Chapter 9
Enhanced RAM Simulation in Succinct Space

Taku Onodera

Abstract We describe two recent results on space-efficient functional random access memory (RAM), which is RAM with non-standard functionalities. The first is about oblivious RAM, which enables a remote database to be accessed without revealing to the database owner which part of the database is being accessed. The other is about wear leveling, which enables the number of updates to be balanced among all the memory cells regardless of the content of the computation being performed on the memory.

9.1 Introduction

Random access memory (RAM) underlies most modern computers, and improvements to the RAM itself can have a positive impact on a wide range of applications. For example, faster RAM access makes all RAM-based computations correspondingly faster. Some types of RAM improvements are not just about efficiency but also about *functionality*. An example is virtual memory in operating systems, which enables, among other things, application programs to utilize the memory without concern about cumbersome management issues such as allocation. Generally speaking, this type of RAM improvement functions by using conventional RAM to simulate "enhanced" RAM while introducing some performance overhead.

In this chapter, we describe two such enhanced RAM simulations—*oblivious RAM* (ORAM) and *wear leveling*—with the emphasis on how to minimize the space overhead. These topics were chosen mainly because the authors' knowledge of them, although there are also some conceptual similarities between ORAM and wear

T. Onodera (✉)
The University of Tokyo, Tokyo, Japan
e-mail: tk-ono@is.s.u-tokyo.ac.jp

© The Author(s) 2022

N. Katoh et al. (eds.), *Sublinear Computation Paradigm*,
https://doi.org/10.1007/978-981-16-4095-7_9

leveling. Other functionality-enhanced RAM simulations include initializable array [2], memory checking [3], locally decodable code [11], and huge random object [8].[1]

9.2 Oblivious RAM

9.2.1 Problem

Suppose you want to outsource a database, stored in RAM, to a server and want to access it in a privacy-preserving way. Although you can hide the data content by encryption, the server can still see which part of the RAM you are accessing. This is a serious issue in the current era of cloud computing. The same problem also appears when one wants to hide the details of software implemented in a physically secure processor that accesses insecure main memory.

Oblivious RAM (ORAM) is the formalization and corresponding solution of this problem. Typically, it works by storing the RAM into some data structure on the server and moves the RAM cells dynamically in the data structure as the user accesses the RAM.

As an example, consider a scheme where the server stores the RAM as-is except that each cell is encrypted by the user's key. To access the ith cell, the user performs the following procedure for $j = 1$ to N where N is the number of cells:

1. Retrieve the ith cell from the server.
2. Decrypt the retrieved cell.
3. If $i = j$:
    - For read access, copy the decrypted value to local memory.
    - For write access, change the decrypted value to the new value.
4. Re-encrypt the possibly changed decrypted value.
5. Store the re-encrypted value back in the ith cell on the server.

We assume semantically secure encryption when encryption is used in this chapter. In particular, there is an overwhelmingly high probability that the re-encrypted ciphertext looks totally different to the server from the ciphertext before re-encryption regardless of whether the plaintext is updated or not. Thus, no matter what actual access is performed, all that the server can see is that random-looking encrypted cells are updated to still random-looking re-encrypted cells in a fixed scan order. Of course, the access overhead of this method is very large since each cell access takes

[1] A huge random object, in this context, is a succinct representation of a pseudorandom object that supports certain queries. For example, a pseudorandom function can be thought of as a huge pseudorandom bitstring that is implicitly represented by a tiny seed and supports efficient random access. This is not a data structure in the conventional sense because the represented object is a pseudorandom bitstring instead of "data."

time linear to the entire RAM size. The purpose of this example is merely to illustrate the kind of security we want to achieve.

We now give a more formal problem description. We have three parties: the user, the server, and the simulator. The simulator models a program that runs in the local environment of the user. The simulator provides the user with an access interface to RAM that we call the virtual RAM while the server provides the simulator with an access interface to RAM that we call the physical RAM. That is, the user gives the simulator a series of queries of the form (type, i, v) where type $\in$ {read, write}, $i \in [N]$, and $v \in \{0, 1\}^B$. We call these virtual queries. The parameter N specifies the number of virtual cells—cells in the virtual RAM—while B specifies the size of each virtual cell. Given a virtual query, the simulator gives the server another series of queries of the form (type, i, v), where type $\in$ {read, write}, $i \in [N']$, and $v \in \{0, 1\}^{B'}$. We call these physical queries. The simulator, and thus the physical queries, is probabilistic in general.[2] The server responds to physical queries in the obvious way. That is, for (read, i, $*$) where "$*$" means that the third component is arbitrary, the server returns the value of the ith physical cell, and for (write, i, v), the server updates the value of the ith physical cell to v. If the virtual query from the user is of the form (read, i, $*$), the simulator derives the value of the ith virtual cell through the interaction with the server and returns it to the user. If the virtual query is of the form (write, i, v), the simulator updates the value of the ith virtual cell to v. The simulator must respond to the virtual queries online. We call the sequence of second components of the virtual queries (resp. physical queries) a virtual access pattern (resp. physical access pattern). For a virtual query sequence q, let $a(q)$ denote the physical access pattern induced by q. Recall that $a(q)$ is a random variable in general. The ORAM scheme is secure if $a(q_1)$ and $a(q_2)$ are indistinguishable for any virtual query sequences q_1 and q_2 of the same length. There are some variations in the exact meaning "indistinguishable". Typically used meanings of indistinguishability in descending order of security are a) equally distributed, b) statistically closely distributed, and c) computationally indistinguishable. The main performance metric of ORAM includes access overhead, which is the number of physical queries processed for each virtual query, the simulator local space size, and the server space size, which is $B'N'$ bits.

As mentioned above, the simulator models a program running in the local environment of the user. Thus, in practice, we do not distinguish the user and the simulator. For example, we refer to the simulator local space as user space.

The ORAM problem is non-trivial only if the user space is smaller than BN bits since otherwise, the simulator can store the entire RAM locally and ignore the server.

[2] The simulator of the scan-based method is deterministic, and it is not hard to see that its linear access overhead is optimal if we restrict the simulator to be deterministic. Thus, all simulators of interest are indeed probabilistic.

Table 9.1 Summary of existing results. † means amortized bound. $\approx \log N$ means $O(f(N))$ for any $f \in \omega(\log N)$. The constant factor of the user space of [26] is $\ll 1$

	Access overhead	Sever space (N')	User space	Technique	Security
[7]	$O(\sqrt{N}\log N)$ †	$N(1+2\sqrt{N})$	$O(1)$	Square root	Computational
[9]	$O(\log^3 N)^\dagger$	$\Theta(N\log N)$	$O(1)$	Hierarchical	Computational
[19]	$O(\sqrt{N}\log N)$	$(1+\Theta(1))N$	$O(1)$		
[19]	$O(\log^3 N)$	$\Theta(N\log N)$	$O(1)$		
[10]	$O(\log^2 N)^\dagger$	$(1+\Theta(1))N$	$O(1)$		
[13]	$O(\frac{\log^2 N}{\log\log N})$	$(1+\Theta(1))N$	$O(1)$		
[20]	$O(\log N\log\log N)^\dagger$	$(1+\Theta(1))N$	$O(1)$		
[12]	$O(\log N)^\dagger$	$(1+\Theta(1))N$	$O(1)$		
[25]	$O(\log^3 N)$	$\Theta(N\log N)$	$O(1)$	Tree	Statistical
[27]	$O(\log^2 N)$	$(1+\Theta(1))N$	$\approx \log N$		
[17]	$O(\log^2 N)$	$(1+o(1))N$	$\approx \log N$		

9.2.2 *Existing Results*

Table 9.1 gives a summary of some of the existing results. Every method has physical cell size $B' = B + \Theta(\log N)$. There is an $\Omega(\log N)$ lower bound for the access overhead if the user space is at most $N^{1-\epsilon}$ for constant $\epsilon > 0$ [14].

There are mainly two types of techniques that are actively studied: hierarchical approaches and tree-based approaches.[3] Asymptotically, the state-of-the-art hierarchical method [12] has access overhead matching the lower bound mentioned above while the state-of-the-art tree-based methods have about $\log N$ times larger asymptotic access overhead. Yet, tree-based methods are still of practical interest because they tend to have much smaller access overhead constant factors than the hierarchical methods. The access overheads of tree-based methods also constrain the worst case while those of the hierarchical methods are often amortized. Although there are techniques to achieve competitive worst-case access overhead via the hierarchical methods [13, 19], they tend to be complex and add further constant factors to the performance bounds.

In the past, the ORAM research community has focused mainly on reducing the access overhead because it was the biggest obstacle to applying ORAM in practice. However, some recent studies have achieved practical access performances [16, 22] by combining tree-based ORAM with special hardware. For example, the PHANTOM secure processor system [16] supports access pattern-hiding SQL queries with a time overhead of 1.2–6 $\times$ compared to the standard insecure version. Thus, at least for tree-based ORAM, exploration of aspects other than access overhead is beginning to make sense.

[3] The square root method [7] was the first non-trivial ORAM and is the origin of some of the ideas underlying the hierarchical methods.

We describe the recent development of techniques for reducing the number of physical cells N' of the tree-based ORAM to $(1 + o(1))N$ [17]. Note that there also is a space overhead originating from the cell size: typically, $B' = B + c \lg N$ where c is a small constant such as 2. We ignore the cell size overhead and focus on the cell number because the typical value of B is 128 bytes in the secure processor setting and the overhead with respect to the cell size is just a few percent.

9.2.3 Tree-Based Methods

The tree-based method of Stefanov et al. [27] works as follows. The server organizes the physical cells into a complete binary tree with N leaves where each node is a bucket—a container that can accommodate a constant number of virtual cells. Each virtual cell has a position label—an integer in $[N]$—and a virtual cell with position label i is stored either in some bucket on the path from the root to the ith leaf or in a stash, which is a container in the user's local memory that can accommodate a small number of virtual cells. Let v_i be the ith virtual cell and let p_i be the position label of v_i. Suppose the user maintains p_i for all $i \in [N]$ in local memory. This requires $\Omega(N)$ user space but simplifies the exposition. We will reduce the user space later. To access v_i, the user retrieves all of the blocks on the path from the root to the p_ith leaf. Let this path be P. At this point, v_i must be in the stash. The user copies the value of v_i to somewhere in its local memory for a read query or changes it to some other value for a write query. Then, the user updates p_i to a fresh random value in $[N]$. After that, the user scans the buckets on P from the leaf to the root, and for each bucket, moves cells in the stash to the bucket greedily while respecting the position labels and the bucket capacity. See Fig. 9.1 for an example.

Sometimes, some virtual cells in the stash cannot be moved back to the tree. For example, if all cells are assigned the same position label, only $\Theta(\log N)$ physical cells can be used to store the virtual cells and thus, most virtual cells must end up in the stash. (Of course, if N is large, such an event happens only extremely rarely.) Stefanov et al. proved that if the bucket size is at least 5, the number of cells left in the stash after processing a query is exponentially small. Thus, if the stash size is $\omega(\log N)$, the stash overflows during processing a polynomial number of queries with only negligible probability.

To reduce the user space for position labels, the user outsources the position labels using the same method recursively. That is, each position label is a $\lceil \lg N \rceil$ bit integer and the table for position labels of all virtual cells can be thought of as a RAM storing the $N\lceil \lg N \rceil$-bit concatenation of the integers. Thus, the original problem of hiding the access pattern to RAM consisting of N cells, each of B bits, is reduced to hiding the access pattern to RAM consisting of $N\lceil \lg N \rceil / B$ cells, each of B bits. If, say, $B \geq 2 \lg N$, which is a completely reasonable assumption for all reasonable N,[4] the problem size (cell number) decreases exponentially and reaches $O(1)$ after

[4] Recall that the typical value of B is 128 bytes in secure processor applications.

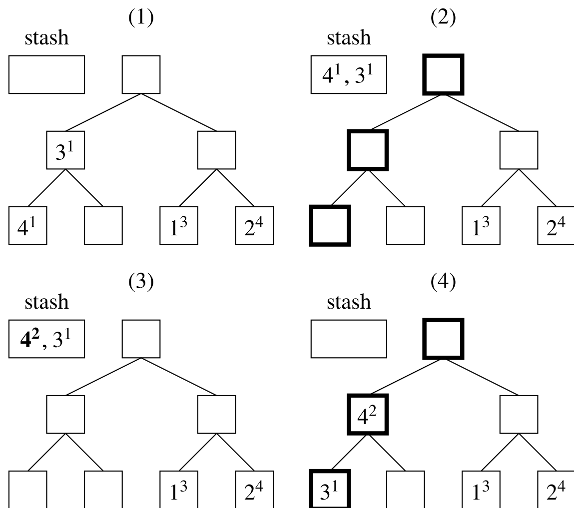

Fig. 9.1 Example access process for reading the 4th virtual cell. $N = 4$. Bucket size is 1. The expression i^j means the ith virtual cell with position label j. The path from the root to the first leaf is scanned from top to bottom in step (2) and from bottom to top in step (4)

$O(\lg N)$ levels of recursion. At that point, the user can store the $O(1)$ size RAM locally terminating the recursion.

The tree at the top level of recursion has size $\Theta(N)$ and the tree at higher recursion levels decreases exponentially. The access overhead is proportional to the sum of the heights of the trees at all recursion levels, which is $O(\log^2 N)$. The server space is proportional to the sum of the sizes of the trees at all recursion levels, which is $\Theta(N)$.

Although each recursion level requires a stash, the numbers of cells left in those stashes are independent and it turns out that the total number of cells left in all stashes is still exponentially small. Thus, $f(N)$ user space is enough for any $f(N) \in \omega(\log N)$.

9.2.4 Succinct Construction

The constant factor hidden in the $\Theta(N)$ server space bound of the method described above is about 10: the top-level tree has $2N$ nodes each of capacity 5 while the size of the recursive trees is negligible because typically, B is much larger than $\lg N$. (Theoretically, we assume $B = \omega(\lg N)$.) Though one can reduce this constant factor to some extent by decreasing the tree height while tuning the bucket size, it is not possible to achieve a factor ≤ 2 while maintaining a meaningful stash overflow probability, at least using the currently known analysis techniques. This method also leads to prohibitively large access overhead as the server space becomes close to $2N$. We now describe a method for achieving $(1 + o(1))N$ server space with a modest sacrifice in access overhead [17].

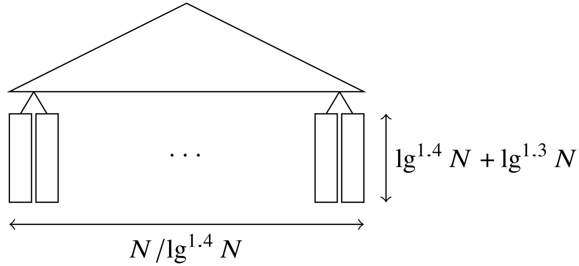

Fig. 9.2 Large leaf layout

The idea is to modify the layout of the tree at the top recursion level so that the leaf number is $N/\lg^{1.4} N$ and the leaf size is $\lg^{1.4} N + \lg^{1.3} N$ (see Fig. 9.2). It is obvious that the tree size is $(1 + 1/\lg^{0.1} N)N$ while access overhead remains $O(\log^2 N)$. We now explain why the stash overflow probability remains small.

Let N_i be the number of cells with position label i for $i \in [N/\lg^{1.4} N]$. Let the load of a bucket be the number of cells stored in the bucket. At each moment in the lifetime of the scheme, N_i follows the binomial distribution with parameters N and $N/\lg^{1.4} N$ for each i. The probability that this becomes larger than $\lg^{1.4} N + \lg^{1.3} N$ is negligible. Thus, no leaf becomes full while processing a polynomial number of queries. Under this assumption, the distribution of the internal bucket loads is dominated by the distribution of the loads of the corresponding $N/\lg^{1.4} N - 1$ internal buckets in the standard N leaf layout scheme described above. This is so because the internal buckets in the large leaf layout do not need to store the cells that overflow from the leaves. Thus, assuming no leaf becomes full, the stash overflow probability of the large leaf case is negligible. The same is true even without this assumption because there is only a negligible probability that the assumed case does not occur.

We can reduce the $N/\lg^{0.1} N$ extra term on the tree size even further by "the power of two choices". That is, we give two random position labels to each virtual cell. One is primary, which determines the path on which the cell can reside, while the other is secondary, which is a dummy needed to hide the access pattern. Now, N_i is the number of virtual cells with primary position label i. We maintain N_i for all i in a sub-ORAM in the same way we store position labels in recursive ORAM. To access a virtual cell v, we retrieve all cells on the path from the root to the p_1th leaf and the path from the root to the p_2th leaf. We choose two random labels p'_1, p'_2 and let p'_1 (resp. p'_2) be the new primary (resp. secondary) label of v if $N_{p'_1} < N_{p'_2}$. Otherwise, we exchange the role of p'_1 and p'_2. We then scan the paths specified by the old labels and greedily move back the cells as in the previous method. Here, N_i is not binomial but concentrated much more tightly around the mean due to the effect of the two choices. Thus, the "head space" for each leaf can be much smaller than $\lg^{1.3} N$, leading to a smaller tree size.

By tuning the parameters, the first technique (large leaf layout) alone can achieve about $(1 + \Theta(\frac{\log N}{B} + \frac{1}{\sqrt{\log N}}))N$ server space while the second technique (two choices) decreases it to $(1 + \Theta(\frac{\log N}{B} + \frac{\log \log N}{\log^2 N}))N$.[5]

9.2.5 Open Problem

It is unknown whether the optimal $O(\log N)$ access overhead and $(1 + o(1))N$ server space can be achieved at the same time. There are two natural approaches for answering this question affirmatively:

- Develop a technique for making hierarchical methods, such as [9], succinct and apply it to the existing optimal method [12]. This seems particularly challenging if we further require a worst-case (instead of amortized) access overhead bound because the existing techniques for achieving a worst-case access overhead bound in the hierarchical approach [13, 19] require maintaining multiple versions of the database.
- Achieve $O(\log N)$ access overhead by a tree-based approach and apply the techniques described here. The first part is already an open problem of sufficient interest.

9.3 Wear Leveling

9.3.1 Problem

Consider the case where you have RAM with the limitation that each cell state can be updated at most a certain number of times. Once the number of updates has reached the limit, the cell dies and you can no longer update it. The utility of the RAM quickly degrades as the cells start to die because the total amount of information that can be stored decreases, and it becomes cumbersome to manage which cells are still alive. Thus, the number of times you can support updates before cells start to die is of primary interest. This number depends heavily on the case. In the best case where the updates are uniform among the cells, you can perform nL updates where n is the number of cells and L is the number of times each cell can be updated. In contrast, in the worst case where all updates fall onto a particular cell, you can perform updates only L times. Wear leveling is the problem of prolonging the memory lifetime as much as possible while keeping the associated overhead, if any, as small as possible.

The system community has been studying wear leveling for decades. Historically, flash memory was the main motivation for studies conducted from the late 1980s to the mid 2000s [1, 6, 15]. Today, the main motivation for wear leveling comes

[5] These bounds include the cell size overhead that we ignored in the main explanation for brevity.

from phase change memory (PCM), which is an emerging next-generation memory technology that has many features, including low latency, energy efficiency, and non-volatility [5]. Each PCM cell supports only 10^8–10^9 updates, which means that cells can start dying within minutes or even seconds if no effort is made to perform wear leveling. PCM differs from flash memory in certain important respects, such as latency, access granularity, and in-place write capability, and thus requires a different wear leveling formalization than flash memory.

Most existing studies on wear leveling are conducted mainly from a practical point of view. Often, they do not have a formal problem statement or rigorous theoretical analyses. While this might not be a serious problem if the only thing that matters is the performance, some relatively recent studies have repeatedly emphasized the security aspects of wear leveling [21, 23, 24, 28, 29]. In particular, it is important to take into account the case of malicious users who actively try to reduce the memory lifetime. (Consider, for example, a computing outsourcing service.)

Below, we describe a recent theoretical study that constructed a problem formalization to capture the wear leveling for PCM explained above, and the corresponding solutions [18].

The formal problem statement is as follows. There are two parties: the user and server. The server has three resources: physical RAM, wear-free memory, and private randomness. The physical RAM is RAM that consists of N B-bit cells while the wear-free memory is RAM that consists of a small number of B-bit cells. The user provides the server with adversarially chosen read/write queries to virtual RAM—a RAM consisting of n b-bit cells—and the server must respond to these queries "correctly." That is, each request is of the form (type, i, v) where type $\in$ {read, write}, $i \in [n]$, and $v \in \{0, 1\}^b$ and, for (read, i, $*$) where "$*$" means that the third component is arbitrary, the server must return the last value written to the ith virtual cell (the v in the last query of the form (write, i, v)). The server not only needs to return the correct responses but also needs to support as many write queries as possible with high probability without updating any physical cell more than L times where L is a parameter. We assume $L = n^\delta$ for some constant $\delta > 0$. Equivalently, we define $\delta := \log_L n$ and assume it is a constant. This assumption is reasonable even though, in reality, L and n are independent, because L is 10^8–10^9 and $\log_L n$ is at most 2 or 3 for reasonable n.

The performance metric for wear leveling includes the physical memory size, the wear-free memory size, the number of write queries supported, and the access overhead, which is the number of physical RAM accesses needed for each virtual RAM access. We say a wear-leveling scheme is "optimal" if it satisfies the following conditions (asymptotic notations are in terms of $n \to \infty$):

- $N = 1 + o(1)$[6];
- With high probability, that is, $1 - O(1/n)$, it can process $(1 - o(1))NL$ write queries without updating any physical cell more than L times;

[6] We ignore the cell size overhead for brevity. Security Refresh [23] described below does not have any cell size overhead ($B' = B$) while the method of Onodera and Shibuya [18] described after that has a cell size of $B' = B + 2\lceil \lg n \rceil + 1$.

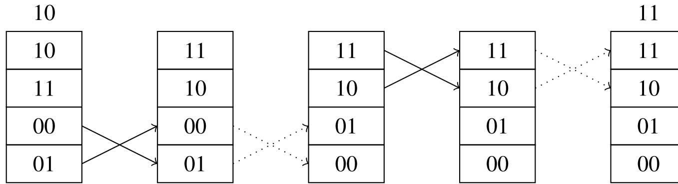

Fig. 9.3 Movement of cells in an epoch of security refresh. All the numbers are binary. $n = 4(= N)$, $r_0 =$"10", and $r_1 =$"11". Solid arrows mean cell swaps while dotted arrows mean skipped cell swaps

- The processing time of each query is $O(1)$;
- It requires only $O(1)$ cells in the wear-free memory.

9.3.2 Security Refresh

The wear leveling scheme of Sewong et al. [23] is optimal if $L = N^\delta$ for $\delta > 1$ while it is non-optimal (in fact, "far from" optimal) for $\delta < 1$ [18].

In this method, n virtual cells are stored in the $N = n$ physical cells in permuted order. The method works in epochs. At each epoch, two random $\lg n$-bit integers r_0 and r_1 are maintained. (We assume n is a power of two for brevity.) At the start of an epoch, for each $i \in [n]$, the ith virtual cell v_i is stored in the $i \oplus r_0$th physical cell $V_{i \oplus r_0}$ where $\oplus$ means bit-wise XOR. During the epoch, each v_i is moved from $V_{i \oplus r_0}$ to $V_{i \oplus r_1}$. Note that the virtual cell stored in the destination $V_{i \oplus r_1}$ of v_i is $v_{i \oplus r_1 \oplus r_0}$ and its destination is $V_{i \oplus r_0}$; that is, v_i and $v_{i \oplus r_0 \oplus r_1}$ swap their positions. This is done as follows. For every t write queries processed where t is a parameter, we perform a remap subroutine. At the ith remap subroutine call in an epoch, we check if $i < i \oplus r_0 \oplus r_1$. If so, v_i still is in $V_{i \oplus r_0}$ and thus, we swap the contents of $V_{i \oplus r_0}$ and $V_{i \oplus r_1}$. Otherwise, v_i is already in $V_{i \oplus r_1}$ and we skip swapping. The epoch ends after the nth remap subroutine finishes. At that point, each cell v_i is stored in $V_{i \oplus r_1}$. We update r_0 to r_1, and r_1 to a fresh random $\lg n$-bit integer. Now every v_i is in $V_{i \oplus r_0}$ as required for the epoch start, and we restart another epoch at this point. See Fig. 9.3 for an example. To access v_i, we access $V_{i \oplus r_0}$ if v_i was already remapped in the epoch. (We have already seen how to check this.) Otherwise, we access $V_{i \oplus r_1}$.

The non-trivial part of the analysis is the proof of a high-probability guarantee of memory lifetime. We outline the key points. Fix a physical cell and let X_i be the number of times it is updated during the ith epoch. We need to place a bound on the probability that the sum of X_is deviates from its expected value. To do this, it suffices to bound the deviation of the sum of odd-indexed variables $X_1, X_3, \ldots$ from its expected value and do the same for the sum of even-indexed variables $X_2, X_4, \ldots$ separately. This is helpful because each X_i is a random variable that depends on r_0, r_1 in the ith epoch (and the queries) and thus, the odd-indexed variables $X_1, X_3, \ldots$ are independent of each other and so are the even-indexed variables. Regardless of

the queries, X_i is bounded by the number of write queries processed in an epoch tn. Although this suggests the use of the Hoeffding inequality, it turns out that it does not work for the case $\delta < 2$, essentially because the condition $X_i \le tn$ alone does not capture the fact that some cell being updated many, say, $\approx tn$, times in an epoch negatively affects the number of times other cells are updated in the epoch. To derive the bound for the case $1 < \delta < 2$, bound the second moment of X_i and apply the Bernstein inequality [4].

If the user tries to keep on updating v_i continuously, one of $V_{i \oplus r_0}$ and $V_{i \oplus r_1}$ is updated $tn/2 = \Omega(n)$ times during the first epoch, and this physical cell dies if $\delta < 1$. Thus, this method is not optimal for $\delta < 1$.

9.3.3 Construction for Small Write Limit Cases

We now briefly describe a method for achieving optimality for the case $\delta < 1$, that is, the memory is large [18]. The idea is to prepare spare cells and remap the frequently updated cells to free spare cells adaptively. (We maintain the write counts of cells by appending a counter to each cell.) We store pointers to the new locations in the old locations to trace the remapped cells. To keep the number of pointers to follow small, we connect pointers in a manner that is similar to the DFS of a complete d-ary tree with d^h leaves where d, h are parameters (see Fig. 9.4). As we continue to process write queries, the data structure gradually degrades: the free spare cells become scarce and the trees become saturated. To reset the degradation, we perform a Security Refresh-style mapping. That is, we treat the structure in Fig. 9.4 as residing in another RAM u and maintain a global mapping—a gradually changing one-to-one map between the cells of u and the physical cells $V_1, V_2, \ldots$. Once we have globally remapped a cell of u corresponding to a tree root, we reset the "DFS" starting from that cell. For example, if we globally remap u_i in state (5) of Fig. 9.4, we free u_{n+1}, u_{n+4}, resetting DFS for the tree from u_i to the root. Garbage such as u_{n+3} are also reclaimed sooner or later when they are globally remapped. To access v_i, the tree path traversal in u starting from i is simulated translating between addresses in u and addresses in V.

Although analysis of the bound on memory lifetime is cumbersome, the same idea as the analysis of Security Refresh applies. Indeed, the core argument is easier because the Hoeffding bound suffices.

9.3.4 Open Problem

The access overhead of the method for the small write limit case described above is about $1/\delta$. It is easy to obtain amortized $1 + o(1)$ and worst-case $\Theta(n)$ access overhead if we allow relatively large wear-free memory, for example, $O(n^\epsilon)$ for an appropriate constant $0 < \epsilon < 1$. It seems possible and practically relevant to

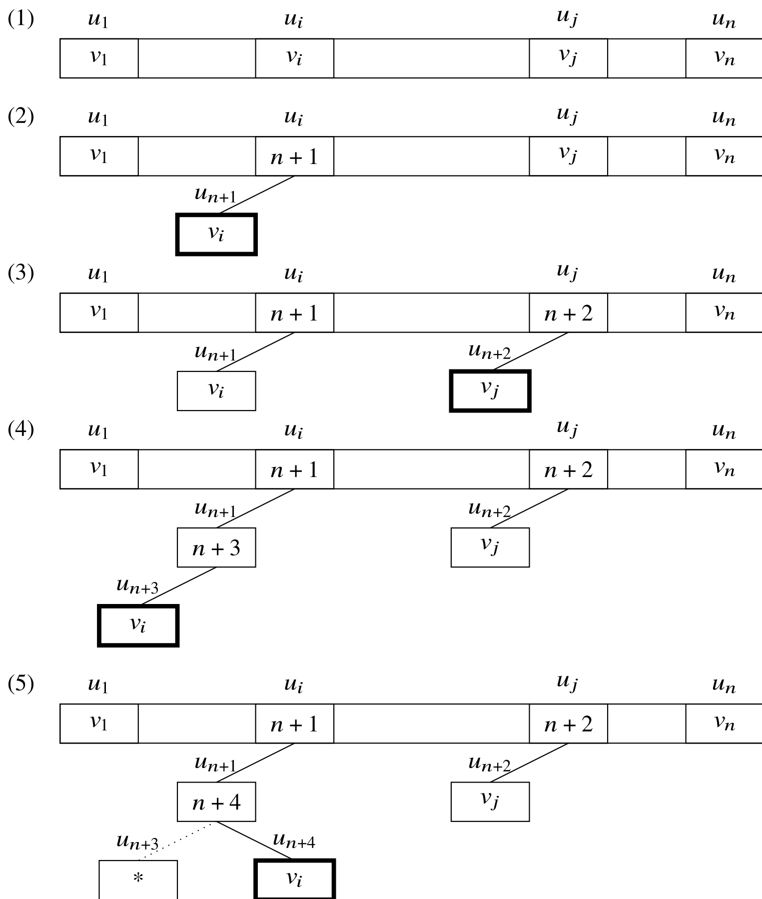

Fig. 9.4 Example evolution of u. $d = h = 2$. $u_1, \ldots, u_n$ are default locations while the rest are spare cells. Each panel shows the state just after the thick-bordered cell was allocated because the cell previously storing its content was updated and the write count reached the threshold

achieve amortized $1 + o(1)$ and worst-case $O(1)$ access overhead in this setting. A theoretically more interesting challenge is to give negative results that justify the use of such large wear-free memory.

9.4 Conclusion

We reviewed two recent studies on ORAM and wear leveling that achieve succinct space usage. Though these objects have totally different motivations and are studied in different communities, there are some similarities between them. As we mentioned

in the introduction, several other concepts with similar flavors are known, including initializable RAM, memory checking, locally decodable code, and huge random objects. There are probably many more such enhanced RAM instances yet to be found, and trying to find them can be an avenue for making progress in studies of data structures.

References

1. A. Ban, Wear leveling of static areas in flash memory. US Patent 6,732,221, May 2004
2. J. Bentley, *Programming Pearls* (Addison-Wesley, Column 1, 1989)
3. Manuel Blum, Will Evans, Peter Gemmell, Sampath Kannan, Moni Naor, Checking the Correctness of Memories. Algorithmica **12**(2-3), 225–244 (1994)
4. S. Boucheron, G. Lugosi, P. Massart, *Concentration Inequalities: A Nonasymptotic Theory of Independence* (Oxford University Press, 2013). Equation (2.10)
5. J. Boukhobza, S. Rubini, R. Chen, Z. Shao, Emerging NVM: a survey on architectural integration and research challenges. ACM Trans. Des. Autom. Electron. Syst. **23**(2), 14:1–14:32 (2017)
6. Eran Gal, Sivan Toledo, Algorithms and Data Structures for Flash Memories. ACM Comput. Surv. **37**(2), 138–163 (2005)
7. O. Goldreich, Towards a theory of software protection and simulation by oblivious RAMs, in *Proceedings of the 19th Annual ACM Symposium on Theory of Computing (STOC)* (1987), pp. 182–194
8. Oded Goldreich, Shafi Goldwasser and Asaf Nussboim. On the implementation of huge random objects. In SIAM J. Comput. 39.7, 2010, pp. 2761–2822
9. Oded Goldreich, Rafail Ostrovsky, Software Protection and Simulation on Oblivious RAMs. J. ACM **43**(3), 431–473 (1996)
10. M.T. Goodrich, M. Mitzenmacher, Privacy-preserving access of outsourced data via oblivious RAM simulation, in *Proceedings of the 38th International Conference on Automata, Languages and Programming (ICALP)*, vol. II (2011), pp. 576–587
11. J. Katz, L. Trevisan, On the efficiency of local decoding procedures for error-correcting codes, in *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing (STOC)* (2000), pp. 80–86
12. I. Komargodski, W.-K. Lin, K. Nayak, E. Peserico, E. Shi, OptORAMa: optimal oblivious RAM, in *Proceedings of the 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques (Eurocrypt)* (2020), pp. 403–432
13. E. Kushilevitz, S. Lu, R. Ostrovsky, On the (in)security of hash-based oblivious RAM and a new balancing scheme, in *Proceedings of the 23rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)* (2012), pp. 143–156
14. K.G. Larsen, J.B. Nielsen, Yes, there is an oblivious RAM lower bound!, in *Proceedings of the 38th International Cryptology Conference (CRYPTO)* (2018), pp. 523–542
15. K.M.J. Lofgren, R.D. Norman, G.B. Thelin, A. Gupta, Wear leveling techniques for flash EEPROM systems. US Patent 6,230,233, May 2001
16. M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, D. Song, PHANTOM: practical oblivious computation in a secure processor, in *Proceedings of the 20th ACM SIGSAC Conference on Computer & Communications Security (CCS)* (2013), pp. 311–324
17. T. Onodera, T. Shibuya, Succinct oblivious RAM, in *Proceedings of the 35th Symposium on Theoretical Aspects of Computer Science (STACS)* (2018), pp. 1–16
18. T. Onodera, T. Shibuya, Wear leveling revisited. To appear in *The 31st International Symposium on Algorithms and Computation (ISAAC2020)*

19. R. Ostrovsky, V. Shoup, Private information storage, in *Proceedings of the 29th Annual ACM Symposium on Theory of Computing (STOC)* (1997), pp. 294–303
20. S. Patel, G. Persiano, M. Raykova, K. Yeo, PanORAMa: oblivious RAM with logarithmic overhead, in *Proceedings of the 59th Annual Symposium on Foundations of Computer Science (FOCS)* (2018), pp. 871–882
21. M.K. Qureshi, M. Franceschini, V. Srinivasan, L. Lastras, B. Abali, J. Karidis, Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling, in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2009), pp. 14–23
22. Ling Ren, Christopher W. Fletcher, Albert Kwon, Marten van Dijk, Srinivas Devadas, Design and Implementation of the Ascend Secure Processor. IEEE Trans. Depend. Secure Comput. **16**(2), 204–216 (2019)
23. N.H. Seong, D.H. Woo, H.-H. Lee, Security refresh: protecting phase-change memory against malicious wear out. IEEE Micro. **31**(1), 119–127 (2011)
24. André Seznec, A Phase Change Memory as a Secure Main Memory. IEEE Computer Architecture Letters **9**(1), 5–8 (2010)
25. E. Shi, T.-H.H. Chan, E. Stefanov, M. Li, Oblivious RAM with $O((\log N)^3)$ worst-case cost, in *Proceedings of the 17th International Conference on the Theory and Application of Cryptology and Information Security (Asiacrypt)* (2011), pp. 197–214
26. E. Stefanov, E. Shi, D.X. Song, Towards practical oblivious RAM, in *19th Annual Network and Distributed System Security Symposium (NDSS)* (2012)
27. E. Stefanov, M. van Dijk, E. Shi, T.-H.H. Chan, C. Fletcher, L. Ren, X. Yu, S. Devadas, Path ORAM: an extremely simple oblivious RAM protocol J. ACM **65**(4) (2018)
28. G. Wu, H. Zhang, Y. Dong, J. Hu, CAR: securing PCM main memory system with cache address remapping, in *Proceedings of the 18th IEEE International Conference on Parallel and Distributed Systems* (2012), pp. 628–635
29. H. Yu, Y. Du, Increasing endurance and security of phase-change memory with multi-way wear-leveling. IEEE Trans. Comput. **63**(5), 1157–1168 (2014)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.