Chair of Applied Computer Science III
UNIVERSITY OF MANNHEIM
Prof. Dr. Guido Moerkotte
Email: moerkotte@uni-mannheim.de
Daniel Flachs
Email: daniel.flachs@uni-mannheim.de

Database Systems II
Spring Semester 2019

Solution to Exercise Sheet 4
Created March 21, 2019

Exercise 1

Download the `zip` archive for this exercise sheet from the website.
Compile `experiment4.cc` with the output name set to `experiment4`, e.g., by compiling as `g++ experiment4.cc -o experiment4`.
Run the script `runExperiment4.sh` and analyze the output. `runExperiment4.sh` is a bash script.
If you run Windows, you are able to run this file in case your Windows version is 10 (or newer), cf. `https://www.howtogeek.com/261591/how-to-create-and-run-bash-shell-scripts-on-windows-10/`
The script `runExperiment4.sh` runs `experiment4` for different arguments. If you're having trouble to run the script, you can also perform the instructions of the script by hand.

Solution

The code illustrates the difference between predicated code and code with a branch, cf. Script, Section 2.2.3 "(Cost of) Branch (Mis-) Prediction", p. 21:

- The code with an `if`-statement (branching) runs slower if the selectivity of the predicate is closer to 0.5. This is due to the fact that branch prediction works worst in this case.
- The predicated code achieves the same result without a branch. It does an unconditional write in each iteration, which is more costly than doing writes only if the predicate is fulfilled. However, there is no penalty for branch misprediction since there is simply no branch. Therefore, the predicated code has constant runtime which is indifferent to the selectivity of the predicate.

Exercise 2

This exercise deals with compression.

Exercise 2 a)

In the `zip` file for this exercise sheet, you find a file named `country_or_area.csv`. This file contains the attribute values for the column *country_or_area* of the *International Financial Statistics* data set `https://www.kaggle.com/unitednations/international-financial-statistics`.
Implement a dictionary that allows you to compress the *country_or_area* attribute values given in `country_or_area.csv`.
You are free to choose on the implementation details. Your dictionary may be based on a hash table, a tree, or something simple (= inefficient), like an unsorted vector.

Solution

See code.

Exercise 2 b)

What is the compression rate and the percentage of space savings of your compression?
The compression ratio is computed as

$$compRatio = \frac{uncompressedSize}{compressedSize},$$

where *compressedSize* is the size of the data plus the size of the dictionary.
The space savings is computed as

$$spaceSavings = 1 - \frac{compressedSize}{uncompressedSize}.$$

Solution

Numbers for the provided solutions:

- $compRatio = 10.2689$
- $spaceSavings = 0.902\,618 = 90.2\,\%$

Raw output:

Size in bytes of uncompressed data:	36 462 713
Size in bytes of compressed data:	3 536 976
Size in bytes of dictionary:	13 825
Size of compressed data and dictionary:	3 550 801
Compression ratio:	10.2689
Percentage space savings:	0.902 618

Exercise 3

This exercise deals with the difference in memory access costs for cache-aligned and unaligned accesses. It simulates tuples of a database relation that are stored in row store format, cf. Script, p. 53 ff. (especially the figure on p. 56).

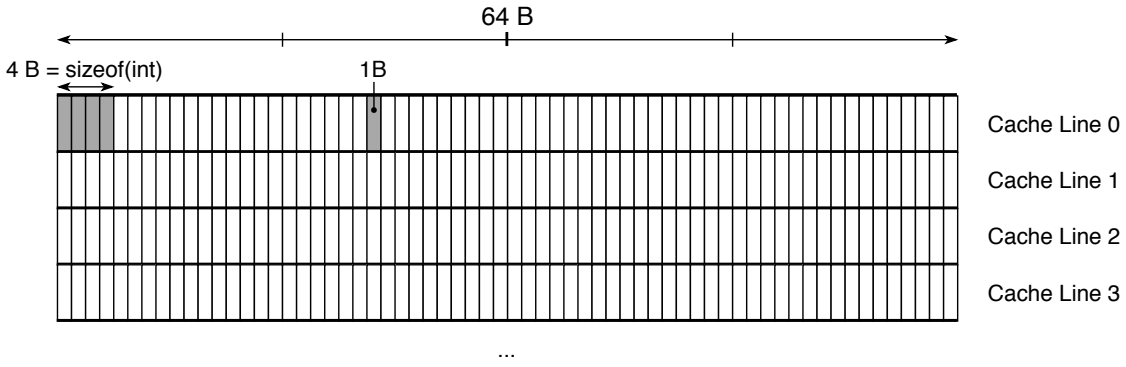
Consider the following piece of code. The `sum` function sums up `m` elements of integer array `B`, starting at `B[0]` in steps of `s` integers, i. e., `s = 3` would sum up `B[0]`, `B[3]`, `B[6]` etc. The main method uses the `sum` function to sum up the same array `A` with different access patters (step size and offset), denoted by `(0)`, `(1)` and `(2)` in the code.

```
1  int32_t sum (int32_t* B, const uint m, const int s) {
2    int32_t lSum = 0;
3    for(uint32_t i = 0; i < m; ++i) { // m: number of elements
4      lSum += *B;
5      B += s;       // s: step size
6    }
7    return lSum;
8  }
9
10
11 int main() {
12   const uint32_t n = 1000*1000*100;
13     // number of ints in array
14   const uint32_t m = (n - 16) / 16;
15     // 16*4 = 64 = sizeof(cacheline)
16
17   void* A = 0;
18   int lRc = posix_memalign(&A, 64, (n * sizeof(int32_t)));
19   if(lRc) {
20     printf("memalign failed.");
21     return 1;
22   }
23
24   int32_t* B = nullptr;
25
26   // *** (0) ***
27   B = (int32_t*) A;
28   for(uint32_t i = 0; i < n; ++i) { B[i] = 1; }
29   const int32_t lSum0 = sum(B, m, 1);
30
31   // *** (1) ***
32   B = (int32_t*) A;
33   for(uint32_t i = 0; i < n; ++i) { B[i] = 1; }
34   const int32_t lSum1 = sum(B, m, 16);
35
36   // *** (2) ***
37   B = (int32_t*) (A + 62);
```

```
38    for(uint32_t i = 0; i < n - 15; ++i) { B[i] = 1; }
39    const int32_t lSum2 = sum(B, m, 16);
40
41    printf("sum0/1/2: %d, %d, %d\n", lSum0, lSum1, lSum2);
42
43    assert((void*) B < A + n);
44    free(A);
45  }
```

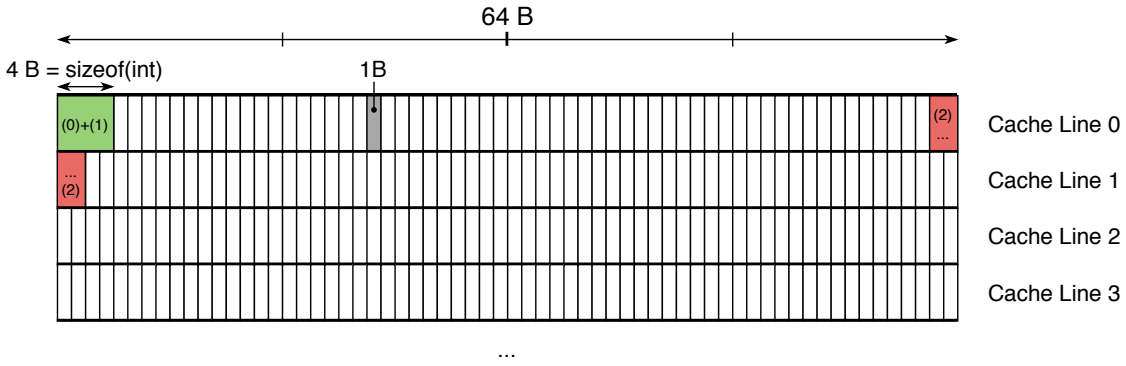
Exercise 3 a)

In the following figure, for each access pattern `(0)`, `(1)` and `(2)`, indicate the location of the first array element that is summed up when calling the `sum` function. Suppose that `int* A` points to the leftmost integer in cache line 0.



Solution

- (0) and (1) both start at the first element of `A`, i. e., `A[0]`. They only differ in the step size.
- (2) is shifted by 62 B, i. e., the first two bytes of the four-byte integer are still in cache line 0, the other two bytes are in the subsequent cache line.



Exercise 3 b)

Use your findings of the previous subtask to determine which of the access patterns is cache-aligned and which is not.

Solution

(0) and (1) are cache-aligned, (2) is not. Note that unaligned access is more expensive (or even impossible on some architectures) than aligned access.

Note also that an offset of 2 B would be unaligned as well with respect to the data type `int32_t`: `B = (int32_t*) (A + 2);` However, a step size of 16 would mean summing up the integers at bytes 2 to 5 of each cache line, so none of the accesses would span more than one cache line. Such accesses are cheaper than those crossing a cache line boundary (like (2)), so the effect might not be measurable in this case.

Exercise 4

Note: This exercise was discussed in class, but was originally not on the sheet. It was taken from Cormen et al., 3e, p. 277, Ex. 11.4-1.

Consider inserting the keys 10, 22, 31, 4, 15, 28, 17, 88, 59 into a hash table of length $m = 11$ using open addressing with the auxiliary hash function $h'(k) := k$. For inserting, use

a) linear probing,

b) quadratic probing with $c_1 := 1$, $c_2 := 3$,

c) double hashing with the two auxiliary hash functions $h_1(k) := k$ and $h_2(k) := (k \bmod (m-1)) + 1$.

Solution

The hash functions for the three probing variants linear probing (LP), quadratic probing (QP) and double hashing (DH) are

- $h_{\mathrm{LP}}(k, i) := (h'(k) + i) \bmod m$,
- $h_{\mathrm{QP}}(k, i) := (h'(k) + i + 3i^2) \bmod m$,
- $h_{\mathrm{DH}}(k, i) := (h_1(k) + i \cdot h_2(k)) \bmod m$.

The following two tables show the state of the three hash tables after the last insert and the probing sequences for each key insertion under each probing variant.

Hash Table

	LP	QP	DH
0	22	22	22
1	88		
2		88	59
3		17	17
4	4	4	4
5	15		15
6	28	28	28
7	17	59	88
8	59	15	
9	31	31	31
10	10	10	10

Probing Sequences

Key	LP	QP	DH
10	10	10	10
22	0	0	0
31	9	9	9
4	4	4	4
15	4, 5	4, 8	4, 10, 5
28	6	6	6
17	6, 7	6, 10, 9, 3	6, 3
88	0, 1	0, 4, 3, 8, 8, 3, 4, 0, 2	0, 9, 7
59	4, 5, 6, 7, 8	4, 8, 7	4, 3, 2