Seminar Advanced Algorithms and Data Structures HS 2018: Resilient Dictionaries [1]

Dario Morandini

October 26, 2018

1 Motivation

This paper presents an implementation of a dictionary and its operations which are resilient in an environment with a bounded number of faults. A paper on DRAM errors running over 2.5 years on Google's servers counted around 5,000 to 70,000 errors per billion device hours per Mbit, which amount to 5 to 7 errors per Terabyte per hour [2]. Therefore errors produced by DRAM alone are far more common than assumed.

To remedy this, most servers these days use ECC memory (Error-correcting code memory) to correct single bit and detect multiple bit errors. But if the environment becomes more hazardous, more errors will lead to significantly more unrecoverable data words.

Information processing devices are especially vulnerable when exposed to radiation which will cause single event upset (SEU) in chips like processing units. Those events can possibly lead to major accidents and are well-known to aviation [3]. Because the build resolution of today's commercial processors are getting smaller, they become more vulnerable to radiation. This can be mitigated by building especially hardened CPU's for this kind of applications. This results in a more expensive smaller edition production whereas mass-produced units are much cheaper making the usage of resilient data structures and algorithm for critical systems more appealing.

2 Data structure

The resilient dictionary implemented by the authors of this paper is based on an AVL tree, they balance themselves after insertions and deletions and are well-known for storing data which has an absolute order.

Definition 1 (Resilient)**.** *A data structure or algorithm is resilient, if it operates correctly on uncorrupted value sets.*

The paper chooses a definition where corrupted values are considered lost but aren't interfering with operations on correct values. This definition is sensible because there only exists an expensive naive algorithm for reading partially corrupted values. Because the process of memory corruption is physically hard to reproduce, we will use the faulty-RAM model to be able to reason about the proposed data structures and algorithms.

Definition 2 (Faulty-RAM model)**.** *The faulty ram model consists of two sets of memory words, safe and unsafe memory. Safe memory cannot be corrupted but is only available in constant size. Unsafe memory can be corrupted up to $\delta \in \mathbb{N}$ times by an adversary and is unbounded in size. The quantity $\alpha \in \mathbb{N}$, $0 \leq \alpha \leq \delta$ corresponds the actual number of corrupted memory words in unsafe memory.*

Because safe memory is bounded by $\mathcal{O}(1)$ we can only store information about the general structure of our resilient directory. Everything else that grows with the size of the data structure needs to be stored in unsafe memory. Trees are especially vulnerable to corruptions because they heavily depend on positional data and traversing them requires following vulnerable pointers to find the next node. If at least one of node along a search path would get corrupted, any standard algorithm operating on the tree wouldn't be

able find the required nodes even though the keys and its values were uncorrupted.

To read important variables which should stay readable even if they get partially corrupted we need resilient variables:

Definition 3 (Resilient variable)**.** *A resilient variable consists of* $(2\delta + 1)$ *traditional variables* x_1 *up to* $x_{(2\delta+1)}$*. It will withstand up to* δ *corrupted words. A copy is called faithful or correct if it hasn't been corrupted, otherwise we call it faulty or corrupted.*

Proof. The proof is trivial: In the worst case, all δ corrupted words happen to be in distinct copies of said resilient variable. This leaves us with $(\delta + 1)$ uncorrupted copies. Because the majority of copies are uncorrupted, we return the value occurring the most. □

This property can be used to create a resilient read algorithm which always returns the uncorrupted value in $\mathcal{O}(\delta)$ time:

Algorithm 1: read()

```
1  y = x_1;
2  z = 1;
3  for i ← 2 to (2δ + 1) do
4      if x_i = y then
5          z = z + 1;
6      else
7          z = z − 1;
8          if z = 0 then
9              y = x_i;
10             z = 1;
11 return y;
```

Note that y, z and the algorithm itself need to be stored in safe memory in order to work correctly, otherwise the adversary could manipulate one of the three to produce incorrect results. This procedure only requires $\mathcal{O}(1)$ space and is therefore resilient. We instantiate such a resilient variable by setting all copies to our value. Writing to resilient variables has the same running time as a read to it and involves overwriting every copy:

Algorithm 2: write(value v)

```
1  for i ← 1 to (2δ + 1) do
2      x_i = v;
3  return;
```

2.1 Properties

The nodes v of the AVL tree with subtrees L and R represented in figure 1 keep the following information:

- Unsorted list of keys stored in v: $\mathcal{K}_v$ is stored unreliably in unsafe memory
- Number of keys in v: $|\mathcal{K}_v|$ is stored resiliently in unsafe memory
- Interval of keys stored in v: I(v) is stored resiliently in unsafe memory
- Union of I(v), I(L) and I(R): U(v) is stored resiliently in unsafe memory

It also guarantees us three invariants which are required to reason over algorithms following later:

IV 1 All keys of v are contained in I(v)

IV 2 The number of keys in v is linear in δ

IV 3 The structure of the tree is only modified after $\Omega(\delta)$ insertions and deletions

We get our first invariant **IV 1** from the fact that I and therefore U need to be continuous, because they will be used to direct the interval search to the correct node possibly containing the key.

2.2 Implementation

To guarantee the advertised behavior the following additional information is stored resiliently:

- Addresses of the left, right child and the parent
- Information to keep the AVL tree balanced

They need to be implemented as resilient variables, otherwise navigating or balancing the tree becomes infeasible after corruption. The root node and the number of its nodes are stored in safe memory. Keys are stored unreliably in unsafe memory in an array of size 2δ per node and are maintained to only contain $\delta/2$ to 2δ keys. This restriction makes it easier to argue about the cost of modifying the tree after insertions and deletions. The decision of storing keys unreliably is sensible according to our definition where we only consider uncorrupted keys.

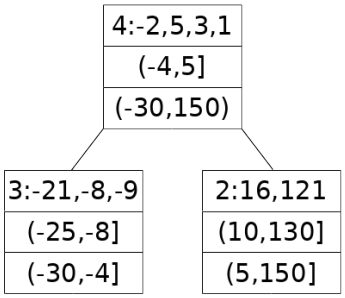

Figure 1: An example of a non-root node and its children: The first row contains its keys and the key count, the second row determines the boundaries of said interval $I(v)$. The third row saves information about the interval $U(v)$ of all nodes below it, consisting of the union of its own interval and all intervals of its children.

3 Algorithms

In the next section I will present the three most important operations on a standard dictionary: Search, Insert and Delete. Both Insert and Delete depend on Search to find the interval to manipulate, therefore an efficient search is vital for a good overall running time. We won't bother about the implementation of a resilient search just yet and rather focus on the impact of insert and delete operations after the corresponding interval has been found.

The insert and delete operations for the resilient dictionary focus on creating or merging intervals such that any new modification of the structure only happens after $\Omega(\delta)$ operations, allowing us to charge fraction of the cost of the modification on said operations.

3.1 Insert

Algorithm 3: insert(key k)

```
1  search node v such that k ∈ I(v);
2  if |K_v|< 2δ then
3  |  insert k into v;
4  |  return;
5  sortedKeys = sort(K + k);
6  lowerHalf = sortedKeys.firstHalf();
7  upperHalf = sortedKeys.secondHalf();
8  deleteNode(v);
9  L = create(lowerHalf);
10 R = create(upperHalf);
11 insertNode(L);
12 insertNode(R);
```

Nodes L and R are created form the lower and upper halves of v's keys, such that $L \cup R = I$. The

authors suggest the usage of a BubbleSort implementation developed in a previous paper [4] for sorting the keys and finding the first and second half. Their implementation allows resilient sorting while only requiring $\mathcal{O}(1)$ space of safe memory. When the interval contains 2δ keys, it needs to be split in half. Based on **IV 2**, this takes an additional $\mathcal{O}(\delta^2)$ time sorting step, giving us two newly created intervals containing exactly δ keys each, leaving space for an equal amount of new keys to insert.

3.2 Delete

Algorithm 4: delete(key k)

```
1  search node v such that k ∈ I(v);
2  if k ∉ I(v) then
3      return;
4  remove k from I(v);
5  if |K_v| > δ/2 or I(v) is the leftmost interval then
6      return;
7  L = predecessor of v;
8  if |K_L| ≤ δ then
9      keys = K_v + K_L;
10     delete(v);
11     v' = create(keys);
12     insertNode(v');
13 else
14     sortedKeys = sort(K_L);
15     top = sortetKeys.lastQuarter();
16     lowerKeys = K_L - top;
17     upperKeys = K_v + top;
18     deleteNode(L);
19     deleteNode(v);
20     L' = create(lowerKeys);
21     v' = create(upperKeys);
22     insertNode(L');
23     insertNode(v');
```

The line 1 to 4 handles the removal of key k from v, the rest of the algorithm distributes the remaining keys. We distinguish on the size of L to avoid the sorting step in the first case and instead merge both v and L into a new interval of size $(\delta - 1)$ to $\frac{6}{4}\delta$ for the cost of $\mathcal{O}(\delta)$. In the other case we will need to pay $\mathcal{O}(\delta^2)$ for the extra sorting step, we create two new intervals v' with size $(\frac{3}{4}\delta - 1)$ and L' where it's size ranges between $(\frac{3}{4}\delta + 1)$ and $(\frac{7}{4}\delta - 1)$. By exploring sequences of insert and delete operations one can see that the next modification only happens after at least $\delta/2$ insertions into a newly merged interval starting at line 8 or $\delta/4$ insertions or deletions in the case starting at line 13. Due to the choice of either splitting or merging we can therefore perform $\Omega(\delta)$ deletion or insertion steps without modifying the structure of the tree, allowing us to establish **IV 3**.

3.3 Search

So far we didn't go into the details on how to perform a resilient interval search. There exists a trivial approach where our search is directed by reading $I(v)$ resiliently, but this always leads to a linear running time penalty of δ until we reach the desired interval where we can perform our two main operations, resulting in the worst case running time of $\mathcal{O}(\delta \log n)$.

For the search operation, the paper proposes a new approach contrasting to the resilient interval search. Instead of performing a resilient read on the variables along the descent, only one copy of $I(v)$, $U(v)$ and the next pointer are chosen per round by an algorithm $\mathcal{A}'$. The result is then checked with another

algorithm $\mathcal{C}$, if the result is incorrect because we encountered a faulty copy along the way, we try again. As it turns out this approach, if done in a clever way, is significantly more efficient than the standard resilient algorithm $\mathcal{A}$.

Theorem 1. *Algorithm $\mathcal{A}'$ and $\mathcal{C}$ solve the same problem as $\mathcal{A}$.*

3.3.1 Randomized Interval Search

If we have access to an oracle which selects memory locations of variables uniformly at random, we can instantiate $\mathcal{A}'$ with such procedure to select a copy of each resilient variable during a round. The algorithm $\mathcal{C}$ is implemented as resilient read of the reached interval $I(v)$ of node v. If $k \notin I(v)$ we repeat both procedures as the check failed.

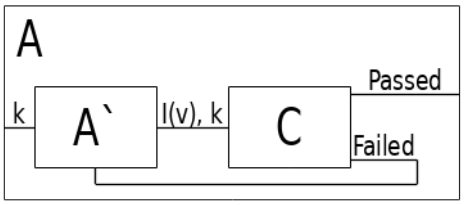

Figure 2: Reduction $\mathcal{A}$ using $\mathcal{A}'$ and $\mathcal{C}$ describing a single round. In this case $\mathcal{A}$ will not return anything if k doesn't exist or got corrupted.

Proof. We use the reduction of figure 2 described in the paper to show that the correctness of $\mathcal{A}'$ and $\mathcal{C}$ from above imply the correctness of $\mathcal{A}$. If $\mathcal{A}'$ returns a wrong answer, $\mathcal{C}$ would always discard it after the check and would rerun $\mathcal{A}'$. Because there are at least $(\delta+1)$ faithful copies in each resilient variable, $\mathcal{A}'$ will eventually return a valid answer which will be approved by $\mathcal{C}$. □

Theorem 2. *This randomized interval search $\mathcal{R}$ constructed from $\mathcal{A}'$ and $\mathcal{C}$ chosen from above is expected to have a worst case running time of $\mathcal{O}(log\ n + \delta)$.*

Proof. The proof follows the decomposition of $\mathcal{R}$ into $\mathcal{A}'$ and $\mathcal{C}$ showing that number of rounds used to find only faithful copies is constant.

We firstly show that $\mathcal{A}'$ takes $\mathcal{O}(log\ n)$ time in the worst case. Because it only accesses one of the copies of a resilient variable and the AVL tree as the basic structure of the dictionary guarantees a worst case running time of $\mathcal{O}(log\ n)$ when performing search operations.

To show that $\mathcal{C}$ runs in $\mathcal{O}(\delta)$ time we use that a resilient read described in algorithm 1 of both interval boundaries takes $\mathcal{O}(2\delta)$ time in the worst case.

Now we need to show that the expected number of rounds r in $\mathcal{O}(r(log\ n + \delta))$ for both $\mathcal{A}'$ and $\mathcal{C}$ are constant. At every node $i \in \{1, 2, .., d\}$ along the path in the dictionary we could encounter a number of actual faults α_i of corrupted interval boundaries or pointers to the next node. The probability that a variable read at the i-th node is faulty or correct is thus:

$$p^i_{\text{faulty}} = \left(\frac{\alpha_i}{2\delta+1}\right), \quad p^i_{\text{correct}} = 1 - p^i_{\text{faulty}} = \left(1 - \frac{\alpha_i}{2\delta+1}\right) \tag{1}$$

The probability that all d encountered nodes are faithful is therefore:

$$\prod_{i=1}^{d} p^i_{corr} = \left(1 - \frac{\alpha_1}{2\delta+1}\right)\left(1 - \frac{\alpha_2}{2\delta+1}\right)...\left(1 - \frac{\alpha_d}{2\delta+1}\right) \geq \left(1 - \frac{\sum_{i=1}^{d}\alpha_i}{2\delta+1}\right) \tag{2}$$

We can bound equation 2 by observing that the probability of encountering no error in d rounds is larger than the probability of seeing no errors if they were all in the same resilient variable by union bounds and set a lower bound for equation 3 with the knowledge that the number of faults $\sum_{i=1}^{d}\alpha_i$ along the path can be at most δ:

$$\left(1 - \frac{\sum_{i=1}^{d}\alpha_i}{2\delta+1}\right) \geq \left(1 - \frac{\delta}{2\delta+1}\right) > \frac{1}{2} \tag{3}$$

This means that the expected number of rounds until success are at most 2 and therefore constant. With this observation we can finally show that the expected worst case running time of this algorithm $\mathcal{R}$ is $\mathcal{O}(2(log\ n + \delta)) = \mathcal{O}(log\ n + \delta)$. □

3.3.2 Deterministic Interval Search

If we have no efficient way of sampling such a probability distribution, we are restricted to deterministic methods only. To select copies of resilient variables encountered along the path, the paper suggests a different solution with the usage of so called prejudice numbers.

Definition 4 (Prejudice number)**.** *The prejudice number p is a positive integer ranging from 1 to* $(2\delta+1)$*. It stands for the lowest index of copies of resilient variables which is still trusted. It starts with* $p = 1$ *and if any p-th copy of a resilient variable is discovered to be corrupted, p is incremented by 1.*

Furthermore the traversal with $\mathcal{A}'$ works in the following way: We start with a checkpoint, a node in safe memory representing the lowest node of which subinterval U possibly contains the key. The checkpoint is initially chosen to be the root of the tree. We then perform δ interval search steps based on one copy of $I(v_i)$ and $U(v_i)$ selected by the prejudice number in safe memory. We end up at node v', if $k \in I(v')$ we reached our destination. If not and $k \in U(v')$ we set v' as our new checkpoint to continue the descent, otherwise we increase the prejudice number and backtrack to the last check point due to being outside of the search interval at the current node. The algorithm can even follow corrupted pointers during one of the unchecked steps. This leads to the problem shown in figure 3 where it visits previous nodes and ends up less than δ steps below the last check point.

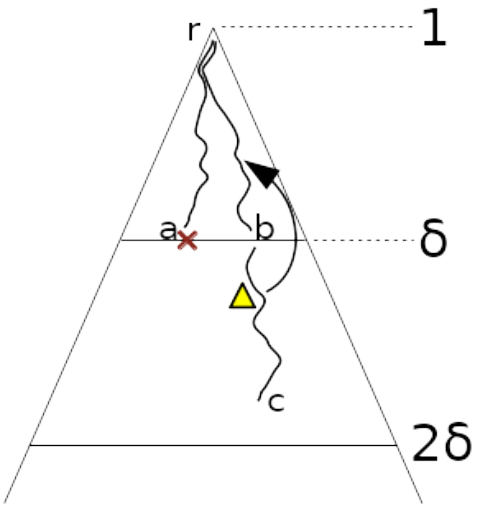

Figure 3: On the first traversal to node a we encountered a corrupted copy such that $U(a) \not\subset U(r)$ and have to increase our prejudice number by one. The second pass consists again of δ nodes where the check succeeds with $U(b) \subset U(r)$ and it is therefore our new checkpoint. In our third pass we follow a corrupted pointer (yellow triangle) which stays unnoticed. We pass the check and end up at node c which is less that δ levels below b.

Theorem 3. *The deterministic interval search has a worst case running time of* $\mathcal{O}(log\ n + \delta + \alpha\delta)$*.*

Proof. We look at all possible outcomes after each round to find suitable upper bounds. A round has a cost of $\mathcal{O}(2\delta)$ time, originating from traversing δ nodes and the $\mathcal{O}(\delta)$ check.

If we notice that $U(v') \not\subset U(v)$ holds at the end of a round, we need to backtrack to the last check point. By increment the prejudice number we avoid at least one corrupted copy meaning that after at most α failed rounds every copy chosen by the prejudice number has to be faithful. This implies that there are at most α failed rounds accountable for $\mathcal{O}(\alpha\delta)$ time.

When pointers can get corrupted, we could jump $\delta - 1$ levels up in the tree and all $\delta - 1$ selected copies of U encountered are conveniently corrupted we end up potentially only one node below the last checkpoint. This can only happen if the adversary spends his δ-sized budget on corrupting values such that the algorithm does not notice faults during the descent and the check at the end of the round succeeds because we are at least one node below the checkpoint. In the worst case we lose one round worth $\mathcal{O}(\delta)$ time.

If a round is successful, meaning that we did not cycle on previous nodes, we progressed by δ levels. If all rounds are successful, it will take $\mathcal{O}(2\delta)$ time for $(\log n)/\delta$ rounds summing up to a $\mathcal{O}(\log n)$ running time.

Considering all cases the total running time is therefore $\mathcal{O}(log\ n + \delta + \alpha\delta)$. □

3.3.3 Deterministic Interval Search (Hierarchy of prejudice numbers)

This approach can be expanded to smaller subintervals where additional subchecks are cheaper to run but in general less reliable to detect a fault. The paper is suggesting to split the δ rounds into $\sqrt{\delta}$, $\sqrt{\delta}$-sized subrounds, assuming that $\sqrt{\delta}$ is an integer number. The same mental partitioning is done on the copies

of resilient variables, resulting in $\sqrt{\delta}$ partitions of size $\mathcal{O}(\sqrt{\delta})$. To index a copy, two prejudice numbers p_0 and p_1 both residing in safe memory are used. The second one indexes the p_1-th partition of a resilient variable and the first one chooses the p_0-th copy within this partition. After each subround we perform a subcheck on the p_1-th interval only which might return a wrong answer because the subinterval is smaller and therefore susceptible to only $\sqrt{\delta}$ faults. If this check fails, we retry from the last subround check point saved in safe memory and increment p_0. If $p_0 \geq \sqrt{\delta}/2$, p_1 is incremented and p_0 is set to 1.

Theorem 4. *This implementation of deterministic interval search has a worst case running time of* $\mathcal{O}(log\ n + \delta + \alpha\sqrt{\delta})$.

Proof. Similar to the last proof, we first consider the cost of a round to be 3δ coming from the δ node traversals, $\sqrt{\delta}\mathcal{O}(\sqrt{\delta})$ subchecks and the $\mathcal{O}(\delta)$ check at the end.

If we fail a subcheck and there are less than $\sqrt{\delta}$ faulty copies in subinterval, the check executed correctly, we need to backtrack to the last subinterval. This can happen α times when only one copy is corrupted, this therefore costs us $\mathcal{O}(\alpha\sqrt{\delta})$ of additional time.

If a subcheck succeeds but is not executed correct, at least $\sqrt{\delta}$ faulty copies reside in subinterval p_1, we will only notice it at the end of another subround or at the end of a proper round. In both cases we load the check point of the last round, but we can eliminate least $\sqrt{\delta}$ faulty copies per subround backtracked by incrementing p_1. In the worst case we need to redo an entire round resulting in a $\mathcal{O}(\delta)$ time overhead.

If no faults are encountered at all we pay $\mathcal{O}(\log n)$ like in the previous proof. Together we get a worst case running time of $\mathcal{O}(log\ n + \delta + \alpha\sqrt{\delta})$. □

This approach can be extended to even smaller steps. This paper presents a version with subrounds of size $1/\varepsilon$, ε being a small constant in the range of $0 < \varepsilon \leq 1/2$, achieving a worst case running time of $\mathcal{O}(log\ n + \delta + \alpha\delta^{\varepsilon})$. There will be δ^{ε} subintervals of size $\delta^{1-\varepsilon}$. For $\varepsilon = 1/2$ we get the same algorithm as before. I will therefore not present both the procedure and the proof due to their similarity. This leaves us with the following result as our last theorem:

Theorem 5. *The search, insert and delete operation of the resilient dictionary operating on the faulty-RAM model has an expected worst case running time of* $\mathcal{O}(log\ n + \delta)$ *when uniform random sampling is available. It takes* $\mathcal{O}(log\ n + \delta + \alpha\delta^{\varepsilon})$ *time in the worst case for the deterministic variant.*

4 Commentary

They are able to achieve the same running time complexity of search with insert and delete by charging the sorting step on the by **IV 3** established $\Omega(\delta)$ previous insertions/deletions.

Over all the paper was fairly readable, some explanations how the interval search proceeds were ambiguous, they never mention how one spots a corruption outside of the checks other than using U during the search.

Most importantly it introduces interesting concepts like prejudice numbers while, also showing a new approach, as discussed in chapter 3.3.2, to speed up the interval search on the faulty-RAM model by a factor of δ.

References

[1] *Resilient dictionaries*, I. Finocchi, F. Grandoni, G. F. Italiano, 2009

[2] *DRAM Errors in the Wild: A Large-Scale Field Study*, B. Schroeder, E. Pinheiro, W. Weber, 2009

[3] *Single Event Effects in Avionics*, E. Normand, 1996

[4] *Optimal sorting and searching in the presence of memory faults*, I. Finocchi, F. Grandoni, F. Italiano, 2006