Oblivious Parallel RAM

Elette Boyle*
Technion Israel
eboyle@alum.mit.edu

Kai-Min Chung
Academica Sinica
kmchung@iis.sinica.edu.tw

Rafael Pass†
Cornell University
rafael@cs.cornell.edu

August 2, 2014

Abstract

A machine is said to be *oblivious* if the sequences of memory accesses made by the machine for two inputs with the same running time are identically (or close to identically) distributed. Oblivious RAM (ORAM) compilers—compilers that turn any RAM program Π into a oblivious RAM Π', while only incurring a "small", polylogarithmic, slow-down—have been extensively studied since the work of Goldreich and Ostrovsky [GO96] and have numerous fundamental applications. These compilers, however, do not leverage parallelism: even if Π can be heavily parallelized, Π' will be inherently sequential.

In this work, we present the first *Oblivious Parallel RAM (OPRAM)* compiler, which compiles any PRAM into an oblivious PRAM while only incurring a polylogarithmic slowdown.

*The research of the first author has received funding from the European Union's Tenth Framework Programme (FP10/ 2010-2016) under grant agreement no. 259426 ERC-CaC.

†Pass is supported in part by a Alfred P. Sloan Fellowship, Microsoft New Faculty Fellowship, NSF Award CNS-1217821, NSF CAREER Award CCF-0746990, NSF Award CCF-1214844, AFOSR YIP Award FA9550-10-1-0093, and DARPA and AFRL under contract FA8750-11-2- 0211. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

1 Introduction

Oblivious Machines. A machine is said to be *memory oblivious*, or simply *oblivious*, if the sequences of memory accesses made by the machine on two inputs with the same running time are identically (or close to identically) distributed. For instance, if we restrict to programs which always have the same running time, the notion of an *Oblivious Turing Machine*, introduced by Pippenger and Fischer [PF79], is a Turing machine where the movement of the heads on the tape is independent of the input, and the notion of an *Oblivous RAM*, introduced by Goldreich [Gol87] and Goldreich and Ostrovsky [GO96], is a Random Access Machine (RAM) where the distribution of memory access locations are independent of the input. Already in the late 1970s, Pippenger and Fischer [PF79] showed that any Turing Machine Π can be compiled into an oblivious one Π' with only a logarithmic slowdown in running-time. Roughly ten years later, Goldreich and Ostrovsky [GO96] showed a similar result for RAMs, but this time the slowdown was polylogarithmic; in recent years several improvements to both the asymptotic and concrete efficiency of such ORAM compilers have been developed (e.g., [Ajt10, DMN11, GMOT11, KLO12, CP13, CLP13, GGH$^+$13, SvDS$^+$13, PR14]).

In parallel with these recent developments, several important applications of ORAMs have surfaced, for example in the context of software protection (already in the original work by [GO96]), in the context of secure two-party [OS97, GKK$^+$11, LO13, GGH$^+$13] and multi-party [DMN11, BCP14] computation, in the context of building secure hardware with untrusted memory [FDD12], and more recently in the context of outsourcing data (we can view the CPU of a RAM program as a client, and the memory of the RAM program as being outsourced to the cloud) [SS13], and more complex server delegation scenarios (e.g., [CKW13, GHRW14]).

Oblivious Parallel RAM (OPRAM). But for many of the above-mentioned applications, the RAM model of computations is not appropriate—modern computing architectures heavily leverage *parallelism* (e.g., using multiple cores), and the above-mentioned ORAM compilers obliterate all such gains: even if Π can be heavily parallelized, the ORAM-compiled program Π' will be inherently sequential. The notion of a Parallel RAM (PRAM) better captures such computing architectures. In the PRAM model of computation, several (polynomially many) CPUs are simultaneously running, potentially communicating with one another, while accessing the same shared "external" memory. We consider the general setting of Concurrent Read, Concurrent Write (CRCW) PRAM, where CPUs may read the same data simultaneously, and simultaneous write conflicts are resolved in a canonical way.

Our focus here is on the notion of an *Oblivious Parallel Random Access Machines (OPRAM)*. Our main contribution is presenting an OPRAM compiler that turns any m-processor PRAM into an m-processor OPRAM, while only inducing a polylogarithmic slowdown.

Theorem 1.1 (Informally stated)**.** *There exists an OPRAM compiler with* $\mathsf{polylog}(n)$ *worst-case computational overhead and* $\omega(\log n)$ *memory overhead, where* n *is the memory size.*

Note that any PRAM can trivially be turned into a RAM (just as a RAM can be turned into a Turing machine), and we could then just rely on standard ORAM compilers; but, in general, such a transformation would result in very large overhead (as would a generic RAM to Turing machine transformation).

We proceed to provide a high-level overview of our construction.

1.1 Construction Overview

Our construction is based on a recent ORAM construction due to Chung and Pass (CP) [CP13], which in turn relies on the tree-based ORAM construction due to Shi, Chan, Stefanov and Li (SCSL) [SCSL11]. To explain our approach, let us first recall the SCSL-CP construction.

The SCSL-CP ORAM Compiler. At a high level, each memory cell r in the original database will be associated with a random leaf *pos* in a binary tree, as specified by a so-called "position map" with $pos = Pos(r)$. Each node in the tree consists of a "bucket," which stores a collection of elements. The content *val* of memory cell r will be found inside one of the buckets *along the path* from the root to the leaf *pos*, and will be stored in a tuple of the form (r, val, pos). Originally, the tuple corresponding to memory cell r will be placed in the root bucket of the tree, and later on, the tuple will be moved to a bucket lower down the tree through a "flush" procedure, which pushes tuples down the tree as far as they can go along a random path, while ensuring that each memory cell r is still found on its appropriate path from the root to its assigned leaf $Pos(r)$. Each time the content of a memory cell r is accessed (either through a read or a write) the following steps are performed:

- We look up the position $pos = Pos(r)$ of the memory cell r in the position map Pos.
- We access all memory buckets in the tree from the root to the leaf *pos*. Once the tuple (r, val, pos) corresponding to the memory cell r is found (in a bucket on the path), it is removed from the bucket. Note that even if the memory cell if found high up in the tree, we still traverse the whole path.
- We assign memory cell r to a new random leaf pos', and store the new position in position map $Pos(r) = pos'$.
- The content (potentially updated in case of a write operation) of the memory cell r is put back in the root as the new tuple (r, val', pos').
- We finally perform a "flush": we pick an independently random leaf pos'', traverse the tree from the root to pos'', and push down tuples as far as possible along this path, as long as contents of memory cells are still stored on the path to their assigned destination leafs.

The analysis in [CP13] shows that as long as the buckets are size $\omega(\log n)$, then except with negligible probability buckets will never overflow, and hence any sequence of data accesses will appear simply as traversals down random paths of the tree. We will make use of this analysis in the present work. (We remark that the original compiler of [SCSL11] showed a similar result but relies on a somewhat more complicated eviction procedure in the place of the of flush step, requiring a more elaborate analysis.)

The above solution has the disadvantage that the CPUs needs to store the whole position map *Pos* which is only smaller than the external memory by a constant factor. However, by simply recursively outsourcing the position map to a (smaller) ORAM, one eventually obtains a solution where the CPU only stores memory of size $\mathsf{polylog}(n)$ (where n is the memory size); this whole internal CPU memory can now also be stored externally at polylogarithmic (multiplicative) slow-down—at each operation we simply read/write this chunk of memory.

Further works [SvDS+13, CLP13] have considered optimized variants of this tree-based approach using even smaller bucket size, but require the use of a "stash" into which potential "overflows" of the buckets in the tree are placed. These solutions appear less amenable to parallelization.

Dealing with Parallel RAMs. To support m-processor Parallel RAM programs, first consider allowing all the m CPUs to access the above-mentioned tree structure in parallel. Several problems immediately arise when attempting to implement this (naïve) idea.

- **Parallel accesses to the same memory cell:** If multiple CPUs attempt to access the *same* memory cell r, both CPUs will traverse the same path from the root to the leaf $Pos(r)$, thus blatantly leaking the fact that the same data was accessed.

  We resolve this issue by letting the CPUs check—through an oblivious aggregation operation—whether two (or more) of them wish to access the same memory cell; if so, a representative is selected (the CPU with the smallest id) to actually perform the memory access, and all the others simply choose a random leaf and make "dummy" accesses to the nodes along the path from the root to that leaf. This ensures that each parallel access always behaves (in terms of memory accesses) as if m distinct memory cells were requested. Finally, the representative CPU needs to communicate the read value *val* back to all the other CPUs that wanted to access r; this is done using a oblivious multi-cast operation.

- **Parallel accesses to the same bucket:** The paths traversed by the different CPUs in the binary tree may intersect, which may lead to read/write collisions. (For example, all CPUs will wish to access the bucket at the root). To resolve this issue, again, the CPUs need to select some representative to perform the appropriate operations to prevent conflicts; again, this is resolved using an aggregation operation.

- **Dealing with parallel "put-backs":** Recall that in the above-described SCSL-CP ORAM solution, after a memory cell has been accessed and removed from the tree, the tuple corresponding to it is put back into the root of the tree. Now, however, we have m CPUs that in parallel will attempt to insert tuples into the root, and since the root bucket can only hold $\mathsf{polylog}(n)$ elements, it will directly overflow if $m \in \omega(\mathsf{polylog}(n))$ (recall that in the PRAM model the number m of parallel processors may be polynomially related to the input).

  To overcome this problem, instead of returning tuples back to the root, we directly insert them into level $\log m$ of the tree, while ensuring that they are placed into the correct bucket along the path to their assigned destination leaf. Note that level $\log m$ contains m buckets, and since the m tuples are each assigned to random leaves, each bucket will in expectation be assigned exactly 1 tuple.

  The challenge in this step is specifying how the m CPUs can insert elements into the tree while maintaining obliviousness. For example, if each CPU simply inserts their own tuple into its assigned node, we immediately leak information about its destination leaf node. To resolve this issue, we have the CPUs obliviously route tuples between each other, so that eventually the i'th CPU holds the tuples to be insert to the ith node, and all CPU finally perform either a real or a dummy write to their corresponding node.

- **Preventing overflows:** While the above parallel "put-back" strategy ensures that the expected number of *new* tuples per node in level $\log m$ is one for each operation of the underlying PRAM, it is not clear that we do not get overflows after executing the PRAM for multiple operations. The flush operations in the ORAM are meant to prevent this. To ensure that no new overflows are introduced, we now flush m times instead of once (since we have reinserted m tuples), and all these m flushes are done in parallel: each CPU simply performs an independent flush. Again, these parallel flushes may lead to conflict in nodes accessed (the paths may intersect) and, as before, we resolve this issue by having the CPUs elect some representative to perform the appropriate operations.

But so far we have not discussed how the above operations can be executed. In particular, in the above description, we require an oblivious routing, an oblivious aggregation, and an oblivious multi-cast.

As an initial step, in Section 3.2, we provide a full description of an OPRAM with desired polylogarithmic slowdown in terms of the number of external memory accesses, but which requires large size local memory stored by each CPU, and large communication between the CPUs (i.e., polynomial in the number of processors). In such "large-bandwidth" setting, the above operations can be trivially implemented by simply having the CPU broadcast all information to one another. Then, in our actual solution, presented in Section 3.3, we show how to instantiate the above procedures using efficient, oblivious, distributed protocols.

We achieve oblivious routing of data items to secret destinations via a fixed-topology routing procedure in $\log m$ rounds, which essentially amounts to routing the messages along the edges of a boolean hypercube. Namely, in each round t, all messages are routed in the correct direction along edges in the tth dimension of the hypercube toward their target destinations; at the conclusion of $\log m$ rounds, each message will be successfully delivered. We show that for our application, the target addresses will be distributed randomly, and hence that we will not encounter message congestion at any node in any of the intermediate routing steps.

The oblivious aggregation and oblivious multi-cast procedures are duals of one another: in the former, a number of CPUs want to efficiently communicate some pieces of information to a single CPU, whereas in the latter a single CPU wants to efficiently communicate some piece of information to a set of CPUs. We rely on a similar procedure to solve both. At a high level, we perform the following steps. First the CPUs' data items are obliviously sorted based on some key (e.g., the memory cell the CPU is trying to access). Then their data is aggregated when possible (e.g., combining requests for the same cell r) by exchanging data with each CPU neighbor along the boolean hypercube (equivalently, "passing data to the left" to CPUs of increasing distance $2^0, 2^1, 2^2, ...2^{\log m}$ and aggregating), and then reversing the original sort to return each aggregated request for cell r to an appropriate CPU who originally requested r.

1.2 Applications of OPRAM

We briefly discuss some immediate applications of secure OPRAM.

Improved/Parallelized Outsourced Data. Standard ORAM has been shown to yield effective, practical solutions for securely outsourcing data storage to an untrusted server (e.g., the ObliviStore system of [SS13]). Efficient OPRAM compilers will enable these systems to support secure efficient *parallel* accesses to outsourced data. For example, OPRAM procedures securely aggregate parallel data requests and resolve conflicts client-side, minimizing expensive client-server communications (as was explored in [WST12], at a smaller scale). As network latency is a major bottleneck in ORAM implementations, such parallelization may yield significant improvements in efficiency.

Multi-Client Outsourced Data. In a similar vein, use of OPRAM further enables secure access and manipulation of outsourced shared data by multiple (mutually trusting) clients. Here, each client can simply act as an independent CPU, and will execute the OPRAM-compiled program corresponding to the parallel concatenation of their independent tasks.

Secure Multi-Processor Architecture. Much recent work has gone toward implementing secure hardware architectures by using ORAM to prevent information leakage via access patterns of the secure processor to the potentially insecure memory (e.g., the Ascend project of [FDD12]). Relying instead on OPRAM opens the door to achieving secure hardware in the multi-processor setting.

Secure Two-Party and Multi-Party Computation of PRAMs. Secure multi-party computation (MPC) enables mutually distrusting parties to jointly evaluate functions on their secret inputs, without revealing information on the inputs beyond the desired function output. ORAM has become a central tool in achieving efficient MPC protocols for securely evaluating RAM programs. By instead relying on OPRAM, these protocols can leverage parallelizability of the evaluated programs, yielding round complexities that scale with the parallel complexity of a PRAM, instead of its (sequential) time complexity as described as a RAM program.

In the two-party setting, protocol constructions generically make *black-box* use of an ORAM compiler [OS97, GKK$^+$11, LO13, GGH$^+$13]. (Roughly, these protocols work by ORAM-compiling the program Π to be evaluated and parties' inputs (i.e., the "database"), and then executing the secret ORAM CPU instructions via a standard two-party secure protocol). In such constructions, the ORAM can be directly replaced by an OPRAM scheme.

In the multi-party setting, a recent work of Boyle, Chung, and Pass [BCP14] demonstrated how to obtain secure computation of RAM programs in a manner that scales well with the number of parties. The [BCP14] protocol relies on the specific ORAM of [CP13] in a *non-black-box* way;[1] however, by inspecting [BCP14], it can be seen that our OPRAM construction in the present work in fact satisfies the same required properties (indeed, as it also builds upon [CP13]).[2]

1.3 Related Prior Work

The notion of oblivious RAM was first studied by Goldreich [Gol87] and Goldreich and Ostrovsky [GO96]. Since then, several constructions have been presented, culminating in a recent line of works following a tree-based approach due to Shi *et al.* [SCSL11] (e.g., [CP13, GGH$^+$13, SvDS$^+$13, PR14]). In this work, we focus on the construction of Chung and Pass [CP13], which achieves near optimal parameters and enjoys a particularly simple description.

The question of parallelizing the *overhead* of (standard) ORAM, or equivalently, the client-server communication required *per data access*, was studied by Lorch, Parno, Mickens, Raykova, and Schiffman [LPM$^+$13]. Note that this question is orthogonal to ours, which instead wishes to support parallelization *of data accesses*. Lorch *et al.* present a modified version of the Shi *et al.* ORAM [SCSL11] compiler, where the $O(\log^2 n)$ sequential communications between the server and CPU per data access (for database size n) can be replaced by a single round of communication between the server and $O(\log^2 n)$ distinct CPUs, in parallel. To do so, Lorch *et al.* provide a specific data aggregation procedure in which the CPUs zero out non-targeted blocks and then "obliviously aggregate" the xor of these values to learn the sought value. In comparison, our oblivious aggregate procedure requires greater computation/communication but supports more general aggregations, enabling us to parallelize over different data queries.

[1] In particular, the load-balancing and communication-locality techniques in [BCP14] rely on specific properties of the server-side data access patterns dictated by the [CP13] ORAM, which are not satisfied generically.

[2] For those readers familiar with the structure of the [BCP14] protocol: A separate committee of parties will be elected for each CPU_i; as before, the committees maintain the corresponding $\mathsf{polylog}(n)$-size CPU_i secret state, and communicate with other CPU_j committees and ORAM memory bucket committees as dictated by the OPRAM-compiled program Π'.

It was observed by Williams, Sion, and Tomescu [WST12] in their PrivateFS work that existing ORAM schemes can support parallelization across data accesses up to the "size of the top level."[3] Explicitly, Williams *et al.* focus on the original Goldreich-Ostrovsky [GO96] ORAM, and provide a means for up to $O(\log n)$ data queries to be simultaneously executed when coordinated through a central CPU (corresponding to our "high-bandwidth" case). However, this approach does not extend to the more general PRAM setting, where the number of parallel processors may be large (even $\Theta(n)$).

Goodrich and Mitzenmacher [GMOT12] provided an ORAM construction that handles some parallelism but requires a large $O(n^\epsilon)$ secret CPU memory state.

2 Preliminaries

2.1 Parallel RAM (PRAM) Programs

A Concurrent Read Concurrent Write (CRCW) m-processor *parallel random-access machine (PRAM)* with memory size n consists of numbered processors $CPU_1, \ldots, CPU_m$, each with local memory registers of size $\log n$, which operate synchronously in parallel and can make access to shared "external" memory of size n.

A PRAM program Π (given m, n, and some input x stored in shared memory) provides CPU-specific execution instructions, which can access the shared data via commands $\mathsf{Access}(r, v)$, where $r \in [n]$ is an index to a memory location, and v is a word (of size $\log n$) or $\perp$. Each $\mathsf{Access}(r, v)$ instruction is executed as:

1. **Read** from shared memory cell address r; denote value by v_{old}.
2. **Write** value $v \neq \perp$ to address r (if $v = \perp$, then take no action).
3. **Return** v_{old}.

In the case that two or more processors simultaneously initiate $\mathsf{Access}(r, v_i)$ with the same address r, then all requesting processors receive the previously existing memory value v_{old}, and the memory is rewritten with the value v_i corresponding to the lowest-numbered CPU i for which $v_i \neq \perp$.

We more generally support PRAM programs with a dynamic number of processors (i.e., m_i processors required for each time step i of the computation), as long as this sequence of processor numbers $m_1, m_2, \ldots$ is public information. The complexity of our OPRAM solution will scale with the number of required processors in each round, instead of the maximum number of required processors. For simplicity of notation, we will describe our compiler for the simplistic fixed-m setting; however, our procedures can be directly extended to the dynamic m_i setting in a straightforward manner.

The *(parallel) time complexity* of a PRAM program Π is the maximum number of time steps taken by any processor to evaluate Π, where each Access execution is charged as a single step. The PRAM complexity of a function f is defined as the minimal parallel time complexity of any PRAM program which evaluates f. We remark that the PRAM complexity of any function f is bounded above by its circuit depth complexity.

2.2 Tree-Based ORAM

Concretely, our solution relies on the ORAM due to Chung and Pass [CP13], which in turn closely follows the tree-based ORAM construction of Shi *et al.* [SCSL11]. We now recall the [CP13]

[3] In the example of the SCSL-CP tree-based ORAMs, this would correspond roughly to the number of data items that can be inserted into the root node before it overflows.

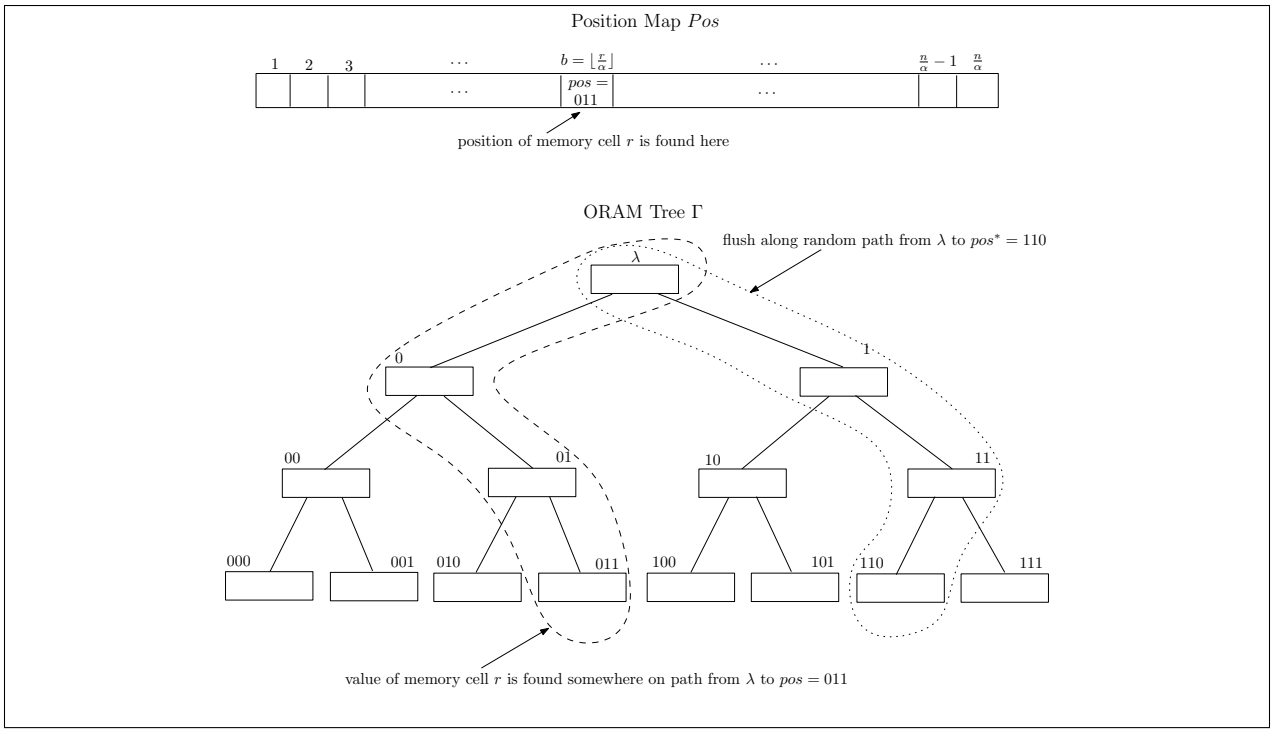

Figure 1: Illustration of the basic [CP13] ORAM construction.

construction in greater detail, in order to introduce notation for the remainder of the paper.

The [CP13] construction (as with [SCSL11]) proceeds by first presenting an intermediate solution achieving obliviousness, but in which the CPU must maintain a large number of registers (specifically, providing a means for securely storing n data items requiring CPU state size $\tilde{\Theta}(n/\alpha)$, where $\alpha > 1$ is any constant). Then, this solution is recursively applied $\log_\alpha n$ times to store the resulting CPU state, until finally reaching a CPU state size $\mathsf{polylog}(n)$, while only blowing up the computational overhead by a factor $\log_\alpha n$. The overall compiler is fully specified by describing one level of this recursion.

Step 1: Basic ORAM with $O(n)$ registers. The compiler $ORAM$ on input $n \in \mathbb{N}$ and a program Π with memory size n outputs a program Π' that is identical to Π but each $\mathsf{Read}(r)$ or $\mathsf{Write}(r, val)$ is replaced by corresponding commands $\mathsf{ORead}(r)$, $\mathsf{OWrite}(r, val)$ to be specified shortly. Π' has the same registers as Π and additionally has n/α registers used to store a *position map* Pos plus a polylogarithmic number of additional *work* registers used by ORead and OWrite. In its external memory, Π' will maintain a complete binary tree Γ of depth $\ell = \log(n/\alpha)$; we index nodes in the tree by a binary string of length at most ℓ, where the root is indexed by the empty string λ, and each node indexed by γ has left and right children indexed $\gamma 0$ and $\gamma 1$, respectively. Each memory cell r will be associated with a random leaf pos in the tree, specified by the position map Pos; as we shall see shortly, the memory cell r will be stored at one of the nodes on the path from the root λ to the leaf pos. To ensure that the position map is smaller than the memory size, we assign a *block* of α consecutive memory cells to the same leaf; thus memory cell r corresponding to block $b = \lfloor r/\alpha \rfloor$ will be associated with leaf $pos = \mathsf{Pos}(b)$.

Each node in the tree is associated with a *bucket* which stores (at most) K tuples (b, pos, v), where v is the content of block b and pos is the leaf associated with the block b, and $K \in \omega(\log n) \cap$

$\mathsf{polylog}(n)$ is a parameter that will determine the security of the ORAM (thus each bucket stores $K(\alpha+2)$ words). We assume that all registers and memory cells are initialized with a special symbol $\perp$.

The following is a specification of the $\mathsf{ORead}(r)$ procedure:

Fetch: Let $b = \lfloor r/\alpha \rfloor$ be the block containing memory cell r (in the original database), and let $i = r \mod \alpha$ be r's component within the block b. We first look up the position of the block b using the position map: $pos = \mathsf{Pos}(b)$; if $\mathsf{Pos}(b) = \perp$, set $pos \leftarrow [n/\alpha]$ to be a uniformly random leaf.

Next, traverse the data tree from the root to the leaf pos, making exactly one read and one write operation for the memory bucket associated with each of the nodes along the path. More precisely, we read the content once, and then we either write it back (unchanged), or we simply "erase it" (writing $\perp$) so as to implement the following task: search for a tuple of the form (b, pos, v) for the desired b, pos in any of the nodes during the traversal; if such a tuple is found, remove it from its place in the tree and set v to the found value, and otherwise take $v = \perp$. Finally, return the ith component of v as the output of the $\mathsf{ORead}(r)$ operation.

Update Position Map: Pick a uniformly random leak $pos' \leftarrow [n/\alpha]$ and let $\mathsf{Pos}(b) = pos'$.

Put Back: Add the tuple (b, pos', v) to the root λ of the tree. If there is not enough space left in the bucket, abort outputting $\mathsf{overflow}$.

Flush: Pick a uniformly random leaf $pos^* \leftarrow [n/\alpha]$ and traverse the tree from the roof to the leaf pos^*, making exactly one read and one write operation for every memory cell associated with the nodes along the path so as to implement the following task: "push down" each tuple (b'', pos'', v'') read in the nodes traversed so far as possible along the path to pos^* while ensuring that the tuple is still on the path to its associated leaf pos'' (that is, the tuple ends up in the node $\gamma =$ longest common prefix of pos'' and pos^*.) Note that this operation can be performed trivially as long as the CPU has sufficiently many work registers to load two whole buckets into memory; since the bucket size is polylogarithmic, this is possible. If at any point some bucket is about to overflow, abort outputting $\mathsf{overflow}$.

$\mathsf{OWrite}(r, v)$ proceeds identically in the same steps as $\mathsf{ORead}(r)$, except that in the "Put Back" steps, we add the tuple (b, pos', v'), where v' is the string v but the ith component is set to v (instead of adding the tuple (b, pos', v) as in ORead). (Note that, just as ORead, OWrite also outputs the ordinal memory content of the memory cell r; this feature will be useful in the "full-fledged" construction.)

The full-fledged construction: ORAM with polylog registers. The full-fledged construction of the CP ORAM proceeds as above, except that instead of storing the position map in registers in the CPU, we now recursively store them in another ORAM (which only needs to operate on n/α memory cells, but still using buckets that store K tuples). Recall that each invocation of ORead and OWrite requires reading one position in the position map and updating its value to a random leaf; that is, we need to perform a *single* recursive OWrite call (recall that OWrite updates the value in a memory cell, and returns the old value) to emulate the position map.

At the base of the recursion, when the position map is of constant size, we use the trivial ORAM construction which simply stores the position map in the CPU registers.

Theorem 2.1 ([CP13])**.** *The compiler ORAM described above is a secure Oblivious RAM compiler with* $\mathsf{polylog}(n)$ *worst-case computation overhead and* $\omega(\log n)$ *memory overhead, where* n *is the database memory size.*

2.3 Sorting Networks

Our protocol will employ an *n-wire sorting network*, which can be used to sort values on n wires via a fixed topology of comparisons. A sorting network consists of a sequence of *layers*, each layer in turn consisting of one or more comparator gates, which take two wires as input, and swap the values when in unsorted order. Formally, given input values $\vec{x} = (x_1, \ldots, x_n)$ (which we assume to be integers wlog), a comparator operation $\mathsf{compare}(i, j, \vec{x})$ for $i < j$ returns $\vec{x'}$ where $\vec{x} = \vec{x'}$ if $x_i \leq x_j$, and otherwise, swaps these values as $x'_i = x_j$ and $x'_j = x_i$ (whereas $x'_k = x_k$ for all $k \neq i, j$). Formally, a layer in the sorting network is a set $L = \{(i_1, j_1), \ldots, (i_k, j_k)\}$ of pairwise-disjoint pairs of distinct indices of $[n]$. A d-depth sorting network is a list $SN = (L_1, \ldots, L_d)$ of layers, with the property that for any input vector $\vec{x}$, the final output will be in sorted order $x_i \leq x_{i+1}\ \forall i < n$.

Ajtai, Komlós, and Szemerédi demonstrated a sorting network with depth logarithmic in n.

Theorem 2.2. *[AKS83] There exists an n-wire sorting network of depth $O(\log n)$ and size $O(n \log n)$.*

While the AKS sorting network is asymptotically optimal, in practical scenarios one may wish to use the simpler alternative construction due to Batcher [Bat68] which achieves significantly smaller linear constants.

Theorem 2.3. *[Bat68] There exists an n-wire sorting network of depth $O(\log^2 n)$ and size $O(n \log^2 n)$.*

3 Oblivious PRAM

The definition of an Oblivious PRAM (OPRAM) compiler directly mirrors that of standard ORAM, with the exception that the compiler takes as input and produces as output a *parallel* RAM program. Namely, denote the sequence of shared memory cell accesses made during an execution of a PRAM program Π on input (m, n, x) as $\tilde{\Pi}(m, n, x)$. We present a definition of an OPRAM compiler following Chung and Pass [CP13], which in turn follows Goldreich [Gol87].

Definition 3.1 (Oblivious Parallel RAM)**.** A polynomial-time algorithm O is an *Oblivious Parallel RAM (OPRAM)* compiler with computational overhead $\mathsf{comp}(\cdot)$ and memory overhead $\mathsf{mem}(\cdot)$, if O given $m, n \in \mathbb{N}$ and a deterministic m-processor PRAM program Π with memory size n, outputs a program Π' with memory size $\mathsf{mem}(n) \cdot n$ such that for any input x, the parallel running time of $\Pi'(m, n, x)$ is bounded by $\mathsf{comp}(n) \cdot T$, where T is the parallel runtime of $\Pi(m, n, x)$, and there exists a negligible function μ such that the following properties hold:

- **Correctness:** For any $m, n \in \mathbb{N}$ and any string $x \in \{0,1\}^*$, with probability at least $1 - \mu(n)$, it holds that $\Pi(m, n, x) = \Pi'(m, n, x)$.
- **Obliviousness:** For any two PRAM programs Π_1, Π_2, any $m, n \in \mathbb{N}$, and any two inputs $x_1, x_2 \in \{0,1\}^*$, if $|\Pi_1(m, n, x_1)| = |\Pi_2(m, n, x_2)|$, then $\tilde{\Pi}'_1(m, n, x_1)$ is μ-close to $\tilde{\Pi}'_2(m, n, x_2)$ in statistical distance, where $\Pi'_i \leftarrow O(m, n, \Pi_i)$ for $i \in \{1, 2\}$.

3.1 Solution Overview

Our OPRAM compiler O, on input $m, n \in \mathbb{N}$ and a m-processor PRAM program Π with memory size n, will output a program Π' that is identical to Π, but where each $\mathsf{Access}(r, v)$ operation is replaced by a sequence of operations defined by subroutine $\mathsf{OPAccess}(r, v)$, which we will construct over the following subsections.

The $\mathsf{OPAccess}$ procedure begins with m CPUs, each with a requested data cell r (within some block b) and some action to be taken (either $\perp$ to denote read, or v to denote rewriting cell r with

value v). Recall that the primary challenges in implementing oblivious parallel data accesses within the tree-based ORAM structure of [SCSL11, CP13] are in handling collisions between processor accesses, and in reinserting data to the ORAM (and flushing data down the tree) in parallel.

Intuitively, our subroutine $\mathsf{OPAccess}$ addresses these challenges by the following sequence of tasks:

1. **Conflict Resolution:**
   - Choose one representative CPU per requested data block b (in the real database). This representative will perform the real data fetch and computation on b in later steps, while the other CPUs will simply make "dummy" accesses into the ORAM structure.
   - Aggregate all CPU instructions to take place on each requested block b.
2. **Read/Write Position Map:**
   - Each representative CPU: Sample a fresh random leaf id ℓ'. Perform a (recursive) Read/Write access command on the position map database $\ell \leftarrow \mathsf{OPAccess}(b_i, \ell')$ to fetch the current position map value ℓ and rewrite it with the newly sampled value ℓ'.
   - Each dummy CPU: Perform a dummy access to an arbitrary cell in the position map database, say the first. (Recall that the position map database is itself protected by a layer of ORAM). That is, execute $\ell \leftarrow \mathsf{OPAccess}(1, \emptyset)$, and ignore the read value ℓ.
3. **Look Up Current Memory Values:** Each representative CPU fetches memory from ORAM database nodes corresponding to accessing his desired data block b (i.e., the collection of buckets down the relevant path in the ORAM tree) and copies the values into local memory. Non-chosen CPUs choose a random path ℓ (independent of the position map above) and make analogous dummy data fetches along the path to ℓ, ignoring all read values. Recall that simultaneous data *reads* do not yield conflicts.[4]
4. **Remove Old Data:** Consider the paths down the ORAM tree accessed in the previous step.
   - Aggregate instructions across CPUs accessing the same "buckets" of memory on the server side. Each representative CPU $\mathsf{rep}(b)$ begins with the instruction of "remove block b if it occurs" and dummy CPUs hold the empty instruction. (Aggregation is as before, but at bucket level instead of the block level).
   - For each bucket to be modified, the CPU with the *smallest* id from those who wish to modify it executes the aggregated block-removal instructions for the bucket.
5. **Insert Updated Data into Database in *Parallel*:** All CPUs execute a parallel insertion procedure into the ORAM database at the appropriate level (corresponding to the number of active CPUs) in order to insert the updated data tuples (b, ℓ', v') with new leaf node ℓ' as sampled in Step 1 and new value v' into the bucket along the path to ℓ'.
6. **Flush the ORAM Database:** In parallel, each CPU initiates an independent flush of the ORAM tree. (Recall that this corresponds to selecting a random path down the tree, and pushing all data blocks in this path as far as they will go). To implement the simultaneous flush commands, as before, commands are aggregated across CPUs for each bucket to be modified, and the CPU with the smallest id performs the corresponding aggregated set of commands. (For example, all CPUs will wish to access the root node in their flush; the aggregation of all corresponding commands to the root node data will be executed by the lowest-numbered CPU who wishes to access this bucket, in this case CPU 1).

[4] We will in fact reveal to the adversary that only read actions are occurring, by not rewriting the data values. But, this will not be an issue, as this step always induces read-only operations, independent of the data values.

7. **Return Output:** Each representative CPU $\mathsf{rep}(b)$ communicates the *original* value of the data block b to the subset of CPUs that originally requested it.

We now proceed to flesh out this OPAccess procedure. We begin in Section 3.2 by considering a simplified setting, in which CPUs are able to cheaply communicate amongst themselves and to store large information (comparable to the number of processors). Then in Section 3.3 we show how to replace this large CPU-to-CPU communication and memory cost via more sophisticated procedures for oblivious aggregation, oblivious multi-cast, and oblivious route.

3.2 Rudimentary Solution: Requiring Large Bandwidth.

We first provide a solution for a simplified case, where we are not concerned with minimizing communication between CPUs or the size of required CPU local memory. In such setting, communicating and aggregating information between all CPUs is "for free."

The compiler Heavy-O, on input $m, n \in \mathbb{N}$ and m-processor PRAM program Π with memory size n, outputs a program Π' identical to Π, but with each $\mathsf{Access}(r, v)$ operation replaced by the modified procedure Heavy-OPAccess as defined in Figure 2.

Lemma 3.2. *For any $n, m \in \mathbb{N}$, The compiler* Heavy-O *is a secure Oblivious PRAM compiler with computational overhead* $\mathsf{polylog}(n)$ *and memory overhead* $\mathsf{polylog}(n)$*, assuming each CPU has $\tilde{\Omega}(m)$ local memory.*

We will address the desired claims of correctness, security, and complexity of the Heavy-O compiler by induction on the number of levels of recursion. Namely, Lemma 3.2 follows directly from the following claim, applied with $t = \lceil \log n \rceil$.

By Heavy-O_t we will denote the Heavy-O compiler implemented with $t \leq \lceil \log n \rceil$ recursion levels. That is, the exit condition "If $t' \geq \log_\alpha n$" in Step 0 (see Figure 2) is replaced by "If $t' \geq t$" (denoting the current recursion counter by t').

Claim 3.3. Heavy-O_t *is a secure Oblivious PRAM compiler with computational overhead* $\mathsf{polylog}(n)$ *and memory overhead* $\mathsf{polylog}(n)$*, assuming each CPU has (large) local memory $\Omega(m + n/\alpha^t)$.*

Proof. Observe the claim holds trivially for $t = 0$, in which case the entire size-n database is simply stored locally by each CPU. Suppose it holds for some $0 \leq t \leq \lceil \log n \rceil - 1$; we now prove it holds also for $t+1$. We first analyze the correctness, security, and complexity overhead of the Heavy-O_{t+1} conditioned on never reaching the event overflow. Then, we prove that the probability of overflow is negligible in n.

Correctness (w/o overflow). Consider the state of the memory (of the CPUs and server) in each step of Heavy-OPAccess, assuming no overflow. In Step 1, each CPU learns the instruction pairs of all other CPUs; thus all CPUs agree on single representative $\mathsf{rep}(b_i)$ for each requested block b_i, and a correct aggregation of all instructions to be performed on this block. Step 2 is a recursive execution of Heavy-OPAccess. By the inductive hypothesis, this access successfully returns the correct value ℓ_i of $\mathsf{Pos}(b_i)$ for each b_i queried, and rewrites it with the freshly sampled value ℓ_i' when specified (i.e., for each $\mathsf{rep}(b_i)$ access; the dummy accesses are read-only). We are thus guaranteed that each $\mathsf{rep}(b_i)$ will find the desired block b_i in Step 3 when accessing the memory buckets in the path down the tree to leaf ℓ_i (as we assume no overflow was encountered), and so will learn the current stored data value v_{old}.

In Step 4, each CPU learns the target block b_i and associated leaf ℓ_i of every representative CPU $\mathsf{rep}(b_i)$. By construction, each requested block b_i appears in some bucket B in the tree along

$\mathsf{Heavy\text{-}OPAccess}(t, (r_i, v_i))$**: The Large Bandwidth Case**

To be executed by m processors $CPU_1, \ldots, CPU_m$ w.r.t. (recursive) database size $n_t := n/(\alpha^t)$.

Input: Each CPU_i holds: recursion level t, instruction pair (r_i, v_i) with $r_i \in [n_t]$, global parameter α.

Each CPU_i performs the following steps, in parallel:

0. Exit Case: If $t \geq \log_\alpha n$, access *local memory*.
   Set $v_i^{old} \leftarrow \mathsf{Mem}[r_i]$. Write $\mathsf{Mem}[r_i] \leftarrow v_i$. return v_i^{old}.
1. Conflict Resolution
   (a) Broadcast the instruction pair (r_i, v_i) to all CPUs. (Note: high bandwidth & memory).
   (b) Let $b_i = \lfloor r_i/\alpha \rfloor$. Locally aggregate incoming instructions to block b_i as $\bar{v}_i = \bar{v}_i[1] \cdots \bar{v}_i[\alpha]$, resolving write conflicts (i.e., $\forall s \in [\alpha]$, take $\bar{v}_i[s] \leftarrow v_j$ for minimal j such that $r_j = b_i\alpha + s$). Denote by $\mathsf{rep}(b_i) := \min\{j : \lfloor r_j/\alpha \rfloor = b_i\}$ the smallest index j of *any* CPU whose r_j is in this block b_i. (CPU $\mathsf{rep}(b_i)$ will actually access b_i, while others perform dummy accesses).
2. Read/Write Position Map
   If $i = \mathsf{rep}(b_i)$: Sample a fresh random leaf id $\ell_i' \leftarrow [n_t]$. Recursively initiate $\ell_i \leftarrow \mathsf{Heavy\text{-}OPAccess}(t+1, (b_i, \ell_i'))$ to read the current value ℓ_i of $\mathsf{Pos}(b_i)$ and rewrite it with ℓ_i'.
   Else: Recursively initiate *dummy* access $x \leftarrow \mathsf{Heavy\text{-}OPAccess}(t+1, (1, \perp))$ at arbitrary address (say 1); ignore the read value x. Sample a fresh random leaf id $\ell_i' \leftarrow [n_t]$ for a dummy lookup.
3. Look Up Current Memory Values
   Read the memory contents of all buckets down the path to leaf node ℓ_i defined in the previous step, copying all buckets into local memory.
   If $i = \mathsf{rep}(b_i)$: locate and store target block triple (b_i, v_i^{old}, ℓ_i). Update $\bar{v}$ from Step 1 with existing data: $\forall s \in [\alpha]$, replace any non-written cell values $\bar{v}_i[s] = \emptyset$ with $\bar{v}_i[s] \leftarrow v_i^{old}[s]$. $\bar{v}_i$ now stores the entire data block to be rewritten for block b_i.
4. Remove Old Data from ORAM Database
   (a) If $i = \mathsf{rep}(b_i)$: Broadcast the pair (b_i, ℓ_i) to all CPUs. Otherwise: broadcast $(\perp, \ell_i)$.
   (b) Initiate $\mathsf{UpdateBuckets}\left(n_t, (\mathsf{remove}\text{-}b_i, \ell_i), \{(\mathsf{remove}\text{-}b_j, \ell_j)\}_{j \in [m] \setminus \{i\}}\right)$.
5. Insert New Data into Database *in Parallel*
   (a) If $i = \mathsf{rep}(b_i)$: Broadcast $(b_i, \bar{v}_i, \ell_i')$, including updated value $\bar{v}_i$ and target leaf node ℓ_i'.
   (b) Let $\mathsf{lev}^* := \lfloor \log m \rfloor$ be the ORAM tree level with number of buckets equal to number of CPUs (the level where data will be inserted). Locally aggregate all incoming instructions whose path ℓ_j' has lev^*-bit prefix i: $\mathsf{Insert}_i := \{(b_j, \bar{v}_j, \ell_j') : (\ell_j')^{(\mathsf{lev}^*)} = i\}$.
   (c) Access memory bucket i (at level lev^*) and rewrite contents, inserting data items Insert_i.
6. Flush the ORAM Database
   (a) Sample a random leaf node $\ell_i^{\mathsf{flush}} \leftarrow [n_t]$ along which to flush. Broadcast ℓ_i^{flush} to all CPUs.
   (b) If $i \leq n_t$: Initiate $\mathsf{UpdateBuckets}\left(n_t, (\mathsf{flush}, \ell_i^{\mathsf{flush}}), \{(\mathsf{flush}, \ell_j^{\mathsf{flush}})\}_{j \in [m] \setminus \{i\}}\right)$.
   Recall that flush denotes that for each encountered data triple (b, ℓ, v), "pushes" the triple down to the lowest point at which his chosen flush path and ℓ agree.
7. Update CPUs
   If $i = \mathsf{rep}(b_i)$: broadcast the *old* value v_i^{old} of block b_i to all CPUs.

Figure 2: Pseudocode for oblivious parallel data access procedure $\mathsf{Heavy\text{-}OPAccess}$, in the case where we are not concerned with the per-round bandwidth/memory of the protocol.

$\mathsf{UpdateBuckets}\left(n_t, (\mathsf{mycommand}, \mathsf{mypath}), \{(\mathsf{command}_j, \mathsf{path}_j)\}_{j\in[m]\setminus\{i\}}\right)$
Let $\mathsf{path}^{(0)}, \mathsf{path}^{(1)}, \ldots, \mathsf{path}^{(\log n_t)}$ denote the bit prefixes of length 0 (i.e., $\emptyset$) to $\log n_t$ of path.

For each level $\mathsf{lev} = 0, \ldots, \log n_t$ of the tree, each CPU i does the following (at bucket $\mathsf{mypath}^{(\mathsf{lev})}$):

1. Define $\mathsf{CPUs}(\mathsf{mypath}^{(\mathsf{lev})}) := \{i\} \cup \{j : \mathsf{path}_j^{(\mathsf{lev})} = \mathsf{mypath}^{(\mathsf{lev})}\}$ to be the set of CPUs requesting changes to bucket $\mathsf{mypath}^{(\mathsf{lev})}$. Let $\mathsf{bucket\text{-}rep}(\mathsf{mypath}^{(\mathsf{lev})})$ denote the *minimal* index in the set.
2. If $i \neq \mathsf{bucket\text{-}rep}(\mathsf{mypath}^{(\mathsf{lev})})$, do nothing. Otherwise:

   Case 1: $\mathsf{mycommand} = \mathsf{remove}\text{-}b_i$.

   Interpret each $\mathsf{command}_j = \mathsf{remove}\text{-}b_j$ as a target block id b_j to be removed. Access memory bucket $\mathsf{mypath}^{(\mathsf{lev})}$ and rewrite contents, removing any block b_j for which $j \in \mathsf{CPUs}(\mathsf{mypath}^{(\mathsf{lev})})$.

   Case 2: $\mathsf{mycommand} = \mathsf{flush}$.

   Define $\mathsf{Flush} \subset \{L, R\}$ as $\{v : \exists\ \mathsf{path}_j \text{ s.t. } \mathsf{path}_j^{(\mathsf{lev}+1)} = \mathsf{mypath}^{(\mathsf{lev})}||v\}$, associating $L \equiv 0$, $R \equiv 1$. This determines whether data will be flushed left and/or right from this bucket. Access memory bucket $\mathsf{mypath}^{(\mathsf{lev})}$; denote its collection of stored data blocks b by $\mathsf{ThisBucket}$. Partition $\mathsf{ThisBucket} = \mathsf{ThisBucket\text{-}L} \cup \mathsf{ThisBucket\text{-}R}$ into those blocks whose associated leaves continue to the left or right (i.e., $\mathsf{ThisBucket\text{-}L} := \{b_j \in \mathsf{ThisBucket} : \bar{\ell}_j^{(\mathsf{lev}+1)} = \mathsf{mypath}^{(\mathsf{lev})}||0\}$, and similar for 1).

   - If $L \in \mathsf{Flush}$, then set $\mathsf{ThisBucket} \leftarrow \mathsf{ThisBucket} \setminus \mathsf{ThisBucket\text{-}L}$, access memory bucket $\mathsf{mypath}^{(\mathsf{lev})}||0$, and insert data items $\mathsf{ThisBucket\text{-}L}$ into it.
   - If $R \in \mathsf{Flush}$, then set $\mathsf{ThisBucket} \leftarrow \mathsf{ThisBucket} \setminus \mathsf{ThisBucket\text{-}R}$, access memory bucket $\mathsf{mypath}^{(\mathsf{lev})}||1$, and insert data items $\mathsf{ThisBucket\text{-}R}$ into it.

   Rewrite the contents of bucket $\mathsf{mypath}^{(\mathsf{lev})}$ with the updated value of $\mathsf{ThisBucket}$.

Figure 3: Procedure for combining CPUs' instructions for buckets and implementing them by a single representative CPU. (Used for correctness, not security).

his path, and there there will necessarily be some CPU assigned as bucket-rep(B) in UpdateBuckets, who will then successfully remove the block b_i from B. At this point, none of the requested blocks b_i appear in the tree.

In Step 5, the CPUs insert each block b_i (with updated data value v_i) into the ORAM data tree at level $\min\{\log n/\alpha^t, \lfloor \log(m) \rfloor\}$ along the path to its (new) leaf ℓ'_i.

Finally, the flushing procedure in Step 6 maintains the necessary property that each block b_i appears along the path to Pos(b_i), and in Step 7 all CPUs learn the collection of all queried values v_{old} (in particular, including the value they initially requested).

Thus, assuming no overflow, correctness holds.

Obliviousness (w/o overflow). Consider the access patterns to server-side memory in each step of Heavy-OPAccess, assuming no overflow. Step 1 is performed locally without communication to the server. Step 2 is a recursive execution of Heavy-OPAccess, which thus yields access patterns independent of the vector of queried data locations (up to statistical distance negligible in n). In Step 3, each CPU accesses the buckets along a single path down the tree, where representative CPUs rep(b_i) access along the path given by Pos(b_i) (for *distinct* b_i), and non-representative CPUs each access down an independent, random path. Since the adversarial view so far has been independent of the values of Pos(b_i), conditioned on this view all CPU's paths are independent and random.

In Step 4, all data access patterns are publicly determinable based on the accesses in the previous step (that is, the complication in Step 4 is to ensure correctness without access collisions, but is not needed for security). In Step 5, each CPU i accesses his corresponding bucket i in the tree. In the flushing procedure of Step 6, each CPU selects an independent, random path down the tree, and the communication patterns to the server reveal no information beyond the identities of these paths. Finally, Step 7 is performed locally without communication to the server.

Thus, assuming no overflow, obliviousness holds.

Protocol Complexity (w/o overflow). First note that the server-side memory storage requirement is simply that of the [CP13] ORAM construction; namely, polylog(n) memory overhead.

Consider the per-CPU required local memory. Each CPU must be able to store: $O(\log n)$-size requests from each CPU (due to the broadcasts in Steps 1(a), 4(a), 5(a), and 7); the data contents of at most 3 memory buckets (due to the flushing procedure in UpdateBuckets); and local data storage of size $n_t = n/\alpha^t$ due to terminating the recursion at level t. Overall, this yields a per-CPU local memory requirement of $\tilde{\Omega}(m + n/\alpha^t)$ (where $\tilde{\Omega}$ notation hides $\log n$ factors).

Consider the parallel complexity of the OPRAM-compiled program $\Pi' \leftarrow$ Heavy-$O(m, n, \Pi)$. For each parallel memory access in the underlying program Π, the processors perform: Conflict resolution (1 local communication round), Read/writing the position map (which has parallel complexity polylog(n) by the inductive hypothesis), Looking up current memory values (sequential steps = depth of level-t ORAM tree $\in O(\log n)$), Removing old data from the ORAM tree (1 local communication round, plus depth of the ORAM tree $\in O(\log n)$ sequential steps), Inserting the new data in parallel (1 local communication round, plus 1 communication round to the server), Flushing the ORAM database (1 local communication round, and 2× the depth of the ORAM tree rounds of communication with the server, since each bucket along a flush path is accessed once to receive new data items and once to flush its own data items down), and Updating CPUs with the read values (1 local communication round). Altogether, this yields parallel complexity overhead $O(\text{polylog}(n))$.

It remains to address the probability of encountering overflow.

Claim 3.4. *There exists a negligible function μ such that for any deterministic m-processor PRAM program Π, any database size n, and any input x, the probability that the* Heavy-O*-compiled program $\Pi'(m, n, x)$ outputs* overflow *is bounded by $\mu(n)$.*

Proof. We consider separately the probability of overflow in each of the level-t recursive ORAM trees. Since there are $\lceil \log n \rceil$ of them, the claim follows by a straightforward union bound.

Taking inspiration from [CP13], we analyze the ORAM-compiled execution via an abstract dart game. The game consists of black and white darts. In each round of the game, m black darts are thrown, followed by m white darts. Each dart independently hits the bullseye with probability $p = 1/m$. The game continues until exactly K darts have hit the bullseye, or after the end of the Tth round for some fixed polynomial bound $T = T(n)$, whichever comes first. The game is "won" (which will correspond to overflow in a particular bucket) if K darts hit the bullseye, and all of them are black.

Let us analyze the probability of winning in the above dart game.

Subclaim 1: With overwhelming probability in n, no more than $K/2$ darts hit the bullseye in any round. In any single round, associate with each of the $2 \cdot m$ darts thrown an indicator variable X_i for whether the dart strikes the target. The X_i are independent random variables each equal to 1 with probability $p = 1/m$. Thus, the probability that more than $K/2$ of the darts hit the target is bounded (via a Chernoff tail bound[5]) by

$$\Pr\left[\sum_{i=1}^{2m} X_i > K/2\right] \leq e^{\frac{2(K/4-1)^2}{2+(K/4-1)}} \leq e^{-\Omega(K)} \leq e^{-\omega(\log n)}.$$

Since there are at most $T = \mathsf{poly}(n)$ distinct rounds of the game, the subclaim follows by a union bound.

Subclaim 2: Conditioned on no round having more than $K/2$ bullseyes, the probability of winning the game is negligible in d. Fix an arbitrary such winning sequence s, which terminates sometime during some round r of the game. By assumption, the final partial round r contains no more than $K/2$ bullseyes. For the remaining $K/2$ bullseyes in rounds 1 through $r-1$, we are in a situation mirroring that of [CP13]: for each such winning sequence s, there exist $2^{K/2} - 1$ distinct other "losing" sequences s' that each occur with the same probability, where any non-empty subset of black darts hitting the bullseye are replaced with their corresponding white darts. Further, every two distinct winning sequences s_1, s_2 yield disjoint sets of losing sequences, and all such constructed sequences have the property that no round has more than $K/2$ bullseyes (since this number of total bullseyes per round is preserved). Thus, conditioned on having no round with more than $K/2$ bullseyes, the probability of winning the game is bounded above by $2^{-K/2} \in e^{-\omega(\log n)}$.

We now relate the dart game to the analysis of our OPRAM compiler.

We analyze the memory buckets at the nodes in the t-th recursive ORAM tree, via three subcases.

Case 1: Nodes in level $\mathsf{lev} < \log m$. Since data items are inserted to the tree in parallel directly at level $\log m$, these nodes do not receive data, and thus will not overflow.

Case 2: Consider any internal node (i.e., a node that is not a leaf) γ in the tree at level $\log m \leq \mathsf{lev} < \log n_t$. Note that when $m > n_t$, this case is vacuous. For purposes of analysis, consider the contents of γ as split into two parts: γ_L containing the data blocks whose leaf path continues to the left from γ (i.e., leaf $\gamma||0||\cdot$), and γ_R containing the data blocks whose leaf path

[5]Explicit Chernoff bound used: for $X = X_1 + \cdots X_{2m}$ (X_i independent) and mean μ, then for any $\delta > 0$, it holds that $\Pr[X > (1+\delta)\mu] \leq e^{-\delta^2\mu/(2+\delta)}$.

continues right (i.e., $\gamma||1||\cdot$). For the bucket of node γ to overflow, there must be K tuples in it. In particular, either γ_L or γ_R must have $K/2$ tuples.

For each parallel memory access in $\Pi(m, n, x)$, in the t-th recursive ORAM tree for which $n_t \geq m$, (at most) m data items are inserted, and then m independent paths in the tree are flushed. By definition, an inserted data item will enter our bucket γ_L (respectively, γ_R) only if its associated leaf has the prefix $\gamma||0$ (resp., $\gamma||1$); we will assume the worst case in which *all* such data items arrive directly to the bucket. On the other hand, the bucket γ_L (resp., γ_R) will be completely emptied after any flush whose path contains this same prefix $\gamma||0$ (resp., $\gamma||1$). Since all leaves for inserted data items and data flushes are chosen randomly and independently, these events correspond directly to the black and white darts in the game above. Namely, the probability that a randomly chosen path will have the specific prefix $\gamma||0$ of length lev is $2^{-\mathsf{lev}} \leq 1/m$ (since we consider $\mathsf{lev} \geq \log m$); this corresponds to the probability of a dart hitting the bullseye. The bucket can only overflow if $K/2$ "black darts" (inserts) hit the bullseye without any "white dart" (flush) hitting the bullseye in between. By the analysis above, we proved that for any sequence of $K/2$ bullseye hits, the probability that all $K/2$ of them are black is bounded above by $2^{-K/4}$, which is negligible in n. However, since there is a fixed polynomial number $T = \mathsf{poly}(n)$ of parallel memory accesses in the execution of $\Pi(m, n, x)$ (corresponding to the number of "rounds" in the dart game), and in particular, $T(2m) \in \mathsf{poly}(n)$ total darts thrown, the probability that the sequence of bullseyes contains $K/2$ sequential blacks *anywhere* in the sequence is bounded via a direct union bound by $(T2m)2^{-K/4} \in e^{-\omega(\log n)}$, as desired.

Case 3: Consider any leaf node γ. This analysis follows the same argument as in [CP13]. For there to be an overflow in γ at time t, there must be $K+1$ out of n_t/α elements in the position map that map to the leaf γ. Since all positions are sampled uniformly and independently among the n_t/α different leaves, the expected number of elements mapping to γ is $\mu = 1$, and by a standard multiplicative Chernoff bound,[6] the probability that $K + 1$ elements are mapped to γ is upper bounded by

$$\left(\frac{e^K}{(K+1)^{(K+1)}}\right)^{\mu} \leq 2^{K/2} \in e^{-\omega(\log n)}.$$

□

Thus, the total probability of $\mathsf{overflow}$ is negligible in n, and the theorem follows.

□

Remark 3.5 (Truncating OPRAM for Fixed m)**.** In the case that the number of CPUs m is fixed and known a priori, the OPRAM construction can be directly trimmed in two places.

Trimming tops of recursive data trees: As was mentioned in the OPRAM $\mathsf{overflow}$ analysis, data items are always inserted into the OPRAM trees at level $\log m$, and flushed down from this level. Thus, if the number of CPUs m is unchanging throughout the course of the program and known a priori (or at least a lower bound), then it will be the case the the top levels in the ORAM tree are *never utilized*. In such case, the data buckets in the corresponding tops of the trees, from the root node to level $\log m$ for this bound, can simply be removed without affecting the OPRAM.

Truncating recursion: In the t-th level of recursion, the corresponding database size shrinks to n/α^t. Trimming the tree tops in the fashion above, we see that starting at recursion level $\log_\alpha m$, the *entire* OPRAM tree can be trimmed, leaving only the leaves. At this point, in contrast to the standard ORAM setting (in which the final "secret database" must fit in the local registers of a

[6] We use the following version of the Chernoff bound: Let $X_1, \ldots, L_n$ be independent $[0, 1]$-valued random variables. Let $X = \sum_i X_i$ and $\mu = E[X]$. For every $\delta > 0$, $\Pr[X \geq (1 + \delta)\mu] \leq \left(\frac{e^\delta}{(1+\delta)^{(1+\delta}}\right)^{\mu}$.

single CPU), we may discontinue the recursion and store the remaining size-$\tilde{O}(m)$ secret database across the local storage of the m processors. We can then trivially achieve oblivious data accesses for this data via local communication between processors.

For simplicity of description and analysis, in our exposition we do not explicitly consider trimming tree-tops, and additionally follow the same recursive structure until the end (i.e., where *each* CPU can store the recursed database, without singling out this special final round in which *collectively* each CPU can store the recursed database).

3.3 New Tools: Space-Efficient Distributed Oblivious Insertion, Aggregation, and Multicasting

In this section, we construct space-efficient distributed procedures for oblivious parallel insertion, aggregation, and multicasting. We will show how to use these procedures to reduce the CPU local memory and inter-CPU communication of our rudimentary solution.

3.3.1 Oblivious Parallel Insertion (Oblivious Routing)

At a certain point in the $\mathsf{OPAccess}$ procedure, the processors must reinsert data blocks in parallel into secret locations within the appropriate level of the ORAM tree. In the rudimentary $\mathsf{Heavy\text{-}OPAccess}$ solution, this insertion was achieved trivially, simply by having each processor i broadcast his data block and target insertion address, gathering those data blocks whose target addresses match his id i, and then inserting these data blocks into bucket i at the correct level of the ORAM tree. We now wish to achieve a memory and communication efficient insertion procedure whose data access patterns remain independent of the secret insertion locations.

We solve this problem by delivering memory blocks to their target locations via a *fixed-topology routing network*. Namely, the m processors $CPU_1, \ldots, CPU_m$ will first write the relevant m data items msg_i (and their corresponding destination addresses addr_i) to memory in fixed order, and then rearrange them in $\log m$ sequential rounds to the proper locations via the routing network. At the conclusion of the routing procedure, each node j should hold all messages msg_i for which $\mathsf{addr}_i = j$. We now proceed to describe this routing network.

For simplicity, assume $m = 2^\ell$ for some $\ell \in \mathbb{N}$ (otherwise, consider the smallest ℓ for which $2^\ell \geq m$). The routing network has depth ℓ; in each level $t = 1, \ldots, \ell$, each node communicates with the corresponding node whose id agrees in all bit locations except for the tth (corresponding to his tth neighbor in the $\log m$-dimensional boolean hypercube). These nodes exchange messages according to the tth bit of their destination addresses addr_i. This is formally described in Figure 4. After the tth round, each message msg_i is held by a party whose id agrees with the destination address addr_i in the first t bits. Thus, at the conclusion of ℓ rounds, all messages are properly delivered.

Note that this pairwise communication structure frequently appears within *sorting networks*. In our setting, in contrast to sorting networks, data items are not simply maintained or swapped in each step, but rather may also be both directed to one node or the other. This will be necessary for us since (unlike sorting) the source-to-target mapping is not a one-to-one function.

We now show that, if the destination addresses addr_i are uniformly sampled, then with overwhelming probability no node will ever need to hold too many messages at any point during the routing network execution.

Lemma 3.6 (Routing Network)**.** *If L messages begin with target destination addresses addr_i distributed independently and uniformly over $[L]$ in the L-to-L node routing network in Figure 4,*

Parallel Insertion Routing Protocol $\mathsf{Route}(m, (\mathsf{msg}_i, \mathsf{addr}_i))$
Input: CPU_i holds: message msg_i with target destination addr_i, and global overflow threshold K.
Output: CPU_i holds $\{\mathsf{msg}_j : \mathsf{addr}_j = i\}$.

Let $\mathsf{lev}^* = \log m$ (assumed for simplicity to be an integer). Each CPU_i performs the following.

Initialize $M_{i,0} \leftarrow \mathsf{msg}_i$. For $t = 1, \ldots, \mathsf{lev}^*$:

1. Perform the following symmetric message exchange with $CPU_{i \oplus 2^t}$:

$$M_{i,t+1} \leftarrow \{\mathsf{msg}_j \in M_{i,t} \cup M_{i \oplus 2^t, t} : (\mathsf{addr}_j)_t = (i)_t\}.$$

2. If $|M_{i,t+1}| > K$ (i.e., memory overflow), then CPU_i aborts.

Figure 4: Fixed-topology routing network for delivering m messages originally held by m processors to their corresponding destination addresses within $[m]$. Used to simultaneously insert values into the ORAM tree.

then with probability bounded by $(L \log L)2^{-K}$, no intermediate node will ever hold greater than K messages at any point during the course of the protocol execution.

Proof. Consider an arbitrary node $a \in \{0,1\}^\ell$, at some level t of execution of the protocol. There are precisely 2^t possible messages m_i that could be held by node a at this step, corresponding to those originating in locations $b \in \{0,1\}^\ell$ whose final $\ell - t$ bits agree with those of a. Node a will hold message m_b at the conclusion of round t precisely if the *first* t bits of addr_b agree with those of a. For each such message m_b, the associated destination address addr_b is a random element of $[L]$, which agrees with a on the first t bits with probability 2^t.

For each $b \in \{0,1\}^\ell$ agreeing with a on the final $\ell - t$ bits, define X_b to be the indicator variable that is equal to 1 if addr_b agrees with a on the first t bits. Then the collection of 2^t random variables $\{X_b : b_i = a_i \ \forall i = t+1, \ldots, \ell\}$ are independent, and $X = \sum X_b$ has mean $\mu = 2^t \cdot 2^{-t} = 1$. Note that X corresponds to the number of messages held by node a at level t. By a Chernoff bound,[7] it holds that

$$\Pr[X \geq K] = \Pr[X \geq (1 + (K-1))\mu] < \left(\frac{e^{K-1}}{K^K}\right) < 2^{-K}.$$

Then, taking a union bound over the total number of nodes L and levels $\ell = \log L$, we have that the probability of any node experiencing an overflow at any round is bounded by $(L \log L)2^{-K}$. □

3.3.2 Oblivious Aggregation

In a number of places in the rudimentary $\mathsf{OPAccess}$ procedure, the CPUs wish to aggregate data pertaining to the same block/bucket/etc. For example, in the first step, the CPUs each begin with a target data address r_i and access instruction v_i, and they wish to aggregate all instructions pertaining to data in the same α-size data block (i.e., considering $b_i := r_i \mod \alpha$), so that a single CPU can perform all these instructions. To achieve security and efficiency in the overall OPRAM construction, we need that this aggregation procedure is oblivious (i.e., the access patterns are independent of the data), and is efficient.

Formally, we want to achieve the following aggregation goal, with communication patterns independent of the inputs, using only $\tilde{O}(\mathsf{polylog}(m))$ local memory and communication per CPU,

[7] Exact Chernoff bound used: $\Pr[X > (1+\delta)\mu] < \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^\mu$ for any $\delta > 0$.

in only $\tilde{O}(\mathsf{polylog}(m))$ sequential time steps. (Note that without requiring efficiency, this goal is trivially achieved in the rudimentary solution with a single round of CPU-to-CPU broadcast communication; but, this step requires large $\Omega(m)$ local memory and communication per CPU.) An illustrative example to keep in mind is that discussed above, where $\mathsf{key}_i = b_i$, $\mathsf{data}_i = v_i$, and Agg is the process that combines instructions to the same data block, resolving conflicts as necessary. In Section 3.4, we will describe exactly where and how the oblivious aggregation procedure is used to replace the expensive steps within the rudimentary OPRAM solution.

Oblivious aggregation:

Input: Each CPU $i \in [m]$ holds $(\mathsf{key}_i, \mathsf{data}_i)$. Let $\mathsf{K} = \bigcup\{\mathsf{key}_i\}$ denote the set of distinct keys. We assume that any (subset of) data associated with the same key can be aggregated by an aggregation function Agg to a short digest of size at most $\mathsf{poly}(\ell, \log m)$, where $\ell = |\mathsf{data}_i|$.

Goal: Each CPU i outputs out_i such that the following holds.

- for every $\mathsf{key} \in \mathsf{K}$, there exists unique agent i with $\mathsf{key}_i = \mathsf{key}$ such that $\mathsf{out}_i = (\mathsf{rep}, \mathsf{key}, \mathsf{agg}_{\mathsf{key}})$, where $\mathsf{agg}_{\mathsf{key}} = \mathsf{Agg}(\{\mathsf{data}_j : \mathsf{key}_j = \mathsf{key}\})$.
- for every remaining agent i, $\mathsf{out}_i = (\mathsf{dummy}, \perp, \perp)$.

At a high level, we achieve this via the following steps. (1) First, the CPUs sort their data list with respect to the corresponding key values. This can be achieved via an implementation of a $\mathsf{polylog}(m)$-depth sorting network, and provides the useful guarantee that all data pertaining to the same key are necessarily held by an block of adjacent CPUs. (2) Second, we pass data among CPUs in a sequence of $\log(m)$ steps such that at the conclusion the "left-most" (i.e., lowest indexed) CPU in each key-block will learn the aggregation of *all* data pertaining to this key. Explicitly, in each step i, each CPU sends all held information to the CPU 2^i to the "left" of him, and simultaneously accepts any received information pertaining to his key. (3) Third, each CPU will learn whether he is the "left-most" representative in each key-block, by simply checking whether his left-hand neighbor holds the same key. From here, the CPUs have succeeded in aggregating information for each key at a single representative CPU; (4) in the fourth step, they now reverse the original sorting procedure to return this aggregated information to one of the CPUs who originally requested it.

We present the complete protocol $\mathsf{OblivAgg}$ for achieving oblivious aggregation in a space-efficient fashion in Figure 5.

We now proceed to prove the correctness and efficiency of the protocol $\mathsf{OblivAgg}$.

Lemma 3.7 (Space-Efficient Oblivious Aggregation)**.** *Suppose m processors initiate protocol $\mathsf{OblivAgg}$ w.r.t. aggregator Agg, on respective inputs $\{(\mathsf{key}_i, \mathsf{data}_i)\}_{i\in[m]}$, each of size ℓ. Then at the conclusion of execution, each processor $i \in [m]$ outputs a triple $(\mathsf{rep}'_i, \mathsf{key}'_i, \mathsf{data}'_i)$ such that the following properties hold (where asymptotics are w.r.t. m):*

1. *The protocol terminates in $\tilde{O}(1)$ rounds.*
2. *The local memory and computation required by each processor is $\tilde{O}(\ell)$.*
3. *(Correctness). For every key $\mathsf{key} \in K := \bigcup\{\mathsf{key}_i\}$, there exists a unique processor i with output $\mathsf{key}'_i = \mathsf{key}$. For each such processor, it further holds that $\mathsf{key}'_i = \mathsf{key}_i$, $\mathsf{rep}'_i =$ "rep", and $\mathsf{data}'_i = \mathsf{Agg}(\{\mathsf{data}_j : \mathsf{key}_j = \mathsf{key}_i\})$. For every remaining processor, the output tuple is $(\mathsf{dummy}, \perp, \perp)$.*
4. *(Obliviousness). The inter-CPU communication patterns are independent of the inputs $(\mathsf{key}_i, \mathsf{data}_i)$.*

Oblivious Aggregation Procedure OblivAgg **(w.r.t.** Agg**)**
Input: Each CPU $i \in [m]$ holds a pair $(\mathsf{key}_i, \mathsf{data}_i)$.
Output: Each CPU $i \in [m]$ outputs a triple $(\mathsf{rep}_i, \mathsf{key}_i, \mathsf{aggdata}_i)$ corresponding to either $(\mathsf{dummy}, \perp, \perp)$ or with $\mathsf{aggdata}_i = \mathsf{Agg}(\{\mathsf{data}_j : \mathsf{key}_j = \mathsf{key}_i\})$, as further specified in Section 3.3.

1. **Sort on** key_i**.** Each CPU_i initializes a triple $(\mathsf{sourceid}_i, \mathsf{keytemp}_i, \mathsf{datatemp}_i) \leftarrow (i, \mathsf{key}_i, \mathsf{data}_i)$.
   For each layer $L_1, \ldots, L_d$ in the sorting network:
   - Let $L_\ell = ((i_1, j_1), \ldots, (i_{m/2}, j_{m/2}))$ be the comparators in the current layer ℓ.
   - In *parallel*, for each $t \in [m/2]$, the corresponding pair of CPUs (CPU_{i_t}, CPU_{j_t}) perform the following pairwise sort w.r.t. key:
     $$\text{If } \mathsf{keytemp}_{j_t} < \mathsf{keytemp}_{i_t}, \text{ then}$$
     $$\text{swap } (\mathsf{sourceid}_{i_t}, \mathsf{keytemp}_{i_t}, \mathsf{datatemp}_{i_t}) \leftrightarrow (\mathsf{sourceid}_{j_t}, \mathsf{keytemp}_{j_t}, \mathsf{datatemp}_{j_t}).$$
2. **Aggregate to left.** For $t = 0, 1, \ldots, \log m$:
   - (Pass to left). Each CPU_i for $i > 2^t$ sends his current pair $(\mathsf{keytemp}_i, \mathsf{datatemp}_i)$ to CPU_{i-2^t}.
   - (Aggregate). Each CPU_i for $i < m - 2^t$ receiving a pair $(\mathsf{keytemp}_j, \mathsf{datatemp}_j)$ will aggregate it into own pair if the keys match. That is, if $\mathsf{keytemp}_i = \mathsf{keytemp}_j$, then set $\mathsf{datatemp}_i \leftarrow \mathsf{Agg}(\mathsf{datatemp}_i, \mathsf{datatemp}_j)$. In both cases, the received pair is then erased.

   The left-most CPU_i with $\mathsf{keytemp}_i = \mathsf{key}$ now has $\mathsf{Agg}(\{\mathsf{datatemp}_j : \mathsf{keytemp}_j = \mathsf{key}\}))$.
3. **Identify representatives.** For each value key_j, the left-most CPU i currently holding $\mathsf{keytemp}_i = \mathsf{key}_j$ will identify himself as (temporary) representative.
   - Each CPU_i for $i < m$: send $\mathsf{keytemp}_i$ to right-hand neighbor, CPU_{i+1}.
   - Each CPU_i for $i > 1$: If the received value $\mathsf{keytemp}_{i-1}$ matches his own $\mathsf{keytemp}_i$, then set $\mathsf{rep}_i \leftarrow$ "dummy" and zero out $\mathsf{keytemp}_i \leftarrow \perp, \mathsf{datatemp}_i \leftarrow \perp$. Otherwise, set $\mathsf{rep}_i \leftarrow$ "rep". (CPU_1 always sets $\mathsf{rep}_1 \leftarrow$ "rep").
4. **Reverse sort (i.e., sort on** $\mathsf{sourceid}_i$**).** Return aggregated data to a requesting CPU.
   For each layer $L_1, \ldots, L_d$ in the sorting network:
   - Let $L_\ell = ((i_1, j_1), \ldots, (i_{m/2}, j_{m/2}))$ be the comparators in the current layer ℓ.
   - Each CPU_i initializes $\mathsf{idtemp} \leftarrow \mathsf{sourceid}_i$. In *parallel*, for each $t \in [m/2]$, the corresponding pair of CPUs (CPU_{i_t}, CPU_{j_t}) perform the following pairwise sort w.r.t. $\mathsf{sourceid}$:
     $$\text{If } \mathsf{idtemp}_{j_t} < \mathsf{idtemp}_{i_t}, \text{ then}$$
     $$\text{swap } (\mathsf{idtemp}_{i_t}, \mathsf{rep}_{i_t}, \mathsf{keytemp}_{i_t}, \mathsf{datatemp}_{i_t}) \leftrightarrow (\mathsf{idtemp}_{j_t}, \mathsf{rep}_{j_t}, \mathsf{keytemp}_{j_t}, \mathsf{datatemp}_{j_t}).$$

   At the conclusion, each CPU_i holds a tuple with $(\mathsf{idtemp}_i, \mathsf{rep}_i, \mathsf{keytemp}_i, \mathsf{datatemp}_i)$ with $\mathsf{idtemp}_i = i$ and $\mathsf{keytemp}_i = \mathsf{key}_i$.
5. **Output.** Each CPU_i outputs the triple $(\mathsf{rep}_i, \mathsf{key}_i, \mathsf{datatemp}_i)$.

Figure 5: Space-efficient oblivious data aggregation procedure.

Proof. Property (1): Steps 1 and 4 of $\mathsf{OblivAgg}$ each execute a sorting network, and require communication rounds equal to the depth $d \in \tilde{O}(1)$ of the sorting network implemented. Step 2 takes place in $\log m$ sequential steps. Step 3 requires a single round. And Step 5 (output) takes place locally. Thus, the combined round complexity of $\mathsf{OblivAgg}$ is $\tilde{O}(1)$. (Recall that asymptotics are with respect to m).

Property (2): We first address the size the individual items stored, and then ensure the number of stored items is never too large.

- Keys (e.g., $\mathsf{key}_i, \mathsf{tempkey}_i$): Each key is bounded in size by the initial input size ℓ.
- Data (e.g., $\mathsf{data}_i, \mathsf{datatemp}_i, \mathsf{aggdata}_i$): Similarly, by the property of the aggregation function Agg, we are guaranteed that each data item is bounded in size by the original data size, which is in turn bounded by size ℓ.
- CPU identifiers (e.g., $\mathsf{sourceid}_i, \mathsf{idtemp}_i$): Each processor can be identified by bit string of length $\log m$.
- Representative flag (rep_i): The $\mathsf{rep}/\mathsf{dummy}$ flag can be stored as a single bit.

Each processor begins with input size ℓ. In each round of executing the first sorting network (Step 1 of $\mathsf{OblivAgg}$), a processor must hold *two* sets of data $(\mathsf{sourceid}, \mathsf{keytemp}, \mathsf{datatemp})$, corresponding to at most $2(\log m + 2\ell)$ storage. Note that no more than 2 tuples are required to be held at any time within this step, as the processors exchange tuples but need not maintain both values. In each round of the Aggregation phase (Step 2), processors may need to store two pairs $(\mathsf{keytemp}, \mathsf{datatemp})$ in addition to the information held from the conclusion of the previous step (namely, a single value $\mathsf{sourceid}_i$), which totals to $\log m + 2(2\ell)$ memory. Note that by the properties of the aggregation scheme Agg, the size of the aggregated data does not grow beyond ℓ (and recall that parties do not maintain data associated with any different key). In the Representative Identification phase (Step 3), each processor receives one additional key value key_{i-1}, which requires memory $\log m$, and is then translated to a single-bit flag rep_i and then deleted. In the Reverse Sort phase (Step 4), processors within each round must again store two tuples, this time of the form $(\mathsf{idtemp}, \mathsf{rep}, \mathsf{keytemp}, \mathsf{datatemp})$, which corresponds to $2(\log m + 1 + \ell + \ell)$ memory. Thus, the total local memory requirement per processor is bounded by $\tilde{O}(1)$.

Property (3): We now prove that the protocol results in the desired output. Consider the values stored by each processor at the conclusion of each phase of the protocol.

After the completion of Step 1, by the correctness of the utilized sorting network, it holds that each CPU_i holds a tuple $(\mathsf{sourceid}_i, \mathsf{keytemp}_i, \mathsf{datatemp}_i)$ such that the list $(\mathsf{sourceid}_1, \ldots, \mathsf{sourceid}_m)$ is some permutation of $[m]$, and $\mathsf{keytemp}_i \leq \mathsf{keytemp}_j$ for every $i < j$. Note that for each i it always the case that the pair $(\mathsf{keytemp}_i, \mathsf{datatemp}_i)$ currently held by CPU_i is precisely the original input pair of CPU_j for $j = \mathsf{sourceid}_i$.

For the Aggregation phase in Step 2, we make the following claim.

Claim 3.8. *At the conclusion of Aggregate Left (Step 2), the CPU of lowest index i for which $\mathsf{keytemp}_i = \mathsf{key}$ holds $\mathsf{datatemp}_i = \mathsf{Agg}(\{\mathsf{data}_j : \mathsf{key}_j = \mathsf{key}\})$ (for each value key).*

Proof. Fix an arbitrary value key, and let $S_{\mathsf{key}} \subset [m]$ denote the subset of processors for which $\mathsf{keytemp}_i = \mathsf{key}$. From the previous sorting step, we are guaranteed that S_{key} consists of an interval of consecutive processors $i_{\mathsf{start}}, \ldots, i_{\mathsf{stop}}$. Now, consider any $j \in S_{\mathsf{key}}$ (whose data CPU i_{start} wishes to learn).

For any pair of indices $i < j \in S_{\mathsf{key}}$, denote by $t_{i,j} := \max\{t \in [\log m] : (j \oplus i_{\mathsf{start}})_t = 1\} \in \{0, 1, \ldots, \log m - 1\}$ the highest index in which the bit representations of j and i_{start} disagree. We

now prove that for each such pair i, j, CPU_i will learn CPU_j's data after round $t_{i,j} \leq \log m$. The claim will follow, by applying this statement to each pair (i_{start}, j) with $j \in S_{\mathsf{key}}$.

Induct on $t_{i,j}$. Base case $t_{i,j} = 0$: follows immediately from the protocol construction; namely, in the 0-th round, each CPU j sends his data to CPU $(j-1)$, which in this case is precisely CPU i. Now, suppose the inductive hypothesis holds for all $i < j$ with $t_{i,j} = t$, and consider a pair $i < j$ with $t_{i,j} = t+1$. In round $t+1$ of the protocol, processor i receives from processor $(i + 2^{t+1})$ the collection of all information it has aggregated up to round t. By the definition of $t_{i,j}$, we know that $i < (i + 2^{t+1}) \leq j$, and that $t_{(i+2^{t+1}),j} \leq t$. Indeed, we know that i and j differ in bit index $(t+1)$, and no higher; thus, $(i + 2^t)$ must agree with j in index $(t+1)$ in addition to all higher indices. But, this means by the inductive hypothesis that CPU $(i + 2^t)$ has learned CPU j's data in a previous round. Thus, CPU i will learn CPU j's data in round $t+1$, as desired.

□

In Step 3, each processor learns whether his left-hand neighbor holds the same temporary key as he does; that is, he learns whether or not he is the left-most CPU holding $\mathsf{tempkey}_i$ (and, in turn, holds the complete aggregation of all data relating to this key). Each processor for whom this is not the case sets his tuple to $(\mathsf{dummy}, \perp, \perp)$.

At this point in the protocol, the processors have successfully reached the state where a single self-identified representative for each queried key holds the desired data aggregation. The final step is to return these information tuples to some CPU who originally requested this key. This is achieved in the final reverse sort (Step 4). Namely, by the correctness of the implemented sorting network, at the conclusion of Step 4 each CPU_i holds a tuple $(\mathsf{idtemp}_i, \mathsf{rep}_i, \mathsf{keytemp}_i, \mathsf{datatemp}_i)$ such that the ordered list $(\mathsf{idtemp}_1, \ldots, \mathsf{idtemp}_m)$ is precisely the ordered list $1, \ldots, m$. Since the tuples $(\mathsf{idtemp}_i, \mathsf{rep}_i, \mathsf{keytemp}_i, \mathsf{datatemp}_i)$ are never modified (only swapped between processors), it remains to show that each non-dummy $(\mathsf{rep}_i, \mathsf{keytemp}_i, \mathsf{datatemp}_i)$ tuple is received by an appropriate requesting CPU. But, that is precisely the information held by idtemp_i: the identity of the CPU who made the original request with respect to key $\mathsf{keytemp}_i$. Thus, the reverse sort successfully routes the aggregated tuples back to a CPU making the correct key request.

Property (4): Since we utilize a sorting network with fixed topology, and the aggregate-to-left functionality has fixed communication topology, the inter-CPU communication patterns are constant, independent of the initial CPU inputs.

□

3.3.3 Oblivious Multicasting

In this section, we provide a memory/communication-efficient procedure for oblivious multicasting. Our goal here is dual to that of the previous section: Namely, a subset of CPUs must deliver information to (unknown) collections of other CPUs who request it. This is abstractly modeled as follows, where key_i denotes which data item is requested by each CPU i.

Oblivious Multicasting:

Input: Each CPU i holds $(\mathsf{key}_i, \mathsf{data}_i)$ with the following promise. Let $\mathsf{K} = \bigcup\{\mathsf{key}_i\}$ denote the set of distinct keys. For every $\mathsf{key} \in \mathsf{K}$, there exists a unique agent i with $\mathsf{key}_i = \mathsf{key}$ such that $\mathsf{data}_i \neq \perp$; let $\mathsf{data}_{\mathsf{key}}$ denote such data_i.

Goal: Each agent i outputs $\mathsf{out}_i = (\mathsf{key}_i, \mathsf{data}_{\mathsf{key}_i})$.

Oblivious Multicasting Procedure OblivMCast

Input: Each CPU i holds $(\mathsf{key}_i, \mathsf{data}_i)$ with the following promise. Let $\mathsf{K} = \bigcup\{\mathsf{key}_i\}$ denote the set of distinct keys. For every $\mathsf{key} \in \mathsf{K}$, there exists a unique agent i with $\mathsf{key}_i = \mathsf{key}$ such that $\mathsf{data}_i \neq \perp$; let $\mathsf{data}_{\mathsf{key}}$ denote such data_i.

Output: Each agent i outputs $\mathsf{out}_i = (\mathsf{key}_i, \mathsf{data}_{\mathsf{key}_i})$.

1. **Sort on** $(\mathsf{key}_i, \mathsf{data}_i)$**.** Each CPU_i initializes $(\mathsf{sourceid}_i, \mathsf{keytemp}_i, \mathsf{datatemp}_i) \leftarrow (i, \mathsf{key}_i, \mathsf{data}_i)$. For each layer $L_1, \ldots, L_d$ in the sorting network:
   - Let $L_\ell = ((i_1, j_1), \ldots, (i_{m/2}, j_{m/2}))$ be the comparators in the current layer ℓ.
   - In *parallel*, for each $t \in [m/2]$, the corresponding pair of CPUs (CPU_{i_t}, CPU_{j_t}) perform the following pairwise sort w.r.t. key, additionally pushing payloads $\mathsf{data}_{\mathsf{key}}$ to the left:

     If (i) $\mathsf{keytemp}_{j_t} < \mathsf{keytemp}_{i_t}$, or (ii) $\mathsf{keytemp}_{j_t} = \mathsf{keytemp}_{i_t}$ and $\mathsf{datatemp}_{j_t} \neq \perp$, then swap $(\mathsf{sourceid}_{i_t}, \mathsf{keytemp}_{i_t}, \mathsf{datatemp}_{i_t}) \leftrightarrow (\mathsf{sourceid}_{j_t}, \mathsf{keytemp}_{j_t}, \mathsf{datatemp}_{j_t})$.
2. **Multicast to right.** For $t = 0, 1, \ldots, \log m$:
   - (Pass to right). Each CPU_i for $i \leq m - 2^t$ sends his current pair $(\mathsf{keytemp}_i, \mathsf{datatemp}_i)$ to CPU_{i+2^t}.
   - (Aggregate). Each CPU_i for $i > 2^t$ receiving a pair $(\mathsf{keytemp}_j, \mathsf{datatemp}_j)$ with $j = i - 2^t$ update its data as follows. If $\mathsf{keytemp}_i = \mathsf{keytemp}_j$ and $\mathsf{datatemp}_j \neq \perp$, then set $\mathsf{datatemp}_i \leftarrow \mathsf{datatemp}_j$.

   Every CPU i now holds $(\mathsf{keytemp}_i, \mathsf{datatemp}_i) = (\mathsf{key}, \mathsf{data}_{\mathsf{key}})$ for some $\mathsf{key} \in \mathsf{K}$.
3. **Reverse sort (i.e., sort on** $\mathsf{sourceid}_i$**).** Return received data to an original requesting CPU. For each layer $L_1, \ldots, L_d$ in the sorting network:
   - Let $L_\ell = ((i_1, j_1), \ldots, (i_{m/2}, j_{m/2}))$ be the comparators in the current layer ℓ.
   - Each CPU_i initializes $\mathsf{idtemp} \leftarrow \mathsf{sourceid}_i$. In *parallel*, for each $t \in [m/2]$, the corresponding pair of CPUs (CPU_{i_t}, CPU_{j_t}) perform the following pairwise sort w.r.t. $\mathsf{sourceid}$:

     If $\mathsf{idtemp}_{j_t} < \mathsf{idtemp}_{i_t}$, then swap $(\mathsf{idtemp}_{i_t}, \mathsf{keytemp}_{i_t}, \mathsf{datatemp}_{i_t}) \leftrightarrow (\mathsf{idtemp}_{j_t}, \mathsf{keytemp}_{j_t}, \mathsf{datatemp}_{j_t})$.

   At the conclusion, each CPU_i holds a tuple with $(\mathsf{idtemp}_i, \mathsf{keytemp}_i, \mathsf{datatemp}_i)$ with $\mathsf{idtemp}_i = i$, $\mathsf{keytemp}_i = \mathsf{key}_i$, and $\mathsf{datatemp}_i = \mathsf{data}_{\mathsf{key}_i}$.
4. **Output.** Each CPU_i outputs $\mathsf{output}_i = (\mathsf{key}_i, \mathsf{data}_{\mathsf{key}_i})$.

Figure 6: Space-efficient oblivious data multicasting procedure.

We present a protocol $\mathsf{OblivMCast}$ for achieving oblivious multicasting in a space-efficient fashion in Figure 6. This procedure is roughly the "dual" of the $\mathsf{OblivAgg}$ protocol in the previous section. We now proceed to prove the correctness and efficiency of the protocol $\mathsf{OblivMCast}$.

Lemma 3.9 (Space-Efficient Oblivious Multicasting)**.** *Suppose m processors initiate protocol $\mathsf{OblivMCast}$ on respective inputs $\{(\mathsf{key}_i, \mathsf{data}_i)\}_{i\in[m]}$ of size ℓ that satisfies the promise specified above. Then at the conclusion of execution, each processor $i \in [m]$ outputs a pair $(\mathsf{key}'_i, \mathsf{data}'_i)$ such that the following properties hold (where asymptotics are w.r.t. m):*

1. *The protocol terminates in $\tilde{O}(1)$ rounds.*
2. *The local memory and computation required by each processor is $\tilde{O}(\ell)$.*
3. *(Correctness). For every i, $\mathsf{key}'_i = \mathsf{key}_i$, and $\mathsf{data}'_i = \mathsf{data}_{\mathsf{key}_i}$.*
4. *(Obliviousness). The inter-CPU communication patterns are independent of the inputs $(\mathsf{key}_i, \mathsf{data}_i)$.*

Proof. Property (1): Steps 1 and 3 of $\mathsf{OblivMCast}$ each execute a sorting network, and require communication rounds equal to the depth $d \in \tilde{O}(1)$ of the sorting network implemented. Step 2 takes place in $\log m$ sequential steps. And Step 4 (output) takes place locally. Thus, the combined round complexity of $\mathsf{OblivMCast}$ is $\tilde{O}(1)$. (Recall that asymptotics are with respect to m).

Property (2): We first address the size the individual items stored, and then ensure the number of stored items is never too large.

- Keys (e.g., $\mathsf{key}_i, \mathsf{tempkey}_i$): Each key is bounded in size by the initial input size ℓ.
- Data (e.g., $\mathsf{data}_i, \mathsf{datatemp}_i$): Similarly, each data item is bounded by the initial input size ℓ.
- CPU identifiers (e.g., $\mathsf{sourceid}_i, \mathsf{idtemp}_i$): Each processor can be identified by bit string of length $\log m$.

Each processor begins with input size ℓ. In each round of executing the first sorting network (Step 1 of $\mathsf{OblivAgg}$), a processor must hold *two* sets of data $(\mathsf{sourceid}, \mathsf{keytemp}, \mathsf{datatemp})$, corresponding to at most $2(\log m + 2\ell)$ storage. Note that no more than 2 tuples are required to be held at any time within this step, as the processors exchange tuples but need not maintain both values. In each round of the Multicast phase (Step 2), processors may need to store two pairs $(\mathsf{keytemp}, \mathsf{datatemp})$ in addition to the information held from the conclusion of the previous step (namely, a single value $\mathsf{sourceid}_i$), which totals to $\log m + 2(2\ell)$ memory. In the Reverse Sort phase (Step 3), processors within each round must again store two tuples, this time of the form $(\mathsf{idtemp}, \mathsf{keytemp}, \mathsf{datatemp})$, which corresponds to $2(\log m + \ell + \ell)$ memory. Thus, the total local memory requirement per processor is bounded by $\tilde{O}(1)$.

Property (3): We now prove that the protocol results in the desired output. Consider the values stored by each processor at the conclusion of each phase of the protocol.

After the completion of Step 1, by the correctness of the utilized sorting network, it holds that each CPU_i holds a tuple $(\mathsf{sourceid}_i, \mathsf{keytemp}_i, \mathsf{datatemp}_i)$ such that the list $(\mathsf{sourceid}_1, \ldots, \mathsf{sourceid}_m)$ is some permutation of $[m]$, and $\mathsf{keytemp}_i \leq \mathsf{keytemp}_j$ for every $i < j$. Furthermore, for every $\mathsf{key} \in \mathsf{K}$, let i be the smallest index such that $\mathsf{keytemp}_i = \mathsf{key}$. It holds that $\mathsf{datatemp}_i = \mathsf{data}_{\mathsf{key}}$ and $\mathsf{datatemp}_j = \perp$ for every $j > i$ with $\mathsf{keytemp}_j = \mathsf{key}$. Note that for each i it always the case that the pair $(\mathsf{keytemp}_i, \mathsf{datatemp}_i)$ currently held by CPU_i is precisely the original input pair of CPU_j for $j = \mathsf{sourceid}_i$.

For the Multicast phase in Step 2, we make the following claim, asserting that each key have received corresponding data.

Claim 3.10. *At the conclusion of Multicast to Right (Step 2), every CPU i holds* $\mathsf{datatemp}_i = \mathsf{data}_{\mathsf{keytemp}_i}$.

Proof. Fix an arbitrary value key, and let $S_{\mathsf{key}} \subset [m]$ denote the subset of processors for which $\mathsf{keytemp}_i = \mathsf{key}$. From the previous sorting step, we are guaranteed that S_{key} consists of an interval of consecutive processors $i_{\mathsf{start}}, \ldots, i_{\mathsf{stop}}$. Furthermore, we know that $\mathsf{datatemp}_{i_{\mathsf{start}}} = \mathsf{data}_{\mathsf{key}}$ and $\mathsf{datatemp}_i = \perp$ for $i_{\mathsf{start}} < i \leq i_{\mathsf{stop}}$.

We now show by induction that for $t \in \{0, \ldots, \log m + 1\}$, at the beginning of iteration t, $\mathsf{datatemp}_i = \mathsf{data}_{\mathsf{key}}$ for $i_{\mathsf{start}} \leq i \leq \min\{i_{\mathsf{start}+2^t-1}, i_{\mathsf{stop}}\}$. The base case $t = 0$ holds trivially. Suppose the induction holds for t, we show that the induction holds for $t+1$. It suffices to show that for every $i_{\mathsf{start}+2^t} \leq i \leq \min\{i_{\mathsf{start}+2^{t+1}-1}, i_{\mathsf{stop}}\}$, $\mathsf{datatemp}_i$ is set to $\mathsf{data}_{\mathsf{key}}$ in the t-st iteration. Note that for every such i, CPU_i receives $(\mathsf{keytemp}_j, \mathsf{datatemp}_j)$ from CPU_j with $j = i - 2^t$. By the induction hypothesis, $\mathsf{datatemp}_j = \mathsf{data}_{\mathsf{key}} \neq \perp$, and thus $\mathsf{datatemp}_i \leftarrow \mathsf{datatemp}_j = \mathsf{data}_{\mathsf{key}}$. □

The final step is to return these information tuples to some CPU who originally requested this key. This is achieved in the final reverse sort (Step 3). Namely, by the correctness of the implemented sorting network, at the conclusion of Step 4 each CPU_i holds a tuple $(\mathsf{idtemp}_i, \mathsf{keytemp}_i, \mathsf{datatemp}_i)$ such that the ordered list $(\mathsf{idtemp}_1, \ldots, \mathsf{idtemp}_m)$ is precisely the ordered list $1, \ldots, m$. Since the tuples $(\mathsf{idtemp}_i, \mathsf{keytemp}_i, \mathsf{datatemp}_i)$ are never modified (only swapped between processors), it means that each CPU i now holds $\mathsf{keytemp}_i$ being the original key_i. Thus, the reverse sort successfully routes the updated key-data pair back to a CPU making the correct key request.

Property (4): Since we utilize a sorting network with fixed topology, and the multicast-to-right functionality has fixed communication topology, the inter-CPU communication patterns are constant, independent of the initial CPU inputs.

□

3.4 Putting Things Together

In this section, we reduce the CPU local memory size and inter-CPU communication of our rudimentary Heavy-OPAccess solution to $\mathsf{polylog}(n)$ using the space-efficient distributed oblivious procedures from Section 3.3. Let us first recall the steps in Heavy-OPAccess where large memory/bandwidth are required.

- In Step 1., each CPU i broadcasts (r_i, v_i) to all CPUs. Let $b_i = \lfloor r_i/\alpha \rfloor$. This is used to aggregate instructions to each b_i and determine its representative CPU $\mathsf{rep}(b_i)$.
- In Step 4., each CPU i broadcasts (b_i, ℓ_i) or $(\perp, \ell_i)$. This is used to aggregate instructions to each buckets along path ℓ_i about which blocks b_i's to be removed.
- In Step 5., each (representative) CPU i broadcasts $(b_i, \bar{v}_i, \ell'_i)$. This is used to aggregate blocks to be inserted to each bucket in appropriate level of the tree.
- In Step 6., each CPU i broadcasts ℓ_i^{flush}. This is used to aggregate information about which buckets the flush operation should perform.
- In Step 7., each (representative) CPU $\mathsf{rep}(b)$ broadcasts the old value v_{old} of block b to all CPUs so that each CPU receives desired information.

We will use oblivious aggregation procedure to replace broadcasts in Step 1, 4, and 6; the parallel insertion procedure to replace broadcasts in Step 5, and finally the oblivious multicast procedure to replace broadcasts in Step 7.

Let us first consider the aggregation steps. For Step 1., to invoke the oblivious aggregation procedure, we set $\mathsf{key}_i = b_i$ and $\mathsf{data}_i = (r_i \mod \alpha, v_i)$, and define the output of $\mathsf{Agg}(\{(u_i, v_i)\})$ to be a vector $\bar{v} = \bar{v}[1] \cdots \bar{v}[\alpha]$ of read/write instructions to each memory cell in the block, where conflicts are resolved by writing the value specified by the smallest CPU: i.e., $\forall s \in [\alpha]$, take $\bar{v}[s] \leftarrow v_j$ for minimal j such that $u_j = s$ and $v_j \neq \perp$. By the functionality of $\mathsf{OblivAgg}$, at the conclusion of $\mathsf{OblivAgg}$, each block b_i is assigned to a unique representative (not necessarily the smallest CPU), who holds the aggregation of all instructions on this block.

Both Step 4 and 6 invoke $\mathsf{UpdateBuckets}$ to update buckets along m random paths. In our rudimentary solution, the paths (along with instructions) are broadcast among CPUs, and the buckets are updated level by level. At each level, each update bucket is assigned to a representative CPU with minimal index, who performs aggregated instructions to update the bucket. Here, to avoid broadcasts, we invoke the oblivious aggregation procedure per level as follows.

- In Step 4., each CPU i holds a path ℓ_i and a block b_i (or $\perp$) to be removed. Also note that the buckets along the path ℓ_i are stored locally by each CPU i, after the read operation in the previous step (Step 3). At each level $\mathsf{lev} \in [\log n]$, we invoke the oblivious aggregation procedure with $\mathsf{key}_i = \ell_i^{(\mathsf{lev})}$ (the lev-bits prefix of ℓ_i) and $\mathsf{data}_i = b_i$ if b_i is in the bucket of node $\ell_i^{(\mathsf{lev})}$, and $\mathsf{data}_i = \perp$ otherwise. We simply define $\mathsf{Agg}(\{\mathsf{data}_i\}) = \{b : \exists \mathsf{data}_i = b\}$ to be the union of blocks (to be removed from this bucket). Since $\mathsf{data}_i \neq \perp$ only when data_i is in the bucket, the output size of Agg is upper bounded by the bucket size K. By the functionality of $\mathsf{OblivAgg}$, at the conclusion of $\mathsf{OblivAgg}$, each bucket $\ell_i^{(\mathsf{lev})}$ is assigned to a unique representative (not necessarily the smallest CPU) with aggregated instruction on the bucket. Then the representative CPUs can update the corresponding buckets accordingly.
- In Step 6., each CPU i samples a path ℓ_i^{flush} to be flushed and the instructions to each bucket are simply left and right flushes. At each level $\mathsf{lev} \in [\log n]$, we invoke the oblivious aggregation procedure with $\mathsf{key}_i = \ell_i^{\mathsf{flush}(\mathsf{lev})}$ and $\mathsf{data}_i = L$ (resp., R) if the $(\mathsf{lev}+1)$-st bit of ℓ_i^{flush} is 0 (resp., 1). The aggregation function Agg is again the union function. Since there are only two possible instructions, the output has $O(1)$ length. By the functionality of $\mathsf{OblivAgg}$, at the conclusion of $\mathsf{OblivAgg}$, each bucket $\ell_i^{\mathsf{flush}(\mathsf{lev})}$ is assigned to a unique representative (not necessarily the smallest CPU) with aggregated instruction on the bucket. To update a bucket $\ell_i^{\mathsf{flush}(\mathsf{lev})}$, the representative CPU loads the bucket and its two children (if needed) into local memory from the server, performs the flush operation(s) locally, and writes the buckets back.

Note that since we update m *random* paths, we do not need to hide the access pattern, and thus the dummy CPUs do not need to perform dummy operations during $\mathsf{UpdateBuckets}$. A formal description of full-fledged $\mathsf{UpdateBuckets}$ can be found in Figure 7.

For Step 5., we rely on the parallel insertion procedure of Section 3.3.1, which routes blocks to proper destinations within the relevant level of the server-held data tree in parallel using a simple oblivious routing network. The procedure is invoked with $\mathsf{msg}_i = b_i$ and $\mathsf{addr}_i = \ell'_i$.

Finally, in Step 7., each representative CPU $\mathsf{rep}(b)$ holds information of the block b, and each dummy CPU i wants to learn the value of a block b_i. To do so, we invoke the oblivious multicast procedure with $\mathsf{key}_i = b_i$ and $\mathsf{data}_i = v_i^{old}$ for representative CPUs and $\mathsf{data}_i = \perp$ for dummy CPUs. By the functionality of $\mathsf{OblivMCast}$, at the conclusion of $\mathsf{OblivMCast}$, each CPU receives the value of the block it originally wished to learn.

$\mathsf{UpdateBuckets}\,(m, (\mathsf{command}_i, \mathsf{path}_i))$
Let $\mathsf{path}^{(1)}, \mathsf{path}^{(2)}, \ldots, \mathsf{path}^{(\log n)}$ denote the bit prefixes of length 1 to $\log n$ of path.

For each level $\mathsf{lev} = 1, \ldots, \log n$ of the tree:

1. The CPUs invoke the oblivious aggregation procedure $\mathsf{OblivAgg}$ as follows.

   Case 1: $\mathsf{command}_i = \mathsf{remove}\text{-}b_i$.
   Each CPU i sets $\mathsf{key}_i = \mathsf{path}_i^{(\mathsf{lev})}$ and $\mathsf{data}_i = b_i$ if b_i is in the bucket of node $\ell_i^{(\mathsf{lev})}$, and $\mathsf{data}_i = \perp$ otherwise. Use the union function $\mathsf{Agg}(\{\mathsf{data}_i\}) = \{b : \exists \mathsf{data}_i = b\}$ as the aggregation function.

   Case 2: $\mathsf{command}_i = \mathsf{flush}$.
   Each CPU i sets $\mathsf{key}_i = \mathsf{path}_i^{(\mathsf{lev})}$ and $\mathsf{data}_i = L$ (resp., R) if the $(\mathsf{lev}+1)$-st bit of path_i is 0 (resp., 1). Use the union function as the aggregation function.

   At the conclusion of the protocol, each bucket $\mathsf{path}_i^{(\mathsf{lev})}$ is assigned to a representative CPU $\mathsf{bucket\text{-}rep}(\mathsf{path}_i^{(\mathsf{lev})})$ with aggregated commands $\mathsf{agg\text{-}command}_i$.

2. Each representative CPU performs the updates:

   If $i \neq \mathsf{bucket\text{-}rep}(\mathsf{path}_i^{(\mathsf{lev})})$, do nothing. Otherwise:

   Case 1: $\mathsf{command}_i = \mathsf{remove}\text{-}b_i$.
   Remove all blocks $b \in \mathsf{agg\text{-}command}_i$ in the bucket $\mathsf{path}_i^{(\mathsf{lev})}$ by accessing memory bucket $\mathsf{path}_i^{(\mathsf{lev})}$ and rewriting contents.

   Case 2: $\mathsf{command}_i = \mathsf{flush}$.
   Access memory buckets $\mathsf{path}_i^{(\mathsf{lev})}, \mathsf{path}_i^{(\mathsf{lev})}||0, \mathsf{path}_i^{(\mathsf{lev})}||1$, perform flush operation locally according to $\mathsf{agg\text{-}command}_i \subset \{L, R\}$, and write the contents back.
   Specifically, denote the collection of stored data blocks b in $\mathsf{path}_i^{(\mathsf{lev})}$ by $\mathsf{ThisBucket}$. Partition $\mathsf{ThisBucket} = \mathsf{ThisBucket\text{-}L} \cup \mathsf{ThisBucket\text{-}R}$ into those blocks whose associated leaves continue to the left or right (i.e., $\{b_j \in \mathsf{ThisBucket} : \ell_j^{(\mathsf{lev}+1)} = \mathsf{mypath}^{(\mathsf{lev})}||0\}$, and similar for 1).
   - If $L \in \mathsf{agg\text{-}command}_i$, then set $\mathsf{ThisBucket} \leftarrow \mathsf{ThisBucket} \setminus \mathsf{ThisBucket\text{-}L}$, and insert data items $\mathsf{ThisBucket\text{-}L}$ into bucket $\mathsf{path}_i^{(\mathsf{lev})}||0$.
   - If $R \in \mathsf{agg\text{-}command}_i$, then set $\mathsf{ThisBucket} \leftarrow \mathsf{ThisBucket} \setminus \mathsf{ThisBucket\text{-}R}$, and insert data items $\mathsf{ThisBucket\text{-}L}$ into bucket $\mathsf{path}_i^{(\mathsf{lev})}||0$.

Figure 7: A space-efficient implementation of the $\mathsf{UpdateBuckets}$ procedure. Unlike the previous version, here each CPU only holds its own path and command but not that of the other CPUs.

References

[Ajt10] Miklós Ajtai. Oblivious rams without cryptogrpahic assumptions. In *STOC*, pages 181–190, 2010.

[AKS83] 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.

[Bat68] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), pages 307–314, New York, NY, USA, 1968. ACM.

[BCP14] Elette Boyle, Kai-Min Chung, and Rafael Pass. Large-scale secure computation. Cryptology ePrint Archive, Report 2014/404, 2014.

[CKW13] David Cash, Alptekin Küpçü, and Daniel Wichs. Dynamic proofs of retrievability via oblivious ram. In *EUROCRYPT*, pages 279–295, 2013.

[CLP13] Kai-Min Chung, Zhenming Liu, and Rafael Pass. Statistically-secure ORAM with $\tilde{O}(\log^2 n)$ overhead. *CoRR*, 2013.

[CP13] Kai-Min Chung and Rafael Pass. A simple oram. Cryptology ePrint Archive, Report 2013/243, 2013.

[DMN11] Ivan Damgård, Sigurd Meldgaard, and Jesper Buus Nielsen. Perfectly secure oblivious ram without random oracles. In *TCC*, pages 144–163, 2011.

[FDD12] Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *Proceedings of the Seventh ACM Workshop on Scalable Trusted Computing*, STC '12, pages 3–8, New York, NY, USA, 2012. ACM.

[GGH+13] 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*, pages 1–18, 2013.

[GHRW14] Craig Gentry, Shai Halevi, Mariana Raykova, and Daniel Wichs. Outsourcing private ram computation. *IACR Cryptology ePrint Archive*, 2014:148, 2014.

[GKK+11] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure computation with sublinear amortized work. *IACR Cryptology ePrint Archive*, 2011:482, 2011.

[GMOT11] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Oblivious ram simulation with efficient worst-case access overhead. In *CCSW*, pages 95–100, 2011.

[GMOT12] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Privacy-preserving group data access via stateless oblivious ram simulation. In *SODA*, pages 157–167, 2012.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996.

[Gol87] Oded Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *STOC*, pages 182–194, 1987.

[KLO12] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious ram and a new balancing scheme. In *SODA*, pages 143–156, 2012.

[LO13] Steve Lu and Rafail Ostrovsky. Distributed oblivious ram for secure two-party computation. In *TCC*, pages 377–396, 2013.

[LPM+13] Jacob R. Lorch, Bryan Parno, James W. Mickens, Mariana Raykova, and Joshua Schiffman. Shroud: ensuring private access to large-scale data in the data center. In *FAST*, pages 199–214, 2013.

[OS97] Rafail Ostrovsky and Victor Shoup. Private information storage (extended abstract). In *STOC*, pages 294–303, 1997.

[PF79] Nicholas Pippenger and Michael J. Fischer. Relations among complexity measures. *J. ACM*, 26(2):361–381, 1979.

[PR14] Benny Pinkas and Tzachy Reinman. A simple recursive tree oblivious ram. Cryptology ePrint Archive, Report 2014/418, 2014.

[SCSL11] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious ram with o((logn)3) worst-case cost. In *ASIACRYPT*, pages 197–214, 2011.

[SS13] Emil Stefanov and Elaine Shi. Oblivistore: High performance oblivious cloud storage. In *IEEE Symposium on Security and Privacy*, pages 253–267, 2013.

[SvDS+13] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: an extremely simple oblivious ram protocol. In *ACM Conference on Computer and Communications Security*, pages 299–310, 2013.

[WST12] Peter Williams, Radu Sion, and Alin Tomescu. PrivateFS: A parallel oblivious file system. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 977–988, New York, NY, USA, 2012. ACM.