Oblivious Hashing Revisited, and Applications to Asymptotically Efficient ORAM and OPRAM

T.-H. Hubert Chan[1(✉)], Yue Guo[2], Wei-Kai Lin[2], and Elaine Shi[2]

[1] The University of Hong Kong, Pokfulam, Hong Kong
hubert@cs.hku.hk
[2] Cornell University, Ithaca, USA
yg393@cornell.edu, wklin@cs.cornell.edu, elaine@cs.cornell.edu

Abstract. Oblivious RAM (ORAM) is a powerful cryptographic building block that allows a program to provably hide its access patterns to sensitive data. Since the original proposal of ORAM by Goldreich and Ostrovsky, numerous improvements have been made. To date, the best asymptotic overhead achievable for general block sizes is $O(\log^2 N/\log\log N)$, due to an elegant scheme by Kushilevitz et al., which in turn relies on the oblivious Cuckoo hashing scheme by Goodrich and Mitzenmacher.

In this paper, we make the following contributions: we first revisit the prior $O(\log^2 N/\log\log N)$-overhead ORAM result. We demonstrate the somewhat incompleteness of this prior result, due to the subtle incompleteness of a core building block, namely, Goodrich and Mitzenmacher's oblivious Cuckoo hashing scheme.

Even though we do show how to patch the prior result such that we can fully realize Goodrich and Mitzenmacher's elegant blueprint for oblivious Cuckoo hashing, it is clear that the extreme complexity of oblivious Cuckoo hashing has made understanding, implementation, and proofs difficult. We show that there is a conceptually simple $O(\log^2 N/\log\log N)$-overhead ORAM that dispenses with oblivious Cuckoo hashing entirely.

We show that such a conceptually simple scheme lends to further extensions. Specifically, we obtain the first $O(\log^2 N/\log\log N)$ Oblivious Parallel RAM (OPRAM) scheme, thus not only matching the performance of the best known sequential ORAM, but also achieving superlogarithmic improvements in comparison with known OPRAM schemes.

Keywords: Oblivious RAM · Oblivious PRAM

1 Introduction

Oblivious RAM [19,20,37], originally proposed in the seminal work by Goldreich and Ostrovsky [19,20], is a powerful cryptographic primitive that provably obfuscates a program's access patterns to sensitive data. Since Goldreich

---

The full version of this paper is available on Cryptology ePrint Archive [7].

and Ostrovsky's original work [19,20], numerous subsequent works have proposed improved constructions, and demonstrated a variety of ORAM applications in both theoretical contexts (e.g., multiparty computation [23,27], Garbled RAMs [18,28]) as well as in secure hardware and software systems (e.g., secure processors [15,16,29,36], and cloud outsourcing [22,35,38,39,43]). To hide access patterns, an ORAM scheme typically involves reading, writing, or shuffling multiple blocks for every data request. Suppose that on average, for each data request, an ORAM scheme must read/write X blocks. In this paper, we refer to X as the overhead (or the total work blowup) of the ORAM scheme.

Goldreich and Ostrovsky [19,20] showed that, roughly speaking, any "natural" ORAM scheme that treats each block as an "opaque ball" must necessarily suffer from at least logarithmic overhead. The recent Circuit ORAM [41] work demonstrated an almost matching upper bound for large enough blocks. Let N denote the total memory size. Circuit ORAM showed the existence of a *statistically* secure ORAM scheme that achieves $O(\alpha \log N)$ overhead for N^ϵ-bit blocks for any constant $\epsilon > 0$ and any super-constant function $\alpha = \omega(1)$. To date, the existence of an almost logarithmic ORAM scheme is only known for large blocks. For general block sizes, the state of affairs is different: the best known construction (asymptotically speaking) is a *computationally* secure scheme by Kushilevitz et al. [26], which achieves $O(\frac{\log^2 N}{\log \log N})$ overhead assuming block sizes of $\Omega(\log N)$[1]. We note that all known ORAM schemes assume that a memory block is at least large enough to store its own address, i.e., at least $\Omega(\log N)$ bits long. Therefore, henceforth in this paper, we use the term "general block size" to refer to a block size of $\Omega(\log N)$.

Although most practical ORAM implementations (in the contexts of secure multi-party computation, secure processors, and storage outsourcing) opted for tree-based ORAM constructions [37,40,41] due to tighter practical constants, we note that hierarchical ORAMs are nonetheless of much theoretical interest: for example, when the CPU has $O(\sqrt{N})$ private cache, hierarchical ORAMs can achieve $O(\log N)$ simulation overhead while a comparable result is not known in the tree-based framework. Recent works [3,8] have also shown how hierarchical ORAMs can achieve asymptotically better locality and IO performance than known tree-based approaches.

Our contributions. In this paper, we make the following contributions:

– **Revisit** $O(\log^2 N/\log \log N)$ **ORAMs.** We revisit how to construct a *computationally* secure ORAM with $O(\frac{\log^2 N}{\log \log N})$ overhead for general block sizes. First, we show why earlier results along this front [22,26] are somewhat incomplete due to the incompleteness of a core building block, oblivious Cuckoo hashing, that is proposed and described by Goodrich and Mitzenmacher [22]. Next, besides fixing and restating the earlier results regarding the existence

---

[1] This $O(\frac{\log^2 N}{\log \log N})$ result for computational security was later matched in the tree-based ORAM framework [9,14] although tree-based ORAMs were initially investigated for the case of statistical security.

of an $O(\log^2 N/\log\log N)$ ORAM, perhaps more compellingly, we show how to obtain an ORAM with the same asymptotic overhead, but in a conceptually much simpler manner, completely obviating the need to perform oblivious Cuckoo hashing [22] which is the center of complexity in the earlier result [26].

– **New results on efficient OPRAMs.** Building on our new ORAM scheme, we next present the first Oblivious Parallel RAM (OPRAM) construction that achieves $O(\frac{\log^2 N}{\log\log N})$ simulation overhead. To the best of our knowledge, our OPRAM scheme is the first one to asymptotically match the best known sequential ORAM scheme for general block sizes. Moreover, we achieve a superlogarithmic factor improvement over earlier works [5,10] and over the concurrent work by Nayak and Katz [31] (see further clarifications in Sect. 1.3).

We stress that our conceptual simplicity and modular approach can open the door for possible improvements. For example, our OPRAM results clearly demonstrate the benefits of having a conceptually clean hierarchical ORAM framework: had we tried to make (a corrected variant of) Kushilevitz et al. [26] into an OPRAM, it is not clear whether we could have obtained the same performance. In particular, achieving $O(\log^2 N/\log\log N)$ worst-case simulation overhead requires deamortizing a parallel version of their oblivious cuckoo hash rebuilding algorithm, and moreover, work and depth have to be deamortized at the same time—and we are not aware of a way to do this especially due to the complexity of their algorithm.

1.1   Background on Oblivious Hashing and Hierarchical ORAMs

In this paper, we consider the hierarchical framework, originally proposed by Goldreich and Ostrovsky [19,20], for constructing ORAM schemes. At a high level, this framework constructs an ORAM scheme by having exponentially growing levels of capacity $1, 2, 4, \ldots, N$ respectively, where each smaller level can be regarded as a "stash" for larger levels. Each level in the hierarchy is realized through a core abstraction henceforth called *oblivious hashing* in the remainder of this paper. Since oblivious hashing is the core abstraction we care about, we begin by explicitly formulating oblivious hashing as the following problem:

– *Functional abstraction.* Given an array containing n possibly dummy elements where each non-dummy element is a (key, value) pair, design an efficient algorithm that builds a hash table data structure, such that after the building phase, each element can be looked up by its key consuming a small amount of time and work. In this paper, we will assume that all non-dummy elements in the input array have distinct keys.

– *Obliviousness.* The memory access patterns of both the building and lookup phases do not leak any information (to a computationally bounded adversary) about the initial array or the sequence of lookup queries Q—as long as *all non-dummy queries in Q are distinct.* In particular, obliviousness must hold even when Q may contain queries for elements *not contained* in the array in which case the query should return the result $\perp$. The correct answer to a dummy query is also $\perp$ by convention.

Not surprisingly, the performance of a hierarchical ORAM crucially depends on the core building block, oblivious hashing. Here is the extent of our knowledge about oblivious hashing so far:

- Goldreich and Ostrovsky [19, 20] show an oblivious variant of normal balls-and-bins hashing that randomly throws n elements into n bins. They show that obliviously building a hash table containing n elements costs $O(\alpha n \log n \log \lambda)$ work, and each query costs $O(\alpha \log \lambda)$ work. If α is any super-constant function, we can attain a failure probability $\mathsf{negl}(\lambda)$. This leads to an $O(\alpha \log^3 N)$-overhead ORAM scheme, where N is the total memory size[2].
- Subsequently, Goodrich and Mitzenmacher [22] show that the Cuckoo hashing algorithm can be made oblivious, incurring $O(n \log n)$ total work for building a hash table containing n elements, and only $O(1)$ query cost (later we will argue why their oblivious hashing scheme is somewhat incomplete). This leads to an ORAM scheme with $O(\log^2 N)$-overhead.
- Kushilevitz et al. [26] in turn showed an elegant reparametrization trick atop the Goodrich and Mitzenmacher ORAM, thus improving the overhead to $O(\frac{\log^2 N}{\log \log N})$. Since Kushilevitz et al. [26] crucially rely on Goodrich and Mitzenmacher's oblivious Cuckoo hashing scheme, incompleteness of the hashing result in some sense carries over to their $O(\frac{\log^2 N}{\log \log N})$ overhead ORAM construction.

1.2   Technical Roadmap

Revisit oblivious Cuckoo hashing. Goodrich and Mitzenmacher [22]'s blueprint for obliviously building a Cuckoo hash table is insightful and elegant. They express the task of Cuckoo hash table rebuilding as a MapReduce task (with certain nice properties), and they show that any such MapReduce algorithm has an efficient oblivious instantiation.

Fundamentally, their construction boils down using a sequence of oblivious sorts over arrays of (roughly) exponentially decreasing lengths. To achieve full privacy, it is necessary to hide the true lengths of these arrays during the course of the algorithm. Here, Goodrich and Mitzenmacher's scheme description and their proof appear inconsistent: their scheme seems to suggest padding each array to the maximum possible length for security—however, this would make their scheme $O(\log^3 N)$ overhead rather than the claimed $O(\log^2 N)$. On the other hand, their proof appears only to be applicable, if the algorithm reveals the true lengths of the arrays—however, as we argue in detail in the online full version [7], the array lengths in the cuckoo hash rebuilding algorithm contain information about the size of each connected component in the cuckoo graph.

---

[2] Henceforth in this paper, we use n to denote the size of a hash table and λ to denote its security parameter. For our ORAM construction, we use N to denote both the logical memory size as well as the ORAM's security parameter. This distinction is necessary since the ORAM will employ hash tables of varying n.

Thus leaking array lengths can lead to an explicit attack that succeeds with non-negligible probability: at a high level, this attack tries to distinguish two request sequences, one repeatedly requesting the same block whereas the other requests disctinct blocks. The latter request sequence will cause the cuckoo graph in the access phase to resemble the cuckoo graph in the rebuild phase, whereas the former request sequence results in a fresh random cuckoo hash graph for the access phase (whose connected component sizes are different than the rebuild phase with relatively high probability).

As metioned earlier, the incompleteness of oblivious Cuckoo hashing also makes the existence proof of an $O(\log^2 N/\log\log N)$-overhead ORAM somewhat incomplete.

Is oblivious Cuckoo hashing necessary for efficient hierarchical ORAM? Goodrich and Mitzenmacher's oblivious Cuckoo hashing scheme is extremely complicated. Although we do show in our online full version [7] that the incompleteness of Goodrich and Mitzemacher's construction and proofs can be patched, thus correctly and fully realizing the elegant blueprint they had in mind—the resulting scheme nonetheless suffers from large constant factors, and is unsuitable for practical implementation. Therefore, a natural question is, *can we build efficient hierarchical ORAMs without oblivious Cuckoo hashing?*

Our first insight is that perhaps oblivious Cuckoo hashing scheme is an overkill for constructing efficient hierarchical ORAMs after all. As initial evidence, we now present an almost trivial modification of the original Goldreich and Ostrovsky oblivious balls-and-bins hashing scheme such that we can achieve an $O(\alpha\log^2 N)$-overhead ORAM for any super-constant function α.

Recall that Goldreich and Ostrovsky [19,20] perform hashing by hashing n elements into n bins, each of $O(\alpha\log\lambda)$ capacity, where λ is the security parameter. A simple observation is the following: instead of having n bins, we can have $\frac{n}{\alpha\log\lambda}$ bins—it is not hard to show that each bin's occupancy will still be upper bounded by $O(\alpha\log\lambda)$ except with $\mathsf{negl}(\lambda)$ probability. In this way, we reduce the size of the hash table by a $\log\lambda$ factor, and thus the hash table can be obliviously rebuilt in logarithmically less time. Plugging in this new hash table into Goldreich and Ostrovsky's ORAM construction [19,20], we immediately obtain an ORAM scheme with $O(\alpha\log^2 N)$ overhead.

This shows that through a very simple construction we can almost match Goodrich and Mitzenmacher's ORAM result [22]. This simple scheme does not quite get us to where we aimed to be, but we will next show that oblivious Cuckoo hashing is likewise an overkill for constructing $(\frac{\log^2 N}{\log\log N})$-overhead ORAMs.

Conceptually simple $(\frac{\log^2 N}{\log\log N})$-overhead ORAM. Recall that a hierarchical ORAM's overhead is impacted by two cost metrics of the underlying oblivious hashing scheme, i.e., the cost of building the hash-table, and the cost of each lookup query. Goodrich and Mitzenmacher's oblivious Cuckoo hashing scheme [22] minimizes the lookup cost to $O(1)$, but this complicates the building of the hash-table.

Our key insight is that in all known hashing-based hierarchical ORAM constructions [19, 20, 22, 26], the resulting ORAM's cost is dominated by the hashtable rebuilding phase, and thus it may be okay if the underlying hashing scheme is more expensive in lookup. More specifically, to obtain an $O(\frac{\log^2 N}{\log \log N})$ ORAM, we would like to apply Kushilevitz et al. [26]'s reparametrized version of the hierarchical ORAM. Kushilevitz et al. [26] showed that their reparametrization technique works when applied over an oblivious Cuckoo hashing scheme. We observe that in fact, Kushilevitz et al. [26]'s reparametrization technique is applicable for a much broader parameter range, and concretely for any oblivious hashing scheme with the following characteristics:

– It takes $O(n \log n)$ total work to build a hash table of n elements—in other words, the per-element building cost is $O(\log n)$.
– The lookup cost is asymptotically smaller than the per-element building cost—specifically, $O(\log^\epsilon \lambda)$ lookup cost suffices where $\epsilon \in (0.5, 1)$ is a suitable constant.

This key observation allows us to relax the lookup time on the underlying oblivious hashing scheme. We thus propose a suitable oblivious hashing scheme that is conceptually simple. More specifically, our starting point is a (variant of a) two-tier hashing scheme first described in the elegant work by Adler et al. [1]. In a two-tier hashing scheme, there are two hash tables denoted H_1 and H_2 respectively, each with $\frac{n}{\log^\epsilon \lambda}$ bins of $O(\log^\epsilon \lambda)$ capacity, where $\epsilon \in (0.5, 1)$ is a suitable constant. To hash n elements (non-obliviously), we first throw each element into a random bin in H_1. For all the elements that overflow its bin capacity, we throw them again into the second hash table H_2. Stochastic bounds show that the second hash table H_2 does not overflow except with $\mathsf{negl}(\lambda)$ probability. Clearly, the lookup cost is $O(\log^\epsilon \lambda)$; and we will show that the hash table building algorithm can be made oblivious through $O(1)$ number of oblivious sorts.

New results on oblivious parallel RAM. The conceptual simplicity of our ORAM scheme not only makes it easier to understand and implement, but also lends to further extensions. In particular, we construct a computationally secure OPRAM scheme that has $O(\log^2 N/\log \log N)$ overhead—to the best of our knowledge, this is the first OPRAM scheme that matches the best known sequential ORAM in performance for general block sizes. Concretely, the hierarchical lookup phase can be parallelized using the standard conflict resolution (proposed by Boyle et al. [5]) as this phase is read-only. In the rebuild phase, our two-tier oblivious hashing takes only $O(1)$ number of oblivious sort and linear scan that marks excess elements, which can be parallelized with known algorithms, i.e. range prefix sum.

As mentioned earlier, our modular approach and conceptual simplicity turned out to be a crucial reason why we could turn our ORAM scheme into an OPRAM—it is not clear whether (a corrected version of) Kushilevitz et al. [26] is amenable to the same kind of transformation achieving the same overhead due to complications in deamortizing their cuckoo hash rebuilding algorithm. Thus we argue that our conceptually simple framework can potentially lend to other possible applications and improvements.

1.3   Related Work

ORAMs. ORAM was first proposed in a seminal work by Goldreich and Ostrovsky [19,20] who showed a computationally secure scheme with $O(\alpha \log^3 N)$ overhead for general block sizes and for any super-constant function $\alpha = \omega(1)$. Subsequent works improve the hierarchical ORAM [22,26] and show that $O(\frac{\log^2 N}{\log \log N})$ overhead can be attained under computational security—our paper points out several subtleties and the incompleteness of the prior results; additionally, we show that it is possible to obtain such an $O(\frac{\log^2 N}{\log \log N})$ overhead in a conceptually much simpler manner.

Besides the hierarchical framework, Shi et al. [37] propose a tree-based paradigm for constructing ORAMs. Numerous subsequent works [11,40,41] improved tree-based constructions. With the exception of a few works [14], the tree-based framework was primarily considered for the construction of *statistically secure* ORAMs. The performance of tree-based ORAMs depend on the block size, since with a larger block size we can reduce the number of recursion levels in these constructions. The recent Circuit ORAM work [41] shows that under block sizes as large as N^ϵ for any arbitrarily small constant ϵ, we can achieve $\alpha \log N$ bandwidth overhead for an arbitrary super-constant function $\alpha = \omega(1)$—this also shows the (near) tightness of the Goldreich-Ostrovsky lower bound [19,20] showing that any ORAM scheme must necessarily incur logarithmic overhead. Note that under block sizes of at least $\log^{1+\epsilon} N$ for an arbitrarily small constant ϵ, Circuit ORAM [41] can also attain $O(\frac{\log^2 N}{\log \log N})$ overhead and it additionally achieves statistical security rather than computational.

OPRAMs. Since modern computing architectures such as cloud platforms and multi-core architectures exhibit a high degree of parallelism, it makes sense to consider the parallel counterpart of ORAM. Oblivious Parallel ORAM (OPRAM) was first proposed by Boyle et al. [5], who showed a construction with $O(\alpha \log^4 N)$ overhead for any super-constant function α. Boyle et al.'s result was later improved by Chen et al. [10], who showed how to achieve $O(\alpha \log^3 N)$ overhead with poly-logarithmic CPU private cache—their result also easily implies an $O(\alpha \log^3 N \log \log N)$ overhead OPRAM with $O(1)$ CPU private cache, the setting that we focus on in this paper for generality.

A concurrent and independent manuscript by Nayak et al. [31] further improves the CPU-memory communication by extending Chen et al.'s OPRAM [10]. However, their scheme still requires $O(\alpha \log^3 N \log \log N)$ CPU-CPU communication which was the dominant part of the overhead in Chen et al. [10]. Therefore, under a general notion of overhead that includes both CPU-CPU communication and CPU-memory communication, Nayak et al.'s scheme still has the same asymptotic overhead[3] as Chen et al. [10] which is more than a logarithmic factor more expensive in comparison with our new OPRAM construction.

---

[3] The title of their paper [31] suggests $O(\log^2 N)$ overhead, since they did not account for the cost of CPU-CPU communication when describing the overhead.

In a companion paper, Chan et al. [9] showed how to obtain statistically secure and computationally secure OPRAMs in the tree-based ORAM framework. Specifically, they showed that for general block sizes, we can achieve statistically secure OPRAM with $O(\log^2 N)$ simulation overhead and computationally secure OPRAM with $O(\log^2 N/\log\log N)$ simulation overhead. For the computationally secure setting, Chan et al. [9] achieves the same asymptotical overhead as this paper, but the two constructions follow different paradigms so we believe that they are both of value. In another recent work, Chan et al. [6] proposed a new notion of depth for OPRAMs where the OPRAM is allowed to have more CPUs than the original PRAM to further parallelize the computation. In this paper, an OPRAM's simulation overhead is defined as its runtime blowup assuming that the OPRAM consumes the same number of CPUs as the PRAM.

Non-oblivious techniques for hashing. Many hashing schemes [4,12,17,25, 30] were considered in the (parallel) algorithms literature. Unfortunately, most of them are *not* good candidates for constructing efficient ORAM and OPRAM schemes since there is no known efficient and oblivious counterpart for the algorithm. We defer detailed discussions of these related works to our online full version [7].

2   Definitions and Building Blocks

2.1   Parallel Random Access Machines

We define a Parallel Random Access Machine (PRAM) and an Oblivious Parallel Random Access Machine (OPRAM) in a similar fashion as Boyle et al. [5] as well as Chan and Shi [9]. Some of the definitions in this section are borrowed verbatim from Boyle et al. [5] or Chan and Shi [9].

Although we give definitions only for the parallel case, we point out that this is without loss of generality, since a sequential RAM can be thought of as a special-case PRAM.

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

We support a more general PRAM model where the number of CPUs in each time step may vary. Specifically, in each step $t \in [T]$, we use m_t to denote the number of CPUs. In each step, each CPU executes a next instruction circuit denoted Π, updates its CPU state; and further, CPUs interact with memory through request instructions $\boldsymbol{I}^{(t)} := (I_i^{(t)} : i \in [m_t])$. Specifically, at time step t, CPU i's instruction is of the form $I_i^{(t)} := (\mathtt{read}, \mathtt{addr})$, or $I_i^{(t)} := (\mathtt{write}, \mathtt{addr}, \mathtt{data})$ where the operation is performed on the memory block with address addr and the block content $\mathtt{data} \in \{0,1\}^D \cup \{\bot\}$.

If $I_i^{(t)} = (\mathtt{read}, \mathtt{addr})$ then the CPU i should receive the contents of $\mathsf{mem}[\mathtt{addr}]$ at the beginning of time step t. Else if $I_i^{(t)} = (\mathtt{write}, \mathtt{addr}, \mathtt{data})$, CPU i should still receive the contents of $\mathsf{mem}[\mathtt{addr}]$ at the beginning of time step t; further, at the end of step t, the contents of $\mathsf{mem}[\mathtt{addr}]$ should be updated to $\mathtt{data}$.

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

– The original PRAM supports concurrent reads and concurrent writes (CRCW) with an arbitary, parametrizable rule for write conflict resolution. In other words, there exists some priority rule to determine which $\mathtt{write}$ operation takes effect if there are multiple concurrent writes in some time step t.
– Our compiled, oblivious PRAM (defined below) is a "concurrent read, exclusive write" PRAM (CREW). In other words, our OPRAM algorithm must ensure that there are no concurrent writes at any time.

   We note that a CRCW-PRAM with a parametrizable conflict resolution rule is among the most powerful CRCW-PRAM model, whereas CREW is a much weaker model. Our results are stronger if we allow the underlying PRAM to be more powerful but the our compiled OPRAM uses a weaker PRAM model. For a detailed explanation on how stronger PRAM models can emulate weaker ones, we refer the reader to the work by Hagerup [24].

CPU-to-CPU communication. In the remainder of the paper, we sometimes describe our algorithms using CPU-to-CPU communication. For our OPRAM algorithm to be oblivious, the inter-CPU communication pattern must be oblivious too. We stress that such inter-CPU communication can be emulated using shared memory reads and writes. Therefore, when we express our performance metrics, we assume that all inter-CPU communication is implemented with shared memory reads and writes. In this sense, our performance metrics already account for any inter-CPU communication, and there is no need to have separate metrics that characterize inter-CPU communication. In contrast, some earlier works [10] adopt separate metrics for inter-CPU communication.

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

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

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

Finally, in this paper, we consider PRAMs that are *stateful* and can evaluate a sequence of inputs, carrying state across in between. Without loss of generality, we assume each input can be stored in a single memory block.

2.2 Oblivious Parallel Random-Access Machines

Randomized PRAM. A *randomized PRAM* is a special PRAM where the CPUs are allowed to generate private random numbers. For simplicity, we assume that a randomized PRAM has a priori known, deterministic runtime, and that the CPU activation pattern in each time step is also fixed a priori and publicly known.

Memory access patterns. Given a PRAM program denoted PRAM and a sequence of inputs $(\mathsf{inp}_1, \ldots, \mathsf{inp}_d)$, we define the notation Addresses[PRAM] $(\mathsf{inp}_1, \ldots, \mathsf{inp}_d)$ as follows:

- Let T be the total number of parallel steps that PRAM takes to evaluate inputs $(\mathsf{inp}_1, \ldots, \mathsf{inp}_d)$.
- Let $A_t := \left\{ (\mathsf{cpu}_1^t, \mathsf{addr}_1^t), (\mathsf{cpu}_2^t, \mathsf{addr}_2^t) \ldots, (\mathsf{cpu}_{m_t}^t, \mathsf{addr}_{m_t}^t) \right\}$ be the list of (CPU id, address) pairs such that cpu_i^t accessed memory address addr_i^t in time step t.
- We define Addresses[PRAM]$(\mathsf{inp}_1, \ldots, \mathsf{inp}_d)$ to be the random variable $[A_t]_{t \in [T]}$.

Oblivious PRAM (OPRAM). A randomized PRAM is said to be computationally *oblivious*, iff there exists a p.p.t. simulator Sim, and a negligible function $\epsilon(\cdot)$ such that for any input sequence $(\mathsf{inp}_1, \ldots, \mathsf{inp}_d)$ where $\mathsf{inp}_i \in \{0,1\}^D$ for $i \in [d]$,

$$\mathsf{Addresses}[\mathsf{PRAM}](\mathsf{inp}_1, \ldots, \mathsf{inp}_d) \overset{\epsilon(N)}{\approx} \mathsf{Sim}(1^N, d, T, (P_t : t \in [T]))$$

where $\overset{\epsilon(N)}{\approx}$ means that no p.p.t. adversary can distinguish the two probability ensembles except with $\epsilon(N)$ probability.

In other words, obliviousness requires that there is a polynomial-time simulator Sim that can simulate the memory access patterns knowing only the memory size N, the number of inputs d, the parallel runtime T for evaluating the inputs, as well as the a-priori fixed CPU activation pattern $(P_t : t \in [T])$. In particular, the simulator Sim does not know anything about the sequence of inputs.

Oblivious simulation and simulation overhead. We say that a oblivious PRAM, denoted as OPRAM, *simulates* a PRAM if for every input sequence $(\mathsf{inp}_1, \ldots, \mathsf{inp}_d)$, OPRAM$(\mathsf{inp}_1, \ldots, \mathsf{inp}_d)$ = PRAM$(\mathsf{inp}_1, \ldots, \mathsf{inp}_d)$, i.e., OPRAM and PRAM output the same outcomes on any input sequence. In addition, an *OPRAM scheme* is a randomized PRAM algorithm such that, given any PRAM, the scheme compiles PRAM into an oblivious PRAM, OPRAM, that simulates PRAM.

For convenience, we often adopt two intermediate metrics in our descriptions, namely, *total work blowup* and *parallel runtime blowup*. We say that an OPRAM scheme has a total work blowup of x and a parallel runtime blowup of y, iff for every PRAM step t in which the PRAM consumes m_t CPUs, the OPRAM can complete this step with $x \cdot m_t$ total work and in y parallel steps—if the OPRAM is allowed to consume any number of CPUs (possibly greater than m_t).

Fact 1. *If there exists an OPRAM with x total work blowup and y parallel runtime blowup such that $x \geq y$, then there exists an OPRAM that has $O(x)$ simulation overhead when consuming the same number of CPUs as the orginal PRAM for simulating at PRAM step.*

In the interest of space, we defer the proof of this simple fact to the online full version [7].

2.3   Oblivious Hashing Scheme

Without loss of generality, we define only the parallel version, since the sequential version can be thought of the parallel version subject to executing on a single CPU.

A parallel oblivious hashing scheme contains the following two parallel, possibly randomized algorithms to be executed on a *Concurrent Read, Exclusive Write* PRAM:

- $\mathsf{T} \leftarrow \mathsf{Build}(1^\lambda, \{(k_i, v_i) \mid \mathsf{dummy}\}_{i \in [n]})$: given a security parameter 1^λ, and a set of n elements, where each element is either a dummy denoted dummy or a (key, value) pair denoted (k_i, v_i), the Build algorithm outputs a memory data structure denoted T that will later facilitate query. For an input array $S := \{(k_i, v_i) \mid \mathsf{dummy}\}_{i \in [n]}$ to be *valid*, we require that any two non-dummy elements in S must have distinct keys.
- $v \leftarrow \mathsf{Lookup}(\mathsf{T}, k)$: takes in the data structure T and a (possibly dummy) query k, outputs a value v.

Correctness. Correctness is defined in a natural manner: given a valid initial set $S := \{(k_i, v_i) \mid \mathsf{dummy}\}_{i \in [n]}$ and a query k, we say that v is the correct answer for k with respect to S, iff

- If $k = \mathsf{dummy}$ (i.e., if k is a dummy query) or if $k \notin S$, then $v = \perp$.
- Else, it must hold that $(k, v) \in S$.

More informally, the answer to any dummy query must be $\perp$; if a query searches for an element non-existent in S, then the answer must be $\perp$. Otherwise, the answer returned must be consistent with the initial set S.

We say that a parallel oblivious hashing scheme is correct, if for any valid initial set S, for any query k, and for all λ, it holds that

$$\Pr\left[\mathsf{T} \leftarrow \mathsf{Build}(1^\lambda, S), v \leftarrow \mathsf{Lookup}(\mathsf{T}, k) : v \text{ is correct for } k \text{ w.r.t. } S\right] = 1$$

where the probability space is taken over the random coins chosen by the Build and Lookup algorithms.

Obliviousness. A query sequence $\mathbf{k} = (k_1, \ldots, k_j)$ is said to be *non-recurrent*, if all non-dummy queries in $\mathbf{k}$ are distinct.

A parallel hashing scheme denoted (Build, Lookup) is said to be *oblivious*, if there exists a polynomial-time simulator Sim, such that for any security parameter λ, for any valid initial set S, for any *non-recurrent* query sequence $\mathbf{k} := (k_1, \ldots, k_j)$ of polynomial length, it holds that

$$\mathsf{Addresses}[\mathsf{Build}, \mathsf{Lookup}](1^\lambda, S, \mathbf{k}) \stackrel{c}{\equiv} \mathsf{Sim}(1^\lambda, |S|, |\mathbf{k}|)$$

where $\stackrel{c}{\equiv}$ denotes computationally indistinguishability, i.e., a computationally bounded adversary can distinguish between the two distributions with an advantage at most $\mathsf{negl}(\lambda)$. Intuitively, this security definition says that a simulator, knowing only the length of the input set and the number of queries, can simulate the memory access patterns.

Definition 1 (($W_{\mathrm{build}}, T_{\mathrm{build}}, W_{\mathrm{lookup}}, T_{\mathrm{lookup}}$)-parallel oblivious hashing scheme). *Let $W_{\mathrm{build}}(\cdot, \cdot)$, $W_{\mathrm{lookup}}(\cdot, \cdot)$, $T_{\mathrm{build}}(\cdot, \cdot)$, and $T_{\mathrm{lookup}}(\cdot, \cdot)$ be functions in n and λ. We say that (Build, Lookup) is a ($W_{\mathrm{build}}, T_{\mathrm{build}}, W_{\mathrm{lookup}}, T_{\mathrm{lookup}}$)-parallel oblivious hashing scheme, iff (Build, Lookup) satisfies correctness and obliviousness as defined above; and moreover, the scheme achieves the following performance:*

- *Building a hash table with n elements takes $n \cdot W_{\mathrm{build}}(n, \lambda)$ total work and $T_{\mathrm{build}}(n, \lambda)$ time with all but $\mathsf{negl}(\lambda)$ probability. Note that $W_{\mathrm{build}}(n, \lambda)$ is the per-element amount of work required for preprocessing.*
- *A lookup query takes $W_{\mathrm{lookup}}(n, \lambda)$ total work and $T_{\mathrm{lookup}}(n, \lambda)$ time.*

As a special case, we say that (Build, Lookup) is a ($W_{\mathrm{build}}, W_{\mathrm{lookup}}$)-oblivious hashing scheme, if it is a ($W_{\mathrm{build}}, \_, W_{\mathrm{lookup}}, \_$)-parallel oblivious hashing scheme for any choice of the wildcard field "$\_$"—in other words, in the sequential case, we do not care about the scheme's parallel runtime, and the scheme's total work is equivalent to the runtime when running on a single CPU.

[Read-only lookup assumption.] When used in ORAM, observe that elements are inserted in a hash table in a batch only in the Build algorithm. Moreover, we will assume that the Lookup algorithm is read-only, i.e., it does not update the hash table data structure T, and no state is carried across between multiple invocations of Lookup.

A note on the security parameter. Since later in our application, we will need to apply oblivious hashing to different choices of n (including possibly small choices of n), throughout the description of the oblivious hashing scheme, we distinguish the security parameter denoted λ and the size of the set to be hashed denoted n.

2.4   Building Blocks

Duplicate suppression. Informally, duplicate suppression is the following building block: given an input array X of length n consisting of (key, value) pairs and possibly dummy elements where each key can have multiple occurrences, and additionally, given an upper bound n' on the number of distinct keys in X, the algorithm outputs a duplicate-suppressed array of length n' where only one occurrence of each key is preserved, and a preference function priority is used to choose which one.

Earlier works have [5,19,20] proposed an algorithm that relies on oblivious sorting to achieve duplicate suppression in $O(n \log n)$ work and $O(\log n)$ parallel runtime where $n := |X|$.

Oblivious select. Select$(X, k, \text{priority})$ takes in an array X where each element is either of the form (k, v) or a dummy denoted $\perp$, a query k, and a priority function priority which defines a total ordering on all elements with the same key; and outputs a value v such that $(k, v) \in X$ and moreover there exists no $(k, v') \in X$ such that v' is preferred over v for the key k by the priority function priority.

Oblivious select can be accomplished using a simple tree-based algorithm [9] in $O(\log n)$ parallel runtime and $O(n)$ total work where $n = |X|$.

Oblivious multicast. Oblivious multicast is the following building block. Given the following inputs:

- a source array $X := \{(k_i, v_i) \mid \text{dummy}\}_{i \in [n]}$ where each element is either of the form (k, v) or a dummy denoted dummy, and further all real elements must have a distinct k; and
- a destination array $Y := \{k'_i\}_{i \in [n]}$ where each element is a query k' (possibly having duplicates).

the oblivious multicast algorithm outputs an array ans $:= \{v_i\}_{i \in [n]}$ such that if $k'_i \notin X$ then $v_i := \perp$; else it must hold that $(k'_i, v_i) \in X$.

Boyle et al. [5] propose an algorithm based on $O(1)$ oblivious sorts that achieves oblivious multicast in $O(\log n)$ parallel runtime and $O(n \log n)$ total work.

Range prefix sum. We will rely on a parallel range prefix sum algorithm which offers the following abstraction: given an input array $X = (x_1, \ldots, x_n)$ of length n where each element of X is of the form $x_i := (k_i, v_i)$, output an array $Y = (y_1, \ldots, y_n)$ where each y_i is defined as follows:

- Let $i' \leq i$ be the smallest index such that $k_{i'} = k_{i'+1} = \ldots = k_i$;
- $y_i := \sum_{j=i'}^{i} v_j$.

In the GraphSC work, Nayak et al. [32] provide an oblivious algorithm that computes the range prefix sum in $O(\log n)$ parallel runtime and $O(n \log n)$ total work—in particular [32] defines a building block called "longest prefix sum" which is a slight variation of the range prefix sum abstraction we need. It is easy to see that Nayak et al.'s algorithm for longest prefix sum can be modified in a straightforward manner to compute our notion of range prefix sum.

3   Oblivious Two-Tier Hashing Scheme

In this section, we present a simple oblivious two-tier hashing scheme. Before we describe our scheme, we make a couple important remarks that the reader should keep in mind:

– Note that our security definition implies that the adversary can only observe the memory access patterns, and we require simulatability of the memory access patterns. Therefore our scheme description does not explicitly encrypt data. When actually deploying an ORAM scheme, all data must be encrypted if the adversary can also observe the contents of memory.
– In our oblivious hashing scheme, we use λ to denote the security parameter, and use n to denote the hash table's size. Our ORAM application will employ hash tables of varying sizes, so n can be small. Observe that an instance of hash table building can fail with $\mathsf{negl}(\lambda)$ probability; when this happens in the context of ORAM, the hash table building is restarted. This ensures that the ORAM is always correct, and the security parameter is related to the running time of the ORAM.
– For small values of n, we need special treatment to obtain $\mathsf{negl}(\lambda)$ security failure probability—specifically, we simply employ normal balls-and-bins hashing for small values of n. Instead of having the ORAM algorithm deal with this issue, we wrap this part inside the oblivious hashing scheme, i.e., the oblivious hashing scheme will automatically decide whether to employ normal hashing or two-tier hashing depending on n and λ.
  This modular approach makes our ORAM and OPRAM algorithms conceptually simple and crystallizes the security argument as well.

The goal of this section is to give an oblivious hashing scheme with the following guarantee.

Theorem 1 (Parallel oblivious hashing). *For any constant $\epsilon > 0.5$, for any $\alpha(\lambda) := \omega(1)$, there exists a $(W_{\mathrm{build}}, T_{\mathrm{build}}, W_{\mathrm{lookup}}, T_{\mathrm{lookup}})$-parallel oblivious hashing scheme where*

$$W_{\mathrm{build}} = O(\log n), \qquad\qquad T_{\mathrm{build}} = O(\log n),$$
$$W_{\mathrm{lookup}} = \begin{cases} O(\alpha \log \lambda) & \text{if } n < e^{3\log^\epsilon \lambda} \\ O(\log^\epsilon \lambda) & \text{if } n \geq e^{3\log^\epsilon \lambda} \end{cases}, \; T_{\mathrm{lookup}} = O(\log\log \lambda)$$

3.1   Construction: Non-oblivious and Sequential Version

For simplicity, we first present a non-oblivious and sequential version of the hashing algorithm, and we can use this version of the algorithm for the purpose of our stochastic analysis. Later in Sect. 3.2, we will show how to make the algorithm both oblivious and parallel. Henceforth, we fix some $\epsilon \in (0.5, 1)$.

Case 1: $n < e^{3\log^\epsilon \lambda}$. When n is sufficiently small relative to the security parameter λ, we simply apply normal hashing (i.e., balls and bins) in the following

manner. Let each bin's capacity $Z(\lambda) = \alpha \log \lambda$, for any $\alpha = \omega(1)$ superconstant function in λ.

For building a hash table, first, generate a secret PRF key denoted $\mathsf{sk} \xleftarrow{\$} \{0,1\}^{\lambda}$. Then, store the n elements in $B := \lceil 5n/Z \rceil$ bins each of capacity Z, where each element $(k, \_)$ is assigned to a pseudorandom bin computed as follows:

$$\text{bin number} := \mathsf{PRF}_{\mathsf{sk}}(k).$$

Due to a simple application of the Chernoff bound, the probability that any bin overflows is negligible in λ as long as Z is superlogarithmic in λ.

To look up an element with the key k, compute the bin number as above and read the entire bin.

Case 2: $n \geq e^{3 \log^{\epsilon} \lambda}$. This is the more interesting case, and we describe our two-tier hashing algorithm below.

– **Parameters and data structure.** Suppose that our memory is organized into two hash tables named H_1 and H_2 respectively, where each hash table has $B := \lceil \frac{n}{\log^{\epsilon} \lambda} \rceil$ bins, and each bin can store at most $Z := 5 \log^{\epsilon} \lambda$ blocks.
– $\mathsf{Build}(1^{\lambda}, \{(k_i, v_i) \mid \mathsf{dummy}\}_{i \in [n]})$:
   a) Generate a PRF key $\mathsf{sk} \xleftarrow{\$} \{0,1\}^{\lambda}$.
   b) For each element $(k_i, v_i) \in S$, try to place the element into the bin numbered $\mathsf{PRF}_{\mathsf{sk}}(1 || k_i)$ in the first-tier hash table H_1. In case the bin is full, instead place the element in the overflow pile henceforth denoted Buf.
   c) For each element (k, v) in the overflow pile Buf, place the element into the bin numbered $\mathsf{PRF}_{\mathsf{sk}}(2 || k)$ in the second-tier hash table H_2.
   d) Output $\mathsf{T} := (\mathsf{H}_1, \mathsf{H}_2, \mathsf{sk})$.
– $\mathsf{Lookup}(\mathsf{T}, k)$: Parse $\mathsf{T} := (\mathsf{H}_1, \mathsf{H}_2, \mathsf{sk})$ and perform the following.
   a) If $k = \bot$, i.e., this is a dummy query, return $\bot$.
   b) Let $i_1 := \mathsf{PRF}_{\mathsf{sk}}(1 || k)$. If an element of the form (k, v) is found in $\mathsf{H}_1[i_1]$, return v. Else, let $i_2 := \mathsf{PRF}_{\mathsf{sk}}(2 || k)$, look for an element of the form (k, v) in $\mathsf{H}_2[i_2]$ and return v if found.
   c) If still not found, return $\bot$.

Overflow event. If in the above algorithm, an element happens to choose a bin in the second-tier hash table H_2 that is full, we say that a bad event called overflow has happened. When a hash building is called in the execution of an ORAM, recall that if an overflow occurs, we simply discard all work thus far and restart the build algorithm from the beginning.

In Sect. 3.4, we will prove that indeed, overflow events occur with negligible probability. Therefore, henceforth in our ORAM presentation, *we will simply pretend that* overflow *events never happen during hash table building.*

Remark 1. Since the oblivious hashing scheme is assumed to retry from scratch upon overflows, we guarantee perfect correctness and computational security failure (due to the use of a PRF). Similarly, our resulting ORAM and OPRAM

schemes will also have perfect correctness and computational security. Obviously, the algorithms may execute longer if overflows and retries take place—henceforth in the paper, whenever we say that an algorithm's total work or runtime is bounded by x, we mean that it is bounded by x except with negligible probability over the randomized execution.

3.2   Construction: Making It Oblivious

Oblivious Building. To make the building phase oblivious, it suffices to have the following Placement building block.

Let B denote the number of bins, let Z denote each bin's capacity, and let R denote the maximum capacity of the overflow pile. Placement is the following building block. Given an array $\mathsf{Arr} = \{(\mathsf{elem}_i, \mathsf{pos}_i) \mid \mathsf{dummy}\}_{i \in [n]}$ containing n possibly dummy elements, where each non-dummy element elem_i is tagged with a pseudo-random bin number $\mathsf{pos}_i \in [B]$, output B arrays $\{\mathsf{Bin}_i\}_{i \in [B]}$ each of size exactly Z and an overflow pile denoted Buf of size exactly R. The placement algorithm must output a valid assignment if one exists. Otherwise if no valid assignment exists, the algorithm should abort outputting hash-failure.

We say that an assignment is valid if the following constraints are respected:

(i) Every non-dummy $(\mathsf{elem}_i, \mathsf{pos}_i) \in \mathsf{Arr}$ exists either in some bin or in the overflow pile Buf.
(ii) For every Bin_i, every non-dummy element in Bin_i is of the form $(\_, i)$. In other words, non-dummy elements can only reside in their targeted bin or the overflow pile Buf.
(iii) For every Bin_i, if there exists a dummy element in Bin_i, then no element of the form $(\_, i)$ appears in Buf. In other words, no elements from each bin should overflow to Buf unless the bin is full.

[*Special case*]. A special case of the placement algorithm is when the overflow pile's targeted capacity $R = 0$. This special case will be used when we create the second-tier hash table.

Below, we show that using standard oblivious sorting techniques [2], Placement can be achieved in $O(n \log n)$ total work:

1. For each $i \in [B]$, add Z copies of filler elements $(\diamond, i)$ where $\diamond$ denotes that this is a filler element. These filler elements are there to make sure that each bin is assigned at least Z elements. Note that filler elements and dummy elements are treated differently.
2. Oblivious sort all elements by their bin number. For elements with the same bin number, break ties by placing real elements to the left of filler elements.
3. In a single linear scan, for each element that is not among the first Z elements of its bin, tag the element with the label "excess".
4. Oblivious sort all elements by the following ordering function:
   – All dummy elements must appear at the very end;
   – All non-excess elements appear before excess elements;

- For two non-excess elements, the one with the smaller bin number appears first (breaking ties arbitrarily).
- For excess elements, place real elements to the left of filler elements.

Oblivious lookups. It remains to show how to make lookup queries oblivious. To achieve this, we can adopt the following simple algorithm:

- If the query $k \neq \perp$: compute the first-tier bin number as $i_1 := \mathsf{PRF_{sk}}(1\|k)$. Read the entire bin numbered i_1 in the first-tier hash table $\mathsf{H_1}$. If found, read an entire random bin in $\mathsf{H_2}$; else compute $i_2 := \mathsf{PRF_{sk}}(2\|k)$ and read the entire bin numbered i_2 in the second-tier hash table $\mathsf{H_2}$. Finally, return the element found or $\perp$ if not found.
- If the query $k = \perp$, read an entire random bin in $\mathsf{H_1}$, and an entire random bin in $\mathsf{H_2}$. Both bin numbers are selected freshly and independently at random. Finally, return $\perp$.

3.3   Construction: Making It Parallel

To make the aforementioned algorithm parallel, it suffices to make the following observations:

(i) Oblivious sorting of n elements can be accomplished using a sorting circuit [2] that involves $O(n \log n)$ total work and $O(\log n)$ parallel runtime.
(ii) Step 3 of the oblivious building algorithm involves a linear scan of the array marking each excessive element that exceeds its bin's capacity.
   This linear scan can be implemented in parallel using the oblivious "range prefix sum" algorithm in $O(n \log n)$ total work and $O(\log n)$ parallel runtime. We refer the reader to Sect. 2.4 for a definition of the range prefix sum algorithm.
(iii) Finally, observe that the oblivious lookup algorithm involves searching in entire bin for the desired block. This can be accomplished obliviously and in parallel through our "oblivious select" building block defined in Sect. 2.4. Since each bin's capacity is $O(\log^\epsilon n)$, the oblivious select algorithm can be completed in $O(\log \log n)$ parallel runtime and tight total work.

Remark 2 (The case of small n). So far, we have focused our attention on the (more interesting) case when $n \geq e^{3 \log^\epsilon \lambda}$. When $n < e^{3 \log^\epsilon \lambda}$, we rely on normal hashing, i.e., balls and bins. In this case, hash table building can be achieved through a similar parallel oblivious algorithm that completes in $O(n \log n)$ total work and $O(\log n)$ parallel runtime; further, each lookup query completes obliviously in $O(\alpha \log \lambda)$ total work and $O(\log \log \lambda)$ parallel runtime.

Performance of our oblivious hashing scheme. In summary, the resulting algorithm achieves the following performance:

- Building a hash table with n elements takes $O(n \log n)$ total work and $O(\log n)$ parallel runtime with all but $\mathsf{negl}(\lambda)$ probability, regardless of how large n is.

– Each lookup query takes $O(\log^\epsilon \lambda)$ total work when $n \geq e^{3\log^\epsilon \lambda}$ and $O(\alpha \log \lambda)$ total work when $n < e^{3\log^\epsilon \lambda}$ where $\alpha(\lambda) = \omega(1)$ can be any superconstant function. Further, regardless of how large n is, each lookup query can be accomplished in $O(\log \log \lambda)$ parallel runtime.

3.4 Overflow Analysis

We give the overflow analysis of the two-tier construction in Sect. 3.1. We use the following variant of Chernoff Bound.

Fact 2 (Chernoff Bound for Binomial Distribution). *Let X be a random variable sampled from a binomial distribution (with any parameters). Then, for any $k \geq 2E[X]$, $Pr[X \geq k] \leq e^{-\frac{k}{6}}$.*

Utilization of first-tier hash. Recall that the number of bins is $B := \left\lceil \frac{n}{\log^\epsilon \lambda} \right\rceil$. For $i \in [B]$, let X_i denote the number of items that are sent to bin i in the first-tier hash. Observe that the expectation $E[X_i] = \frac{n}{B} \geq \log^\epsilon \lambda$.

Overflow from first-tier hash. For $i \in [B]$, let $\widehat{X}_i$ be the number of items that are sent to bin i in the first-tier but have to be sent to the overflow pile because bin i is full. Recall that the capacity of a bin is $Z := 5\log^\epsilon \lambda$. Then, it follows that $\widehat{X}_i$ equals $X_i - Z$ if $X_i > Z$, and 0 otherwise.

Tail bound for overflow pile. We next use the standard technique of moment generating function to give a tail inequality for the number $\sum_i \widehat{X}_i$ of items in the overflow pile. For sufficiently small $t > 0$, we have

$E[e^{t\widehat{X}_i}] \leq 1 + \sum_{k \geq 1} \Pr[X_i = Z + k] \cdot e^{tk} \leq 1 + \sum_{k \geq 1} \Pr[X_i \geq Z + k] \cdot e^{tk} \leq 1 + \frac{\exp(-\frac{Z}{6})}{e^{\frac{1}{6}-t} - 1}$, where the last inequality follows from Fact 2 and a standard computation of a geometric series. For the special case $t = \frac{1}{12}$, we have $E[e^{\frac{\widehat{X}_i}{12}}] \leq 1 + 12\exp(-\frac{Z}{6})$.

Lemma 1 (Tail Inequality for Overflow Pile). *For $k \geq 288Be^{-\frac{Z}{6}}$, $\Pr[\sum_{i \in [B]} \widehat{X}_i \geq k] \leq e^{-\frac{k}{24}}$.*

Proof. Fix $t := \frac{1}{12}$. Then, we have $\Pr[\sum_{i \in [B]} \widehat{X}_i \geq k] = \Pr[t \sum_{i \in [B]} \widehat{X}_i \geq tk] \leq e^{-tk} \cdot E[e^{t \sum_{i \in [B]} \widehat{X}_i}]$, where the last inequality follows from the Markov's inequality.

As argued in [13], when n balls are thrown independently into n bins uniformly at random, then the numbers X_i's of balls received in the bins are negatively associated. Since $\widehat{X}_i$ is a monotone function of X_i, it follows that the $\widehat{X}_i$'s are also negatively associated. Hence, it follows that $E[e^{t \sum_{i \in [B]} \widehat{X}_i}] \leq \prod_{i \in [B]} E[e^{t\widehat{X}_i}] \leq \exp(12Be^{-\frac{Z}{6}})$.

Finally, observing that $k \geq 288Be^{-\frac{Z}{6}}$, we have $\Pr[\sum_{i \in [B]} \widehat{X}_i \geq k] \leq \exp(12Be^{-\frac{Z}{6}} - \frac{k}{12}) \leq e^{-\frac{k}{24}}$, as required.

In view of Lemma 1, we consider $N := 288Be^{-\frac{Z}{6}}$ as an upper bound on the number of items in the overflow pile. The following lemma gives an upper bound on the probability that a particular bin overflows in the second-tier hash.

Lemma 2 (Overflow Probability in the Second-Tier Hash). *Suppose the number of items in the overflow pile is at most $N := 288Be^{-\frac{Z}{6}}$, and we fix some bin in the second-tier hash. Then, the probability that this bin receives more than Z items in the second tier hash is at most $e^{-\frac{Z^2}{6}}$.*

Proof. Observe that the number of items that a particular bin receives is stochastically dominated by a binomial distribution with N items and probability $\frac{1}{B}$. Hence, the probability that it is at least Z is at most $\binom{N}{Z} \cdot (\frac{1}{B})^Z \leq (\frac{Ne}{Z})^Z \cdot (\frac{1}{B})^Z \leq e^{-\frac{Z^2}{6}}$, as required.

Corollary 1 (Negligible Overflow Probability). *Suppose the number n of items is chosen such that both $Be^{-\frac{Z}{6}}$ and Z^2 are $\omega(\log \lambda)$, where $B := \left\lceil \frac{n}{\log^\epsilon \lambda} \right\rceil$ and $Z := \lceil 5 \log^\epsilon \lambda \rceil$. Then, the probability that the overflow event happens in the second-tier hash is negligible in λ.*

Proof. Recall that $B = \lceil \frac{n}{\log^\epsilon \lambda} \rceil$, where $n \geq e^{3 \log^\epsilon \lambda}$ in Theorem 1. By choosing $N = 288Be^{-\frac{Z}{6}}$, from Lemma 1, the probability that there are more than N items in the overflow pile is $\exp(-\Theta(N))$, which is negligible in λ.

Given that the number of items in the overflow pile is at most N, according to Lemma 2, the probability that there exists some bin that overflows in the second-tier hash is at most $Be^{-\frac{Z^2}{6}}$ by union bound, which is also negligible in λ, because we assume $B \leq \text{poly}(\lambda)$.

3.5   Obliviousness

If there is no overflow, for any valid input, Build accesses fixed addresses. Also, Lookup fetches a fresh pseudorandom bin for each dummy or non-dummy request. Hence, the simulator is just running Build and Lookup with all dummy requests. See the online full version [7] for the formal proof.

4   Modular Framework for Hierarchical ORAM

4.1   Preliminary: Hierarchical ORAM from Oblivious Hashing

Goldreich and Ostrovsky [19,20] were the first to define Oblivious RAM (ORAM) and they provide an elegant solution to the problem which was since referred to as the "hierarchical ORAM". Goldreich and Ostrovsky [19,20] describe a special-case instantiation of a hierarchical ORAM where they adopt an oblivious variant of naïve hashing. Their scheme was later extended and improved by several subsequent works [22,26,42].

In this section, we will present a generalized version of Goldreich and Ostrovsky's hierarchical ORAM framework. Specifically, we will show that Goldreich and Ostrovsky's core idea can be interpreted as the following: take any oblivious hashing scheme satisfying the abstraction defined in Sect. 2.3, we can construct a corresponding ORAM scheme that makes blackbox usage of the oblivious hashing scheme.

From our exposition, it will be clear why such a modular approach is compelling: it makes both the construction and the security proof simple. In comparison, earlier hierarchical ORAM works do not adopt this modular approach, and their conceptual complexity could sometimes confound the security proof [34].

Data structure. There are $\log N + 1$ *levels* numbered $0, 1, \ldots, L$ respectively, where $L := \lceil \log_2 N \rceil$ is the maximum level. Each level is a hash table denoted $\mathsf{T}_0, \mathsf{T}_1, \ldots, \mathsf{T}_L$ where T_i has capacity 2^i. At any time, each table T_i can be in two possible states, *available* or *full*. Available means that this level is currently empty and does not contain any blocks, and thus one can rebuild into this level. Full means that this level currently contains blocks, and therefore an attempt to rebuild into this level will effectively cause a cascading merge.

ORAM operations. Upon any memory access request $(\mathtt{read}, \mathsf{addr})$ or $(\mathtt{write}, \mathsf{addr}, \mathsf{data})$, perform the following procedure. For simplicity, we omit writing the security parameter of the algorithms, i.e., let $\mathsf{Build}(\cdot) := \mathsf{Build}(1^N, \cdot)$, and let $\mathsf{Lookup}(\cdot) := \mathsf{Lookup}(1^N, \cdot)$.

1. found := `false`.
2. For each $\ell = 0, 1, \ldots L$ in increasing order,
   - If not found, fetched := $\mathsf{Lookup}(\mathsf{T}_\ell, \mathsf{addr})$: if fetched $\neq \perp$, let found := `true`, data* := fetched.
   - Else $\mathsf{Lookup}(\mathsf{T}_\ell, \perp)$.
3. Let $\mathsf{T}^\emptyset := \{(\mathsf{addr}, \mathsf{data}^*)\}$ if this is a `read` operation; else let $\mathsf{T}^\emptyset := \{(\mathsf{addr}, \mathsf{data})\}$. Now perform the following hash table rebuilding:
   - Let ℓ be the smallest level index such that T_ℓ is marked available. If all levels are marked full, then $\ell := L$. In other words, ℓ is the target level to be rebuilt.
   - Let $S := \mathsf{T}^\emptyset \cup \mathsf{T}_0 \cup \mathsf{T}_1 \cup \ldots \cup \mathsf{T}_{\ell-1}$; if all levels are marked full, then additionally let $S := S \cup \mathsf{T}_L$. Further, tag each non-dummy element in S with its level number, i.e., if a non-dummy element in S comes from T_i, tag it with the level number i.
   - $\mathsf{T}_\ell := \mathsf{Build}(\mathsf{SuppressDuplicate}(S, 2^\ell, \mathsf{pref}))$, and mark T_ℓ as full. Further, let $\mathsf{T}_0 = \mathsf{T}_1 = \ldots = \mathsf{T}_{\ell-1} := \emptyset$ and their status bits set to available. Here we adopt the following priority function pref:
     • When two or more real blocks with the same address (i.e., key) exist, the one with the smaller level number is preferred (and the algorithm maintains the invariant that no two blocks with the same address and the same level number should exist).
4. Return data*.

Deamortization. In the context of hierarchical ORAM, a hash table of capacity n is rebuilt every n memory requests, and we typically describe the ORAM's overhead in terms of the amortized cost per memory request. As one may observe, every now and then, the algorithm needs to rebuild a hash table of size N, and thus a small number of memory requests may incur super-linear cost to complete.

A standard deamortization technique was described by Ostrovsky and Shoup [33] to evenly spread the cost of hash table rebuilding over time, and this deamortization framework only blows up the total work of the ORAM scheme by a small constant factor; the details are in the online full version [7]. In the rest of the paper, we assume that every instance of hash table used in an ORAM scheme is rebuilt in the background using this deamortization technique without explicitly mentioning so. Further, the stated costs in the theorems are applicable to worst-case performance (not just amortized).

Obliviousness. To show obliviousness of the above construction, we make the following observations.

Fact 3 (Non-recurrent queries imply obliviousness). *In the aforementioned ORAM construction, as long as lookup queries to every instance of hash table satisfies the non-recurrent condition specified in Sect. 2.3, the resulting ORAM scheme satisfies obliviousness.*

The proof of this fact is deferred to our online full version [7].

Fact 4 (Non-recurrence condition is preserved). *In the above ORAM construction, it holds that for every hash table instance, all lookup queries it receives satisfy the non-recurrence condition.*

Proof. Due to our ORAM algorithm, every 2^ℓ operations, the old instance of hash table T_ℓ is destroyed and a new hash table instance is created for T_ℓ. It suffices to prove the non-recurrence condition in between every two rebuilds for T_ℓ. Suppose that after T_ℓ is rebuilt in some step, now we focus on the time steps going forward until the next rebuild. Consider when a block block^* is first found in T_ℓ where $\ell \in [L]$, block^* is entered into $\mathsf{T}^\emptyset$. Due to the definition of the ORAM algorithm, until the next time T_ℓ is rebuilt, block^* exists in some $\mathsf{T}_{\ell'}$ where $\ell' < \ell$. Due to the way the ORAM performs lookups—in particular, we would look up a dummy element in T_ℓ if block^* is found in a smaller level—we conclude that until T_ℓ is rebuilt, no lookup query will ever be issued again for block^* to T_ℓ.

Lemma 3 (Obliviousness). *Suppose that the underlying hashing scheme satisfies correctness and obliviousness as defined in Sect. 2.3, then it holds that the above ORAM scheme satisfies obliviousness as defined in Sect. 2.2.*

Proof. Straightforward from Facts 3 and 4.

Theorem 2 (Hierarchical ORAM from oblivious hashing). *Assume the existence of one-way functions and a $(W_{\mathrm{build}}, W_{\mathrm{lookup}})$-oblivious hashing scheme.*

Then, there exists an ORAM scheme that achieves the following blowup for block sizes of $\Omega(\log N)$ bits:

$$ORAM\text{'s blowup} := \max\left(\sum_{\ell=0}^{\log N} W_{\text{build}}(2^\ell, N),\ \sum_{\ell=0}^{\log N} W_{\text{lookup}}(2^\ell, N)\right) + O(\log^2 N)$$

This theorem is essentially proved by Goldreich and Ostrovsky [19,20]—however, they proved it only for a special case. We generalize their hierarchical ORAM construction and express it modularly to work with any oblivious hashing scheme as defined in Sect. 2.3.

Remark 3. We point out that due to the way we define our oblivious hashing abstraction, each instance of oblivious hash table will independently generate a fresh PRF key during Build, and this PRF key is stored alongside the resulting hash table data structure in memory. Throughout this paper, we assume that each PRF operation can be evaluated in $O(1)$ runtime on top of our RAM. *We stress that this implicit assumption (or equivalent) was made by all earlier ORAM works* [19,20,22,26] *that rely on a PRF for security.*

4.2 Preliminary: Improving Hierarchical ORAM by Balancing Reads and Writes

Subsequent to Goldreich and Ostrovsky's ground-breaking result [19,20], Kushilevitz et al. [26] propose an elegant optimization for the hierarchical ORAM framework such that under some special conditions to be specified later, they can shave a (multiplicative) $\log \log N$ factor off the total work for a hierarchical ORAM scheme. Similarly, Kushilevitz et al. [26] describe a special-case instantiation of an ORAM scheme based on oblivious Cuckoo hashing which was proposed by Goodrich and Mitzenmacher [22].

In this section, we observe that the Kushilevitz et al.'s idea can be generalized. For the sake of exposition, we will first ignore the smaller ORAM levels that employ normal hashing in the following discussion, i.e., we assume that the smaller levels that employ normal hashing will not be a dominating factor in the cost. Now, imagine that there is an oblivious hashing scheme such that for sufficiently large n, the per-element cost for preprocessing is more expensive than the cost of a lookup by a $\log^\delta n$ factor for some constant $\delta > 0$. In other words, imagine that there exists a constant $\delta > 0$ such that the following condition is met for sufficiently large n:

$$\frac{W_{\text{build}}(n, \lambda)}{W_{\text{lookup}}(n, \lambda)} \geq \log^\delta n.$$

If the underlying oblivious hashing scheme satisfies the above condition, then Kushilevitz et al. [26] observes that Goldreich and Ostrovsky's hierarchical ORAM construction is suboptimal in the sense that the cost of fetch phase is asymptotically smaller than the cost of the rebuild phase. Hence, the resulting

ORAM's total work will be dominated by the rebuild phase, which is then determined by the building cost of the underlying hashing scheme, i.e., $W_{\mathrm{build}}(n, \lambda)$.

Having observed this, Kushilevitz et al. [26] propose the following modification to Goldreich and Ostrovsky's hierarchical ORAM [19,20]. In Goldreich and Ostrovsky's ORAM, each level is a factor of 2 larger than the previous level— henceforth the parameter 2 is referred to the *branching factor*. Kushilevitz et al. [26] proposes to adopt a branching factor of $\mu := \log N$ instead of 2, and this would reduce the number of levels to $O(\log N / \log \log N)$—in this paper, we will adopt a more general choice of $\mu := \log^{\phi} N$ for a suitable positive constant ϕ. To make this idea work, they allow up to $\mu - 1$ simultaneous hash table instances for any ORAM level. If for all levels below ℓ, all instances of hash tables are full, then all levels below ℓ will be merged into a new hash table residing at level $\ell + 1$. The core idea here is to *balance the cost of the fetch phase and the rebuild phase by having a larger branching factor*; and as an end result, we could shave a $\log \log N$ factor from the ORAM's total work.

We now elaborate on this idea more formally.

Data structure. Let $\mu := \log^{\phi} N$ for a suitable positive constant ϕ to be determined later. There are $O(\log N / \log \log N)$ *levels* numbered $0, 1, \ldots, L$ respectively, where $L = \lceil \log_{\mu} N \rceil$ denotes the maximum level. Except for level L, for every other $\ell \in \{0, 1, \ldots, L-1\}$: the ℓ-th level contains up to $\mu - 1$ hash tables each of capacity μ^{ℓ}. Henceforth we use the notation T_{ℓ} to denote level ℓ, and T_{ℓ}^{i} to denote the i-th hash table within level ℓ. The largest level L contains a single hash table of capacity N denoted T_{L}^{0}. Finally, every level $\ell \in \{0, 1, \ldots, L\}$ has a counter c_{ℓ} initialized to 0. Effectively, for every level $\ell \neq L$, if $c_{\ell} = \mu - 1$, then the level is considered full; else the level is considered available.

ORAM operations. Upon any memory access query $(\mathtt{read}, \mathtt{addr})$ or $(\mathtt{write}, \mathtt{addr}, \mathtt{data})$, perform the following procedure.

1. $\mathsf{found} := \mathtt{false}$.
2. For each $\ell = 0, 1, \ldots L$ in increasing order, for $\tau = c_{\ell} - 1, c_{\ell} - 2 \ldots 0$ in decreasing order:

    If not found: $\mathsf{fetched} := \mathsf{Lookup}(\mathsf{T}_{\ell}^{\tau}, \mathtt{addr})$; if $\mathsf{fetched} \neq \bot$, let $\mathsf{found} := \mathtt{true}$, $\mathsf{data}^{*} := \mathsf{fetched}$. Else $\mathsf{Lookup}(\mathsf{T}_{\ell}^{\tau}, \bot)$.
3. Let $\mathsf{T}^{\emptyset} := \{(\mathtt{addr}, \mathsf{data}^{*})\}$ if this is a $\mathtt{read}$ operation; else let $\mathsf{T}^{\emptyset} := \{(\mathtt{addr}, \mathtt{data})\}$. Now, perform the following hash table rebuilding.
    - Let ℓ be the smallest level index such that its counter $c_{\ell} < \mu - 1$. If no such level index exists, then let $\ell := L$. In other words, we plan to rebuild a hash table in level ℓ.
    - Let $S := \mathsf{T}^{\emptyset} \cup \mathsf{T}_{0} \cup \mathsf{T}_{1} \cup \ldots, \cup \mathsf{T}_{\ell-1}$; and if $\ell = L$, additionally, let $S := S \cup \mathsf{T}_{L}^{0}$ and let $c_{L} = 0$. Further, in the process, tag each non-dummy element in S with its level number and its hash table number within the level. For example, if a non-dummy element in S comes from T_{i}^{τ}, i.e., the τ-th table in the i-th level, tag it with (i, τ).
    - Let $\mathsf{T}_{\ell}^{c_{\ell}} := \mathsf{Build}(\mathsf{SuppressDuplicate}(S, \mu^{\ell}, \mathsf{pref}))$, and let $c_{\ell} := c_{\ell} + 1$.

Here we adopt the following priority function pref: when two or more blocks with the same address (i.e., key) exist, the one with the *smaller* level number is preferred; if there is a tie in level number, the one with the *larger* hash table number is preferred.

– Let $\mathsf{T}_0 = \mathsf{T}_1 = \ldots = \mathsf{T}_{\ell-1} := \emptyset$ and set $c_0 = c_1 = \ldots = c_{\ell-1} := 0$.

4. Return data^*.

Goldreich and Ostrovsky's ORAM scheme [19,20] is a special case of the above for $\mu = 2$.

Deamortization. The deamortization technique of Ostrovsky and Shoup [33] (described in the online full version [7]) applies in general to hierarchical ORAM schemes for which each level is some data structure that is rebuilt regularly. Therefore, it can be applied to our scheme as well, and thus the work of rebuilding hash tables is spread evenly across memory requests.

Obliviousness. The obliviousness proof is basically identical to that presented in Sect. 4.1, since the only change here from Sect. 4.1 is that the parameters are chosen differently due to Kushilevitz et al.'s elegant idea [26].

Theorem 3 (Hierarchical ORAM variant.). *Assume the existence of one-way functions and a $(W_{\mathrm{build}}, W_{\mathrm{lookup}})$-oblivious hashing scheme. Then, there exists an ORAM scheme that achieves the following blowup for block sizes of $\Omega(\log N)$ bits where $L = O(\log N / \log \log N)$:*

$$ORAM\text{'s blowup} := \max\left(\sum_{\ell=0}^{L} W_{\mathrm{build}}(\mu^\ell, N), \ \log^\phi N \cdot \sum_{\ell=0}^{L} W_{\mathrm{lookup}}(\mu^\ell, N) \right)$$
$$+ O(L \log N)$$

We note that Kushilevitz et al. [26] proved a special case of the above theorem, we now generalize their technique and describe it in the most general form.

4.3   Conceptually Simpler ORAM for Small Blocks

In the previous section, we presented a hierarchical ORAM scheme, reparametrized using Kushilevitz et al. [26]'s technique, consuming any oblivious hashing scheme with suitable performance characteristics as a blackbox.

To obtain a conceptually simple ORAM scheme with $O(\log^2 N / \log \log N)$ overhead, it suffices to plug in the oblivious two-tier hashing scheme described earlier in Sect. 3.

Corollary 2 (Conceptually simpler ORAM for small blocks). *There exists an ORAM scheme with $O(\log^2 N / \log \log N)$ runtime blowup for block sizes of $\Omega(\log N)$ bits.*

Proof. Using the simple oblivious two-tier hashing scheme in Sect. 3 with $\epsilon = \frac{3}{4}$, we can set $\phi = \frac{1}{4}$ in Theorem 3 to obtain the result.

4.4   IO Efficiency and the Case of Large CPU Cache

Besides the ORAM's runtime, we often care about its IO performance as well, where IO-cost is defined as the number of cache misses as in the standard external-memory algorithms literature. When the CPU has a large amount of private cache, e.g., N^ϵ blocks where $\epsilon > 0$ is an arbitrarily small constant, several works have shown that oblivious sorting $n \leq N$ elements can be accomplished with $O(n)$ IO operations [8, 21, 22]. Thus, a direct corollary is that for the case of N^ϵ CPU cache, we can construct a computationally secure ORAM scheme with $O(\log N)$ IO-cost (by using the basic hierarchical ORAM construction with $O(\log N)$ levels with an IO-efficient oblivious sort).

5   Asymptotically Efficient OPRAM

In this section, we show how to construct an $O(\frac{\log^2 N}{\log \log N})$ OPRAM scheme. To do this, we will show how to parallelize our new $O(\frac{\log^2 N}{\log \log N})$-overhead ORAM scheme. Here we benefit tremendously from the conceptual simplicity of our new ORAM scheme. In particular, as mentioned earlier, our oblivious two-tier hashing (Build, Lookup) algorithms have efficient parallel realizations. We will now present our OPRAM scheme. For simplicity, we first present a scheme assuming that the number of CPUs in each step of the computation is fixed and does not change over time. In this case, we show that parallelizing our earlier ORAM construction boils down to parallelizing the (Build and Lookup) algorithms of the oblivious hashing scheme. We then extend our construction to support the case when the number of CPUs varies over time.

5.1   Intuition

Warmup: uniform number of CPUs. We first describe the easier case of uniform m, i.e., the number of CPUs in the PRAM does not vary over time. Further, we will consider the simpler case when the branching factor $\mu := 2$.

– *Data structure.* Recall that our earlier ORAM scheme builds an exponentially growing hierarchy of oblivious hash tables, of capacities $1, 2, 4, \ldots, N$ each. Here, we can do the same, but we can start the level of hierarchy at capacity $m = 2^i$ (i.e., skip the smaller levels).
– *OPRAM operations.* Given a batch of m simultaneous memory requests, suppose that all addresses requested are distinct—if not, we can run a standard conflict resolution procedure as described by Boyle et al. [5] incurring only $O(\log m)$ parallel steps consuming m CPUs. We now need to serve these requests in parallel. In our earlier ORAM scheme, each request has two stages: (1) reading one block from each level of the exponentially growing hierarchy; and (2) perform necessary rebuilding of the levels. It is not hard to see that the fetch phase can be parallelized easily—particularly, observe that the fetch

phase is read-only, and thus having m CPUs performing the reads in parallel will not lead to any write conflicts.

It remains to show how to parallelize the rebuild phase. Recall that in our earlier ORAM scheme, each level has a status bit whose value is either *available* or *full*. Whenever we access a single block, we find the available (i.e., empty) level ℓ and merge all smaller levels as well as the updated block into level ℓ. If no such level ℓ exists, we simply merge all levels as well as the updated block into the largest level.

Here in our OPRAM construction, since the smallest level is of size m, we can do something similar. We find the smallest available (i.e., empty) level ℓ, and merge all smaller levels as well as the possibly updated values of the m fetched blocks into level ℓ. If no such level ℓ exists, we simply merge all levels as well as possibly updated values of the m fetched blocks into the largest level. Rebuilding a level in parallel effectively boils down to rebuilding a hash table in parallel (which boils down to performing $O(1)$ number of oblivious sorts in parallel)—which we have shown to be possible earlier in Sect. 3.

Varying number of CPUs. Our definitions of PRAM and OPRAMs allow the number of CPUs to vary over time. In this case, oblivious simulation of a PRAM is more sophisticated. First, instead of truncating the smaller levels whose size are less than m, here we have to preserve all levels—henceforth we assume that we have an exponentially growing hierarchy with capacities $1, 2, 4, \ldots, N$ respectively. The fetch phase is simple to parallelize as before, since the fetch phase does not make modifications to the data structure. We now describe a modified rebuild phase when serving a batch of $m = 2^\gamma$ requests: note that in the following, γ is a level that matches the current batch size, i.e., the number of CPUs in the present PRAM step of interest:

(a) Suppose level γ is marked available. Then, find the first available (i.e., empty) level ℓ greater than γ. Merge all levels below γ and the updated values of the newly fetched m blocks into level ℓ.
    If no such level ℓ exists, then merge all blocks and the updated values of the newly fetched m blocks into the largest level L.
(b) Suppose level γ is marked as full. Then, find the first available (i.e., empty) level ℓ greater than γ. Merge all levels below or equal to γ (but not the updated values of the m fetched blocks) into level ℓ; rebuild level γ to contain the updated values of the m fetched blocks.
    Similarly, if no such level ℓ exists, then merge all blocks and the updated values of the newly fetched m blocks into the largest level L.

One way to view the above algorithm is as follows: let us view the concatenation of all levels' status bits as a binary counter (where full denotes 1 and available denotes 0). If a single block is accessed like in the ORAM case, the counter is incremented, and if a level flips from 0 to 1, this level will be rebuilt. Further, if there would be a carry-over to the $(L+1)$-st level, then the largest level L is rebuilt. However, now m blocks may be requested in a single batch—in

this case, the above procedure for rebuilding effectively can be regarded as incrementing the counter by some value v where $v \leq 2m$—in particular, the value v is chosen such that only $O(1)$ levels must be rebuilt by the above rule.

We now embark on describing the full algorithm—specifically, we will describe for a general choice of the branching factor μ that is not necessarily 2. Further, our description supports the case of varying number of CPUs.

5.2   Detailed Algorithm

Data structure. Same as in Sect. 4.2. Specifically, there are $O(\log N / \log \log N)$ *levels* numbered $0, 1, \ldots, L$ respectively, where $L = \lceil \log_\mu N \rceil$ denotes the maximum level. Except for level L, for every other $\ell \in \{0, 1, \ldots, L-1\}$: the ℓ-th level contains up to $\mu - 1$ hash tables each of capacity μ^ℓ. Henceforth, we use the notation T_ℓ to denote level ℓ. Moreover, for $0 \leq i < \mu - 1$, we use T_ℓ^i to denote the i-th hash table within level ℓ. The largest level L contains a single hash table of capacity N denoted T_L^0. Finally, every level $\ell \in \{0, 1, \ldots, L\}$ has a counter c_ℓ initialized to 0.

We say that a level $\ell < L$ is *available* if its counter $c_\ell < \mu - 1$; otherwise, $c_\ell = \mu - 1$, and we say that the level $\ell < L$ is *full*. For the largest level L, we say that it is available if $c_L = 0$; else we say that it is full. Note that for the case of general $\mu > 2$, available does not necessarily mean that the level's empty.

OPRAM operations. Upon a batch of m memory access requests $Q := \{\mathsf{op}_p\}_{p \in [m]}$ where each op_p is of the form $(\mathsf{read}, \mathsf{addr}_p)$ or $(\mathsf{write}, \mathsf{addr}_p, \mathsf{data}_p)$, perform the following procedure. Henceforth we assume that $m = 2^\gamma$ where γ denotes the level whose capacity matches the present batch size.

1. **Conflict resolution.** $Q' := \mathsf{SuppressDuplicate}(Q, m, \mathsf{PRAM.priority})$, i.e., perform conflict resolution on the batch of memory requests Q, and obtain a batch Q' of the same size but where each distinct address appears only once—suppressing duplicates using the PRAM's priority function $\mathsf{priority}$, and padding the resulting set with dummies to length m.
2. **Fetch phase.** For each $\mathsf{op}_i \in Q'$ in parallel where $i \in [m]$, parse $\mathsf{op}_i = \bot$ or $\mathsf{op}_i = (\mathsf{read}, \mathsf{addr}_i)$ or $\mathsf{op}_i = (\mathsf{write}, \mathsf{addr}_i, \mathsf{data}_i)$:
   (a) If $\mathsf{op}_i = \bot$, let $\mathsf{found} := \mathsf{true}$; else let $\mathsf{found} := \mathsf{false}$.
   (b) For each $\ell = 0, 1, \ldots L$ in increasing order, for $\tau = c_\ell - 1, c_\ell - 2 \ldots 0$ in decreasing order:
       – If not found: $\mathsf{fetched} := \mathsf{Lookup}(\mathsf{T}_\ell^\tau, \mathsf{addr}_i)$; if $\mathsf{fetched} \neq \bot$, let $\mathsf{found} := \mathsf{true}$, $\mathsf{data}_i^* := \mathsf{fetched}$.
       – Else, $\mathsf{Lookup}(\mathsf{T}_\ell^\tau, \bot)$.
3. **Rebuild phase.** For each $\mathsf{op}_i \in Q'$ in parallel where $i \in [m]$: if op_i is a read operation add $(\mathsf{addr}_i, \mathsf{data}_i^*)$ to $\mathsf{T}^\emptyset$; else if op_i is a write operation, add $(\mathsf{addr}_i, \mathsf{data}_i)$ to $\mathsf{T}^\emptyset$; else add $\bot$ to $\mathsf{T}^\emptyset$.
   Perform the following hash table rebuilding—recall that γ is the level whose capacity matches the present batch size:

(a) If level γ is full, then skip this step; else, perform the following:
Let $S := \mathsf{T}_0 \cup \mathsf{T}_1 \cup \ldots \cup \mathsf{T}_{\gamma-1}$, and $\mathsf{T}_\gamma^{c_\gamma} := \mathsf{Build}$ $(\mathsf{SuppressDuplicate}(S, \mu^\gamma, \mathsf{pref}))$ where pref prefers a block from a smaller level (i.e., the fresher copy) if multiple blocks of the same address exists. Let $c_\gamma := c_\gamma + 1$, and for every $j < \gamma$, let $c_j := 0$.

(b) At this moment, if level γ is still available, then let $\mathsf{T}_\gamma^{c_\gamma} := \mathsf{Build}(\mathsf{T}^\emptyset)$, and $c_\gamma := c_\gamma + 1$.
Else, if level γ is full, perform the following:
Find the first available level $\ell > \gamma$ greater than γ that is available; if no such level ℓ exists, let $\ell := L$ and let $c_L := 0$.
Let $S := \mathsf{T}^\emptyset \cup \mathsf{T}_0 \cup \ldots \cup \mathsf{T}_{\ell-1}$; if $\ell = L$, additionally include $S := S \cup \mathsf{T}_L$. Let $\mathsf{T}_\ell^{c_\ell} := \mathsf{Build}(\mathsf{SuppressDuplicate}(S, \mu^\ell, \mathsf{pref}))$, and let $c_\ell := c_\ell + 1$. For every $j < \ell$, reset $c_j := 0$.

Deamortization. The deamortization technique (described in the online full version [7]) of Ostrovsky and Shoup [33] applies here as well, and thus the work of rebuilding hash tables are spread evenly across memory requests.

Obliviousness. The obliviousness proof is basically identical to that presented in Sect. 4.1. Since we explicitly resolve conflict before serving a batch of m requests, we preserve the non-recurrence condition. The only remaining differences here in comparison with Sect. 4.1 is that (1) here we use a general branching factor of μ rather than 2 (as in Sect. 4.1); and (2) here we consider the parallel setting. It is clear that neither of these matter to the obliviousness proof.

Theorem 4 (OPRAM from oblivious parallel hashing). *Assume the existence of one-way functions and a $(W_{\mathrm{build}}, T_{\mathrm{build}}, W_{\mathrm{lookup}}, T_{\mathrm{lookup}})$-oblivious hashing scheme. Then, there exists an ORAM scheme that achieves the following performance for block sizes of $\Omega(\log N)$ bits where $L = O(\frac{\log N}{\log \log N})$:*

$$ total\ work\ blowup := \max\left(\sum_{\ell=0}^{L} W_{\mathrm{build}}(\mu^\ell, N),\ \log^\phi N \cdot \sum_{\ell=0}^{L} W_{\mathrm{lookup}}(\mu^\ell, N) \right) $$

$$ + O(L \log N), $$

and para. runtime blowup :=

$$ \max\left(\{T_{\mathrm{build}}(\mu^\ell, N)\}_{\ell \in [L]},\ \log^\phi N \cdot \sum_{\ell=0}^{L} T_{\mathrm{lookup}}(\mu^\ell, N) \right) + O(L). $$

Proof. Basically, the proof is our explicit OPRAM construction from any parallel oblivious hashing scheme described earlier in this section. For total work and parallel runtime blowup, we basically take the maximum of the ORAM's fetch phase and rebuild phase. The additive term $O(L \log N)$ in the total work stems from additional building blocks such as parallel duplicate suppression and other steps in our OPRAM scheme; and same for the additive term $O(L)$ in the parallel runtime blowup.

Using the simple oblivious hashing scheme in Sect. 3 with $\epsilon = \frac{3}{4}$, we can set $\phi = \frac{1}{4}$ to obtain the following corollary.

Corollary 3 (Asympototically efficient OPRAM for small blocks). *Assume that one-way functions exist. Then, there exists a computationally secure OPRAM scheme that achieves $O(\log^2 N / \log \log N)$ simulation overhead when the block size is at least $\Omega(\log N)$ bits.*

References

1. Adler, M., Chakrabarti, S., Mitzenmacher, M., Rasmussen, L.E.: Parallel randomized load balancing. Random Struct. Algorithms **13**(2), 159–188 (1998)
2. Ajtai, M., Komlós, J., Szemerédi, E.: An O(N Log N) sorting network. In: STOC (1983)
3. Asharov, G., Chan, H., Nayak, K., Pass, R., Ren, L., Shi, E.: Oblivious computation with data locality. IACR Cryptology ePrint Archive 2017/772 (2017)
4. Bast, H., Hagerup, T.: Fast and reliable parallel hashing. In: SPAA (1991)
5. Boyle, E., Chung, K.-M., Pass, R.: Oblivious parallel RAM and applications. In: Kushilevitz, E., Malkin, T. (eds.) TCC 2016. LNCS, vol. 9563, pp. 175–204. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49099-0_7
6. Chan, T.-H.H., Chung, K.-M., Shi, E.: On the depth of oblivious parallel RAM. In: Asiacrypt (2017)
7. Chan, T.-H.H., Guo, Y., Lin, W.-K., Shi, E.: Oblivious hashing revisited, and applications to asymptotically efficient ORAM and OPRAM. Online full version of this paper, IACR Cryptology ePrint Archive 2017/924 (2017)
8. Chan, T.-H.H., Guo, Y., Lin, W.-K., Shi, E.: Cache-oblivious and data-oblivious sorting and applications. In: SODA (2018)
9. Chan, T.-H.H., Shi, E.: Circuit OPRAM: unifying statistically and computationally secure ORAMs and OPRAMs. In: TCC (2017)
10. Chen, B., Lin, H., Tessaro, S.: In: TCC, pp. 205–234 (2016)
11. Chung, K.-M., Liu, Z., Pass, R.: Statistically-secure ORAM with $\tilde{O}(\log^2 n)$ overhead. In: Asiacrypt (2014)
12. Dietzfelbinger, M., Karlin, A., Mehlhorn, K., Meyer auF der Heide, F., Rohnert, H., Tarjan, R.E.: Dynamic perfect hashing: upper and lower bounds. SIAM J. Comput. **23**(4), 738–761 (1994)
13. Dubhashi, D.P., Ranjan, D.: Balls and bins: a study in negative dependence. Random Struct. Algorithms **13**(2), 99–124 (1998)
14. Fletcher, C.W., Ren, L., Kwon, A., van Dijk, M., Devadas, S.: Freecursive ORAM: [nearly] free recursion and integrity verification for position-based oblivious RAM. In: ASPLOS (2015)
15. Fletcher, C.W., Ren, L., Kwon, A., van Dijk, M., Stefanov, E., Devadas, S.: RAW Path ORAM: a low-latency, low-area hardware ORAM controller with integrity verification. IACR Cryptology ePrint Archive 2014/431 (2014)

16. Fletcher, C.W., Ren, L., Yu, X., van Dijk, M., Khan, O., Devadas, S.: Suppressing the oblivious RAM timing channel while making information leakage and program efficiency trade-offs. In: HPCA, pp. 213–224 (2014)
17. Fredman, M.L., Komlós, J., Szemerédi, E.: Storing a sparse table with O(1) worst case access time. J. ACM **31**(3), 538–544 (1984)
18. Gentry, C., Halevi, S., Lu, S., Ostrovsky, R., Raykova, M., Wichs, D.: Garbled RAM revisited. In: Nguyen, P.Q., Oswald, E. (eds.) EUROCRYPT 2014. LNCS, vol. 8441, pp. 405–422. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-55220-5_23
19. Goldreich, O.: Towards a theory of software protection and simulation by oblivious RAMs. In: STOC (1987)
20. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious RAMs. J. ACM **43**, 431–473 (1996)
21. Goodrich, M.T.: Data-oblivious external-memory algorithms for the compaction, selection, and sorting of outsourced data. In: SPAA (2011)
22. Goodrich, M.T., Mitzenmacher, M.: Privacy-preserving access of outsourced data via oblivious RAM simulation. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011. LNCS, vol. 6756, pp. 576–587. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22012-8_46
23. Gordon, S.D., Katz, J., Kolesnikov, V., Krell, F., Malkin, T., Raykova, M., Vahlis, Y.: Secure two-party computation in sublinear (amortized) time. In: CCS (2012)
24. Hagerup, T.: Fast and optimal simulations between CRCW PRAMs. In: Finkel, A., Jantzen, M. (eds.) STACS 1992. LNCS, vol. 577, pp. 45–56. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-55210-3_172
25. Hagerup, T.: The log-star revolution. In: Finkel, A., Jantzen, M. (eds.) STACS 1992. LNCS, vol. 577, pp. 257–278. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-55210-3_189
26. Kushilevitz, E., Lu, S., Ostrovsky, R.: On the (in)security of hash-based oblivious RAM and a new balancing scheme. In: SODA (2012)
27. Liu, C., Wang, X.S., Nayak, K., Huang, Y., Shi, E.: ObliVM: a programming framework for secure computation. In S & P (2015)
28. Lu, S., Ostrovsky, R.: How to garble ram programs. In: Eurocrypt (2013)
29. Maas, M., Love, E., Stefanov, E., Tiwari, M., Shi, E., Asanovic, K., Kubiatowicz, J., Song, D.: Phantom: practical oblivious computation in a secure processor. In: CCS (2013)
30. Matias, Y., Vishkin, U.: Converting high probability into nearly-constant time—with applications to parallel hashing. In: STOC, pp. 307–316 (1991)
31. Nayak , K., Katz, J.: An oblivious parallel ram with $o(\log^2 n)$ parallel runtime blowup. https://eprint.iacr.org/2016/1141
32. Nayak, K., Wang, X.S., Ioannidis, S., Weinsberg, U., Taft, N., Shi, E.: GraphSC: parallel secure computation made easy. In: IEEE S & P (2015)
33. Ostrovsky, R., Shoup, V.: Private information storage (extended abstract). In: STOC, pp. 294–303 (1997)
34. Pinkas, B., Reinman, T.: Oblivious RAM revisited. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 502–519. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14623-7_27
35. Ren, L., Fletcher, C.W., Kwon, A., Stefanov, E., Shi, E., van Dijk, M., Devadas, S.: Constants count: practical improvements to oblivious RAM. In: USENIX Security Symposium, pp. 415–430 (2015)
36. Ren, L., Yu, X., Fletcher, C.W., van Dijk, M., Devadas, S.: Design space exploration and optimization of path oblivious RAM in secure processors. In: ISCA (2013)

37. Shi, E., Chan, T.-H.H., Stefanov, E., Li, M.: Oblivious RAM with $O((\log N)^3)$ worst-case cost. In: Asiacrypt (2011)
38. Stefanov, E., Shi, E.: Multi-cloud oblivious storage. In: CCS (2013)
39. Stefanov, E., Shi, E.: ObliviStore: high performance oblivious cloud storage. In: IEEE Symposium on Security and Privacy (S & P) (2013)
40. Stefanov, E., van Dijk, M., Shi, E., Fletcher, C., Ren, L., Yu, X., Devadas, S.: Path ORAM - an extremely simple oblivious ram protocol. In: CCS (2013)
41. Wang, X., Chan, H., Shi, E.: Circuit ORAM: on tightness of the Goldreich-Ostrovsky lower bound. In: CCS (2015)
42. Williams, P., Sion, R., Carbunar, B.: Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In: CCS, pp. 139–148 (2008)
43. Williams, P., Sion, R., Tomescu, A.: PrivateFS: a parallel oblivious file system. In: CCS (2012)