Database Systems II – Exercise #4

Sheet #4: Branch Misprediction, Compression, Cache Alignment

Daniel Flachs

Chair of Practical Computer Science III:
Database Management Systems

20/03/2019

UNIVERSITY OF MANNHEIM
School of Business Informatics and Mathematics

Created March 21, 2019

Contents

Contents

Task 1

```
size_t selectBranch(int* aInput, int* aOutput, size_t aSize, int aValue) {
  size_t j = 0;
  for (size_t i = 0; i < aSize; ++i) {
    if (aInput[i] <= aValue) {
      aOutput[j++] = aInput[i];
    }
  }
  return j;
}

size_t selectPredicated(int* aInput, int* aOutput, size_t aSize, int aValue) {
  size_t j = 0;
  for (size_t i = 0; i < aSize; ++i) {
    aOutput[j] = aInput[i];
    j += (aInput[i] <= aValue);
  }
  return j;
}
```

Task 1

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:

1. 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.
2. 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.

Task 1

- Which alternative is better?

Task 1

- Which alternative is better? This is determined by

Task 1

- Which alternative is better? This is determined by
  1. selectivity of the predicate (i. e., the fraction of cases in which the condition becomes true),

Task 1

- Which alternative is better? This is determined by
  1. selectivity of the predicate (i. e., the fraction of cases in which the condition becomes true),
  2. cost for executing the conditional code (in the example, this was aOutput[j] = aInput[i];)

Task 1

- Which alternative is better? This is determined by
  1. selectivity of the predicate (i. e., the fraction of cases in which the condition becomes true),
  2. cost for executing the conditional code (in the example, this was $\mathrm{aOutput[j] = aInput[i];}$)
- Rule of thumb:

Task 1

- Which alternative is better? This is determined by
  1. selectivity of the predicate (i. e., the fraction of cases in which the condition becomes true),
  2. cost for executing the conditional code (in the example, this was aOutput[j] = aInput[i];)
- Rule of thumb:
  - The more selective the predicate/condition, the better the performance of branch prediction and the lower the total penalty of branch misprediction. Branching might therefore be better in this case.

Task 1

- Which alternative is better? This is determined by
  1. selectivity of the predicate (i. e., the fraction of cases in which the condition becomes true),
  2. cost for executing the conditional code (in the example, this was aOutput[j] = aInput[i];)
- Rule of thumb:
  - The more selective the predicate/condition, the better the performance of branch prediction and the lower the total penalty of branch misprediction. Branching might therefore be better in this case.
  - If the cost for executing the conditional code is high, branching might be better suited than predicated code.

Task 1

- Which alternative is better? This is determined by
  1. selectivity of the predicate (i. e., the fraction of cases in which the condition becomes true),
  2. cost for executing the conditional code (in the example, this was aOutput[j] = aInput[i];)
- Rule of thumb:
  - The more selective the predicate/condition, the better the performance of branch prediction and the lower the total penalty of branch misprediction. Branching might therefore be better in this case.
  - If the cost for executing the conditional code is high, branching might be better suited than predicated code.
- Note that transforming branching code into predicated code might not always be possible, especially for more sophisticated conditions and statements.

Task 1

- Which alternative is better? This is determined by
  1. selectivity of the predicate (i. e., the fraction of cases in which the condition becomes true),
  2. cost for executing the conditional code (in the example, this was `aOutput[j] = aInput[i];`)
- Rule of thumb:
  - The more selective the predicate/condition, the better the performance of branch prediction and the lower the total penalty of branch misprediction. Branching might therefore be better in this case.
  - If the cost for executing the conditional code is high, branching might be better suited than predicated code.
- Note that transforming branching code into predicated code might not always be possible, especially for more sophisticated conditions and statements.
- **Exercise**: Write a function that sums up all element in an `int` array that are greater than a certain value. Avoid branching.

Task 2

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.

Hash Table Implementations

[1] See Cormen et al., 3e, p. 269ff.

Hash Table Implementations

1. Hashing by Chaining

[1] See Cormen et al., 3e, p. 269ff.

Hash Table Implementations

1. Hashing by Chaining
2. Open Addressing[1]

[1] See Cormen et al., 3e, p. 269ff.

Hash Table Implementations

1. Hashing by Chaining
2. Open Addressing[1] $\leftarrow$ used by the solution

[1] See Cormen et al., 3e, p. 269ff.

Hash Table Implementations

1. Hashing by Chaining
2. Open Addressing[1] ← used by the solution
   - Insert a key k at some free slot in case slot $h(k)$ is already occupied (i. e., there is a collision). ⇒ No collision chain is built.

[1] See Cormen et al., 3e, p. 269ff.

Hash Table Implementations

1. Hashing by Chaining
2. Open Addressing[1] ← used by the solution
   - Insert a key k at some free slot in case slot $h(k)$ is already occupied (i. e., there is a collision). ⇒ No collision chain is built.
   - Consequence: Can insert at most as many keys as there are slots in the table, i. e., $n \leq m$. ⇒ Load factor α can never exceed 1.

[1] See Cormen et al., 3e, p. 269ff.

Hash Table Implementations

1. Hashing by Chaining
2. Open Addressing[1] ← used by the solution
   - Insert a key k at some free slot in case slot $h(k)$ is already occupied (i. e., there is a collision). ⇒ No collision chain is built.
   - Consequence: Can insert at most as many keys as there are slots in the table, i. e., $n \leq m$. ⇒ Load factor α can never exceed 1.
   - Probing sequence determines which slots are probed when inserting and searching. For this, extend the hash function h by a probe number i.

[1] See Cormen et al., 3e, p. 269ff.

Hash Table Implementations

1. Hashing by Chaining
2. Open Addressing[1] ← used by the solution
   - Insert a key k at some free slot in case slot $h(k)$ is already occupied (i. e., there is a collision). $\Rightarrow$ No collision chain is built.
   - Consequence: Can insert at most as many keys as there are slots in the table, i. e., $n \leq m$. $\Rightarrow$ Load factor α can never exceed 1.
   - Probing sequence determines which slots are probed when inserting and searching. For this, extend the hash function h by a probe number i.
     1. Linear Probing: $h(k, i) := (h'(k) + i) \bmod m$

[1] See Cormen et al., 3e, p. 269ff.

Hash Table Implementations

1. Hashing by Chaining
2. Open Addressing[1] ← used by the solution
   - Insert a key k at some free slot in case slot $h(k)$ is already occupied (i. e., there is a collision). ⇒ No collision chain is built.
   - Consequence: Can insert at most as many keys as there are slots in the table, i. e., $n \leq m$. ⇒ Load factor α can never exceed 1.
   - Probing sequence determines which slots are probed when inserting and searching. For this, extend the hash function h by a probe number i.
     1. Linear Probing: $h(k, i) := (h'(k) + i) \bmod m$
     2. Quadratic Probing: $h(k, i) := (h'(k) + c_1 i + c_2 i^2) \bmod m \quad (c_1, c_2 \geq 0)$

[1] See Cormen et al., 3e, p. 269ff.

Hash Table Implementations

1. Hashing by Chaining
2. Open Addressing[1] ← used by the solution
   - Insert a key k at some free slot in case slot $h(k)$ is already occupied (i. e., there is a collision). ⇒ No collision chain is built.
   - Consequence: Can insert at most as many keys as there are slots in the table, i. e., $n \leq m$. ⇒ Load factor α can never exceed 1.
   - Probing sequence determines which slots are probed when inserting and searching. For this, extend the hash function h by a probe number i.
     1. Linear Probing: $h(k, i) := (h'(k) + i) \bmod m$
     2. Quadratic Probing: $h(k, i) := (h'(k) + c_1 i + c_2 i^2) \bmod m \quad (c_1, c_2 \geq 0)$
     3. Double Hashing: $h(k, i) := (h_1(k) + i \cdot h_2(k)) \bmod m$

[1] See Cormen et al., 3e, p. 269ff.

Hash Table Implementations

1. Hashing by Chaining
2. Open Addressing[1] ← used by the solution
   - Insert a key k at some free slot in case slot $h(k)$ is already occupied (i. e., there is a collision). $\Rightarrow$ No collision chain is built.
   - Consequence: Can insert at most as many keys as there are slots in the table, i. e., $n \leq m$. $\Rightarrow$ Load factor α can never exceed 1.
   - Probing sequence determines which slots are probed when inserting and searching. For this, extend the hash function h by a probe number i.
     1. Linear Probing: $h(k,i) := (h'(k) + i) \bmod m$
     2. Quadratic Probing: $h(k,i) := (h'(k) + c_1 i + c_2 i^2) \bmod m \quad (c_1, c_2 \geq 0)$
     3. Double Hashing: $h(k,i) := (h_1(k) + i \cdot h_2(k)) \bmod m$
   - Linear and quadratic probing only generate m instead of the $m!$ possible distinct probing sequences. They suffer from a problem called clustering (collisions in one part of the HT lead to even more collisions).

[1] See Cormen et al., 3e, p. 269ff.

Hash Table Implementations

1. Hashing by Chaining
2. Open Addressing[1] ← used by the solution
   - Insert a key k at some free slot in case slot $h(k)$ is already occupied (i. e., there is a collision). ⇒ No collision chain is built.
   - Consequence: Can insert at most as many keys as there are slots in the table, i. e., $n \leq m$. ⇒ Load factor α can never exceed 1.
   - Probing sequence determines which slots are probed when inserting and searching. For this, extend the hash function h by a probe number i.
     1. Linear Probing: $h(k, i) := (h'(k) + i) \bmod m$
     2. Quadratic Probing: $h(k, i) := (h'(k) + c_1 i + c_2 i^2) \bmod m \quad (c_1, c_2 \geq 0)$
     3. Double Hashing: $h(k, i) := (h_1(k) + i \cdot h_2(k)) \bmod m$
   - Linear and quadratic probing only generate m instead of the $m!$ possible distinct probing sequences. They suffer from a problem called clustering (collisions in one part of the HT lead to even more collisions).
   - Double hashing can generate up to m^2 distinct probing sequences for well-chosen h_1, h_2, m.

[1] See Cormen et al., 3e, p. 269ff.

Exercise

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 $h'(k) := k$.

Exercise

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 $h_2(k) := (k \bmod (m-1)) + 1$.

Taken from Cormen et al., 3e, p. 277, Ex. 11.4-1.

Exercise

- Keys to insert: 10, 22, 31, 4, 15, 28, 17, 88, 59
- Linear and quadratic probing
  - Auxiliary hash function: $h'(k) := k$
  - $h_{LP}(k, i) := (h'(k) + i) \bmod m$
  - $h_{QP}(k, i) := (h'(k) + i + 3i^2) \bmod m$
  - Pre-computation of $i + 3i^2$

i	0	1	2	3	4	5	6	7	8
$i + 3i^2$	0	4	$14 \equiv 3$	$30 \equiv 8$	$52 \equiv 8$	$80 \equiv 3$	$114 \equiv 4$	$154 \equiv 0$	$200 \equiv 2$

- Double Hashing
  - Auxiliary hash functions:
    $h_1(k) := k$
    $h_2(k) := (k \bmod (m - 1)) + 1$
  - $h_{DH}(k, i) := (h_1(k) + i \cdot h_2(k)) \bmod m$

Solutions: see exercise sheet solution

Task 2

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

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

Task 2

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

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

Numbers for the provided solutions:

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

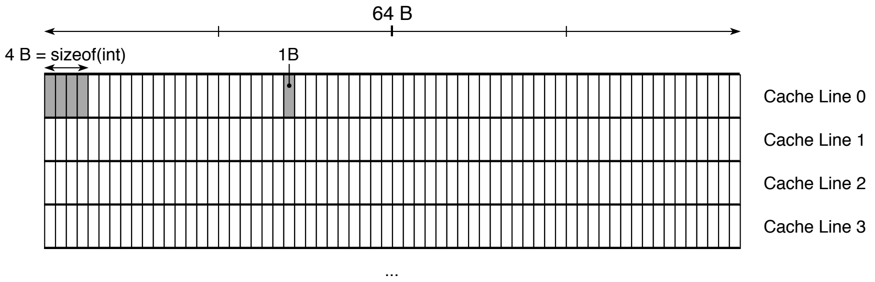
Task 3

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).

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 function 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.

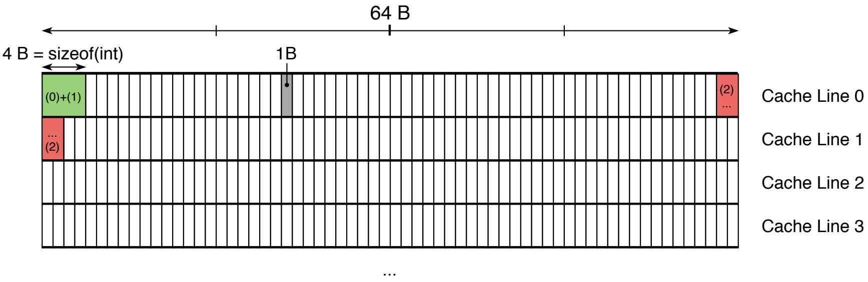
Task 3

In the following figure, 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.

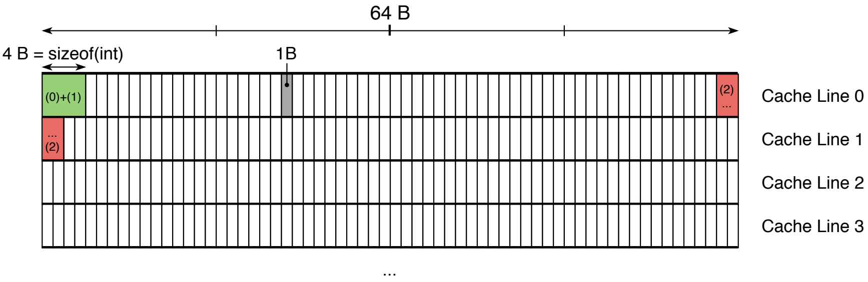


Which of the access patterns is cache-aligned, which is not?

Task 3


64 B
4 B = sizeof(int)
1B
(0)+(1)
(2)
...
...
(2)
Cache Line 0
Cache Line 1
Cache Line 2
Cache Line 3
...

Task 3



- (0) and (1) are cache-aligned,
- (2) is not.