Practical Implementations of Compressed RAM

Seungbum Jo (Chungnam National University, South Korea)
Wooyoung Park (Seoul National University, South Korea)
Kunihiko Sadakane (The University of Tokyo, Japan)
Srinivasa Rao Satti (Norwegian University of Science and Technology, Norway)

DCC 2023

Problem Definition

Given a string S over alphabet $\Sigma=\{1,\ldots,\sigma\}$ of size n, consider a data structure that supports the following operations:

1. **access(i):** Return S[i].

3	7	**5**	2	6	3	1

access(3): Return 5

2. **replace(u, c):** replace S[i] with c.

3	7	5	**2**	6	3	1

→

3	7	5	**7**	6	3	1

replace(3, 7)

- These two operations are basic operations on standard RAM.
- We assume that word-RAM model with word size $\Theta(\log n)$, which means access operation can return any $\Theta(\log_\sigma n)$ consecutive symbols in $O(1)$ time.

Problem Definition

We also consider the following two additional operations on S

3. **insert (i, c):** Insert c between S[i-1] and S[i].

3	7	**5**	2	6	3	1

→

3	7	**6**	**5**	2	6	3	1

insert(3, 6)

4. **delete(i):** Delete S[i].

3	7	5	**2**	6	3	1

→

3	7	5	6	3	1

delete(4)

- Data structure which supports access, replace, insert, and delete is called dynamic RAM.

Problem: Construct dynamic RAM using small space while supporting the operations efficiently.

Here, the small space means close to $H_k(S)$ (the k-th entropy of S).

Previous Results

- Theoretically, there are several works ([Jansson et al. 12], [Grossi et al. 13], [Munro and Nekrich 15]). All of them support the queries efficiently while using o(H_k(S))-bit additional space from H_k(S).

- Also there are some practical implementations

SPSI (Searchable Partial Sums with Insert) [Prezza 17]

: Uncompressed, delete is not implemented

(input can be compressed, delete is implemented when σ =2).

KN [Klitzke and Nicholson 16]

: Based on the work of [Jansson et al. 12] with LZ compression.

Our Results

New Practical implementation of compressed dynamic RAM while supporting access/replace/insert/delete operations

- Based on [Jansson et al. 12] (CRAM) and [Grossi et al. 13] (DCRAM).
- Compressed S based on Huffman code (KN uses LZ-based compression).
- As in theory, the space changes based on the current input's entropy (this is not supported in SPSI and KN).
- Optimized for sequential operations.
- Compared our implementation with SPSI and KN.

The source is available at https://github.com/wyptcs/CRAM

[Jansson et al. 12] (CRAM)

Codetable(before)	
code	block
0	AA
1	AD
00	AB
01	AC
10	BA
11	CB

Codetable(after)	
code	block
0	AA
1	AB
00	AC
01	BA
10	CB
11	AD

r = 2, ε = 0.5, superblock size: r/ε = 4

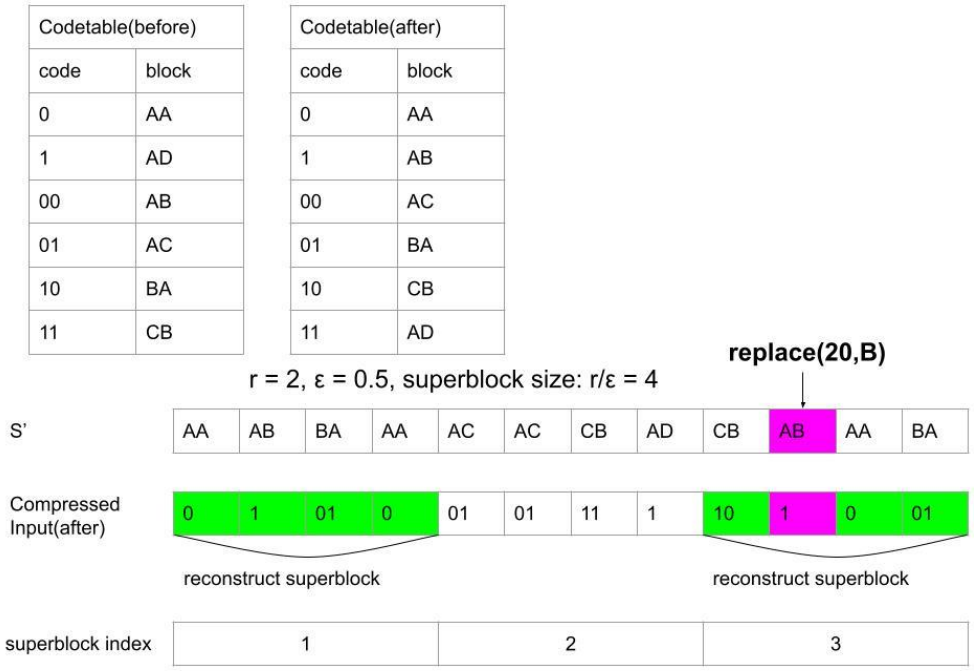
replace(20,B)

S	AA	AB	BA	AA	AC	AC	CB	AD	CB	AD	AA	BA
Compressed Input(before)	0	00	10	0	01	01	11	1	11	1	0	10
superblock index	1				2				3			

CRAM in theory (for access and replace)

- Divide S into blocks of size $r = 1/2 \log_\sigma n$
- Consider S as a string S′ of length n′= n/r over an alphabet 2^r, and construct a code table for S′ based on the frequency of the symbols in S′.
- Consecutive r/ε (0 < ε <1) blocks form a single superblock.
- Special DS (darray) for answering the end position of the i-th superblock while supporting updates, access can be answered in O(1) time using darray.

[Jansson et al. 12] (CRAM)



CRAM in theory (for access and replace)

- For replace, maintain two encode and decode tables (old and new).
- When i-th replace operation occurs, re-encode the superblock that contains the cor. Position, and update the frequency table.
- In addition, we update (i mod n') superblock using new encode table.
- After n' updates, reconstruct the tables based on the current frequencies of the symbols.
- In overall, supporting replace in O(1/ε) time.

[Grossi et al. 13] (DCRAM)

Codetable	
code	block
0000	AA
0001	AC
0010	unused
0011	unused
….	unused
1111	unused

Codetable	
code	block
00000	BA
00001	CB
00010	AD
00011	AB

r = 2

replace(20,B)

S	AA	AB	BA	AA	AC	AC	CB	AD	CB	AD	AA	BA
Compressed Input(before)	0000	00011	00000	0000	0001	0001	00001	00010	00001	00010	0000	00000

DCRAM in theory (for access and replace)

- S′ and darray structures on the blocks.
- Each symbol c in S′ is divided into class C_j based iff their frequency is between n′/2^j and n′/2^{j+1}, and we assign a code of length j+3 for the symbols in C_j (Hence, we have **some unused codes** for each class).
- access can be supported in O(1) time, similar as CRAM.

[Grossi et al. 13] (DCRAM)

Codetable	
code	block
0000	AA
0001	AC
0010	AB
0011	unused
....	unused
1111	unused

Codetable	
code	block
00000	BA
00001	CB
00010	AD
00011	AB

r = 2

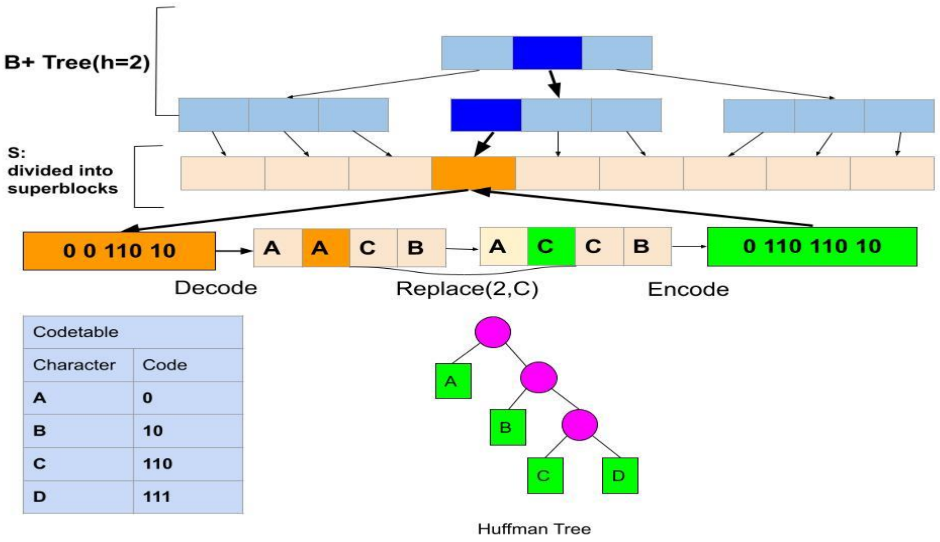
replace(20,B)

S'	AA	AB	BA	AA	AC	AC	CB	AD	CB	AB	AA	BA
Compressed Input(after)	0000	00011	00000	0000	0001	0001	00001	00010	00001	0010	0000	00000

DCRAM in theory (for access and replace)

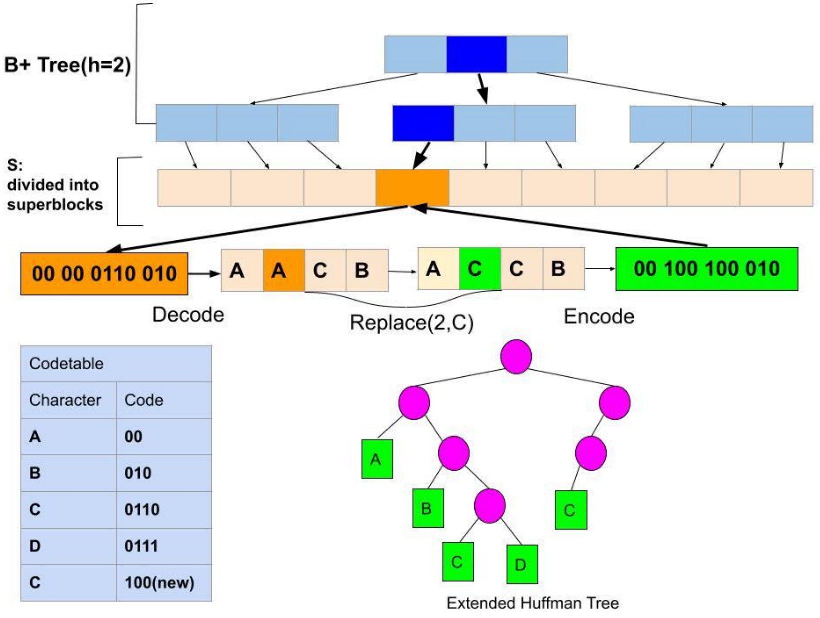
- After replace, if the class of c is changed (increased), we assign a new codeword according to the class of c.
- If there is no unused code for c, reconstruct the entire DS (this can be amortized in theory).
- In theory, $\Omega(n')$ replace operations can be performed before the reconstruction. Hence one can support replace in O(1) time.

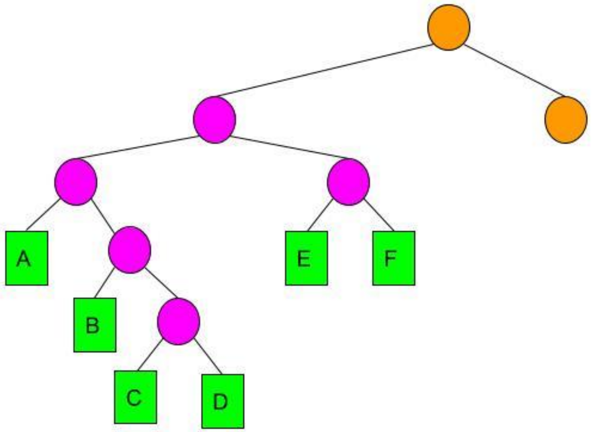
Practical Implementation of CRAM



- Each blocks of S is encoded using Huffman code. We do linear scan on the superblock.
- For managing superblocks, we use **B^+-tree with fixed height h** (i.e., root is not splitted. h is depending on the size of S) instead of darray.
- O(h log (r/ε) + r/ε) for time for both operations.
- For supporting sequential operations efficiently, we store the position of the superblock that recently used, and apply some SIMD operations.

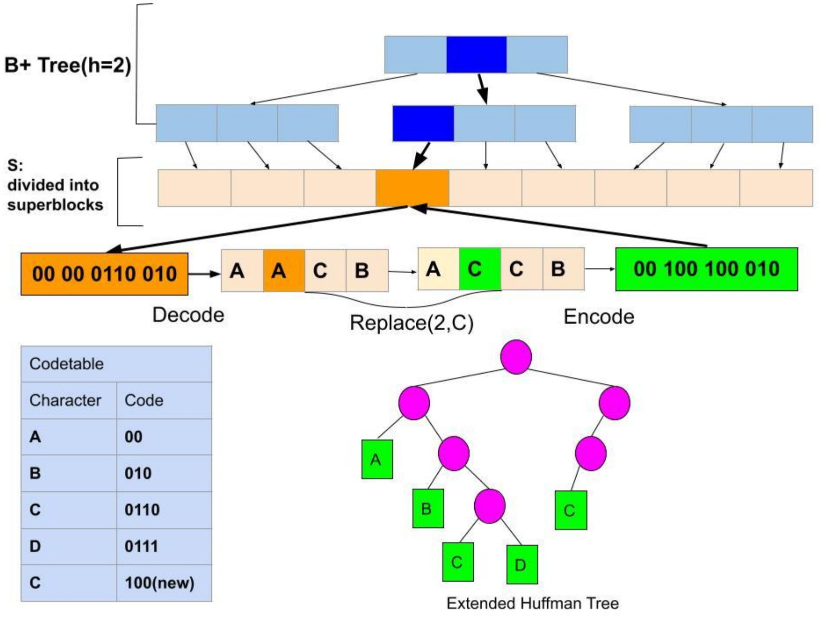
Practical Implementation of DCRAM

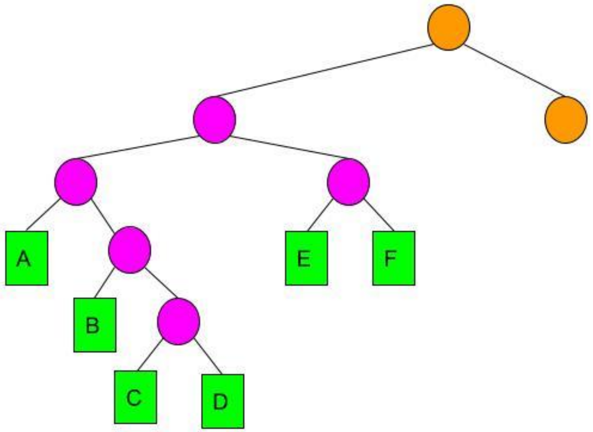




- We need some unused codes for DCAM.
- We make a space for unused codes using **Extended Huffman tree**. For implementation we consider (i) Type-1: codes starting with 1 is initially unused, and (ii) Type-2: codes starting with 11 is initially unused.
- For both Type-1 and -2 trees, at least Ω(n′) update operations are necessary before reconstructing the tree.

Practical Implementation of DCRAM





- In addition to Extended Huffman tree, we use **lazy update** on DCRAM, which combines the update algorithms of CRAM and DCRAM.
- In lazy update, we reconstruct the tree when (i) every superblock is re-encoded, and (ii) there is no unused code.

Experimental Results

Equipment specifications

AMD Ryzen 5 1600 Six-Core Processor (576KB L1, 3MB L2, and 16MB L3 cache) with 32GB RAM.

Datasets from Pizza&Chilli Corpus (http://pizzachili.dcc.uchile.cl/texts.html)

File Name	σ	H_0	H_1
DNA	16	0.247	0.245
ENGLISH	225	0.565	0.509
PROTEINS	25	0.525	0.524
XML	96	0.657	0.547

All files sizes are 200MB

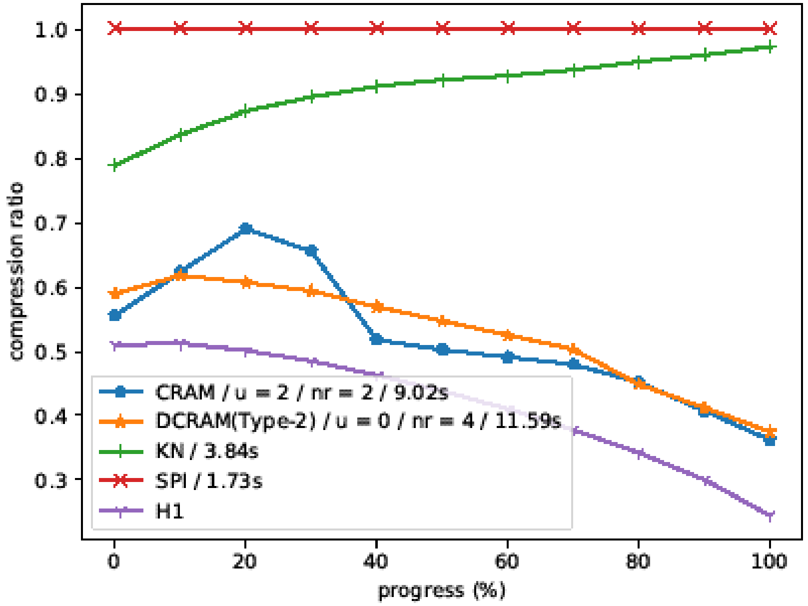
Experimental Results

Some parameters for experiments

- r (block size): 2
- $1/\varepsilon$ (#blocks in each superblock for CRAM): 512 initially
- h (height of fixed B^+-tree): 2, each node contains the information of ~200 consecutive superblocks.
- u (for CRAM and DCAM with lazy update): decides how many superblocks are additionally re-encoded for each updated.
- For CRAM, u = 1 means no reconstruction, and u (>1) reconstructions during n′ updates.

Experimental Results

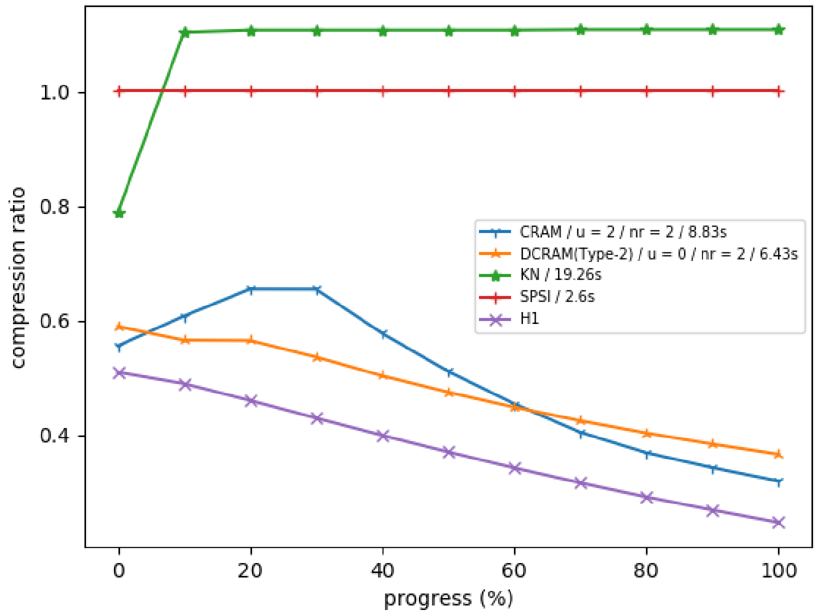
(i) Replace-seq: Overwrites from ENGLISH to DNA sequentially.



- SPSI is ~7 times faster than ours (does not compress the input).
- KN is 2~3 times faster than ours (highly optimized for sequential updates)
- Only our implementations reduce the space usage based on the input's entropy.

Experimental Results

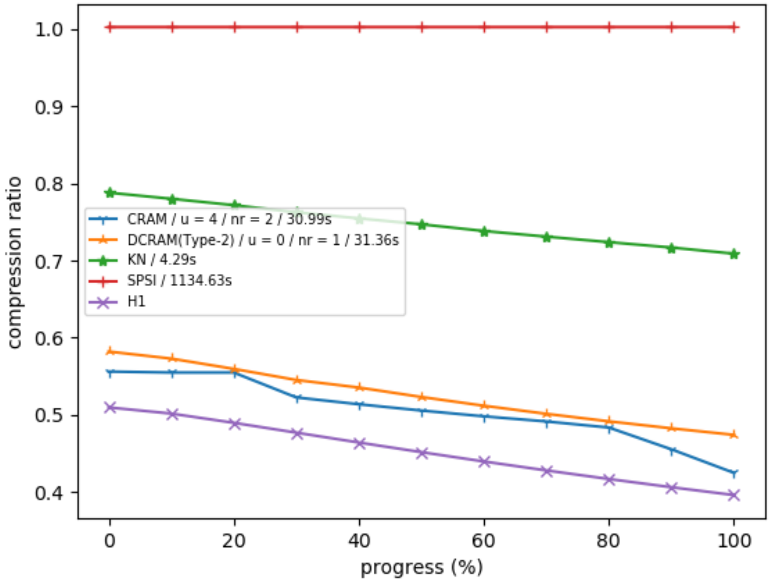
(ii) Replace-random2 (ENGLISH): Perform n′ε replace operations with the same character



- DCRAM with lazy update works faster than KN while using less space during the progress.

Experimental Results

(iii) Insert-seq (ENGLISH): Insert n/2 same characters consecutively.



- KN is ~7 times faster than ours (highly optimized for sequential updates) while our implementations take less space.
- SPSI is worse in both time and space.

Conclusion

- New implementation of compressed RAM, which changes the space adaptively for the input's entropy.
- Supporting decent operations times in both sequential and random tests.

Future work

- More optimization for sequential operations.
- Implementation based on the work of [Munro and Nekrich 15].

Thank You