, we devise an algorithm based on oblivious sorting that incurs $O(\widehat{m} \log \widehat{m})$ total work and $O(\log \widehat{m})$ parallel runtime (for large $\widehat{m}$) — we defer its detailed description to Section 5.4. Probabilistic analysis on the stash utilization in Section 5.6 ensures that no real blocks are lost during this reconstruction with all but negligible probability.

5 Details of the Maintain Phase

An overview of the maintain phase was provided in Section 4.3. In this section, we give the details of each step of the algorithm. Recall that all the following steps are performed in parallel across all recursion levels.

5.1 Removing Fetched Blocks from Subtrees and Group Stashes

Problem statement. In the fetch phase, $2m$ fetch CPUs each look for a block in both a group stash and a subtree (where at most m of them are real blocks). One fetch CPU is responsible for each request instruction that is associated with a fetch path. After completing the fetch operation, each of the $2m$ fetch CPUs obtains the following tuple which serves as the CPU's input to the simultaneous removal algorithm described below.

- The block address that it is supposed to look for (which can be dummy).
- The fetch path scanned (i.e., the block's old position identifier).
- The height of the bucket on the path from which the fetched block should be removed (if the block was found on the path in the subtree).

Now, in the maintain phase, we need to remove fetched blocks from the group stashes and subtrees. We describe how the fetched blocks are removed from the subtrees, and the case for the group stashes is similar. Even though the fetched blocks will be returned to the group stashes, we still have to remove a block if it is found in the group stash. The reason is that the contents of the block (such as its position identifier) have to be updated.

Challenge: write conflicts. The difficulty with simultaneous removal is write conflict resolution. In particular, the fetch paths may intersect, i.e., multiple fetch CPUs may have read the same bucket. Therefore, for each bucket that lies on multiple fetch paths (i.e., read by multiple fetch CPUs), we must accomplish the following: 1) elect a representative CPU to perform removal for this bucket; and 2) have all other fetch CPUs that read the bucket inform the representative CPU whether a removal operation is necessary, and if so, which block to remove. A naïve way to achieve the above is for the $2m$ fetch CPUs to perform oblivious sorting and aggregation for each height of the tree (similar to the conflict resolution algorithm in Section 3.3) — but this approach would cost $O(m \log m \log N)$ total work, which is too expensive.

We observe, however, that the adversary can observe which paths in the subtrees are scanned. Therefore, although we need to hide which CPU wants to remove a block from which bucket, the topology of all the fetch paths are publicly known. This observation allows us to design a new simultaneous removal algorithm that is more efficient.

Simultaneous removal algorithm. We describe our simultaneous removal algorithm in detail. We will show that the entire removal algorithm completes in $O(\log m)$ parallel runtime and $O(\alpha m \log N)$ total work.

- *Sorting fetch paths.* Each of the $2m$ fetch CPU writes down its input tuple containing the fetch path, the block to be removed, and the physical location of the block on the path. All input tuples form an array of size $2m$. We now obliviously sort this array by their fetch path such that the leftmost fetch path appears first. This step takes $O(m \log m)$ total work and $O(\log m)$ parallel steps.

- *Table creation.* In parallel, fill out a table Q where each row corresponds to a height in the tree, and each column corresponds to a fetch path (in sorted order from left to right). Specifically, $Q[\ell][i]$ stores the following: 1) the identifier of the block that the i-th fetch CPU wants to remove from height-ℓ of its fetch path (or dummy if none exists); and 2) the identifier of the corresponding fetch path; and 3) the height ℓ.

  Given the output of the previous step, it is not hard to see that this step can be completed in $O(1)$ parallel steps and in $O(m \log N)$ total work. We assume that a CPU is assigned to each entry of the table Q, and hence sometimes treat an entry in Q as a CPU.

- *Grouping.* For each row ℓ of the table Q, multiple locations in the row $Q[\ell]$ can correspond to the same bucket. Since the fetch paths have been sorted, if a bucket appears multiple times in some row $Q[\ell]$, all these occurrences must be in adjacent locations — henceforth we refer to locations that correspond to the same bucket as a *group*. The leftmost member of a group is said to be the *group representative*.

  The following algorithm is run for each row independently in parallel. The goal is to let each entry know the indices of the leftmost and the rightmost entries in its group. Let J be an upper bound on the number of entries in a group. Trivially, we have $J \leq m$. However, since the m requests are distributed randomly among $\widehat{m} \geq 2m$ subtrees, each subtree receives at most $\alpha \log N$ requests with all but negligible probability. Hence, with all but negligible probability, we have $J \leq \min\{m, \alpha \log N\}$; without loss of generality, we assume that J is a power of 2. It is not hard to see that this procedure in total requires $O(m \log N)$ total work and $O(\log J) = O(\min(\log \log N, \log m))$ parallel runtime over all rows of the table.

  1. Recall that each entry in row $Q[\ell]$ is equipped with a CPU. Each entry looks at its left neighbor and right neighbor in Q and decides whether it is the leftmost entry in its group, i.e., the group representative.
  2. Each entry that is its group's representative performs a binary search to identify the rightmost member of the group. To achieve this, the entry looks at the element $1, 2, 4, \ldots$ positions away to its right, until it reaches the end of the row or an element that does not belong to the same bucket. Then, the representative entry performs a binary search in the range identified above to find the rightmost member of its group. Recall that each group has at most J elements. Therefore, for each row, this step can be accomplished in $O(\log J)$ parallel time and $O(m)$ total work.

     At this point, non-representative entries are *asleep*, and representative entries are *awake*. The invariant is that an awake entry knows the indices of the leftmost and the rightmost entries in its group.
  3. For round from $i = 0$ to $\log_2 J$, the following happens. Each currently asleep entry whose index j satisfies $j \mod \frac{J}{2^i} = 0$ begins to *wake up* and waits for a message from an already awake entry in its group. On the other hand, each entry i that is already awake sends a message (containing the indices of the leftmost and the rightmost entries in its group) to an entry j (if any) that satisfies the following condition: (i) j is the smallest index larger than i such that $j \mod \frac{J}{2^i} = 0$, (ii) entry j is beginning to wake up in this round, and (iii) entry j is in the same group as entry i. After receiving a message, entry j becomes awake.

     It follows that after the last round, every entry is awake and hence knows the leftmost and the rightmost entries in its group. Apart from the representatives, each entry $i = (2k+1) \cdot 2^r$ will be awake for r rounds. Hence, the total work is $O(m)$ for each row and the parallel runtime is $O(\log J)$.

- *Gathering removal instruction.* The goal of this step is for all non-representative CPUs of a bucket to inform the representative CPU whether it wants to remove a block from the bucket, and if so, which block. To achieve this, we repeat the oblivious select algorithm (see Section 3.3) $Z = O(1)$ times, where Z denotes the bucket size. Hence, for each location in the bucket, the representative CPU gathers information from all CPUs in its group on whether to remove the block from that specific location.

  Since each row has size $2m$, at most $2m$ locations in the table Q can correspond to the same bucket. Therefore, this step can be accomplished in $O(\log m)$ parallel steps and $O(m \log N)$ total work.

- *Removal.* All representative CPUs perform the actual removal from the bucket it represents.

The above description did not take group stashes into account. However, we point out that the case for removing blocks from the group stashes is similar, because each stash can be treated like a path, where the index of a block within a stash behaves like a height in the path. When we take the group stashes into account, effectively the height of the tree becomes $O(\alpha \log N)$ instead of $O(\log N)$. Therefore, overall, the simultaneous removal can be accomplished in $O(\log m)$ parallel time and $O(\alpha m \log N)$ total work.

5.2 Convert Group Stashes to Pool

For completeness, we repeat two relatively simple steps that were described in the overview in Section 4.3.

Interpreting the group stashes as a single pool. After the fetched blocks are removed, we will treat all the group stashes as a single pool without any particular structure. If $\widehat{m} < 2m$, i.e., the number of subtrees is too small, we set $\widehat{m} := 2m$, and also treat all buckets at heights smaller than $\log_2 \widehat{m}$ as part of the pool (by simply ignoring part of the tree structure). Note that the pool has a capacity of $O(\widehat{m} + \alpha \log N)$ blocks.

If $\widehat{m} > 2m$, i.e., the number of subtrees is too large, we do not change $\widehat{m}$ at this point; we modify $\widehat{m}$ at the end of the maintain phase during cleanup.

Passing updated blocks to the pool. Each fetch CPU updates its fetched block and tags the block with its new position identifier. The new position identifier was in fact chosen by the previous recursion level and passed down during the level-to-level routing, when the next level's instruction array is being completed — the details of this will be presented in Section 6.

The updated blocks are merged into the pool. The pool temporarily increases its capacity to hold these extra blocks, but the extra memory will be released at the end of the maintain phase when we reconstruct group stashes from the pool.

5.3 Evictions

Recall that in the sequential Circuit ORAM [34], whenever a fetched (and possibly updated) block is added to the root, two path evictions must be performed. The goal of Circuit OPRAM is to simulate the stochastic process of Circuit ORAM. However, since Circuit OPRAM does not maintain the tree structure for lower heights of the tree, we only need to partially simulate Circuit ORAM's stochastic process for the $\widehat{m}$ disjoint subtrees that Circuit OPRAM does maintain. Our algorithms described below make use of certain non-blackbox properties of the Circuit ORAM algorithm [34]. In our description below, we will point out these crucial properties as the need arises, without re-explaining the entire Circuit ORAM construction [34].

Select $2m$ distinct eviction paths in $2m$ distinct subtrees. At this point, a batch of $2m$ requests have been made, of which at m real blocks are fetched and possibly updated blocks (together with the dummy blocks). All the $2m$ fetched (real or dummy) blocks have been added back to the pool which is an overflowing structure containing all smaller heights of the big Circuit ORAM tree. To perfectly simulate Circuit ORAM's stochastic process, at this point we must perform $2m$ evictions on $2m$ distinct paths. We leverage Circuit ORAM's deterministic, reverse-lexicographical order for determining the next $2m$ eviction paths. The specifics of the eviction path selection criterion is not important here, and the reader only needs to be aware that this selection criterion is fixed a priori and data independent. For more details on eviction path selection, we refer the reader to Circuit ORAM [34].

Fact 2. *Observe that at this point, $\widehat{m} \geq 2m$. Due to Circuit ORAM's eviction path selection criterion, all $2m$ eviction paths will not only be distinct, but also correspond to distinct subtrees — henceforth we refer to these subtrees as evicting subtrees. For the special case when m stays a constant over time, it always holds that $\widehat{m} = 2m$. In this case, we have that all $2m$ subtrees are evicting subtrees, and exactly one path is evicted in each subtree.*

Select $2m$ eviction candidates. We will now leverage a special property of the Circuit ORAM's eviction algorithm described earlier by Fact 1 such that we perform a "partial eviction" only on the subtrees maintained by our Circuit OPRAM. Recall that Fact 1 says the following:

- For Circuit ORAM's eviction algorithm, at most one block passes from $\mathsf{path}[: i]$ to $\mathsf{path}[i + 1 :]$ for each height i on the eviction path denoted path. In this case we also say that the block passes through the boundary between height i and height $i + 1$.
- Moreover, if a block does pass through the boundary between height i and height $i + 1$, it must be the block that is *deepest* with respect to the eviction path, where *deepest* is a criterion defined by Circuit ORAM [34]. Since the specifics of the *deepest* function are not important here, we simply refer the reader to Circuit ORAM [34] for details.

Therefore, we only need to elect one candidate block from the pool for each of the $2m$ eviction paths on which we would like to perform eviction. We describe an algorithm for performing such selection based on two different cases:

- **Case 1: when $\widehat{m} > \alpha \log N$.** We devise an algorithm based on a constant number of oblivious sorts. Since the pool contains $O(\widehat{m})$ blocks, the parallel runtime of this algorithm is $O(\log \widehat{m})$ and the total work is $O(\widehat{m} \log \widehat{m})$.

(a) In the beginning, each block in the pool is tagged with the block's position identifier.

Now, assign one auxiliary CPU to each block in the pool, compute and tag the block with the additional metadata $(\mathsf{treeid}, \mathsf{priority})$ which will later be used as a sort key, where treeid denotes the block's destined subtree if the destined subtree is an evicting subtree, otherwise $\mathsf{treeid} := \perp$; further, $\mathsf{priority}$ denotes the block's eviction priority within the subtree. The block's $\mathsf{priority}$ value can be computed using the *deepest* rule of Circuit ORAM algorithm [34], using the eviction path (corresponding to the subtree identified by treeid), and the block's position identifier as input.

(b) We add $2m$ dummy blocks to the pool, each tagged with a distinct treeid from the $2m$ evicting subtrees. The $\mathsf{priority}$ values of these dummy blocks are set to $-\infty$, i.e., least preferred. These dummy blocks are added only to make sure that in the end, every evicting subtree gets a block even though the block may be dummy.

(c) In parallel, oblivious sort all blocks based first on their treeid, and second on their $\mathsf{priority}$. In other words, if we order all evicting subtrees from left to right, blocks destined for a smaller evicting subtree occurs before blocks destined for a larger evicting subtree. Further, all blocks not destined for an evicting subtree occurs at the end. For the same evicting subtree, a block with a higher $\mathsf{priority}$ value occurs before a block with a lower $\mathsf{priority}$ value. If there are still ties, we can adopt an arbitrary way to break the tie.

(d) Now assign one CPU to each block in this sorted list. Each CPU looks at its left neighbor to determine if it is the top priority block for an evicting subtree. For any block that is not the top priority block for an evicting subtree, the CPU changes the block's metadata tag to $\perp$.

The result of this step is an array containing all blocks in the original pool, such that only those that are the final eviction candidates are tagged with a real tag $(\mathsf{treeid}, \mathsf{priority})$; all other blocks are tagged with $\perp$.

(e) Finally, oblivious sort the above list based on the block's treeid, and moving all blocks tagged with $\perp$ to the end. Now, the first $2m$ blocks in the sorted list are the selected eviction candidates for the $2m$ evicting subtrees respectively. The remaining blocks are returned to the pool.

- **Case 2: when $\widehat{m} \leq \alpha \log N$.** In this case, the pool contains $\Theta(\alpha \log N)$ blocks, and performing oblivious sort will cause a total work of $\Omega(\log N \log \log N)$, which is too expensive if $\widehat{m}$ is small. Instead, we perform the following, which can be accomplished in $O(\log m + \log \log N)$ parallel runtime and $O(\alpha m \log N)$ total work:

(a) For each of the $2m$ eviction paths, we assign $O(\alpha \log N)$ auxiliary CPUs, such that each auxilary CPU looks at one block in the pool and computes its priority with respect to the eviction path (if the block does not belong to the relevant eviction path, its priority is defined to be $-\infty$).

(b) Now, for each of the $2m$ eviction paths, its group of $O(\alpha \log N)$ auxiliary CPUs perform an oblivious select to determine the highest priority block with respect to the eviction path.

(c) Finally, for each block in the pool, the $2m$ CPUs that have looked at the block perform an oblivious select operation to determine a bit indicating whether the block should be removed. Then, we assign a CPU for each block in the pool that reads this bit, and performs removal if necessary.

Evictions. At this point, each of the $2m$ eviction paths has received one candidate block (which can be dummy). Hence, these $2m$ evictions can be carried out in parallel, each according to the (sequential) eviction algorithm of Circuit ORAM [34]. More specifically, we first expand each eviction path by one block that corresponds to the eviction candidate selected earlier (we henceforth refer to this as the smallest block on the path); we then run Circuit ORAM's (sequential) eviction algorithm on each of these $2m$ (expanded) paths in parallel.

At the end, the smallest block on each eviction path is returned to the pool. Note that if the eviction candidate has been successfully evicted into the path, then the smallest block on the path would be dummy, and thus a dummy block is returned to the pool. In a final cleanup step described later, we restore the group stashes from the pool, suppressing a subset of the dummy blocks in the process, thus ensuring that the group stashes/pool data structures do not keep growing.

Doing this according to Circuit ORAM's eviction algorithm [34] takes $O(\log N)$ parallel runtime and $(m \log N)$ total work. This would suffice for our basic result for general block sizes. However, as an optimization for larger block sizes, we will later describe how to reduce the parallel runtime of the eviction algorithm to $O(\log \log N)$, at the cost of increasing the total work to $O(m \log N \log \log N)$ (see Section A).

5.4 Cleanup: Reconstructing the Group Stashes

Adjusting the number of subtrees. If $\widehat{m} > 2m$, i.e., the number of subtrees is too large, we will halve the number of subtrees by letting $\widehat{m} := \frac{\widehat{m}}{2}$. This means that we must reconstruct one more height of the big Circuit ORAM. Let Z be the bucket size of the ORAM tree. To reconstruct a height of size $\widehat{m}$, we must reconstruct $\widehat{m}$ buckets each of size Z. This can be achieved by repeating the eviction candidate selection algorithm Z number of times (see Section 5.3).

Note that when $\widehat{m} > 2m$, we do not immediately adjust the value of $\widehat{m}$ to $2m$, since if $\widehat{m}$ is much larger than $2m$, this would require us to reconstruct multiple heights of big Circuit ORAM tree in one shot, and the cost of this can be too expensive. We therefore adopt a lazy adaptation algorithm that only halves $\widehat{m}$ should it ever be too large.

Convert pool back to group stashes. We now restore the $\lceil \frac{\widehat{m}}{\alpha \log N} \rceil$ number of group stashes from the pool. Each group stash is given $K = \Theta(\alpha \log N)$ space as given by Corollary 1. Although the pool may contain more than $K \cdot \lceil \frac{\widehat{m}}{\alpha \log N} \rceil$ blocks, a subset of these blocks are dummy — we prove later in Section 5.6 that no real block is removed during this pool to group stash conversion. In other words, during the pool to group stash conversion, we also effectively suppress a subset of the dummy blocks in the pool, thus releasing some extra space previously allocated to hold dummy blocks in the pool.

We now describe the pool to group stash conversion algorithm. We consider two cases:

- **Case 1: when $\widehat{m} > \alpha \log N$.** The group stashes can be restored using a constant number of oblivious sorts consuming $O(\log \widehat{m})$ parallel time and $O(\widehat{m} \log \widehat{m})$ total work.

(a) Let $K := \Theta(\alpha \log N)$ denote the maximum capacity of each group stash as indicated by Corollary 1. Assign one CPU per block of the pool, and in parallel tag each block with the metadata stashid denoting which group stash the block belongs to. Note that the stashid of a block can be computed from the block's position identifier and any block always carries around its position identifier. At this time, all dummy blocks are assigned stashid $:= \perp$.

(b) For each group stash $i \in [\lceil \frac{\widehat{m}}{\alpha \log N} \rceil]$, add K dummy blocks to the pool whose stashid is set to i. This step is necessary to ensure that later each group stash receives at least K possibly dummy blocks.

(c) Oblivious sort all blocks in the pool based on their stashid. In this way, blocks with a smaller stashid occurs before blocks with a larger stashid. All blocks with the same stashid occur consecutively in a row.

(d) Now assign one auxiliary CPU to each block in the above sorted array. A CPU for the i-th block looks at the $(i-K)$-th block: if $i - K > 0$ and the $(i-K)$-th block has the same stashid as the i-th block, then CPU i overwrites its block's stashid with $\perp$. In this way, exactly K blocks for each stashid will be preserved.

(e) Finally, oblivious sort the above array again based on each block's stashid. Now, the first $K \cdot \lceil \frac{\widehat{m}}{\alpha \log N} \rceil$ locations in the array correspond to the $\lceil \frac{\widehat{m}}{\alpha \log N} \rceil$ group stashes respectively, each of which has size exactly K.

- **Case 2: when $\widehat{m} \leq \alpha \log N$.** In this case, there is only one stash, and $O(m)$ blocks of memory is supposed to be released from the stash. We further consider two sub-cases.

(i) Case $m \geq \log \log N$. In this case, we can release $O(m)$ blocks from the stash by oblivious sorting, which takes parallel runtime $O(\log K) = O(\log \log N)$ and total work $O(K \log K) = O(mK)$.

(ii) Case $m < \log \log N$. In this case, oblivious sorting on the stash causes too much total work. In this case, we do not release the extra $O(m)$ blocks from the stash and just let it grow.

Observe that $\log \log N << K = O(\alpha \log N)$. As long as there is some PRAM step in which m is at least $\log \log N$, we can afford to perform oblivious sorting to reduce the capacity of the stash. On the other hand, if m is small for consecutive $\log \log N$ steps, then the stash will grow by at most $O(\log \log N)^2 << K$ blocks. Then, at the end of the $\log \log N$ steps, we can perform oblivious sorting to clean up the stash, with total work $O(K \log K) = O(K \log \log N)$, which causes only $O(K)$ total work when amortized over $\log \log N$ steps.

5.5 Performance of the Maintain Phase

Accounting for the cost of all of the above steps, we can easily derive the following lemma for the performance of the maintain phase.

Lemma 1 (Performance of the maintain phase)**.** *Suppose that the block size $B = \Omega(\log N)$, and at the beginning of serving the batch of m requests the OPRAM has $\widehat{m}$ number of subtrees. Then, to serve the batch of m requests, the maintain phase, over all $O(\log N)$ levels of recursion, incurs a parallel runtime of $O(\log N)$ and a total work of $O(\alpha m \log^2 N + \widehat{m} \log^2 N)$.*

Remark 3. To be precise, the parallel runtime is $O(\log m + \log \widehat{m} + \log\log N + \log N)$, where the $O(\log N)$ term comes from a sequential implementation of Circuit ORAM's eviction procedure [34]. This suffices for our basic result for generic block sizes. Later, as an optimization for large block sizes, we shall see that Circuit ORAM's eviction procedure can be carried out in parallel runtime $O(\log\log N)$, at the cost of increasing the total work of eviction to $O(\log N \log\log N)$.

5.6 Analysis of Stash Usage

Recall that in the maintain phase, the pool increases its capacity temporarily to receive the newly updated blocks. However, after performing eviction and restoring the group stashes, the temporary extra memory is released. We need to prove a stochastic bound to show that except with negligible probability, none of the real blocks ever get lost during the process of rebuilding group stashes (during which we effective compress the space).

Recall that in Circuit ORAM, for every updated block added to the root stash, evictions are performed on two paths, each for the left and the right branch at the root. The details of the eviction strategy are given in the Circuit ORAM paper [34]. To illustrate block utilization, the non-oblivious Algorithm 1 in Section A shows what happens to the blocks during a path eviction. Most of the details are not important for our proof, but the important point is the following result on the stash usage of Circuit ORAM. We consider Circuit ORAM's stash as part of the root bucket.

Fact 3 (Stash usage in Circuit ORAM [34])**.** *Let R be an integer. After each oblivious access, the probability that Circuit ORAM's stash contains more than R blocks is at most $O(e^{-R})$.*

Batched Circuit ORAM. For the purpose of our stochastic analysis, it helps to think of an alternative random process which we call "Batched Circuit ORAM". Imagine that we run a variant of the (sequential) Circuit ORAM algorithm. Specifically, there is a complete Circuit ORAM binary tree. For a batch of m memory requests:

1. We first perform the read phase of all m requests sequentially following the Circuit ORAM algorithm, placing all fetched blocks in the root bucket (and therefore the root bucket is allowed to be larger).
2. Then, we perform $2m$ path evictions also following Circuit ORAM's strategy of choosing eviction paths and following its eviction algorithm.

In other words, Batched Circuit ORAM is almost identical to Circuit ORAM, except that Circuit ORAM interleaves reads and evictions such that each read is followed by two evictions; however, here we perform $2m$ evictions after every m reads, i.e., reads and evictions are batch-processed.

For Batched Circuit ORAM, Circuit ORAM, or Circuit OPRAM, an execution trace is defined by the pair $(\psi, \vec{S})$ where ψ is a random string that defines position identifiers for all accessed blocks in temporal order, and $\vec{S} := \{S_t : t \in [T]\}$ denotes the memory request sequence. Now consider an execution trace defined by $(\psi, \vec{S})$ for Batched Circuit ORAM. Imagine that after serving the t-th

batch of requests, we truncate the ORAM tree at a height $\ell^* := \log_2 \widehat{m}$ that contains $\widehat{m}$ nodes, which defines $\widehat{m}$ subtrees. We define $\mathsf{BatchedORAMStash}^{t,\widehat{m},i,j}(\psi, \vec{S})$ to be the set of blocks that reside at a height smaller than $\ell^* := \log_2 \widehat{m}$ but can legally reside in any subtree numbered $k \in [i..j]$.

We now consider an execution defined by $(\psi, \vec{S})$ of our Circuit OPRAM. Let $\mathsf{OPRAMGrpStash}^{t,i,j}(\psi, \vec{S})$ denote the set of blocks that can legally reside in a group stash that covers subtrees i to j at the end of the t-th batch of memory requests.

Fact 4 (Stash equivalence between Circuit OPRAM and Batched Circuit ORAM)**.** *For any ψ, any $\vec{S}$, for any t, for any $i..j$ that correspond to the subtrees covered by some group stash in the OPRAM algorithm after serving t batches of requests, it holds that*

$$\mathsf{OPRAMGrpStash}^{t,i,j}(\psi, \vec{S}) = \mathsf{BatchedORAMStash}^{t,\widehat{m},i,j}(\psi, \vec{S})$$

where $\widehat{m} := \widehat{m}(\psi, \vec{S})$ is the number of subtrees in the OPRAM algorithm immediately after serving the t-th batch of requests in the execution trace defined by $(\psi, \vec{S})$.

More informally, the utilization of a particular group stash in our OPRAM exactly corresponds to the blocks in the complete binary tree of the Batched Circuit ORAM that can reside in one of the group stash's subtrees and are currently at heights less than $\log_2 \widehat{m}$.

Circuit ORAM stochastically dominates Batched Circuit ORAM. As explained earlier, in Batched Circuit ORAM, we perform m reads, add m blocks to the root, and then perform $2m$ evictions. By contrast, in Circuit ORAM, each read is immediately followed by 2 evictions. We now show that Circuit ORAM stochastically dominates Batched Circuit ORAM in terms of stash usage. The intuition is that as opposed to eviction performed in a batch, a block added later in Circuit ORAM cannot benefit from the eviction due to accessing an earlier block. This intuition is formalized in the following Lemma 2.

Consider an execution trace defined by $(\psi, \vec{S})$ of Batched Circuit ORAM, for any given t, any path, any height i, let $X^{t,\mathsf{path},i}(\psi, \vec{S})$ denote the number of blocks in $\mathsf{path}[: i-1]$ that can legally reside in $\mathsf{path}[i]$ after serving the t-th batch of requests in this execution trace Similarly, we can define an analogous variable denoted $\widetilde{X}^{t,\mathsf{path},i}(\psi, \vec{S})$ for an execution trace defined by $(\psi, \vec{S})$ of the (non-batched) Circuit ORAM.

Lemma 2 (Circuit ORAM stochastically dominates Batched Circuit ORAM)**.** *For any random string ψ, any memory request sequence $\vec{S} := \{S_t : t \in [T]\}$, for any t, any eviction* path*, and any i, it holds that*

$$X^{t,\mathsf{path},i}(\psi, \vec{S}) \leq \widetilde{X}^{t,\mathsf{path},i}(\psi, \vec{S})$$

Proof. Henceforth we use the terms *sequential eviction* and *batch eviction* to differentiate the eviction processes of the regular Circuit ORAM and the Batched Circuit ORAM, respectively. We assume that the statement holds just before a batch eviction, and it suffices to show that the statement holds again after a batch eviction corresponding to adding m blocks is performed. Observe that the same $2m$ eviction paths are used in both the sequential and the batch cases. Hence, if none of these paths intersect $\mathsf{path}[i]$, then the statement certainly holds.

According to the eviction strategy given in Algorithm 1, even when the eviction path intersects $\mathsf{path}[0..i]$ and there is a block in $\mathsf{path}[0..i-1]$ that can legally reside in $\mathsf{path}[i]$, no block might be passed from $\mathsf{path}[0..i-1]$ to $\mathsf{path}[i]$ because there could be no available slots in some descendants of $\mathsf{path}[i-1]$. To eliminate this complication in the analysis, another (imaginary) abstraction known as the ∞-ORAM is often used in previous work as in [34].

The ∞-ORAM is a variant in which each bucket has infinite capacity. When it is applied, we assume all other rules remain the same. A non-trivial result [34] is that when path eviction is

performed according to Algorithm 1, the bucket utilization of the ORAM with bounded bucket capacity can be obtained exactly from a post-processing procedure applied to the corresponding ∞-ORAM.

The post-processing procedure repeatedly applies the following to the ∞-ORAM. If there is a non-root bucket that contains more blocks than its capacity, then extra blocks are removed and placed in the bucket's parent. It is possible to select which blocks to remove to achieve the exact configuration as the normal ORAM with bounded capacity. However, we just need to care about how many blocks are in each bucket, and hence we can push back arbitrary extra blocks from a bucket to its parent.

Therefore, we just need to prove the statement for the case of the ∞-ORAM variant. Observe that in the case of batch eviction, all the newly added m blocks that can legally reside in $\mathsf{path}[i]$ will be considered for each of the $2m$ eviction paths that intersect $\mathsf{path}[i]$. For the sequential case, because of the interleaving between block insertion at the root and path eviction, it is possible that a newly added block that can legally reside in $\mathsf{path}[i]$ might miss a previous eviction path that intersects $\mathsf{path}[i]$. Hence, the statement must also hold for ∞-ORAM after a batch eviction.

In the ∞-ORAM variant, the statement actually holds for every height i. Hence, the statement will still hold after applying the post-processing procedure. This completes the proof. □

Utilization of a group stash. Let $\mathsf{ORAMStash}^{t,\widehat{m},i,j}(\psi,\vec{S})$ be defined in the same way as $\mathsf{BatchedORAMStash}^{t,\widehat{m},i,j}(\psi,\vec{S})$, but for the (non-batched) Circuit ORAM instead.

To bound the random variable $\mathsf{OPRAMGrpStash}^{t,i,j}$ which denotes the utilization of a group stash in our OPRAM, due to Fact 4, we can bound $\mathsf{BatchedORAMStash}^{t,\widehat{m},i,j}$. Now, due to Lemma 2, we can instead bound the random variable $\mathsf{ORAMStash}^{t,\widehat{m},i,j}$.

Lemma 3. *For any request sequence $\vec{S}$, any t, any $\widehat{m}$, any $i,j \in [\widehat{m}]$ such that $i < j$, it holds that*

$$\Pr_{\psi}\left[\mathsf{ORAMStash}^{t,\widehat{m},i,j}(\psi,\vec{S}) > 2Z(j-i+1) + 2\log N + R\right] \leq O(e^{-R}),$$

where $Z = O(1)$ is the capacity of a on-root bucket.

Proof. Henceforth we consider Circuit ORAM's stash to be part of the root. Let $\ell^* := \log_2 \widehat{m}$ be a height of the ORAM tree containing $\widehat{m}$ buckets. We can thereby define $\widehat{m}$ disjoint subtrees numbered $1, 2, \ldots, \widehat{m}$ respectively. For any t, after serving the t-th batch of requests, consider the set of blocks that currently reside in any height smaller than ℓ^*: if any block in this set can legally reside in a subtree numbered $k \in [i,j]$, it holds that this block currently must be in a bucket that is an ancestor of the k-th node at height ℓ^*.

Let Ω_k be the set of buckets that are ancestors of the k-th node at height ℓ^*. Let $\Omega_{i:j} := \cup_{k\in[i,j]}\Omega_k$. Note that all buckets have capacity of $Z = O(1)$, except the root bucket which has a capacity of R if we want the scheme's failure probability to be $O(e^{-R})$ — see Fact 3. It is not hard to see that the capacity of all buckets in $\Omega_{i:j}$ is upper bounded by

$$2Z(j-i+1) + 2\log N + R.$$

□

Corollary 1 (Utilization of a group stash)**.** *For any request sequence $\vec{S}$, any t, for any group stash corresponding to subtrees i to $j := i + \alpha \log N - 1$ at the end of serving the t-th batch of requests in Circuit OPRAM, it holds that*

$$\Pr_{\psi}\left[\mathsf{OPRAMGrpStash}^{t,i,j}(\psi,\vec{S}) > (2Z+2)\alpha\log N\right] \leq e^{-\Omega(\alpha\log N)}$$

Proof. The proof is straightforward due to Fact 4, Lemma 2, and Lemma 3. □

6 Details of the Fetch Phase

An overview of the fetch phase was presented in Section 4.3. In this section, we present the details — specifically, we shall focus on how to efficiently route fetched position identifiers from one recursion level to the next, and the offline preparation necessary to enable such efficient online routing. All other steps of the fetch phase (see Section 4.3), including conflict resolution, parallel read and gather are straightforward.

Notation. Let $D := \log_2 N$ be the total number of recursion levels, and we assume that each block is large enough to store the position identifiers of two block addresses. For ease of exposition, we assume that in each level of recursion, a block holds exactly two position identifiers, i.e., the fan-out is two. Our solution easily generalizes to fan-out values up to $O(\log N)^{O(1)}$. For large fan-out values such as N^ϵ, the number of recursion levels is $O(\frac{1}{\epsilon})$ and we can just perform operations level-by-level sequentially.

The highest level D of the recursion refers to the original N logical addresses. In general, for $0 \leq d < D$, level d of the recursion stores 2^d blocks with addresses in $\{0,1\}^d$. If a block has virtual address addr at level d, we sometimes write $\mathsf{addr}^{\langle d \rangle}$ to emphasize its level; if addr is a bit string of length longer than d, then $\mathsf{addr}^{\langle d \rangle}$ means that we truncate the string to include the d most significant bits. Each block in a pos-OPRAM level d contains the position identifiers (i.e., leaf identifiers) for two consecutive blocks in the level $d+1$. Specifically, the block $\mathsf{addr} \in \{0,1\}^d$ in level d stores the leaf identifiers of blocks $(\mathsf{addr}||0)^{\langle d+1 \rangle}$ and $(\mathsf{addr}||1)^{\langle d+1 \rangle}$ in level $d+1$.

We say that $\mathsf{addr}||0$ and $\mathsf{addr}||1$ are *siblings* — two addresses are siblings if they differ only in the last bit. We also say that $\mathsf{addr}^{\langle d \rangle}$ is the *parent* of $(\mathsf{addr}||0)^{\langle d+1 \rangle}$ and $(\mathsf{addr}||1)^{\langle d+1 \rangle}$. Similarly, we say that $(\mathsf{addr}||0)^{\langle d+1 \rangle}$ and $(\mathsf{addr}||1)^{\langle d+1 \rangle}$ are the left and the right child of $\mathsf{addr}^{\langle d \rangle}$, respectively.

Fact 5. *Observe that for a recursive tree-based ORAM/OPRAM, to access a data block with logical address* $\mathsf{addr}^{\langle D \rangle}$*, at level d of the recursion we would access a logical address* $\mathsf{addr}^{\langle d \rangle}$*, i.e., the full address* $\mathsf{addr}^{\langle D \rangle}$ *truncated to the most significant d bits.*

Main technical challenge. The main technical challenge addressed in this section is how to route fetched position identifiers to the next recursion level. The routing must be oblivious: for example, each block accessed in recursion level d contains the position identifiers for two blocks (with adjacent addresses) in recursion level $d+1$. For example, the block at $\mathsf{addr}^{\langle d \rangle}$ in recursion level d must be accessed if one or both of the blocks $(\mathsf{addr}||0)^{\langle d+1 \rangle}$ and $(\mathsf{addr}||1)^{\langle d+1 \rangle}$ need to be accessed in level $d+1$. When we route position identifiers from recursion level d to $d+1$, we must not reveal information such as whether one or both position identifiers are needed in the next level.

Routing position identifiers to the next recursion level can be trivially achieved through oblivious routing (which in turn relies on a constant number of oblivious sorts), but this naïve approach takes $\Theta(\log m)$ parallel runtime and is too expensive for our purpose. Our goal is to achieve such level-to-level routing in $O(\log \log N)$ *online* parallel time, with appropriate offline preparation. In this way, across all levels of recursion, we can ensure that the OPRAM's parallel runtime is bounded by $O(\log N \log \log N)$.

6.1 Basic Building Blocks Related to Permutations

To describe our ideas, we need a few basic building blocks as depicted in Figures 2, 3, and 4.

1. **Apply a pre-determined permutation to an array.** Figure 2 shows how to in parallel apply a pre-determined permutation to an array in a single parallel step.

Apply Permutation to Array

Inputs:

- An initial array $A[0..k)$ of size k.
- A permutation $\pi : [k] \to [k]$ written down as an array.

Outputs: A new array $\widehat{A}[0..k)$, where for each $i \in [k]$, $\widehat{A}[i] = A[\pi^{-1}(i)]$; in other words, each element $A[i]$ is copied to $\widehat{A}[\pi(i)]$.

Algorithm: Parallel runtime $O(1)$, total work $O(k)$

- Each CPU with index i reads $A[i]$ and $\pi(i)$, and then set $\widehat{A}[\pi(i)] := A[i]$.

Figure 2: Applying a pre-determined permutation to an array

Oblivious Random Permutation

Input: An array A of size k.

Outputs: An array $\widehat{A}$ which is obtained by applying a uniformly random permutation on array A in an oblivious manner.

Algorithm: Expected parallel runtime $O(\log k)$, expected total work $O(k \log k)$.
(We remark that we can also prove that with probability at least $1 - \frac{1}{N^\alpha}$, the parallel runtime is $O(\alpha \log k)$ and the total work is $O(\alpha k \log k)$.)

1. Each CPU with index $i \in [k]$ copies $A[i]$ and samples $3 \log_2 N$ bits which form a key k_i in $[3N^3]$.
2. The k CPUs perform oblivious sorting on the data tuple $(k_i, A[i])$ using k_i's to determine the total order.
3. Each CPU can check whether there are any duplicated k_i's with CPUs having neighboring indices. The process is repeated for each subarray of duplicated keys, until there are no more duplicated keys.
4. When there is no more duplicated keys, each CPU with index i copies the second term of its data tuple to $\widehat{A}[i]$.

Figure 3: Randomly permuting an array obliviously, without revealing the secret permutation.

Construct a Permutation that Maps a Source to a Destination

Inputs:

- A source array snd of length k containing distinct real elements and dummies;
- A destination array rcv also of length k containing distinct real elements and dummies;

Assume:

- The real elements in snd are guaranteed to be the same as the real elements in rcv.
- The dummy elements in snd each has a unique label that decides their relative order; similarly, the dummies in rcv each has a unique label too.

Outputs: The routing permutation π such that $\mathsf{rcv}[\pi(i)] = \mathsf{snd}[i]$ for all $i \in [k]$.

Algorithm: Parallel runtime $O(\log k)$, total work $O(k \log k)$

1. Tag each element in both arrays with an index within the array. More specifically, in parallel, write down the following two arrays:

$$\big[(1, \mathsf{snd}[1]), (2, \mathsf{snd}[2]), \ldots, (k, \mathsf{snd}[k])\big], \quad \big[(1, \mathsf{rcv}[1]), (2, \mathsf{rcv}[2]), \ldots, (k, \mathsf{rcv}[k])\big]$$

2. Obliviously sort each of the above two arrays by the second value. Suppose the two output arrays are the following where "_" denotes a wildcard value that we do not care about.

$$\big[(s_1, _), (s_2, _), \ldots (s_k, _)\big], \quad \big[(t_1, _), (t_2, _), \ldots (t_k, _)\big]$$

   Now in parallel write down $\big[(s_1 \to t_1), (s_2 \to t_2), \ldots (s_k \to t_k)\big]$

3. Now sort the above output by the s_i values, and let the output be

$$\big[(1 \to t'_1), (2 \to t'_2), \ldots, (k \to t'_k)\big]$$

   The output routing permutation is defined as $\pi(i) := t'_i$.

Figure 4: Obliviously construct a routing permutation that maps a source to a destination.

2. **Permute an array by a secret random permutation.** Figure 3 shows how to generate a secret random permutation and apply it to an array obliviously, without revealing any information about the permutation.

3. **Obliviously construct a routing permutation that aligns source and destination arrays.** Figure 4 shows how to accomplish the following task: given a source array snd of length k containing distinct real elements and dummies (where each dummy element contains unique identifying information as well), and a destination array rcv also of length k containing distinct real elements and dummies, with the guarantee that the set of real elements in snd are the same as the set of real elements in rcv. Now, construct a routing permutation $\pi : [k] \to [k]$ such that for all $i \in [k]$, if $\mathsf{snd}[i]$ contains a real element, then $\mathsf{rcv}[\pi[i]] = \mathsf{snd}[i]$.

6.2 Oblivious Compression (Dummy Removal)

Recall that in each level, $2m$ fetch processors each fetch two (possibly dummy) next-level position identifiers. Among the fetched $4m$ (possibly dummy) position identifiers, at most m are needed by the next recursion level. For obliviousness, the level-to-level routing must hide whether one or

Oblivious Compression

Inputs: A initial array of $2m$ randomly permuted entries, where each entry contains two items (each of which can be real or dummy), such that the total number of real items is at most m.

Outputs: A array of $2m$ items that is a subset of the $4m$ items in the original array such that all the real items in the initial array are preserved.

Algorithm: Parallel runtime $O(\log\log N)$, total work $O(m \log\log N)$

1. Fix some $\omega(1) \leq \alpha \leq \log N$. Partition the entries in the initial array such that each segment of $\frac{\alpha \log N}{2}$ consecutive entries forms a group. Since each entry contains two items, each group contains $\alpha \log N$ items.
2. For each group of items, perform (in parallel) oblivious sorting where real items have priorities over dummy items. After oblivious sorting, in each group, the lower priority half of the items are removed.

   Note that here we assume that $4m$ is a multiple of $\alpha \log N$. If not, we can simply pad the array with dummy entries up to the nearest multiple of $\alpha \log N$.
3. Concatenate the remaining elements of all groups to form an array of size $2m$.

Figure 5: Oblivious compression (dummy removal) algorithm with small (online) parallel runtime

two position identifiers from a block in a level are needed in the next level. One naïve approach is to forward all $4m$ position identifiers to the next level — but then the next level must have $4m$ fetch CPUs receiving them, and the next level $8m$, and so on. Clearly this approach will result in exponential blowup in the number of recursion levels.

Our idea is to obliviously compress the $4m$ position identifiers back to $2m$, suppressing dummies in the process. A naïve way to perform the compression is through oblivious sorting which would take $\Theta(\log m)$ parallel runtime per recursion level, and $O(\log m \log N)$ parallel runtime over all recursion levels — clearly this would be too expensive for our purpose.

Localized work. Instead of performing oblivious sorting on the entire $\Theta(m)$-sized array, our idea is to use a random permutation to partition the array into groups each with size $\alpha \log N$, for some $\omega(1) \leq \alpha \leq \log N$. If at most $\frac{1}{4}$ fraction of the items are real in the original array, it follows that with all but negligible in N probability, every group has at most $\frac{1}{2}$ fraction of real items. Hence, we can perform oblivious sorting on each group (in parallel) to remove $\frac{1}{2}$ fraction of items in the group while keeping all real items. The parallel runtime is only $O(\log\log N)$. The details are given in Figure 5, where we assume that a random permutation is given, which we shall see is generated (in parallel) in an offline phase.

The reason that no real elements are removed is due to the following lemma.

Lemma 4. *Let* arr *denote an array of* $2m$ *randomly permuted pairs, each of which contains two items such that out of the* $4m$ *items, at most* m *are real and the rest are dummy.*

Without loss of generality assume that $4m$ *is a multiple of* $\alpha \log N$ *(if not, we can pad the array to the nearest multiple incurring only* $O(1)$ *multiplicative overhead). Then, with probability at least* $1 - \frac{1}{N^{\Theta(\alpha)}}$*, a consecutive group of* $\frac{\alpha \log N}{2}$ *pairs contains at most* $\frac{\alpha \log N}{2}$ *real items.*

Proof. We use the Hoeffding Inequality for sampling without replacement [29]. Consider $2m$ integers $a_1, a_2, \ldots, a_{2m}$, where the integer a_i denotes the number of real items contained in the ith pair in the array. Hence, we have $\sum_{i=1}^{2m} a_i \leq m$.

Since the pairs are randomly permuted in the initial array, each group of consecutive $\frac{\alpha \log N}{2}$ pairs can be viewed as a sample of size $n := \frac{\alpha \log N}{2}$ without replacement from the $2m$ integers. Let Z be the sum of the integers in the size-n sample. Then, $E[Z] = \frac{n}{2m} \sum_{i=1}^{2m} a_i \leq \frac{\alpha \log N}{4}$.

Hence, the Hoeffding inequality for sampling without replacement [29] gives

$\Pr[Z > \frac{\alpha \log N}{2}] \leq \Pr[Z - E[Z] > \frac{\alpha \log N}{4}] \leq \exp\{-\frac{(\frac{\alpha}{4} \cdot \log N)^2}{n \cdot 2^2}\} = N^{-\frac{\alpha}{32}}$, as required. □

6.3 Offline Phase: Conflict Resolution and Preparation

Recall that at each level $d = 0, 1, \ldots, D$, the level-d OPRAM needs the position identifiers of its addresses to carry out its fetch phase. To make the online routing of position identifiers efficient, we must perform appropriate offline pre-processing to prepare auxiliary data that is used in the online routing phase.

In the offline preparation phase, the following data structures are prepared in parallel for each recursion level d:

1. An incomplete instruction array of size $2m$ denoted

$$\mathsf{Instr}^{\langle d \rangle} := \left\{ (\mathsf{addr}^{\langle d \rangle}, \mathsf{flag}, _) \mid \mathsf{dummy} \right\}_{2m}$$

   where flag is a 2-bit indicator denoting whether each of the level-$(d+1)$ addresses $\mathsf{addr}||0^{\langle d+1 \rangle}$ and $\mathsf{addr}||1^{\langle d+1 \rangle}$ are needed at the next level; and $_$ denotes a blank space to later receive a pair $(\mathsf{pos}, \mathsf{pos}')$, denoting the current and the new position identifiers respectively for the level-d address $\mathsf{addr}^{\langle d \rangle}$.

2. A routing permutation $\pi^{(d \to d+1)} : [2m] \to [2m]$ that is applied later in the online phase, to route fetched position identifiers from level d to $d+1$, and thus completing the instruction array.

Prepare the incomplete instruction array. At each recursion level d, an incomplete instruction array $\mathsf{Instr}^{\langle d \rangle}$ of size $2m$ will be constructed *obliviously* to later receive position identifiers from the previous level.

Let $\{\mathsf{addr}_i^{\langle D \rangle}\}_{i \in [m]}$ the batch of m memory requests corresponding to the top recursion level D. At each recursion level d in parallel, the following steps are performed.

1. For each $i \in [m]$, assign one CPU to each address $\mathsf{addr}_i^{\langle D \rangle}$. If the address is dummy, the CPU simply writes down dummy; otherwise, the CPU writes down the truncated address $\mathsf{addr}_i^{\langle d+1 \rangle}$, which is the full address truncated to the $d+1$ most significant bits. The usual oblivious conflict resolution is applied to the truncated addresses to suppress duplication of the truncated addresses using oblivious sort. Duplicates after truncation are replaced by dummies after oblivious sorting on the truncated addresses.

2. By comparing neighboring entries, we can further remove the least significant bit to produce an address $\mathsf{addr}^{\langle d \rangle}$ of d bits together with a flag of 2 bits indicating whether each of $(\mathsf{addr}||0)^{\langle d+1 \rangle}$ and $(\mathsf{addr}||1)^{\langle d+1 \rangle}$ is present in the previous step. (For a non-dummy entry, at least one of them must be present.) Hence, each CPU either contains a tuple of the form $(\mathsf{addr}^{\langle d \rangle}, \mathsf{flag}, _)$ or a dummy such that there is no duplicate address in level d.

3. In parallel, append m dummy entries to form an array of size $2m$.

4. In parallel, tag all dummy elements in the array with a unique label, e.g., representing the index of the element within the array. The unique label is needed later to decide the relative order of the dummy elements during the construction of the routing permutation (Figure 4).

5. The oblivious random permutation algorithm in Figure 3 is applied to the resulting array, to produce an incomplete instruction array $\mathsf{Instr}^{\langle d \rangle}$.

Prepare the routing permutation. For each recursion level $0 \leq d < D$, a permutation $\pi^{(d \to d+1)} : [2m] \to [2m]$ to route data from level d to level $d+1$ is constructed as follows. At this point, both Instr_d and Instr_{d+1} have been constructed, and hence, this step can be performed in parallel across all levels. The following steps emulate the data flow in the online phase but leaving the fetched position identifiers blank.

1. Starting from $\mathsf{Instr}^{\langle d \rangle}$ of size $2m$, construct an array $\overline{\mathsf{Fetched}}^{\langle d \rangle}$ of $2m$ pairs. Speficially, each entry of the $\mathsf{Instr}^{\langle d \rangle}$ array is converted to a pair.
   - If the original entry is dummy, then a pair of dummies will be produced.
   - Otherwise, the original entry is real and has the form $(\mathsf{addr}^{\langle d \rangle}, \mathsf{flag}, _)$, a pair of the form

   $$\Big((\mathsf{addr}||0)^{\langle d+1 \rangle}, _\Big) \mid \mathsf{dummy}, \quad \Big((\mathsf{addr}||1)^{\langle d+1 \rangle}, _\Big) \mid \mathsf{dummy}$$

     is produced, where the two bits in flag are used to determine whether each of the two items will be dummy.
   - We assume that whenever a dummy is written down, it is tagged with a unique label, e.g., representing its index within the array. This will be used to decide the relative order between dummies during the construction of the routing permutation.
2. Note that the entries in Instr_d have already been randomly permuted. Hence, oblivious compression (dummy removal) in Figure 5 is applied to $\overline{\mathsf{Fetched}}_d$ to produce an array $\overline{\mathsf{Send}}_d$ of size $2m$ of the form

   $$\overline{\mathsf{Send}}^{\langle d \rangle} := \Big\{(\mathsf{addr}^{\langle d+1 \rangle}, _) \mid \mathsf{dummy}\Big\}_{2m}$$

   where each entry is either dummy or contains an address at level $d+1$. The order of addresses in this array will agree with the online version of this variable $\mathsf{Send}^{\langle d \rangle}$ where the place-holders denoted "_" in $\overline{\mathsf{Send}}^{\langle d \rangle}$ will be filled in with position identifiers.
3. Finally, the routing permutation $\pi^{(d \to d+1)}$ is produced such that when applying $\pi^{(d \to d+1)}$ to $\overline{\mathsf{Send}}^{\langle d \rangle}$, the order of the addresses will be aligned with $\mathsf{Instr}^{\langle d+1 \rangle}$. This routing construction can be constructed using the algorithm in Figure 4 in $O(m \log m)$ total work and $O(\log m)$ parallel time.

Performance analysis. For block sizes where $B = \Omega(\log N)$, there are in total $O(\log N)$ levels of recursion. Since preparation can be done in parallel across all recursion levels, the parallel runtime is $O(\log m)$, dominated by oblivious sorting. The total work is $(m \log m \log N)$ over all recursion levels.

6.4 Online Phase: Routing Position Identifiers to the Next Recursion Level

We now describe the online phase and how fetched position identifiers are routed from one recursion level to the next. We highlight how the work performed in the offline phase can help ensure small parallel runtime in the online phase.

1. *Fetch for* $d = 0$. The smallest recursion level $d = 0$ has $O(1)$ number of blocks. Therefore, all CPUs simply fetch all blocks in level $d = 0$ in a single parallel step.

2. *Fetch for $d > 0$.* For every other recursion level $d > 0$, assume that we start with a completed instruction array $\mathsf{CInstr}^{\langle d \rangle}$ of length $2m$:

$$\mathsf{CInstr}^{\langle d \rangle} := \left\{ (\mathsf{addr}^{\langle d \rangle}, \mathsf{flag},\ \mathsf{pos}^{\langle d \rangle}, \widetilde{\mathsf{pos}}^{\langle d \rangle}) \mid \mathsf{dummy} \right\}_{2m}$$

where $\mathsf{addr}^{\langle d \rangle}$ is the level-d address to read, flag is a 2-bit indicator representing whether each of the two fetched addresses contained in a block is needed at the next recursion level $d+1$, $\mathsf{pos}^{\langle d \rangle}$ is the current position identifier (i.e., which path should be read), and $\widetilde{\mathsf{pos}}^{\langle d \rangle}$ is the new position identifier to assign to the block after it is fetched. Notice that the new position identifier $\widetilde{\mathsf{pos}}^{\langle d \rangle}$ is chosen at random by the previous recursion level.

Now, $2m$ fetch CPUs each read one (possibly dummy) instruction. Next, each fetch CPU recruits $O(\alpha \log N)$ auxiliary CPUs to read the path specified by $\mathsf{pos}^{\langle d \rangle}$ (including the group stash) in a single parallel step. The requested block (along with metadata that is needed later in the maintain phase such as the physical location where the block is found) is aggregated to the fetch CPU through an oblivious select algorithm (see Section 3.3) in $O(\log \log N)$ time and $O(\alpha \log N)$ total work. At this moment, the fetched block is not removed from the path yet — this will be done in the offline maintain phase; moreover, after the block is removed later, its position identifier should be updated to $\widetilde{\mathsf{pos}}^{\langle d \rangle}$. If the instruction is dummy, the fetch CPU simply chooses a path at random to read.

3. *Construct fetched array.* Unless $d = D$ (i.e., this is the final data-OPRAM), at this moment, each of the $2m$ fetch CPUs has read either two position identifiers $(\mathsf{pos}_0^{\langle d+1 \rangle}, \mathsf{pos}_1^{\langle d+1 \rangle})$ or dummy. Now each fetch CPU will write down a pair of entries as follows:

   - If the fetch CPU has read dummy, it writes down a pair $(\mathsf{dummy}, \mathsf{dummy})$.
   - Else, the fetch CPU looks at the flag entry in its instruction to see whether the two fetched position identifiers are needed in the next recursion level. Let $\mathsf{addr}^{\langle d \rangle}$ be the level-d address assigned to the fetch CPU. For $b \in \{0, 1\}$, if $(\mathsf{addr}||b)^{\langle d+1 \rangle}$ is needed in the next recursion level, the fetch CPU writes down $((\mathsf{addr}||b)^{\langle d+1 \rangle}, \mathsf{pos}_b^{\langle d+1 \rangle})$; observe that in this case, a fresh random position identifier $\widetilde{\mathsf{pos}}_b^{\langle d+1 \rangle}$ has to be generated, which is used to update the block in level d fetched by the CPU. If $(\mathsf{addr}||b)^{\langle d+1 \rangle}$ is not needed, the fetch CPU simply writes down dummy.

   The result of this step is a $\mathsf{Fetched}^{\langle d \rangle}$ array of size $4m$ of the form:

$$\mathsf{Fetched}^{\langle d \rangle} := \left\{ (\mathsf{addr}^{\langle d+1 \rangle}, \mathsf{pos}^{\langle d+1 \rangle}) \mid \mathsf{dummy} \right\}_{4m}$$

4. *Compress.* Next, we apply the oblivious compression procedure described in Figure 5 to compress the $\mathsf{Fetched}^{\langle d \rangle}$ array down to $2m$ entries, suppressing a subset of the dummies in the process. For every non-dummy entry in the resulting array, we also append the fresh random position identifier generated earlier to the entry (note that later the maintain phase will overwrite the block back with these new position identifiers).

   The result of this step is a send array denoted $\mathsf{Send}^{\langle d \rangle}$ of the form:

$$\mathsf{Send}^{\langle d \rangle} := \left\{ (\mathsf{addr}^{\langle d+1 \rangle}, \mathsf{pos}^{\langle d+1 \rangle}, \widetilde{\mathsf{pos}}^{\langle d+1 \rangle},) \mid \mathsf{dummy} \right\}_{2m}$$

5. *Route to next level.* Apply the routing permutation $\pi^{(d \to d+1)}$ to the send array $\mathsf{Send}^{\langle d \rangle}$ in a single parallel step. At this moment, the addresses in $\mathsf{Send}^{\langle d \rangle}$ are aligned with the addresses in the next level's incomplete instruction array $\mathsf{Instr}^{\langle d+1 \rangle}$.

   In parallel, perform a coordinate-wise copy operation such that the next level's instruction array $\mathsf{Instr}^{\langle d+1 \rangle}$ is now completed, resulting in $\mathsf{CInstr}^{\langle d+1 \rangle}$. The next recursion level is now ready to be executed.

6. *Oblivious multicast.* If $d = D$, i.e., this is the final data-OPRAM, we perform a standard oblivious multicast procedure to route the fetched data blocks to the request CPUs. This can be attained through a standard oblivious routing algorithm (see Section 3.3) in $O(\log m)$ parallel time and $O(m \log m)$ total work.

Performance analysis. For general block sizes where $B = \Omega(\log N)$, there are in total $O(\log N)$ levels of recursion. Therefore, it is not hard to see that the entire fetch phase can be completed in $O(\log N \log \log N + \log m)$ parallel time across all recursion levels, and with $O(\alpha m \log^2 N + m \log m)$ total work across all recursion levels.

6.5 Performance of the Fetch Phase

Accounting for the cost of all the steps in the fetch phase, we can easily derive the following lemma for the performance of the fetch phase.

Lemma 5 (Performance of the fetch phase)**.** *Suppose that the block size $B = \Omega(\log N)$. Then, to serve the batch of m requests, the fetch phase incurs a parallel runtime of $O(\log m + \log N \log \log N)$, and a total work of $O(\alpha m \log^2 N + m \log m \log N)$. Observe that the performance of the fetch phase does not depend on the number $\widehat{m}$ of subtrees kept by the OPRAM, even when $\widehat{m}$ is much larger than m.*

7 Performance and Security of Circuit OPRAM

7.1 Performance

Earlier in Sections 5 and 6, we analyzed the performance of the maintain and fetch phases respectively for simulating each PRAM step. The following lemma describes the performance of both phases.

Lemma 6 (Performance for simulating a single PRAM step)**.** *Suppose that the block size $B = \Omega(\log N)$, and at the beginning of serving a batch of m requests, the OPRAM has $\widehat{m}$ subtrees. Then, our OPRAM takes $O(\log N \log \log N)$ parallel runtime and $O(\alpha(m + \widehat{m}) \log^2 N)$ total work to serve this batch of requests.*

Proof. Straightforward by Lemmas 1 and 5. □

Amortized work for varying m. Recall that the number m of CPUs in each PRAM step can be different. In Section 5, we see that the OPRAM internally maintains $\widehat{m}$ subtrees and $\widehat{m}$ is meant to track m. If m increases, $\widehat{m}$ immediately increases to m. However, when m decreases suddenly, $\widehat{m}$ lazily tracks m each time decreasing only by a factor of 2. In Lemma 6, we see that the total work has a dependence on $\widehat{m}$. However, since $\widehat{m}$ decreases geometrically at each simulation step, we can consider amortized analysis and charge the part of the total work depending on $\widehat{m}$ to the step before m decreases. This gives the following amortized result.

Theorem 3 (Performance of Circuit OPRAM for general block sizes)**.** *For a general block size $B = \Omega(\log N)$, Circuit OPRAM has an amortized total work blowup of $O(\alpha \log^2 N)$, and a parallel runtime blowup of $O(\log N \log\log N)$.*

7.2 Security

Theorem 4 (Security of Circuit OPRAM)**.** *Circuit OPRAM satisfies obliviousness as formulated in Section 3.2.*

Proof. First, observe that the entire maintain phase has deterministic, a-priori known memory access patterns. Next, observe that for the fetch phase, the offline preparation stage also has deterministic, a-priori known memory access patterns. (In particular, any rearranging of sensitive data is handled by oblivious sort, which has a deterministic access pattern.)

The only part of the OPRAM whose access patterns are not a-priori known is the online stage of the fetch phase. Here, it is not hard to see that a simulator that does not know the inputs to the OPRAM can simulate the access patterns as follows:

- For simulating all fetch paths, the simulator can simply pick a path at random;
- For simulating the online routing permutation, the simulator simply picks a random permutation.

It is not hard to see that conditioned on the fact that no bad event (e.g., stash overflow) happens, the simulated access patterns are identically distributed as the real access patterns. More specifically, the argument for simulating the fetch paths is the same as the security argument for all known tree-based ORAMs [30,33,34] and tree-based OPRAMs [3,6] — whenever a block was last accessed, the OPRAM algorithm chooses a random path for the block without revealing the position identifier. Therefore, when the block is accessed next, an independent random position identifier is revealed to the adversary. For simulating the online routing permutation, observe that each $\mathsf{Instr}^{\langle d+1 \rangle}$ is permuted by an independent secret random permutation, and therefore regardless of what the $\mathsf{Send}^{\langle d \rangle}$ is, the routing permutation that maps $\mathsf{Send}^{\langle d \rangle}$ to $\mathsf{Instr}^{\langle d+1 \rangle}$ is an independent random permutation. □

8 Extensions and Optimizations

8.1 Results for Large Block Sizes

Table 2 provides Circuit OPRAM's asymptotical performance under various parametrizations.

Larger block sizes. When Circuit OPRAM's block size is $\Omega(\log^{1+\epsilon} N)$ for an arbitrarily small constant $\epsilon > 0$, clearly in this case the depth of recursion becomes $O(\log N / \log\log N)$ rather than $O(\log N)$. Therefore, we have that the total work blowup is $O(\log^2 N / \log\log N)$, and the parallel runtime blowup is $O(\log N)$ — see Table 2.

When the block size is $\Omega(N^\epsilon)$ for an arbitrarily small constant $\epsilon > 0$, the depth of recursion becomes $O(1)$ and so the fetch phase across different recursion levels can just be executed sequentially. In Appendix A, we describe a technique that parallelizes Circuit ORAM's eviction algorithm, such that an eviction can be performed in $O(\log\log N)$ parallel time, but with a slight $O(\log\log N)$ factor increase in total work. In this way, we have the following corollary for sufficiently large block sizes.

Corollary 2 (Result for sufficiently large blocks)**.** *Let $\epsilon > 0$ be any positive constant. For any sufficiently small $\alpha = \omega(1)$,*

Block size	**Total work blowup**	**Parallel runtime blowup**
Any*	$O(\alpha \log^2 N)$	$O(\log N \log\log N)$
Any*	$O(\log^{2+\epsilon} N)$	$O(\log N)$
$\Omega(\log^{1+\epsilon} N)$	$O(\alpha \log^2 N / \log\log N)$	$O(\log N)$
$\Omega(N^\epsilon)$	$O(\alpha \log N)$	$O(\log N)$
$\Omega(N^\epsilon)$	$O(\log N \log\log N)$	$O(\log m + \log\log N)$

Table 2: Circuit OPRAM performance under different block sizes. Here we assume that the OPRAM adopts consistent block size both for the pos-OPRAMs and the data-OPRAM. Throughout this table, ϵ denotes an arbitrarily small positive constant. We also assume non-trivial $m \leq N$.
*: As in all prior ORAM/OPRAM works, we by default assume that a block is at least $\Omega(\log N)$ bits long such that the block can at least store its own address.

- *Circuit ORAM (under appropriate parameterizations) achieves $O(\alpha \log N)$ total work blowup and $O(\log N)$ parallel runtime blowup for block size $B = \Omega(N^\epsilon)$.*
- *There exists a Circuit ORAM variant (denoted Circuit ORAM*) that achieves $O(\alpha \log N \log\log N)$ total work blowup and $O(\log m + \log\log N)$ parallel runtime blowup for block size $B = \Omega(N^\epsilon)$.*

Proof. The former result is straightforward by pointing out that the depth of recursion is $O(1)$. The result for Circuit ORAM* relies on a further optimization that parallelizes the eviction algorithm as described below and in Appendix A. □

We remark that Chan et al. [5] show a $\Omega(\log m)$ lower bound for the parallel runtime blowup of OPRAM; further, Goldreich and Ostrovsky's $\Omega(\log N)$ lower bound [16, 17] directly applies to the total work of OPRAM. In this sense, Circuit OPRAM is tight in total work for large block sizes; further, Circuit OPRAM* has tight parallel runtime for large blocks and when $m = \Omega(\log N)$.

Parallelizing the path eviction algorithm. For our result on general block sizes, the depth of recursion dominates the parallel runtime. Therefore, we can simply run Circuit ORAM's path eviction algorithm sequentially which completes in $O(L)$ parallel time for a path length of $O(L)$. For blocks larger than N^ϵ bits, the recursion depth becomes $O(1)$. In this case, path eviction's runtime will dominate if we naïvely run the sequential version.

Therefore we propose a parallel variant of Circuit ORAM's eviction algorithm. We give an overview of the intuition here but defer the details of the parallel eviction algorithm to Appendix A. Recall that Circuit ORAM's path eviction algorithm involves two metadata scans (the `PrepareDeepest` and `PrepareTarget` algorithms in Circuit ORAM [34]) followed by a single data scan (the `EvictFast` algorithm in Circuit ORAM [34]).

To parallelize these algorithms, our idea is to rely on the *divide-and-conquer* paradigm. More specifically, to solve a problem instance of length n, we divide the problem into two sub-problems each of length $\frac{n}{2}$; and we solve the sub-problems in parallel. Then, in $O(1)$ parallel steps and $O(n)$ time, we reconstruct the solution to the length-n problem from the solutions of the two length-$\frac{n}{2}$ sub-problems. Finally, this divide-and-conquer strategy is applied recursively.

In Appendix A, we show that all three algorithms `PrepareDeepest`, `PrepareTarget`, and `EvictFast` of Circuit ORAM's path eviction are amenable to such a divide-and-conquer paradigm. It is therefore not difficult to calculate the total work and parallel runtime for the parallel version, by solving the following recurrences, where $T(n)$ denotes the parallel runtime for a problem instance

of length n, and $W(n)$ denotes the total work for a problem instance of length n:

$$T(n) = T(\frac{n}{2}) + O(1), \qquad W(n) = 2W(\frac{n}{2}) + O(n)$$

We henceforth derive the following lemma.

Lemma 7 (Parallelizing the path eviction)**.** *Circuit ORAM's path eviction algorithm can be parallelized: for a path length of $O(\log N)$, the parallel variant completes in $O(\log \log N)$ parallel runtime with $O(\log N \log \log N)$ total work.*

8.2 Additional Extensions

Trading off total work and parallel runtime. We describe a simple parametrization trick to reduce Circuit OPRAM's parallel runtime from $O(\log N \log \log N)$ to simply $O(\log N)$, with only a slight increase in total work. The idea is to use block bundles for the position map levels. In particular, let $\epsilon > 0$ be an arbitrarily small constant. We group $\log^{\epsilon} N$ blocks into a bundle for the position map levels. Of course, to read a block bundle would now require $O(\log^{\epsilon} N)$ total work, but can be trivially parallelized by having $O(\log^{\epsilon} N)$ auxiliary CPUs. This block bundle trick allows us to reduce the depth of the recursion by a $\Theta(\log \log N)$ factor. In particular, with each step of the recursion, the total number of block bundles reduce by a factor of $\Theta(\log^{\epsilon} N)$, and therefore the depth of the recursion is now $O(\log N / \log \log N)$. The saving in parallel runtime comes at an $O(\log^{\epsilon} N)$ increase in total work.

It is not hard to observe that with this reparametrization trick, we will have an OPRAM scheme with $O(\log^{2+\epsilon} N)$ total work blowup and $O(\log N)$ depth, where $\epsilon > 0$ is an arbitrarily small constant.

Non-uniform block sizes and bandwidth blowup. So far, we focused on two metrics, total work blowup and parallel runtime blowup. These metrics are the most general that we know of for OPRAM. In certain application settings, such as cloud storage outsourcing, we may care about other metrics, for example, bandwidth blowup. We define *bandwidth blowup* as the ratio of the total number of bits transferred between the CPU (i.e., client) and memory (i.e., cloud server) in the OPRAM case vs. the PRAM case. Under uniform block sizes, the total work metric immediately implies the bandwidth blowup.

However, if the OPRAM is allowed to adopt different block sizes for the data-OPRAM and the pos-OPRAMs, then as in earlier tree-based ORAM/OPRAM works [30, 33, 34], we can use the "big data block, little metadata block" trick to reduce the bandwidth blowup to $O(\alpha \log N)$ for data blocks of only $\Omega(\log^2 N)$ bits long. The idea is to let data blocks be $\Omega(\log^2 N)$ bits long, but let pos-OPRAM blocks to be only $\Theta(\log N)$ bits long — in this way, fetching $O(\log N)$ pos-OPRAM blocks costs the same (or less) bandwidth than fetching a data block.

Acknowledgments

We are extremely grateful to Rafael Pass without whose insights and support this work would not have been possible.

Additionally, we are grateful to Joshua Gancher for helpful discussions in an earlier phase of the project, and to Kai-Min Chung for many supportive conversations. We thank the beanbag in the systems lab that played a crucial role in the initial phase of the project, as well as the beautiful Beebe lake and Watkins Glen state park in the summer. This work is supported in part by NSF grants CNS-1314857, CNS-1514261, CNS-1544613, CNS-1561209, CNS-1601879, CNS-1617676, an Office of Naval Research Young Investigator Program Award, a Packard Fellowship, a Sloan Fellowship, Google Faculty Research Awards, and a VMWare Research Award.

References

[1] M. Ajtai, J. Komlós, and E. Szemerédi. An 0(n log n) sorting network. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, STOC '83, pages 1–9, New York, NY, USA, 1983. ACM.

[2] Daniel Apon, Jonathan Katz, Elaine Shi, and Aishwarya Thiruvengadam. Verifiable oblivious storage. In *Public Key Cryptography*, pages 131–148, 2014.

[3] Elette Boyle, Kai-Min Chung, and Rafael Pass. Oblivious parallel RAM and applications. In *Theory of Cryptography - 13th International Conference, TCC 2016-A, Tel Aviv, Israel, January 10-13, 2016, Proceedings, Part II*, pages 175–204, 2016.

[4] Elette Boyle and Moni Naor. Is there an oblivious RAM lower bound? In *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science, Cambridge, MA, USA, January 14-16, 2016*, pages 357–368, 2016.

[5] T-H. Hubert Chan, Kai-Min Chung, Wei-Kai Lin, Feng-Hao Liu, and Elaine Shi. New lower bounds for oblivious simulation of PRAMs. Manuscript in preparation.

[6] Binyi Chen, Huijia Lin, and Stefano Tessaro. Oblivious parallel RAM: improved efficiency and generic constructions. In *Theory of Cryptography - 13th International Conference, TCC 2016-A, Tel Aviv, Israel, January 10-13, 2016, Proceedings, Part II*, pages 205–234, 2016.

[7] Kai-Min Chung, Zhenming Liu, and Rafael Pass. Statistically-secure ORAM with $\tilde{O}(\log^2 n)$ overhead. In *Asiacrypt*, 2014.

[8] Dana Dachman-Soled, Chang Liu, Charalampos Papamanthou, Elaine Shi, and Uzi Vishkin. Oblivious network ram and leveraging parallelism to achieve obliviousness. In *Asiacrypt*, 2015.

[9] Srinivas Devadas, Marten van Dijk, Christopher W. Fletcher, Ling Ren, Elaine Shi, and Daniel Wichs. Onion ORAM: A constant bandwidth blowup oblivious RAM. In *Theory of Cryptography - 13th International Conference, TCC 2016-A, Tel Aviv, Israel, January 10-13, 2016, Proceedings, Part II*, pages 145–174, 2016.

[10] Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *STC*, 2012.

[11] Christopher W. Fletcher, Ling Ren, Albert Kwon, Marten van Dijk, and Srinivas Devadas. Freecursive ORAM: [nearly] free recursion and integrity verification for position-based oblivious RAM. In *ASPLOS*, 2015.

[12] Christopher W. Fletcher, Ling Ren, Albert Kwon, Marten van Dijk, Emil Stefanov, and Srinivas Devadas. RAW Path ORAM: A low-latency, low-area hardware ORAM controller with integrity verification. *IACR Cryptology ePrint Archive*, 2014:431, 2014.

[13] Christopher W. Fletcher, Ling Ren, Xiangyao Yu, Marten van Dijk, Omer Khan, and Srinivas Devadas. Suppressing the oblivious RAM timing channel while making information leakage and program efficiency trade-offs. In *HPCA*, pages 213–224, 2014.

[14] Craig Gentry, Kenny A. Goldman, Shai Halevi, Charanjit S. Jutla, Mariana Raykova, and Daniel Wichs. Optimizing ORAM and using it efficiently for secure computation. In *Privacy Enhancing Technologies Symposium (PETS)*, 2013.

[15] Craig Gentry, Shai Halevi, Charanjit Jutla, and Mariana Raykova. Private database access with he-over-oram architecture. Cryptology ePrint Archive, Report 2014/345, 2014. `http://eprint.iacr.org/`.

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

[17] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 1996.

[18] Michael T. Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *ICALP*, 2011.

[19] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In *CCS*, 2012.

[20] Torben Hagerup. Fast and optimal simulations between CRCW prams. In *STACS 92, 9th Annual Symposium on Theoretical Aspects of Computer Science, Cachan, France, February 13-15, 1992, Proceedings*, pages 45–56, 1992.

[21] Marcel Keller and Peter Scholl. Efficient, oblivious data structures for mpc. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology ASIACRYPT 2014*, volume 8874 of *Lecture Notes in Computer Science*, pages 506–525. Springer Berlin Heidelberg, 2014.

[22] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *SODA*, 2012.

[23] Chang Liu, Yan Huang, Elaine Shi, Jonathan Katz, and Michael Hicks. Automating Efficient RAM-model Secure Computation. In *S & P*, May 2014.

[24] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Kriste Asanovic, John Kubiatowicz, and Dawn Song. Phantom: Practical oblivious computation in a secure processor. In *CCS*, 2013.

[25] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. GraphSC: Parallel Secure Computation Made Easy. In *IEEE S & P*, 2015.

[26] Benny Pinkas and Tzachy Reinman. Oblivious RAM revisited. In *CRYPTO*, 2010.

[27] Ling Ren, Christopher W. Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten van Dijk, and Srinivas Devadas. Ring oram: Closing the gap between small and large client storage oblivious ram. Cryptology ePrint Archive, Report 2014/997, 2014. `http://eprint.iacr.org/`.

[28] Ling Ren, Xiangyao Yu, Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas. Design space exploration and optimization of path oblivious RAM in secure processors. In *ISCA*, pages 571–582, 2013.

[29] Robert J Serfling. Probability inequalities for the sum in sampling without replacement. *The Annals of Statistics*, pages 39–48, 1974.

[30] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *ASIACRYPT*, 2011.

[31] Emil Stefanov and Elaine Shi. Multi-cloud oblivious storage. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

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

[33] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM – an extremely simple oblivious ram protocol. In *CCS*, 2013.

[34] Xiao Shaun Wang, T-H. Hubert Chan, and Elaine Shi. Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound. In *ACM CCS*, 2015.

[35] Xiao Shaun Wang, Yan Huang, T-H. Hubert Chan, Abhi Shelat, and Elaine Shi. SCORAM: Oblivious RAM for Secure Computation. In *CCS*, 2014.

[36] Peter Williams and Radu Sion. Usable PIR. In *Network and Distributed System Security Symposium (NDSS)*, 2008.

[37] Peter Williams and Radu Sion. Round-optimal access privacy on outsourced storage. In *ACM Conference on Computer and Communication Security (CCS)*, 2012.

[38] Peter Williams, Radu Sion, and Bogdan Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *CCS*, pages 139–148, 2008.

A Optimizing Parallel Runtime for Path Eviction

We have so far used Circuit ORAM's eviction algorithm [34] as a blackbox. For each eviction on a path with L buckets, the procedure is sequential, and hence, both the total work and the parallel runtime is $O(L)$. To optimize the parallel runtime, we give a parallel version of the procedure that takes $O(\log L)$ parallel runtime, but with a slightly larger total work of $O(L \log L)$.

Intuition. As mentioned earlier, the high-level idea is to adopt the divide-and-conquer paradigm. To solve a problem instance of length n, we divide the problem into two sub-instances each of length $\frac{n}{2}$ and solve them in parallel. Then, in $O(1)$ parallel steps and $O(n)$ time, we reconstruct the solution to the length-n problem from the solutions of the two length-$\frac{n}{2}$ sub-problems.

Notation. We first revisit the eviction strategy used in Circuit ORAM, which is performed on a path of buckets $\mathsf{path}[0..L]$, where $\mathsf{path}[0]$ represents the (group) stash and $\mathsf{path}[L]$ is the leaf. In our case, $\mathsf{path}[1]$ corresponds to the root of one of the $\widehat{m}$ subtrees. Recall that each block has a leaf position, whose lowest common ancestor with $\mathsf{path}[L]$ gives the deepest height at which the block can legally reside on path. We assume there is a total ordering imposed on the blocks determined by the maximum depths they can legally reside in path, where ties are consistently resolved (for instance by block addresses). The (non-oblivious) procedure in Algorithm 1 illustrates how the blocks are supposed to be evicted along the path.

Metadata Scan. As in Circuit ORAM [34], we first scan the metadata to collect the relevant information for moving the blocks in path. The difference here is that we give a parallel version for scanning the metadata. Algorithm 3 is supposed to produce an array $\mathsf{target}[0..L]$, where the entry $\mathsf{target}[i] = \mathsf{dst}_i$ means that if $\mathsf{dst}_i \neq \perp$, the deepest block in $\mathsf{path}[i]$ will be moved to $\mathsf{path}[\mathsf{dst}_i]$.

Subroutine `PrepareDeepest`. A sequential version of this procedure was given in the Circuit ORAM paper [34].

Algorithm 1 `EvictSlow(path)` *A slow, non-oblivious version of the eviction algorithm, only for illustration purpose [34]*

1: $i := L$ /* *start from leaf* */
2: **while** $i \geq 1$ **do**:
3: **if** $\mathsf{path}[i]$ has empty slot **then**
4: (B, ℓ) := Deepest block in $\mathsf{path}[0..i-1]$ that can legally reside in $\mathsf{path}[i]$, and its corresponding height in path.
/* $\mathsf{B} := \perp$ *if such a block does not exist.* */
5: **if** $\mathsf{B} \neq \perp$ **then**
6: Move B from $\mathsf{path}[\ell]$ to $\mathsf{path}[i]$.
7: $i := \ell$ // *skip to height* ℓ
8: **else** $i := i - 1$

After calling `PrepareDeepest`$(\mathsf{path}[0..L])$, for $1 \leq i \leq L$, the array entry $\mathsf{deepest}[i]$ stores the source height of the deepest block in $\mathsf{path}[0..i-1]$ that can legally reside in $\mathsf{path}[i]$.

We will use the following monotone property.

Fact 6 (Monotone property of `deepest`)**.** *Suppose for some* $1 \leq i \leq L$*,* $\mathtt{deepest}[i] = j$*. Then, for* $j < k \leq i$*,* $\mathtt{deepest}[k] = j$*.*

Subroutine `PrepareTarget`**.** A sequential version was given in the Circuit ORAM paper [34] to process the metadata. We assume `PrepareDeepest`$(\mathsf{path}[0..L])$ has already been called to have $\mathsf{deepest}[0..L]$ ready. For the parallel version, `PrepareTarget`$(\mathsf{path}[i..j], \mathsf{dst})$ takes a path segment and a potential destination height $\mathsf{dst} \geq j+1$. It prepares two arrays $\mathtt{target1}[i..j]$ and $\mathtt{target2}[i..j]$ and returns a pair $(\mathsf{dst}_1, \mathsf{dst}_2)$ with the following properties.

For $i \leq k \leq j$, $\mathtt{target1}[k]$ indicates the destination height that the deepest block in $\mathsf{path}[k]$ should be moved to, assuming that no block will be moved from $\mathsf{path}[0..j]$ to $\mathsf{path}[j+1..L]$; in this case, if $\mathsf{dst}_1 \neq \perp$, then $i \leq \mathsf{dst}_1 \leq j$, and there should be a block moving from $\mathsf{path}[0..i-1]$ to $\mathsf{path}[\mathsf{dst}_1]$. Similarly, $\mathtt{target2}[k]$ indicates the corresponding destination height, assuming that if $\mathsf{dst} \neq \perp$, then $\mathsf{deepest}[j+1] = \mathsf{deepest}[\mathsf{dst}]$ and the deepest block in $\mathsf{path}[\mathsf{deepest}[j+1]]$ will be moved to $\mathsf{path}[\mathsf{dst}]$; similarly, in this case $\mathsf{dst}_2 \neq \perp$ implies that there should be a block moving from $\mathsf{path}[0..i-1]$ to $\mathsf{path}[\mathsf{dst}_2]$, where $\mathsf{dst}_2 \geq i$ and $\mathsf{deepest}[\mathsf{dst}_2] < i$.

Hence, calling `PrepareTarget`$(\mathsf{path}[0..L], \perp)$ will give the desired array $\mathsf{target}[0..L] := \mathsf{target1}[0..L]$ for moving the actual blocks.

Subroutine `EvictFast`**.** After the array $\mathsf{target1}[0..L]$ is prepared by running $\mathsf{PrepareTarget}(\mathsf{path}[0..L], \perp)$, we can run $\mathsf{EvictFast}$, which moves the blocks in parallel, as opposed to doing a linear scan from the root to the leaf. In particular, the effect of procedure $\mathsf{EvictFast}(\mathsf{path}[i..j])$ is to evict along the path starting from the ith height down to height j, following $\mathsf{target}[i..j]$; the procedure returns a pair $(\mathsf{B}, \mathsf{dst})$, where $(\mathsf{B}, \mathsf{dst})$ is any block that is supposed to be moved from $\mathsf{path}[i..j]$ to $\mathsf{path}[\mathsf{dst}]$, where $\mathsf{dst} > j$.

Hence, if suffices to call $\mathsf{EvictFast}(\mathsf{path}[0..L])$ to complete the eviction along the whole path.

Performance Analysis. All the algorithms are recursive and use a divide and conquer paradigm. Specifically, for an instance of size L, each algorithm first solves instances with sizes $\lfloor \frac{L}{2} \rfloor$ and $\lceil \frac{L}{2} \rceil$ in parallel. Then, the two solutions are combined to produce a solution to the original instance using $O(1)$ parallel runtime and and $O(L)$ total work. Hence, each algorithm takes $O(\log L)$ parallel runtime and $O(L \log L)$ total work.

Correctness. We next show that $\mathsf{EvictFast}$ (Algorithm 4) achieves the same effect on the locations

Algorithm 2 PrepareDeepest(path$[i..j]$)

*/*Make parallel metadata scan to prepare the* deepest *array.*
After this algorithm, for each $i < k \leq j$, deepest$[k]$ *stores the source height of the deepest block in* path$[i..k-1]$ *that can legally reside in* path$[k]$*, and* goal$[k]$ *stores the deepest height that the corresponding block can reside. Moreover, the procedure returns a pair* (src, dst)*, where* src *is the source height the deepest block in* path$[i..j]$ *and* dst *is the deepest height the corresponding block can reside.*
Note: *The procedure can be implemented in an oblivious way. For instance, for the case when the condition of an "If" statement is false, dummy operations can access the corresponding variables without actually modifying them. */*

1: Initialize deepest$[i..j] := (\bot, \bot, ..., \bot)$ and goal$[i..j] := (\bot, \bot, ..., \bot)$.
2: **if** $(i == j)$ **then**
3:     **if** path$[i]$ is non-empty **then**
4:         dst := Deepest height that a block in path$[i]$ can legally reside on path.
      return (i, dst)
5:     **else return** $(\bot, \bot)$ ▷ Assume $\bot$ is smaller than any integer.
▷ $i < j$
6: $p := \left\lfloor \frac{i+j}{2} \right\rfloor$
7: Execute the following two statements **in parallel**:
- $(\mathsf{src}_1, \mathsf{dst}_1) :=$ PrepareDeepest(path$[i..p]$)
- $(\mathsf{src}_2, \mathsf{dst}_2) :=$ PrepareDeepest(path$[p+1..j]$)

8: **for** $k = p + 1$ to j in parallel **do**
9:     **if** $\mathsf{dst}_1 \geq \mathsf{goal}[k]$ **then**
10:         deepest$[k] := \mathsf{src}_1$, goal$[k] := \mathsf{dst}_1$
11: **if** $\mathsf{dst}_1 \geq \mathsf{dst}_2$ **then** **return** $(\mathsf{src}_1, \mathsf{dst}_1)$
12: **else return** $(\mathsf{src}_2, \mathsf{dst}_2)$

of the blocks as EvictSlow (Algorithm 1). The intuition is that the sequential version of the path eviction consists of three linear scans along the path. Hence, using the divide-and-conquer strategy in the parallel version, it suffices to keep track of what computation each sub-instance can perform on its own, and what information needs to be passed from one sub-instance to the other in order to complete the computation.

Lemma 8 (Correctness of EvictFast)**.** EvictFast *has the same effect on the locations of the blocks in the eviction path as* EvictSlow*.*

Proof. We show that each of the algorithms PrepareDeepest, PrepareTarget and EvictFast achieve the effects in the above description. In particular, as seen from [34], in the sequential versions of these algorithms, PrepareDeepest and EvictFast each performs a root-to-leaf scan, and PrepareTarget performs a leaf-to-root scan. We explain how the solutions to the two sub-instances in each algorithm can be combined to give a solution to the larger instance.

1. PrepareDeepest. Since the effect of the algorithm can be achieved by a linear scan from the root to the leaf in path, after the sub-instances $[i..p]$ and $[p+1..j]$ have been handled in parallel, we just need to consider what information needs to be passed from the first instance to the second one to get a solution for the larger instance $[i..j]$.

   Observe that for each $k \in [p+1..j]$, deepest$[k]$ is supposed to store the source height of the deepest block from path$[i..k-1]$ that can legally reside in path$[k]$.

Algorithm 3 PrepareTarget(path$[i..j]$, dst)
/ Parallel version to prepare the target arrays. */*

```
1: target1[i..j] := target2[i..j] := (⊥, ⊥, ..., ⊥), dst_1 := dst_2 := ⊥
2: if (i == j) then
3:     if (path[i] has an empty slot) and (deepest[i] ≠ ⊥) then
4:         dst_1 := dst_2 := i
5:     if (deepest[dst] == i) then target2[i] := dst
6:         if deepest[i] ≠ ⊥ then dst_2 := i
7:     else if deepest[dst] < i then dst_2 := dst
8:     return (dst_1, dst_2)
                                                              ▷ i < j
9: p := ⌊(i+j)/2⌋
10: Execute the following two statements in parallel:
     • (dst_1, dst_2) := PrepareTarget(path[p+1..j], dst)
     • (dst'_1, dst'_2) := PrepareTarget(path[i..p], p+1)
11: if dst_2 ≠ ⊥ then                     ▷ deepest[dst_2] = deepest[p+1]
12:     for k = i to p in parallel do
13:         if deepest[p+1] == k then target2[k] := dst_2
14:     if dst_1 ≠ ⊥ then                 ▷ deepest[dst_1] = deepest[p+1]
15:         for k = i to p in parallel do
16:             if (deepest[p+1] == k) then target1[k] := dst_1
17:             else  target1[k] := target2[k]
18:         if (dst'_2 == p+1) then dst'_1 := dst_1
19:         else  dst'_1 := dst'_2
20:     if (dst'_2 == p+1) then dst'_2 := dst_2
                                                   ▷ dst_1 = dst_2 = ⊥
21: if dst_2 = ⊥ then
22:     for k = i to p in parallel do  target2[k] := target1[k]
23:     dst'_2 := dst'_1
24: return (dst'_1, dst'_2)
```

Starting from a solution for the sub-instance $[p+1..j]$, we just need to know if where the deepest block in path$[i..p]$ can reside in path$[p+1..j]$, i.e., the source level src_1 of the deepest block in path$[i..p]$ and the deepest height dst_1 that it can reside. Equipped this information, the solution for $[p+1..j]$ can be updated accordingly as shown in Algorithm 2.

2. PrepareTarget. Assuming that PrepareDeepest has already been called to construct the array deepest, Algorithm 3 performs a leaf-to-root scan. In the sequential version, we can imagine that when the scan passes from height $i+1$ to height i, the algorithm just needs to remember a destination level dst $\geq i+1$ that has an empty slot and is supposed to take a block from deepest[dst]. However, the monotone property from Fact 6 implies that deepest[dst] = deepest$[i+1]$, which means the algorithm only needs to remember dst.

   Now, suppose we break the larger instance $[i..j]$ into two smaller sub-instances $[i..p]$ and $[p+1..j]$. The difficulty is that if we try to solve the sub-instance $[i..p]$, we do not know whether a block is supposed to go from path$[i..p]$ to path$[p+1..j]$, without solving the sub-instance $[p+1..j]$ first.

Algorithm 4 `EvictFast`($\mathsf{path}[i..j]$)
/* *Evicts blocks along* $\mathsf{path}[i..j]$ *according to* $\mathsf{target}[i..j]$ *and return a pair* $(\mathsf{B}, \mathsf{dst})$ *for any block* B *to be moved from* $\mathsf{path}[i..j]$ *down to height* dst. */

1: **if** $(i == j)$ **then**
2:     **if** $\mathsf{target}[i] \neq \perp$ **then**
3:         B := Deepest block in $\mathsf{path}[i]$.
4:         Remove B from $\mathsf{path}[i]$.
        return $(\mathsf{B}, \mathsf{target}[i])$
5:     **else** **return** $(\perp, \perp)$

▷ $i < j$

6: $p := \left\lfloor \frac{i+j}{2} \right\rfloor$
7: Execute the following two statements **in parallel**:
- $(\mathsf{B}_1, \mathsf{dst}_1) := \mathsf{EvictFast}(\mathsf{path}[i..p])$
- $(\mathsf{B}_2, \mathsf{dst}_2) := \mathsf{EvictFast}(\mathsf{path}[p+1..j])$

8: **for** $k = p + 1$ to j in parallel **do**
9:     **if** $(\mathsf{dst}_1 == k)$ **then** put block B_1 in $\mathsf{path}[k]$.
10: **if** $\mathsf{dst}_1 > j$ **then return** $(\mathsf{B}_1, \mathsf{dst}_1)$
11: **else return** $(\mathsf{B}_2, \mathsf{dst}_2)$

The important observation is that if a block is supposed to go from $\mathsf{path}[i..p]$ to some destination height dst in $\mathsf{path}[p+1..j]$, even if we do not know what dst is, we know the source level of this block is given by $\mathsf{deepest}[p+1] = \mathsf{deepest}[\mathsf{dst}]$.

This intuition suggest that we need to prepare for two scenarios. The first scenario is that no block is supposed to go from $\mathsf{path}[i..p]$ to $\mathsf{path}[p+1:]$, and this corresponds to the solution in $\mathsf{target1}[i..p]$. The second scenario is that a block is supposed to go from $\mathsf{path}[i..p]$ to $\mathsf{path}[p+1:]$, and so we pretend that the destination height is $p+1$ for the time being to produce $\mathsf{target1}[i..p]$.

Once the solutions $\mathsf{target1}$ and $\mathsf{target2}$ have been produced for the sub-instance $[p+1..j]$, we know whether a block is supposed to go from $\mathsf{path}[i..p]$ to $\mathsf{path}[p+1:]$ in each case, and we also know the true destination of that block. Hence, we can use this information to correct the solutions for the sub-instance $[i..p]$ accordingly.

3. EvictFast. After PrepareTarget has been called, the sequential version just needs to perform a root-to-leaf scan to move the blocks according to $\mathsf{target1}[0..L-1]$.

   After the sub-instances $[i..p]$ and $[p+1..j]$ are solved, we just need to know if any block is supposed to be moved from $\mathsf{path}[i..p]$ to $\mathsf{path}[p+1:]$. If yes, the block can either be moved to $\mathsf{path}[p+1..j]$ or passed to $\mathsf{path}[j+1:]$, as shown in Algorithm 4.

□