Chia Proof of Space code documentation

This document aims to provide a high level explanation for Chia's Proof of Space plotter implementation (alpha version). For a more theoretical explanation of the construction, please review the other document.

Code structure

The plotter is implemented in both C++ and Python. In this document, we'll refer to the C++ version, as it's the faster of the two. The main plotting algorithm is done in plotter_disk.hpp.

The place where all 7 functions are implemented is calculate_bucket.hpp. All functions use a form of AES. We provide a fast implementation of it, using AES-NI in aes.hpp. The input/output of functions are arrays of bits. Python provides the bitarray class (used in our Python implementation), but there is no C++ alternative for that library, so we have implemented our own in bits.hpp.

We need to sort large amounts of data in order to be able to efficiently calculate the next function, given the previous one. We can set a parameter on how much data is sorted in memory (the constant kMemorySize from plotter_disk.hpp). The external sorting algorithm (sorting more data than memory allows, by using disk operations) is implemented in sort_on_disk.hpp.

The prover/verifier are also implemented in prover.hpp/verifier.hpp. The prover has the role that, given a plot file and a challenge, to retrieve the proofs present in the file for that challenge. The verifier checks if a provided proof is valid and, if so, returns its quality.

Finally, our command-line interface is implemented in cli.cpp. It uses the external library cxxopts.hpp.

You can run plotter by going into the root folder. Then, do:

```
make
make test
```

This will produce an executable that will do the plotting. To run it, for example:

```
time ./ProofOfSpace -k 25 generate
```

Or run tests:

```
./RunTests
```

CLI usage

./ProofOfSpace -k 25 -f "plot.dat" -m "0x1234" generate
./ProofOfSpace -f "plot.dat" prove <32 byte hex challenge>
./ProofOfSpace -k 25 verify <hex proof> <32 byte hex challenge>
./ProofOfSpace -f "plot.dat" check <iterations>

Bits class

This class stores an array of bits. We can perform array-specific operations (eg concatenation or finding a substring with function 'Slice') as well as treating the array as a huge binary number and performing increments, decrements, xor between two bit arrays, and left/right shifts. Another function of the class is to transform the bit array into an array of bytes. This is needed in order to output to disk: by using fstream with reinterpret_cast<char*> we can write byte arrays onto the disk file.

The class is implemented using an array "values_" of unsigned int128. Each uint128 is able to store 128 bits. We use values_[0] to store first 128 bits of the bit array, values_[1] to store next 128 bits of the bit array and so on. In this way, all elements of values_ array contain exactly 128 bits, except for the last one, which may contain fewer. This is done for both memory efficiency as well as speed. The size of the last group of bits is stored into last_size_.

We also need to remember the total size in order to differentiate between objects. For example, if values_ = {5}, the bitarray might be 101 or 0000101. In order to differentiate, we'll say first bit array is Bits(5, 3) and the other one is Bits(5, 7), where the last parameter is the total length. Note that each Bits object corresponds to a single bit array.

Most functions use AppendValue to append a new value at the end of the existing bitarray. This is the function that ensures that values are always grouped into 128-bits chunks, possibly except for the last chunk, which contains last_size_ bits in it.

Finally, Bits is a generic class, accepting the array type as a template. The custom array type must implement the std::vector methods needed in the code. The reason we use custom arrays is that most Bits objects are small, and using the standard std::vector would spend a lot of time with heap allocation/deallocation. That's why, for most Bits objects, a stack-array of length 5 is enough, which spends no time allocating.

We use Bits as an alias for BitsGeneric<SmallVector> (stack-array of length 5), ParkBits as an alias for BitsGeneric<ParkVector> (stack-array of length 128) and LargeBits as an alias for BitsGeneric<std::vector> (dynamic heap array).

Calculate bucket class

Functions are implemented using two classes: F1Calculator (for f1) and FxCalculator (for f2, ..., f7). To calculate the output of f1 for a given Bits object, call CalculateF. CalculateBucket(x) returns the pair (CalculateF(x), x), where x is the Bits object. Finally, the function CalculateBuckets evaluates CalculateBucket for a range of consecutive Bits objects, starting from start_L and ending in start_L + number_of_evaluations - 1. Using CalculateBuckets for a range of consecutive inputs is preferable to calling CalculateBucket with one input at a time, because it will be faster. The reason for this is that some AES calls overlap between consecutive inputs, so we can use a single call in CalculateBuckets for those. Otherwise calling CalculateBucket for each input would have to call the overlapping AES functions each time. F1 function is implemented as AES256 with 14 rounds, producing k + kExtrabits (=5) output bits.

Each entry is represented by (y, metadata). For table 1, we get that by calling f1.CalculateBucket(x) (which calculates the pairs of form {CalculateF(x), x}). To get the content of the next table, we use the pairs (y, metadata) from the previous table. We divide those entries into groups, the group of a pair (y, metadata) is y / kBC. We only consider entries from adjacent groups in calculating the next table (i.e. let (y1, metadata1) and (y2, metadata2) be entries from the previous table. If min(y1, y2) / kBC != max(y1, y2) / kBC + 1, those 2 entries won't contribute in the current table).

Let bucket_L be all entries from a group x and bucket_R be all entries from the group x+1. Calling FindMatches for bucket_L and bucket_R will determine which entries will produce new entries in the next table. The function returns a list of pairs {(i1, j1), (i2, j2), .., (ik, jk)}, meaning bucket_L[i1] and bucket_R[j1] will produce a new entry in the next table, bucket_L[i2] and bucket_R[j2] will produce a new entry in the next table and so on. When two entries from adjacent groups produce an entry in the next table, we call them matching. The logic behind the matching function is explained in the other document.

Say we get two entries (y1, metadata1) and (y2, metadata2), which match. Those two will produce an entry (y', metadata') of the next table. This pair is determined by calling CalculateBucket(y1, y2, metadata1, metadata2). Y' is in turn determined by calling CalculateF of FxCalculator, and metadata' is determined by calling Compose. CalculateF does 2-round AES128 of the metadatas (split into 4 numbers La, Ra, Lb and Rb). Then, the result is xor-ed by y1, to give more randomness of the output. Compose result is determined by metadatas only.

Plotter disk class

The plotter algorithm works by having 3 phases: forward propagation (done in WritePlotFile function) and back propagation (done in Backpropagate function) and compress (done in Compress function). The forward propagation creates a bigger temporary file than the final file

(plot_filename + ".tmp"), which is not needed once the final file is calculated. Backpropagation removes the ephemeral data stored in the temporary file (described in more detail below). Compress creates the final plotfile, where useful entries are stored more compactly in order to save space.

Forward propagation

The first step is to allocate space at the beginning of the file for table pointers (in WriteHeader), which will be used to quickly locate the beginning of each table after they are calculated, as well as other information specific to the plot. Then we write the f1 table, concatenating f1(x) and x, for every x from 0 to 2^k - 1, in Bits format.

The next step is to incrementally build the tables. All tables (except for the first one) have the following information stored: y, pos, offset, and metadata. When writing to the disk, the numbers are concatenated as Bits objects, then converted to bytes arrays and written. When processing them, we read back the byte arrays from disk, then slice from it in order to obtain the information, which is stored into a PlotEntry struct (containing the fields y pos offset and metadata). Note that metadata might exceed 128-bit numbers, in that case we store it as left_metadata and right_metadata (left containing the first 128 bits of metadata, right containing the remaining ones).

Y and metadata fields were explained above, their importance is to calculate the next table. The pos and offset fields represent the position in the plotter file where L and its respective R can be found. L can be found at position pos and R can be found at position L + offset. This is important for retrieving the previous entries that generated the current one (which will be used in proving, to get from a challenge all the way down to table 1 entries).

When we calculate the next table, we first sort on disk the previous table by y. This will ensure that the elements in the adjacent groups are close to each other. The table will contain all entries from group 0, then all entries from group 1, and so on. We'll continue by iterating over the table, reading from disk, until the end of table marker is found (a line full of zeros). We also keep a counter bucket which stores the first bucket we didn't match with the next one. We always keep two vectors with the following contents: bucket_L contains all the PlotEntries in the counter's bucket, and bucket_R contains all the PlotEntries in the counter's bucket plus one (bucket+1). When the bucket of the currently read PlotEntry becomes greater or equal to bucket+2, we finally calculate matches for bucket_L and bucket_R (since we know we have all of them stored in the vectors) and we update the bucket, as well as the vectors bucket_L and bucket_R accordingly.

Back propagation

The logic behind the backpropagation algorithm is to eliminate all of the entries that don't contribute anything to the seventh table. Let's say we want to cleanup the sixth table. Intuitively, we'll erase all positions from the sixth table that do not appear in the seventh table (either as pos or pos+offset of an entry of seventh table). Now we have the "correct" sixth table. We can use that to eliminate irrelevant entries from the fifth table. So if a position from the fifth table doesn't appear in the new sixth table (either as pos or pos+offset of an entry in the new sixth table), we won't store this. This logic is applied all the way down to the first table. Note that when we are deleting entries from tables, some positions will change, so we'll have to consider this in our implementation.

We begin by having read/write file handles for a "left" and "right" table. The left table is the one that we are cleaning up, and the right table is the next one. So, at the beginning, the left table will be table six and right table will be table seven. We firstly sort on disk the right table by positions (by the field PlotEntry.pos).

Imagine for now we have enough memory (i.e. a small plot). We can keep a boolean array used_positions, initially all false. We iterate over the seventh table using the plot_entry variable. Now, we store used_positions[plot_entries.pos] = true and used_positions[plot_entries.pos + plot_entries.offset] = true. Now, we iterate over the sixth table. We have a very simple way to tell if we should keep or delete an entry from position pos: if (used_positions[pos] == true), we keep it, otherwise we delete it.

We can adapt this idea by using a sliding window technique: used_pos vector will have the size cached_positions_size (in the implementation 1024). This permits us to process the first 1024 positions from the left table. After finishing with the first position, the memory will be used for 1025th position. Likewise, after finishing with second position, the memory will be used for 1026th position and so on. This works fine, considering that we don't handle more than 1024 positions from the left table at a time. When we are dealing with a position from the left table, we have to quickly verify if it appears in the right table (either as pos or pos+offset). This is where sorting the right table by pos comes in handy.

We'll read from the right table simultaneously while reading from the left table. Say we are at position 25000 in the left reader. The right reader should be synced with that position as well (i.e. plot_entry.pos is close to 25000 for the read entry in the right table). Luckily, since the right table is sorted by pos, keeping the right table synced means just advancing the right reader handler as long as (plot_entry.pos) < (current position in the left table).

In our implementation, current_pos represents where the read handle is on the left table. This is advanced incrementally as we process the left table. We'll also have a write handle for the left table, that simply rewrites the useful entries from the left table (we rewrite the good entries instead of deleting the bad ones, as it's more efficient). As stated, the right reader will need to

be synced, such as plot_entry.pos is the closest element smaller or equal to current_pos. The right table also has a write handle. This is necessary, as the positions in the left table change since we do not rewrite the bad entries. This in turn means that all right entries will need to be modified (with updated pos and offsets from the new left table).

Putting everything together, the algorithm starts from the beginning of the left table. We read from the right table, until the pos element of the right entry is greater than the current left table position. As we read the right entries, we update the used_positions vector (for a right entry, we update pos and pos+offset in the vector, using the sliding window technique). At this point, the used_positions vector will be updated to be able to answer if the current left position appears anywhere in the right table (the current left position will not appear from this point on in the right table, all positions will be greater than it, since the right table is sorted by pos).

If we see that the current left position appears in the right table somewhere, we rewrite the entry in the left table. We also increment a counter, left_entry_counter. This is the new position of the left entry into the new left table. We keep a map, new_positions, which does the mapping between the old positions in the left table, and where they are now in the new left table. So, after we increment left_entry_counter (for a rewritten “good” left entry), we store new_positions[current_left_position] = left_entry_encounter. This will be used when rewriting the right table content to write the correct pos and offsets.

One more space optimization we do is that we do not rewrite metadata when rewriting the left table element. Recall that metadata was useful only to calculate the next table, but now, as everything is calculated, we don’t have to store it anymore. Also, we keep the new “sort_key” field, meaning the initial position of the entry in the table. As the plotting algorithm progresses, the initial ordering will be lost, so we need to somehow be able to reconstitute it. So, we store sort_key (=left_entry_counter), as well as pos and offset. For table 1, we’ll store only y and x.

In order to write the updated positions in the right table, we keep two additional vectors: old_sort_keys and old_offsets. Say we have a right entry (sort_key, pos, offset). When we read it with the right reader, we store sort_key in old_sort_keys[pos] and pos+offset in old_offsets[pos]. This way we know exactly what right entries contain a particular left position pos. Say we want to “correct” all entries containing given position pos in their PlotEntry. We know that now, pos is no longer pos in the “new” left table, but new_positions[pos]. As well, offset is no longer offset, but new_positions[pos+offset]-new_positions[pos]. The entry itself is located at new_positions[pos+offset] in the new left table, but to get the offset, we need to subtract the new position. So if we wanted to write all the entries that contain pos = x, we’d iterate over old_ys[x] and old_offsets[x]. Say the current index is i. We’ll store y = old_ys[x][i], pos = new_positions[x] and offset = new_positions[old_offsets[x][i]] - new_positions[x]. The tuple (y, pos, offset) is a new entry that can be written to the right table, now having pos and offset updated.

In order to make sure that all the new positions are actually calculated by the time we write all right entries containing a given left position, we keep the number read_minus_write (in the implementation 256). This means, when we read the position current_pos (in the left table), we are allowed to write all the right entries containing pos=current_pos - read_minus_write. For example, when we've arrived at position=500 in the left table, we'll rewrite all right entries containing pos=500-256=254 (in other words we iterate over old_ys[254] and old_offsets[254]). The same sliding window technique is used here, this time using window length = 256.

Compress

This is the step that writes the final file, and happens after Backpropagation, when all the useless entries were dropped and, moreover, all tables were sorted by PlotEntry.pos (except for table 1).

The format of the final file will be: table 2, containing L and R (where L and R are xs from table 1 that generated the table 2 entry), then table 3, containing pos1 and pos2 (where pos1 is the position of the L from the new table 2 that generated the entry, and pos2 is the position of the R from the new table 2 that generated the entry), then table 4, containing pos1 and pos2 (with the same meaning, but they belong to table 3 now) and so on.

Pos1 and Pos2 are grouped into a line_point, which is obtained by applying SquareToLinePoint. We need to sort all the tables by line_points, as this is necessary for our compression algorithm. Pos1 and pos2 will always refer to a previous table sorted by line points (except for table 2, where they simply refer to the metadata). This is different than pos and offset, which refer to tables sorted by forward propagation ordering. We need to find an efficient map between the old way of referencing entries (pos and offset) and the new way (pos1 and pos2, where the previous table is sorted only by line_points).

As before, we'll refer to the left table as table t-1, and the right table as the table t. We always keep the following invariant: the left table is sorted by the ordering of the forward propagation (sort_key field), while the right table is sorted by the PlotEntry.pos ordering. We iterate from table 1-2, then 2-3 and so on.

In the temporary file, the left table with contain the sort_key (needed for restoring the forward propagation order), as well as the new position of that entry (when entries are sorted by line_points, what's the position of that entry in the ordering). This allows the quick mapping between the old positions and the new ones in the right table: if we know pos (in forward propagation ordering) is new_pos (in line_point ordering), then in the right table, everytime we find a pos, we replace it with new_pos (and we apply the same logic with pos+offset).

The algorithm works similar to Backpropagation: we iterate at the same time the left and the right table. Left table is of form sort_key, new_pos, sorted by sort_key (except for table 1, where we store y and metadata). We store the mapping between old pos and new pos in left_new_pos

array, left_new_pos[pos] representing either the line_point sorting position of the entry at position pos, or the metadata that generated the entry at position pos (in case of table 1). Left_new_pos is implemented as a sliding window.

For the right table, we do a similar sliding window technique as described above for Backpropagation. We keep old_offsets[pos] and old_sort_key[pos], representing pairs of offset and old_sort_keys that contain PlotEntry.pos = pos. We then use left_new_pos to get both pos and pos+offset to their line_point sorting ordering (or their metadata in case of table 1).

We then sort the right table, now entries are sorted by line_point ordering. Having the right table in this ordering, we write it in the final file, compressing pos1 and pos2 into Parks (details are described in the other paper).

Finally, the right table will become the left table in the next step and we need to have the right table sorted by forward propagation ordering (sort_key). So, in the temporary file, we iterate over the right table. Let i be the i-th entry read. We know i is the right position in the line_point ordering, so we simply write back sort_key+i. (corresponding to the format old_pos new_pos the left table must have). Finally, we sort again the right table, giving back the forward propagation ordering and allowing to have the map between old positions and new positions next step, when the right table will become the left one.

C Tables

Finally we can discuss the C tables. We need to quickly be able to identify if y (a challenge) appears in f7. After identifying position(s) where it appears, answering full queries just becomes retrieving pos and offset information all the way down to table one. The logic goes as follows: we have 3 tables C1, C2 and C3. Table C1 stores the y value for every 10000 entries of table seven. Similarly, C2 stores the y value for every 10000 entries of the C1 table. So every 10000th entry is stored in C1, every 100 millionth entry is stored in C2. When given a challenge y, we can use C2 to locate the part of the plot where it belongs. Then, we can use C1 to further refine the location. The table C3 stores precisely all ys, so after having some general guidance where the entry y can be, we can find exactly the positions where y appears. Recall the table is sorted by y, so the C1, C2 and C3 tables will give the ranges for all possible locations of the challenge y (C3 finally giving the exact range). The format of C3 is as follows: suppose the current entry has the value y and the previous entry has the value prev_y. Let dif = y - prev_y (always >= 0, since the table is sorted by y). We store then a 0 (delimiter between current and previous entry), followed by the difference in 1s. So, say y = 3 and prev_y = 3, we simply store 0. If y = 5 and prev_y = 3, we store 011. Going firstly over C2, then over C1, then finally over C3, we can finally find the position where y belongs in the seventh table.

Sort on disk

Given that the final plot file can be large, we can't sort all the data in memory. We define a parameter, kMemoryLen (in the current implementation this is 2GB), representing the maximum amount of memory allowed at any time for sorting. The rest of the data will need to be sorted using disk operations, each time dividing the data that needs to be sorted into smaller pieces. Finally, when the length of the data becomes smaller than kMemoryLen, we can sort that using the preallocated memory.

The main idea of sort on disk is as follows: divide all entries into buckets and write them back in the disk, in the bucket ordering. Once this is done, recursively call the sort for each bucket.

Dividing into buckets works as follows: the first four "non-skipped" bits represent the bucket. So we have bucket 0, having all entries in it starting with bits 0000, bucket 1 having all entries in it starting with bits 0001, …, bucket 15 having all entries in it starting with bits 1111. We write back all entries from bucket 0, then all entries from bucket 1, …, then all entries from bucket 15. This guarantees all entries are sorted correctly, considering only the first 4 bits. Then, we recursively call sort on disk, for bucket 0, 1, …, 15, but this time skipping the first 4 bits. So, for example, say we wrote all entries from bucket 0, then calling sort on disk to recursively sort them. Now, the bucket 0 of the new entries will start with 00000000, bucket 1 will start with 00000001, …, bucket 15 will start with 00001111. Note that now the first 4 bits are completely skipped when we calculate the new buckets, instead we use the 5th to 8th bit to determine the new buckets. Bits_begin variable represents the first bit used to determine the buckets. We keep doing this, until it is small enough to sort in memory (number of bytes is less than kMemorySize), then we'll apply the sort in memory phrase.

Writing into the correct locations is done using the vector bucket_sizes, which is always calculated before calling sort on disk. Bucket_sizes[i] represents the number of entries from bucket i. Let disk_begin be the disk position of the first entry and let entry_len how many bytes an entry will occupy. Then, we know all entries from bucket 0 will be from disk position disk_begin to disk position disk_begin+bucket_sizes[0]*entry_len (exclusive). All entries from bucket 1 will be from disk position disk_begin+bucket_sizes[0]*entry_len to disk_begin+(bucket_sizes[0]+bucket_sizes[1])*entry_len, exclusive.

Having this information, it becomes easier to place an entry into its correct bucket. Let written_per_bucket be an array, having written_per_bucket[i] represent how many entries were actually written into their correct bucket i. Say we want to write another entry which should be in the bucket i. Let bucket_begin[i] be bucket_sizes[0] + bucket_sizes[1] + .. + bucket_sizes[i-1].

We know then that the disk position to write the new entry is disk_begin + (bucket_begin[i] + written_per_bucket[i]) * entry_len (i.e. the first unwritten disk position inside the bucket i).

In order to store the buckets, we use BucketStore class. It uses the preallocated memory of size kMemorySize to store the entries. Entries from the same bucket are stored into segments, which contain entries_per_seg entries in each segment (in the current implementation the number is 100).
BucketStore provides the following arrays: bucket_head_ids[b] represents the memory id of last written segment for bucket b, bucket_head_counts_[b] represents how many entries are written into the last segment for bucket b. Moreover, the first 4 bytes of each segment store the memory id of the next fully written segment.

The overall structure of the BucketStore is a linked list. In order to retrieve all entries from a bucket from memory, we first go to bucket_head_ids[b]. We retrieve entries from that segment, then use the first 4 bytes of the segment to find the next segment of the bucket b, next_full_seg_id. We then set bucket_head_ids[b] equal to next_full_seg_id and continue until we get the desired number of entries from a bucket.

Initially, all segments are empty and also organized as a linked list. We store first_empty_seg_id as the head of the linked list, also having the first 4 bytes as the id of the next empty segment. To store a new entry into bucket b, we retrieve bucket_head_ids[b] and check if bucket_head_counts_[b] still has space for one more entry. If it does, we put the entry in the current head bucket. Otherwise, we use first_empty_seg_id to get a new empty segment. This becomes the new head of bucket b, the old head becomes the next segment of the current head, also first_empty_seg_id needs to be updated to the next empty segment.

Let’s put everything together: we initially load entries from each buckets to the “spare” zone. The spare zone will contain about 5 * num_buckets * memory_len entries in it. We remember consumed_per_bucket[i] as the number of entries that are moved from bucket_begins[i] in the original memory zone to spare. Then, we load the BucketStore, as long as it’s not getting full and as long as there are still entries in the spare zone. Let’s also define written_per_bucket[i] as the number of entries written back to bucket_begin[i] from BucketStore, such as they are “correct” entries (i.e. they really belong to bucket i). At any step, we must have written_per_bucket[i] <= consumed_per_bucket[i], for every i (in other words, we never lose any entry by overwriting it to an entry from BucketStore).

The algorithm moves entries from BucketStore to their correct bucket in the original memory zone, then refills BucketStore. In order to empty the BucketStore, we sort the buckets decreasingly by size. Hence, we start to empty the BucketStore starting from the bucket with the most number of entries in it and then decreasing. For any bucket i, we can’t take more than consumed_per_bucket[i] - written_per_bucket[i] entries from it, even if the bucket contains more entries in memory. In order to refill the BucketStore, we sort the buckets increasingly by consumed_per_bucket[i] - written_per_bucket[i] and start filling the BucketStore from buckets in

this order. Each time we load one entry from bucket i into BucketStore, we increment consumed_per_bucket[i] by 1. We stop when BucketStore is full again. If we've filled everything from a bucket (consumed_per_bucket[i] = bucket_sizes[i]), we move to the next bucket according to the sorting order. Finally, if every bucket is consumed, but BucketStore is still not full, we start loading it from the entries left from the spare zone.

Sort in memory (bucketsort)

When the data size is small enough to sort everything in memory, we use two different algorithms: quicksort, when the data is not random, and SortInMemory algorithm, when the data is random. We'll describe the latter.

Say the number of entries is n. Find the smallest power of 2 greater or equal to 2*n (i.e. minimum x such as 2^x >= 2*n). We are going to do a similar thing to bucket sort, using 2^x buckets (i.e. sorting by the first x "not-skipped" bits).

Say all the entries have the bits_begin common prefix. To determine the bucket of an entry, look at the first x bits, starting from bits_begin. Given that the data is random, we can expect the entry will be close to that position in the final sorted array. The intuition is that we are using twice as many buckets as the number of entries, so we can expect each bucket to contain very few entries in it.

For example, say our array is [11110010, 11110101, 11111110, 11111011]. We have n = 4, bits_begin=4 and x=3 (2^3 >= 2*4). This means we have 8 buckets: 000, 001, .., 111.

Entry 1111**001**0 goes into bucket 1, entry 1111**010**1 goes into bucket 2, entry 1111**111**0 goes into bucket 7 and entry 1111**101**1 goes into bucket 5.

Say result is the final array. Then, we do result[1] = 11110010, result[2] = 11110101, result[7] = 11111110, result[5] = 11111011. Result looks as follows after the assignments: result = [0, 11110010, 11110101, 0, 0, 11111011, 0, 11111110]. Removing the 0s (unused buckets) and you get the sorted array.

The problem we need to fix now is what happens when two entries are in the same bucket (both will be written to the same position of result, so one of them will be overwritten and lost). The algorithm proceeds as follows: let pos be the bucket (the expected position of the entry). If result[pos] is not filled yet, write the current entry there. Otherwise, compare result[pos] with the current entry. If the current entry is smaller than result[pos], swap the two. Then, increment pos by 1 and continue until we find an empty position. After we finish inserting all the entries, to extract the entries, simply iterate over the result, skipping the unfilled positions.

For example, say we get the array [0001, 0010, 0100, 0000] with bits_begin=0. We get bucket size = 3 (2^3 >= 2*4). After inserting the first 3 entries, the result looks like [0001, 0010, 0100, 0, 0, 0, 0, 0].

Now we are inserting 0000 and the expected position is 0. Since position 0 is occupied, we compare 0000 and 0001. Since 0001 is greater than 0000, we swap them and get:

Result = [0000, 0010, 0100, 0], pos = 1 and current_entry=0001.
Then, we get result=[0000, 0001, 0100, 0], pos=2 and current_entry=0010
Next, we get result=[0000, 0001, 0010, 0], pos=3 and current_entry=0100
Finally, pos=3 is empty so we insert 0100 and get [0000, 0001, 0010, 0100].

The practical details of the implementation are as follows: instead of reading/writing one entry from disk, we read/write in buffers of length 262K bytes. After reading a buffer, we fill the memory with all of its content, then read another buffer and fill the memory and so on. For the writing phase, we first fill the write buffer completely (iterating over the result and taking entries that are not empty), then write the buffer, then repeat.

Another practical detail is that we don't have to store the whole entry_len bytes in memory. Instead, we are storing only entry_len - bits_begin / 8 bytes (since the others are common) in memory and store the bits_begin / 8 common prefix in a separate array, only once. When reading from the read buffer, we simply ignore the first bits_begin/8 prefix every time when adding an entry to the memory. When writing to the write buffer, we first paste back the bits_begin/8 prefix into the write buffer followed by the memory entry.

Quicksort vs bucket sort

In 1st phrase of Compress, we always do quicksort, as data there is not uniform. Otherwise, we do quicksort as well for the last bucket given by sort on disk algorithm. The reason is the bucket might be incomplete, so this could make it less uniform as well. We use quicksort variable in sort on disk to denote the type of sorting: quicksort=0 means we do bucket sort, quicksort=1 means we are in 1st phrase of Compress, so we do quicksort and quicksort=2 means this is the last sort on disk bucket (note that quicksort=2 isn't permanent, but quicksort=1 is, meaning it keeps its property for all recursive calls as well).