Database Systems II – Exercise #3

Sheet #3: Branch Prediction + Hashing and Hash Tables

Daniel Flachs

Chair of Practical Computer Science III:
Database Management Systems

13/03/2019

UNIVERSITY
OF MANNHEIM
School of Business Informatics
and Mathematics

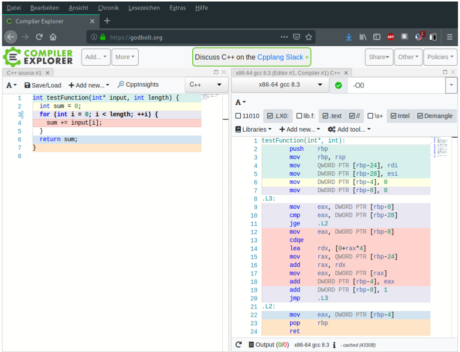
Contents

Contents

godbolt

Compiler Explorer, https://godbolt.org/

- Lets you enter and compile source code to assembler.
- You can choose
  - architecture (Intel, ARM, ...),
  - compiler (gcc, clang, ...),
  - and compiler flags.
- Source and assembler code are shown side by side.



Intel x86 architecture see https://en.wikibooks.org/wiki/X86_Assembly/X86_Architecture.

Contents

Task 1 a+b

In general, the code implements an if-statement that depends on a conjunction of two selection predicates p_1 and p_2, i. e.

if $p_1 \wedge p_2$
<code 1>
else <code 2>

On what factors does the efficiency of the above code depend, assuming that <code 1> and <code 2> take approximately the same time?

- costs of evaluating p_1 and/or p_2
- implementation of the $\wedge$ operator (& vs. &&) and probability that p_1 is true

Recap: Branch Prediction

- CPU loads subsequent instructions in advance and starts executing them (pipelining and speculative execution).
- If a branch (`if`-statement) is encountered, the CPU tries to guess the outcome of the condition (branch prediction) and loads the instructions of the path that is more likely to be executed.
- Branch misprediction: pipeline hazard leading to instruction stall.
- Branch prediction is harder if both branches are equally likely. The probability for a misprediction increases.

Recap: & vs. && in C++

- In C++, there are two variants for a logical AND operator ($\wedge$) that connects two predicates p_1 and p_2.
- &&
  - Short-circuit evaluation, i. e., the second operand is not evaluated if the first yields `false`.
  - Beneficial if the second operand is expensive to evaluate.
  - Introduces a new "invisible branch" that evaluates p_2 only if p_1 is true. $\Rightarrow$ Danger of branch misprediction.
- &
  - Both operands are always evaluated.
  - Does not introduce an additional branch.
  - Unnecessary cost for evaluating the second operand even if the first yields `false`.
- Depending on the use case, one variant might be superior to the other.

Task 1b

```
bool singleAmp(int a, int b) {
  if ((a == 3) & (b == 5)) {
    return true;
  }
  return false;
}

bool doubleAmp(int a, int b) {
  if ((a == 3) && (b == 5))
     {
    return true;
  }
  return false;
}
```

- How does the assembler code differ?
- Optimization levels: -O0 vs. -O3
- Side effects
  - If the predicates do not have side effects, & and && are identical w. r. t. correctness.
  - In that case, the compiler may optimize by just evaluating both predicates (if they are cheap) and disregarding short-circuit evaluation.
  - If a predicate has side effects, short circuit evaluation is a must as the programmer might rely on that behavior!

Task 1b

Code with side effects

What is the expected behavior of the following code snippet?

```
#include <iostream>

bool __attribute__ ((noinline))      // Disallows inlining
secondCondition (int b) {
  puts("first condition was true");
  return (b==5);
}

bool f(int a, int b) {
  if ((a == 3) && secondCondition(b)) {
    return true;
  }
  return false;
}
```

Task 1c

- Both functions sum up the elements in an integer array.
- `sum0` uses a single summation variable, iterates all elements using a for-loop, and always adds the current element: `lSum += aArray[i];`
- `sum1` uses a for-loop iterating the indices of the lower half of the array, and two summation variables, each of the summing up the elements in one half of the array:
  `lSum1 += aArray[i];`
  `lSum2 += aArray[i+lHalf];`
- Observation: `sum1` runs faster than `sum0`.
- Reason: μ-ops parallelism
- How about cache misses?

Task 2a

What does the term *universal hashing* mean?

- Intuition: Universal hashing captures the desired property that distinct keys do not collide too often. The idea is to choose a hash function from a set of hash functions *randomly* and *independent* of the values that are stored in it.
- Formal: Let H be a family of hash functions that map values in U to values in $M := \{0, 1, \ldots, m-1\}$. H is said to be *universal* iff for each pair of distinct keys $k, l \in U$, the number of hash functions $h \in H$ for which $h(k) = h(l)$ is at most
$$\frac{|H|}{|M|}.$$
- Equivalently, we can say that, for a hash function h, randomly chosen from the family of hash function H, we have that
$$\Pr[h(k) = h(l)] \leq \frac{1}{|M|},$$
i.e., the chance of a collision between k and l is no more than $\frac{1}{|M|}$. This equals the probability that $h(k) = h(l)$ for randomly chosen values of $h(k)$ and $h(l)$ from the interval $\{0, 1, \ldots, m-1\}$.

Task 2b

Consider the following four hash functions h_1, h_2, h_3, h_4 that map values from the universe $U := \{0, 1, 2, 3, 4, 5\}$ to the set $D := \{0, 1\}$.

$x \in U$	0	1	2	3	4	5
$h_1(x) \in \{0, 1\}$	0	1	0	1	0	1
$h_2(x) \in \{0, 1\}$	0	0	0	1	1	1
$h_3(x) \in \{0, 1\}$	0	0	1	0	1	1
$h_4(x) \in \{0, 1\}$	1	0	0	1	1	0

i. Let $H := \{h_1, h_2\}$. Is H universal?

ii. Let $H' := \{h_1, h_2, h_3\}$. Is H' universal?

iii. Let $H'' := \{h_1, h_2, h_3, h_4\}$. Is H'' universal?

Task 2b

$x \in U$	0	1	2	3	4	5
$h_1(x) \in \{0,1\}$	0	1	0	1	0	1
$h_2(x) \in \{0,1\}$	0	0	0	1	1	1
$h_3(x) \in \{0,1\}$	0	0	1	0	1	1
$h_4(x) \in \{0,1\}$	1	0	0	1	1	0

i. $H := \{h_1, h_2\}$ is not universal. The relevant bound is $\frac{|H|}{|D|} = \frac{2}{2} = 1$.
   - $\delta_H(1,4) = 0 \leq 1$ ✓, and $\delta_H(1,3) = 1 \leq 1$ ✓, but
   - $\delta_H(0,2) = 2 \not\leq 1$ ✗, and $\delta_H(3,5) = 2 \not\leq 1$ ✗.

ii. $H' := \{h_1, h_2, h_3\}$ is not universal. The relevant bound is $\frac{|H'|}{|D|} = \frac{3}{2} = 1.5$.
   - $\delta_H(1,4) = 0 \leq 1.5$ ✓, but
   - $\delta_H(0,2) = 2 \not\leq 1.5$ ✗, $\delta_H(1,3) = 2 \not\leq 1.5$ ✗, and $\delta_H(4,5) = 2 \not\leq 1.5$ ✗.

iii. $H'' := \{h_1, h_2, h_3, h_4\}$ is universal. The relevant bound is $\frac{|H''|}{|D|} = \frac{4}{2} = 2$.
   - $\delta_H(1,4) = 0 \leq 2$ ✓, $\delta_H(0,2) = 2 \leq 2$ ✓, $\delta_H(1,3) = 2 \leq 2$ ✓, $\delta_H(4,5) = 2 \leq 2$ ✓, ...

Task 3

Implement a hash table that resolves collisions using chaining. Your hash table must be generic: Make the hash function a parameter. Find some data and insert it into your hash table under different hash functions. Output the average length and the maximum length of the buckets/chains in your hash table. What do you observe?