memory

Chhi'med Kunzang

July 27, 2019

Contents

1 ZigZag Replication Memory Requirements (as multiple of sector size)

1.1 Why does it matter?

- RAM is expensive and requires expensive configurations at high levels.
- We consider 64GiB the limit for ordinary operation.
  - Someone might always prefer more, but we ideally will not require it.
- Larger sectors are more economical from both a CPU and proof size perspective.

1.2 Merkle trees

- Most naively, we need enough memory to hold all trees at once.
- Improve by offloading trees to disk and reload for proofs.
- Reduce space usage per tree.
- Best memory usage holds log2 nodes but uses more CPU.

1.3 Encoding

- We want replication time to be strictly limited by linear encoding time.
- This means no waiting for parents to load.
- Parent distribution is random and resists locality.
- Hashing (currently blake2s) a parent for the KDF is faster than random-access latency on disk.
- This seems to imply we need to keep the entire replica in memory, if we don't want to sacrifice speed (which we don't).
- If we could take advantage of sequential reads, using disk might be fast enough to outrun hashing.

1.3.1 Idea: Ensure parents need not be accessed randomly.

- This implies greatly increased total space, since each parent has (on average) P (currently 13) children.
- Two approaches using multiple machines:
  - Allow replicating sectors larger than 64GiB but still requiring at least as much RAM as the sector size (on multiple machines).
  - Allow replicating sectors larger than the total RAM.
- Start with the first, less ambitous target and build up to the least bounded we can.

1. The way that doesn't work
   - Just write each parent to the ~13 locations optimal for encoding each of its children.
     - Unfortunately, this trades random reads for random writes, which are if anything more expensive.
2. Divide responsibility for one sector across multiple machines.
   - If writing to disk, we **can** do so in parallel. With enough machines, this makes the amortized cost of writing fast enough.
   - If holding in memory we can replicate sector sizes greater than our (64 GiB) RAM limit on any given machine.

- We will need much more total memory across all machines (order of * P) than the sector size, though.

3. Algorithm

   (a) Basic Operation

   - Each unit is responsible for encoding N consecutive nodes and runs on a separate machine.
   - Layout is [[Node1...Parents1]...[NodeN...ParentsN]] = N * ((1 + P) * 32) bytes per unit.
   - Units are arranged in a ring, such that each unit has a 'next unit'.
   - Units are ordered such that consecutive units are responsible for consecutive ranges of nodes.
   - After encoding a node, the encoding unit sends the node's index and new value to the next unit and also processes it.
     - The next unit forward the message to its next unit and begins processing.
     - This continues until all units have seen the message.
   - Each unit searches its layout for locations where the new node occurs in a 'parent' location, and stores the value there.
     - Naively, this requires the parent indexes to be colocated with the data. We will eliminate this later.
     - The originating unit also searches and writes the new value anywhere necessary in its remaining (future) layout.
   - At any time, one unit is responsible for encoding the current node.
     - Assuming all messages have been processed, it is guaranteed that all of the current parents have been received.
     - Therefore, the current parents exist in contiguous memory and can be looked up directly.
   - When a given unit reaches the end of the range of nodes for which it has responsibility, encoding passes to the next node.
   - Upon reaching the end of a layer, the direction of encoding reverses, and the process repeats with order of units and nodes reversed.
     - This requires a change of layout for each unit.
   - NOTE: There is clearly room for optimization, since with all data held in RAM, we do not need it to be sequential.

- We will offload some of this contiguous memory to disk.

(b) How are matching parent locations found?

- We can precompute a correctly-ordered map and store it alongside the node-parents layout.
- To precompute: record the locations (in layout coordinates) of each parent in layout.
  - If parent is not contained in unit's layout, no entry for parent is included in the map.
- Maintain a pointer/index to the current map entry, advancing as each matching node is processed.
- As each node is processed, its corresponding map entry can be immediately checked. If no match, do nothing.
- If the procesesd node matches the current map entry, store the node in each of the entry's parent locations, and advance the entry pointer.

(c) How do we reduce memory?

- As described here, more than P (13) times the sector size is required to replicate one sector.
- Since both the map and the layout eventually contain correctly ordered sequence data, they are well-suited to non-random reads.
- We can reduce total memory by choosing a percentage of both the map and the layout to store on disk.
- Writes to the memory-resident portion remain fast.
- Writes to the currently disk-resident portion become random-access and are slow.
- However, since these writes are spread across multiple machines, each with its own disk, the cost of writes can be amortized.
- Define one random-access write to be a factor of W slower than one KDF hash (ignoring the extra hash for the replica id).
- Define the fraction of each unit's map/layout which is stored to disk as D.
- For each node processed, we make P writes.
- Of these, D * P are random-access.

- In order not to be slower than encoding, we need that D * P * W <= 1.
- For example, suppose random writes are 5 times slower than KDF hashes. Then W = 1/5, P = 13, and D <= 5/13.
- That is, we can cache up to 0.385 of each unit.

(d) Is it worth it? And if so, at what scale?

- For now, estimate that we can cache 1/3 of each unit's data.
- Concrete example:
  - Ignoring the cost of the map, for now, total memory for 1TiB would be 14TiB.
  - Caching 1/3, we are left with 9.33TiB, or roughly 145 * 64GiB machines.
  - Assume this allows us to uninterruptedly encode.
  - Then we are able to replicate 1TiB using the same amount of RAM as could otherwise be used to replicate 9.33TiB.
- In order for this enormous overhead to be worthwhile, we would need to see a 9.33X benefit in combined proof size and proving CPU time savings.
- Proof size scales linearly with sector size, proving time logarithmically.
- Therefore, although we pay a premium of 9.3X, we save 145X in proof size, and 20X in proving CPU time.
- [Since we ignored and did not estimate the cost of the map, these numbers may be off by a relatively small factor.]
- So, at scale, maybe it is worth it. In this model, practical sector size is determined by replication speed and desired turnaround time for sealing.

(e) Further considerations

- Estimate the size of the unit's map.
- NOTE: even assuming calculations are correct, all numbers above are based on the hypothetical and unresearched 5x difference between one KDF hash and one random write.
- Which parts of the map/layout are cached, and how?
  - Since we want current reads to be fast, cache the last, not first portion.
  - As replication proceeds, we can reclaim memory which will not be needed again for this layer.

- Periodically flush the encoded nodes' map/layout to disk and replace it with disk-resident map/layout.
- At this point, we're manually managing disk paging.
- Our data is no longer random access.
- Can we just mmap the whole (correctly sized) data and let the system's virtual memory manage the problem for us?
  * Maybe not, since the random writes may wreak havoc with this. Needs better understanding.