C++20's [[likely]] Attribute

Optimizations, Pessimizations, and [[unlikely]] Consequences

AMIR KIRSH & TOMER VROMEN

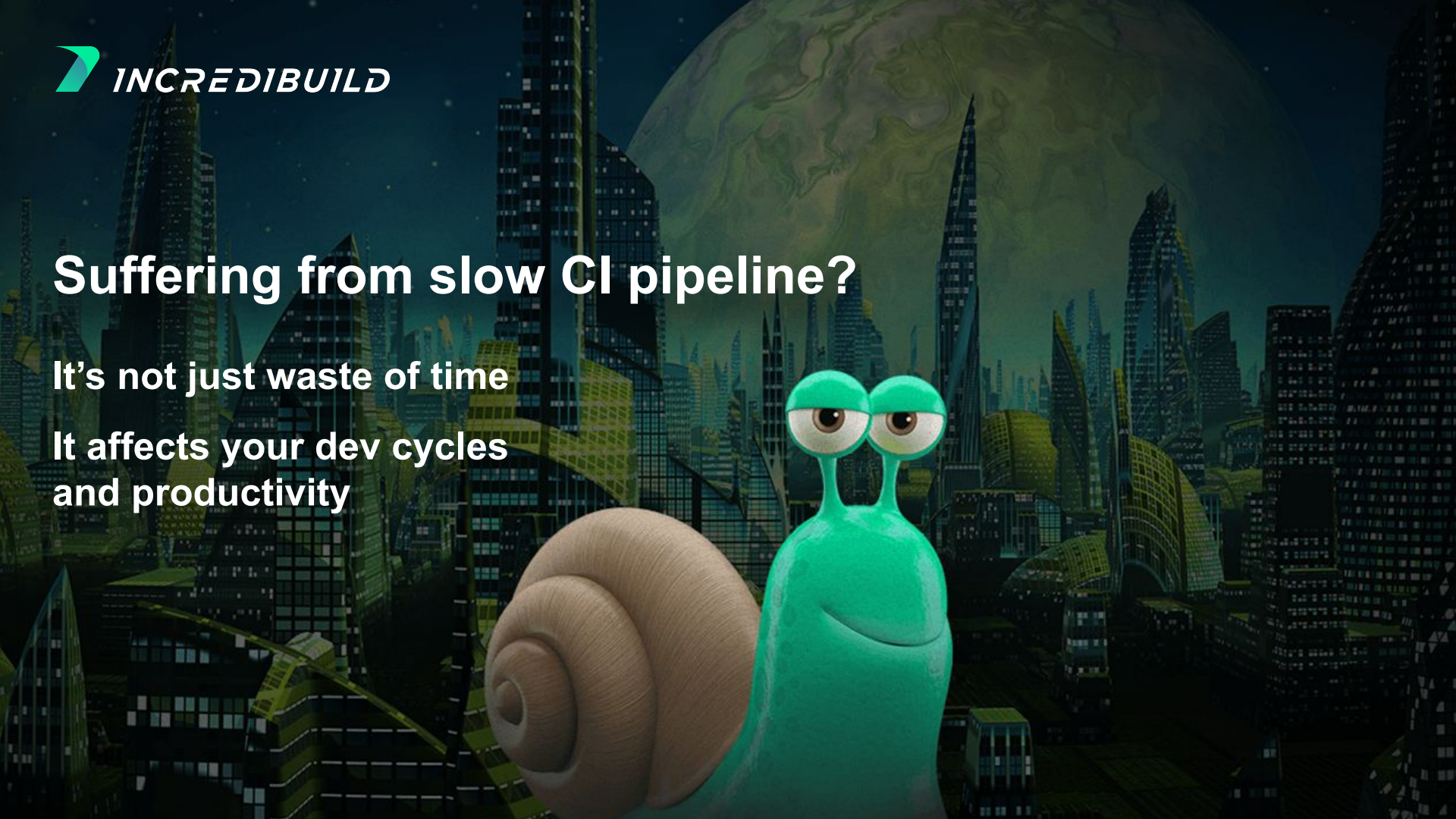
About me

Lecturer
Academic College of Tel-Aviv-Yaffo
and Tel-Aviv University
Visiting researcher at SBU, NY

Developer Advocate at

INCREDIBUILD

Co-Organizer of the **CoreCpp** conference and meetup group


INCREDIBUILD
Suffering from slow CI pipeline?
It’s not just waste of time
It affects your dev cycles
and productivity

About me

Tomer Vromen - תומר פרומן

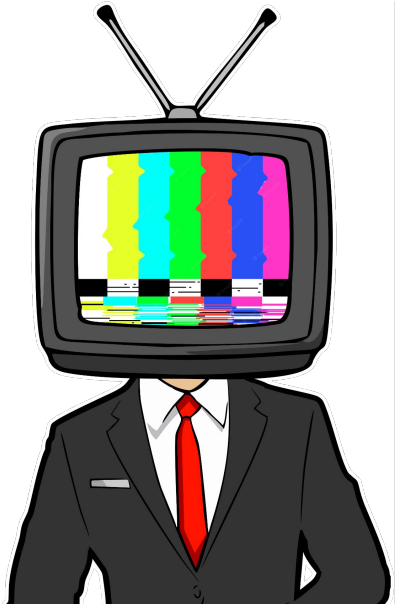
My first time here in CppCon

My 5mo son might be in the audience - have to start early!

github.com/tomerv

↳ Where you can also find code for this talk

Software developer @ Dell PowerFlex



`[[likely]]` Optimizations, `[[unlikely]]` Consequences

Death Valley National Park, United States

Photo by Rosie Steggles on Unsplash

Two new attributes in C++20

```
if (n > 5) [[unlikely]] {
    g(0);
    return n * 2 + 1;
}
```

```
switch (n) {
case 1:
    g(1);
    [[fallthrough]];

[[likely]] case 2:
    g(2);
    break;
}
```

Photo by Abhinav Sharma on Unsplash

https://www.eel.is/c++draft/dcl.attr.likelihood

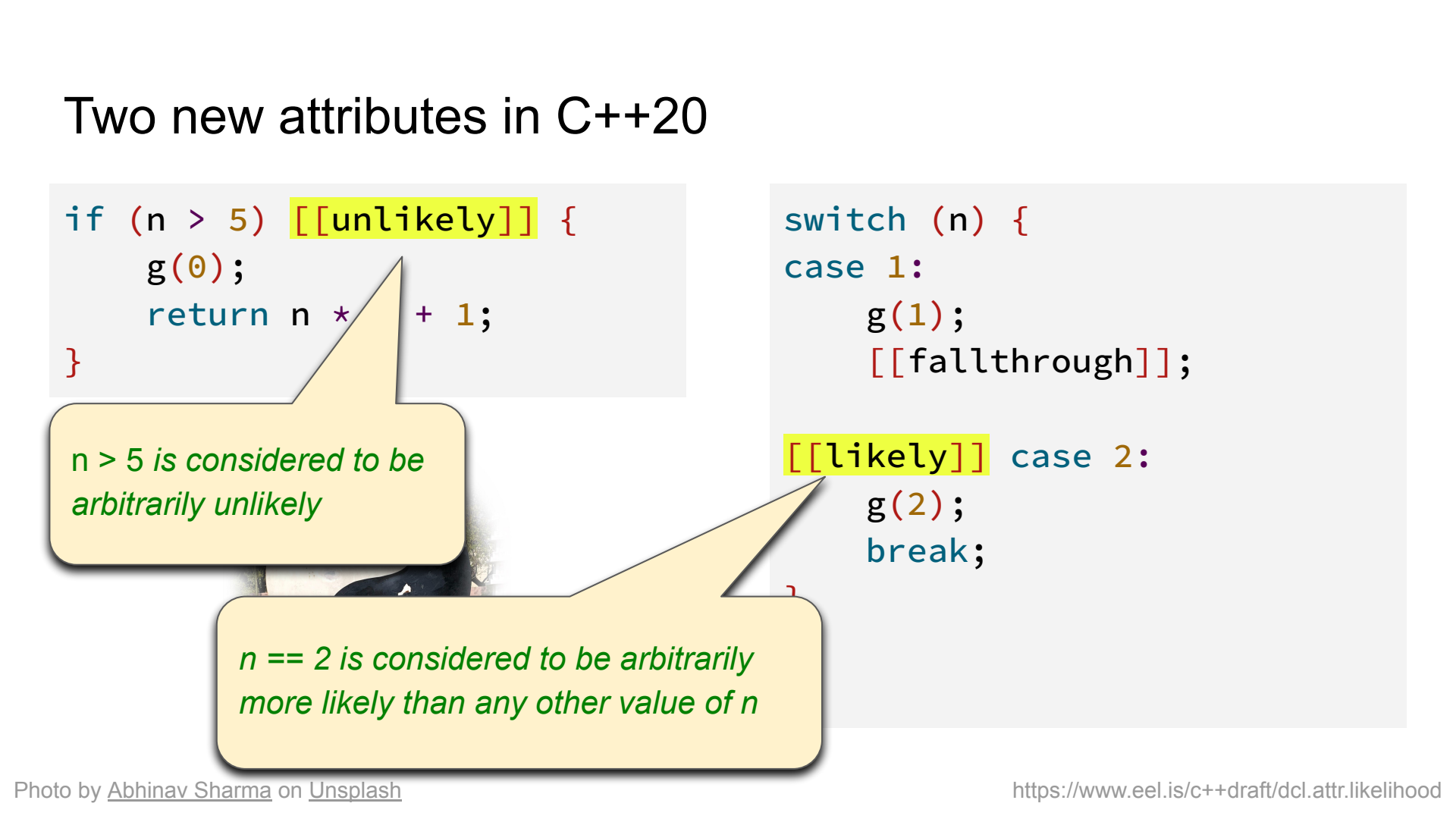
Two new attributes in C++20

```
if (n > 5) [[unlikely]] {
    g(0);
    return n *    + 1;
}
```

n > 5 *is considered to be arbitrarily unlikely*

```
switch (n) {
case 1:
    g(1);
    [[fallthrough]];

[[likely]] case 2:
    g(2);
    break;
}
```

n == 2 is considered to be arbitrarily more likely than any other value of n



https://www.eel.is/c++draft/dcl.attr.likelihood

Back to History (“It was there before”)

- Old intrinsic attributes - e.g. for gcc:
  - `__builtin_expect(expr, val)`
- Usually used as a macro

Stack Overflow: How do the likely/unlikely macros in the Linux kernel work and what is their benefit?

```
#define likely(x)    __builtin_expect((x),1)
#define unlikely(x)  __builtin_expect((x),0)
```

- Proposed in **Attributes for Likely and Unlikely Statements** (P0479)

9 Declarations [dcl.dcl]

9.12 Attributes [dcl.attr]

9.12.7 Likelihood attributes [dcl.attr.likelihood]

The *attribute-tokens* `likely` and `unlikely` may be applied to labels or statements. No *attribute-argument-clause* shall be present. The *attribute-token* `likely` shall not appear in an *attribute-specifier-seq* that contains the *attribute-token* `unlikely`.

Recommended practice: The use of the `likely` attribute is intended to allow implementations to optimize for the case where paths of execution including it are arbitrarily more likely than any alternative path of execution that does not include such an attribute on a statement or label. The use of the `unlikely` attribute is intended to allow implementations to optimize for the case where paths of execution including it are arbitrarily more unlikely than any alternative path of execution that does not include such an attribute on a statement or label. A path of execution includes a label if and only if it contains a jump to that label.

[*Note 1*: Excessive usage of either of these attributes is liable to result in performance degradation. — *end note*]

9 Declarations [dcl.dcl]

9.12 Attributes [dcl.attr]

9.12.7 Likelihood attributes [dcl.attr.likelihood]

The *attribute-tokens* `likely` and `unlikely` may be applied to labels or statements. No *attribute-argument-clause* shall be present. The *attribute-token* `likely` shall not appear in an *attribute-specifier-seq* that contains the *attribute-token* `unlikely`.

Recommended practice: The use of the `likely` attribute is intended to allow implementations to optimize for the case where paths of execution including it are arbitrarily more likely than any alternative path of execution that does not include such an attribute on a statement or label. The use of the `unlikely` attribute is intended to allow implementations to optimize for the case where paths of execution including it are arbitrarily more unlikely than any alternative path of execution that does not include such an attribute on a statement or label. A path of execution includes a label if and only if it contains a jump to that label.

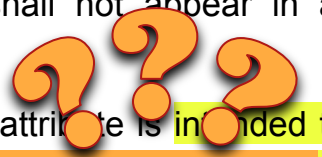
[*Note 1*: Excessive usage of either of these attributes is liable to result in performance degradation. — *end note*]

9 Declarations [dcl.dcl]

9.12 Attributes [dcl.attr]

9.12.7 Likelihood attributes [dcl.attr.likelihood]

The *attribute-tokens* `likely` and `unlikely` may be applied to labels or statements. No *attribute-argument-clause* shall be present. The *attribute-token* `likely` shall not appear in an *attribute-specifier-seq* that contains the *attribute-token* `unlikely`.



Recommended practice: The use of the `likely` attribute is intended to allow implementations to optimize for the case where paths of execution including it are arbitrarily more likely than any alternative path of execution that does not include such an attribute on a statement or label. The use of the `unlikely` attribute is intended to allow implementations to optimize for the case where paths of execution including it are arbitrarily more unlikely than any alternative path of execution that does not include such an attribute on a statement or label. A path of execution includes a label if and only if it contains a jump to that label.

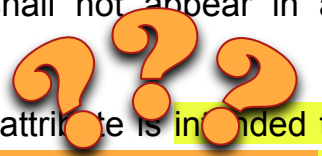
[*Note 1*: Excessive usage of either of these attributes is liable to result in performance degradation. — *end note*]

9 Declarations [dcl.dcl]

9.12 Attributes [dcl.attr]

9.12.7 Likelihood attributes [dcl.attr.likelihood]

The *attribute-tokens* `likely` and `unlikely` may be applied to labels or statements. No *attribute-argument-clause* shall be present. The *attribute-token* `likely` shall not appear in an *attribute-specifier-seq* that contains the *attribute-token* `unlikely`.



Recommended practice: The use of the `likely` attribute is intended to allow implementations to optimize for the case where paths of execution including it are arbitrarily more likely than any alternative path of execution that does not include such an attribute on a statement or label. The use of the `unlikely` attribute is intended to allow implementations to optimize for the case where paths of execution including it are arbitrarily more unlikely than any alternative path of execution that does not include such an attribute on a statement or label. A path of execution includes a label if and only if it contains a jump to that label.

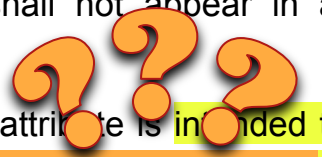
[*Note 1*: Excessive usage of either of these attributes is liable to result in performance degradation. — *end note*]

9 Declarations [dcl.dcl]

9.12 Attributes [dcl.attr]

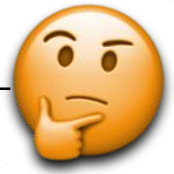
9.12.7 Likelihood attributes [dcl.attr.likelihood]

The *attribute-tokens* `likely` and `unlikely` may be applied to labels or statements. No *attribute-argument-clause* shall be present. The *attribute-token* `likely` shall not appear in an *attribute-specifier-seq* that contains the *attribute-token* `unlikely`.



Recommended practice: The use of the `likely` attribute is intended to allow implementations to optimize for the case where paths of execution including it are arbitrarily more likely than any alternative path of execution that does not include such an attribute on a statement or label. The use of the `unlikely` attribute is intended to allow implementations to optimize for the case where paths of execution including it are arbitrarily more unlikely than any alternative path of execution that does not include such an attribute on a statement or label. A path of execution includes a label if and only if it contains a jump to that label.

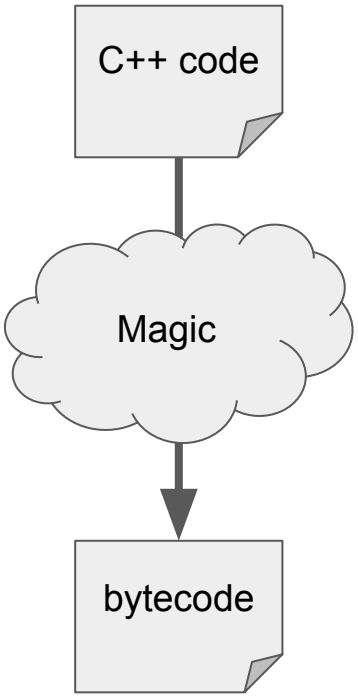
[*Note 1*: Excessive usage of either of these attributes is liable to result in performance degradation. —



https://www.eel.is/c++draft/dcl.attr.likelihood

Why?

What can the compiler (optimizer) do with this information?

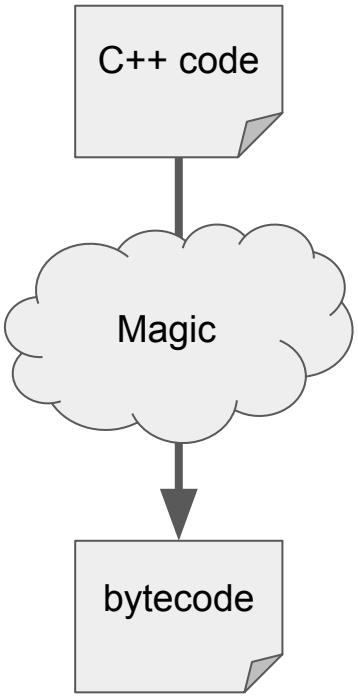


Why?

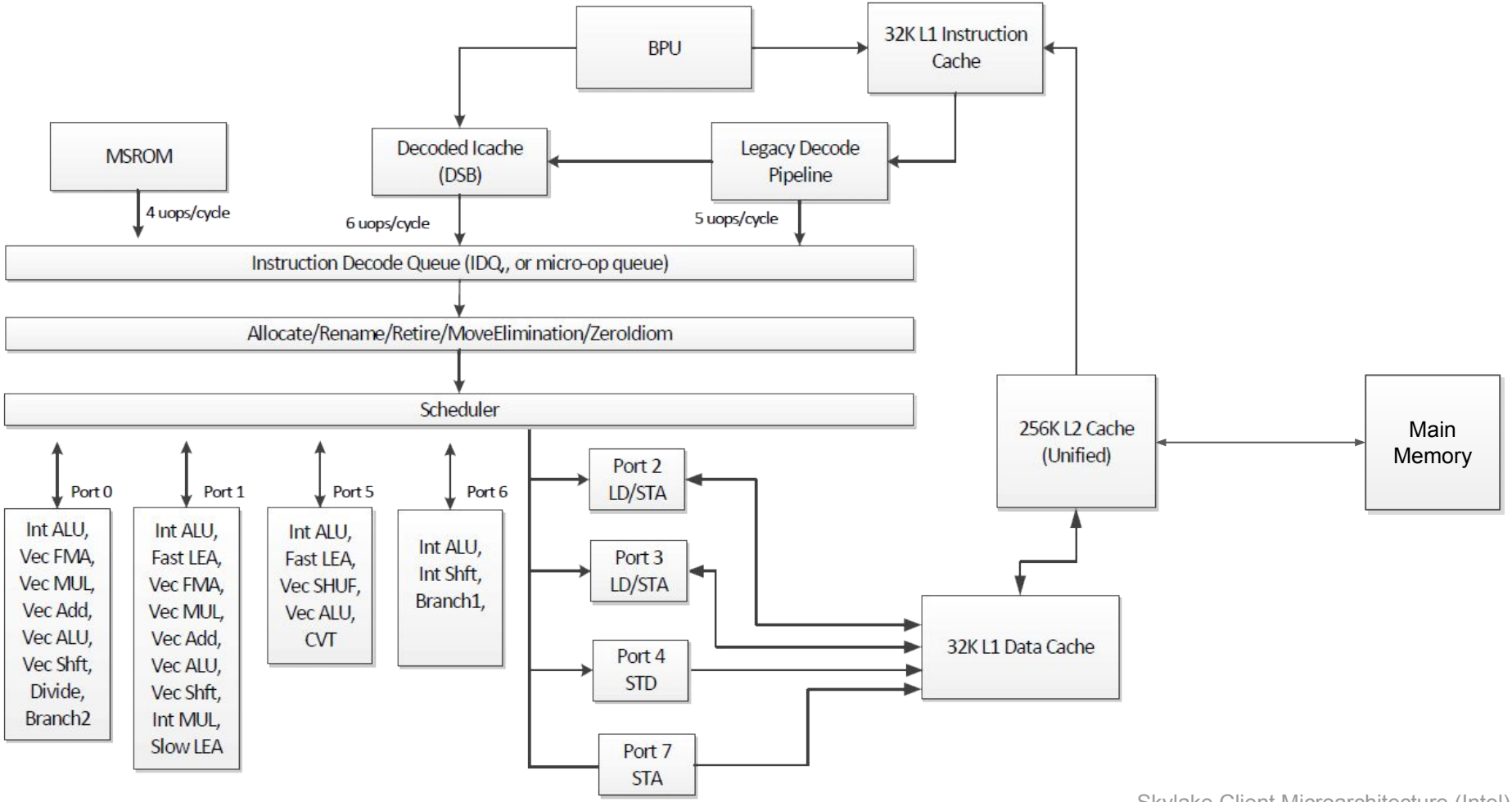
What can the compiler (optimizer) do with this information?

1. Improve code layout?
    - Instruction Cache (I-Cache)
2. Improve branch instructions?
    - Branch prediction
    - Are all branches created equal?
    - Branches vs No branches (cmov)

All of these depend on the *micro-architecture*!




INTEL® CORE™ i9
i9-9900K
SRELS 3.60GHZ
L903F183
e4
06035



Skylake Client Microarchitecture (Intel)

Code layout

Likely (“hot”) code paths should be close

Unlikely (“cold”) code paths should be far

- Reduce I-Cache stress == leave room for hot stuff

Code layout

A simple switch statement.

```
int foo(int x)
{
    switch (x)
    {
        case 0:
            return foo0();
        case 1:
            return foo1();
        case 2:
            return foo2();
        case 3:
            return foo3();
        case 4:
            return foo4();
        case 5:
            return foo5();
        case 6:
            return foo6();
        case 7:
            return foo7();
        case 8:
            return foo8();
        case 9:
            return foo9();
        default:
            return bar(x);
    }
}
```

https://godbolt.org/z/ceK9ddreb

Code layout

A simple switch statement.

- Generally, cases are in arbitrary order.
- Default case is last: Compiler assumes that default case is less likely
- The way we write code gives some hints to the optimizer: the default case is less likely
- gcc has a very good optimizer, assume it knows what it's doing

```
int foo(int x)
{
    switch (x)
    {
        case 0:
            return foo0();
        case 1:
            return foo1();
        case 2:
            return foo2();
        case 3:
            return foo3();
        case 4:
            return foo4();
        case 5:
            return foo5();
        case 6:
            return foo6();
        case 7:
            return foo7();
        case 8:
            return foo8();
        case 9:
            return foo9();
        default:
            return bar(x);
    }
}
```

```
1  foo(int):
2          cmp     edi, 9
3          ja      .L2
4          mov     edi, edi
5          jmp     [QWORD PTR .L4[0+rdi*8]]
6  .L4:
7          .quad   .L13
8          .quad   .L12
9          .quad   .L11
10         .quad   .L10
11         .quad   .L9
12         .quad   .L8
13         .quad   .L7
14         .quad   .L6
15         .quad   .L5
16         .quad   .L3
17 .L5:
18         jmp     foo8()
19 .L3:
20         jmp     foo9()
21 .L13:
22         jmp     foo0()
23 .L12:
24         jmp     foo1()
25 .L11:
26         jmp     foo2()
27 .L10:
28         jmp     foo3()
29 .L9:
30         jmp     foo4()
31 .L8:
32         jmp     foo5()
33 .L7:
34         jmp     foo6()
35 .L6:
36         jmp     foo7()
37 .L2:
38         jmp     bar(int)
```

https://godbolt.org/z/ceK9ddreb

Cod

A simp

- Ger
  ord
- Default case last: Compiler assumes th ss likely
- The way we wri gives some hints to the optimizer: the default case is less likely
- gcc has a very good optimizer, assume it knows what it's doing

Conditional branch

```
cmp     edi, 9
ja      .L2
mov     edi, edi
jmp     [QWORD PTR .L4[0+rdi*8]]
```

Indirect branch

```
int foo(int x)
{
    switch (x)
    {
        case 4:
            return foo4();
        case 5:
            return foo5();
        case 6:
            return foo6();
        case 7:
            return foo7();
        case 8:
            return foo8();
        case 9:
            return foo9();
        default:
            return bar(x);
    }
}
```

```
1  foo(int):
2          cmp     edi, 9
3          ja      .L2
           mov     edi, edi
5          jmp     [QWORD PTR .L4[0+rdi*8]]
6  .L4:
7          .quad   .L13
8          .quad   .L12
9          .quad   .L11
10         .quad   .L10
11         .quad   .L9
12         .quad   .L8
13         .quad   .L7
14         .quad   .L6
15         .quad   .L5
16         .quad   .L3
17 .L5:
18         jmp     foo8()
19 .L3:
20         jmp     foo9()
21 .L13:
22         jmp     foo0()
23 .L12:
24         jmp     foo1()
25 .L11:
26         jmp     foo2()
27 .L10:
28         jmp     foo3()
29 .L9:
30         jmp     foo4()
31 .L8:
32         jmp     foo5()
33 .L7:
34         jmp     foo6()
35 .L6:
36         jmp     foo7()
37 .L2:
38         jmp     bar(int)
```

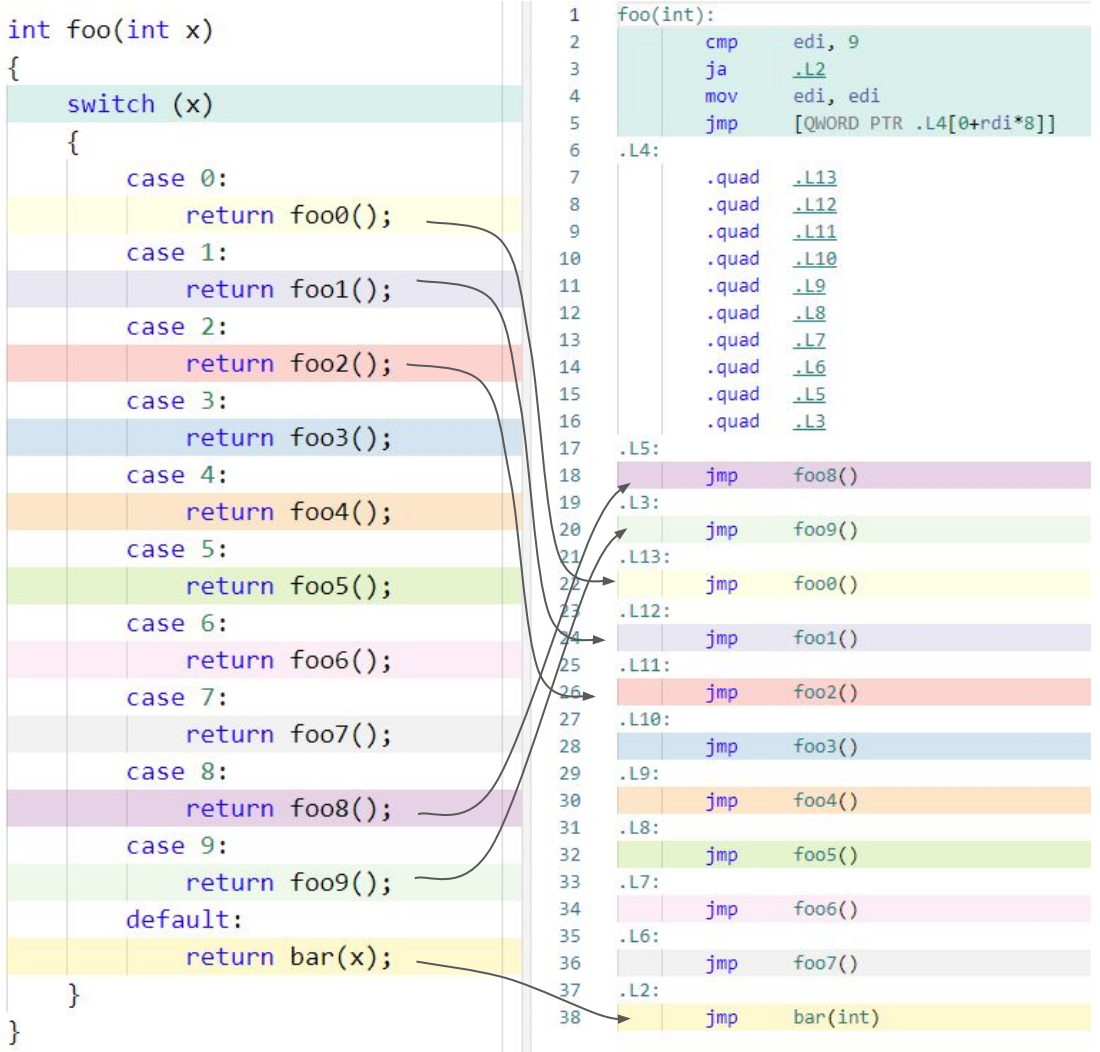
Jump table

https://godbolt.org/z/ceK9ddreb

Code layout

A simple switch statement.

- Generally, cases are in arbitrary order.
- Default case is last: Compiler assumes that default case is less likely
- The way we write code gives some hints to the optimizer: the default case is less likely
- gcc has a very good optimizer, assume it knows what it's doing



```
int foo(int x)
{
    switch (x)
    {
        case 0:
            return foo0();
        case 1:
            return foo1();
        case 2:
            return foo2();
        case 3:
            return foo3();
        case 4:
            return foo4();
        case 5:
            return foo5();
        case 6:
            return foo6();
        case 7:
            return foo7();
        case 8:
            return foo8();
        case 9:
            return foo9();
        default:
            return bar(x);
    }
}
```

```
foo(int):
        cmp     edi, 9
        ja      .L2
        mov     edi, edi
        jmp     [QWORD PTR .L4[0+rdi*8]]
.L4:
        .quad   .L13
        .quad   .L12
        .quad   .L11
        .quad   .L10
        .quad   .L9
        .quad   .L8
        .quad   .L7
        .quad   .L6
        .quad   .L5
        .quad   .L3
.L5:
        jmp     foo8()
.L3:
        jmp     foo9()
.L13:
        jmp     foo0()
.L12:
        jmp     foo1()
.L11:
        jmp     foo2()
.L10:
        jmp     foo3()
.L9:
        jmp     foo4()
.L8:
        jmp     foo5()
.L7:
        jmp     foo6()
.L6:
        jmp     foo7()
.L2:
        jmp     bar(int)
```

https://godbolt.org/z/ceK9ddreb

Code layout

Case 5 is unlikely
=> put it further away

Case 7 is likely
=> put it closer

Default case is still last!

What if we mark default as [[likely]]?

```
int foo(int x)
{
    switch (x)
    {
        case 0:
            return foo0();
        case 1:
            return foo1();
        case 2:
            return foo2();
        case 3:
            return foo3();
        case 4:
            return foo4();
        [[unlikely]] case 5:
            return foo5();
        case 6:
            return foo6();
        [[likely]] case 7:
            return foo7();
        case 8:
            return foo8();
        case 9:
            return foo9();
        default:
            return bar(x);
    }
}
```

```
1  foo(int):
2          cmp     edi, 9
3          ja      .L2
4          mov     edi, edi
5          jmp     [QWORD PTR .L4[0+rdi*8]]
6  .L4:
7          .quad   .L13
8          .quad   .L12
9          .quad   .L11
10         .quad   .L10
11         .quad   .L9
12         .quad   .L8
13         .quad   .L7
14         .quad   .L6
15         .quad   .L5
16         .quad   .L3
17 .L6:
18         jmp     foo7()
19 .L5:
20         jmp     foo8()
21 .L7:
22         jmp     foo6()
23 .L9:
24         jmp     foo4()
25 .L10:
26         jmp     foo3()
27 .L11:
28         jmp     foo2()
29 .L12:
30         jmp     foo1()
31 .L13:
32         jmp     foo0()
33 .L3:
34         jmp     foo9()
35 .L8:
36         jmp     foo5()
37 .L2:
38         jmp     bar(int)
```

https://godbolt.org/z/ceK9ddreb

Code layout

What if we mark default as [[likely]]?

Now it's first!

Q: Why did we remove the [[likely]] from case 7?

A: Otherwise, gcc doesn't like it!

```
int foo(int x)
{
    switch (x)
    {
        case 0:
            return foo0();
        case 1:
            return foo1();
        case 2:
            return foo2();
        case 3:
            return foo3();
        case 4:
            return foo4();
        [[unlikely]] case 5:
            return foo5();
        case 6:
            return foo6();
        case 7:
            return foo7();
        case 8:
            return foo8();
        case 9:
            return foo9();
        [[likely]] default:
            return bar(x);
    }
}
```

```
1   foo(int):
2           cmp     edi, 9
3           ja      .L2
4           mov     edi, edi
5           jmp     [QWORD PTR .L4[0+rdi*8]]
6   .L4:
7           .quad   .L13
8           .quad   .L12
9           .quad   .L11
10          .quad   .L10
11          .quad   .L9
12          .quad   .L8
13          .quad   .L7
14          .quad   .L6
15          .quad   .L5
16          .quad   .L3
17  .L2:
18          jmp     bar(int)
19  .L5:
20          jmp     foo8()
21  .L6:
22          jmp     foo7()
23  .L7:
24          jmp     foo6()
25  .L9:
26          jmp     foo4()
27  .L10:
28          jmp     foo3()
29  .L11:
30          jmp     foo2()
31  .L12:
32          jmp     foo1()
33  .L13:
34          jmp     foo0()
35  .L3:
36          jmp     foo9()
37  .L8:
38          jmp     foo5()
```

https://godbolt.org/z/ceK9ddreb

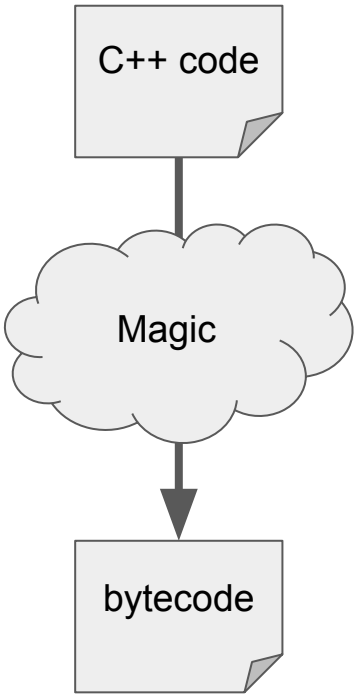
Why?

What can the compiler (optimizer) do with this information?

1. Improve code layout?
   - Instruction Cache (I-Cache)

2. Improve branch instructions?
   - Branch prediction
   - Are all branches created equal?
   - Branches vs No branches (cmov)

All of these depend on the *micro-architecture*!



A tale of two branch predictors

The CPU's branch predictor

The compiler's branch predictor

[1] *The microarchitecture of Intel, AMD, and VIA CPUs*, Agner Fog

A tale of two branch predictors

The CPU's branch predictor

- Predicts most likely execution path
- Sees a stream of bytecodes
- Based on previous runtime data
  - Full details are an industry secret…
- Predictions affect the execution pipeline
- A branch misprediction is costly
  - Throw away the pipeline, 16-20 cycles[1]
- Branch prediction is necessary for good performance - otherwise the pipeline has to stall every time

The compiler's branch predictor

Stack Overflow: Why is processing a sorted array faster than processing an unsorted array?

[1] *The microarchitecture of Intel, AMD, and VIA CPUs*, Agner Fog

A tale of two branch predictors

The CPU's branch predictor

- Predicts most likely execution path
- Sees a stream of bytecodes
- Based on previous runtime data
  - Full details are an industry secret…
- Predictions affect the execution pipeline
- A branch misprediction is costly
  - Throw away the pipeline, 16-20 cycles[1]
- Branch prediction is necessary for good performance - otherwise the pipeline has to stall every time

The compiler's branch predictor

- Predicts most likely execution path
- Sees all the code at once
- Sees code in high-level language (C++)
- Based on static analysis (compile time)
- Predictions affect machine code output
  - Implementation detail: needs to tag the IR with source code information
- PGO (profiler-guided optimization) can help here

What's the connection between them?

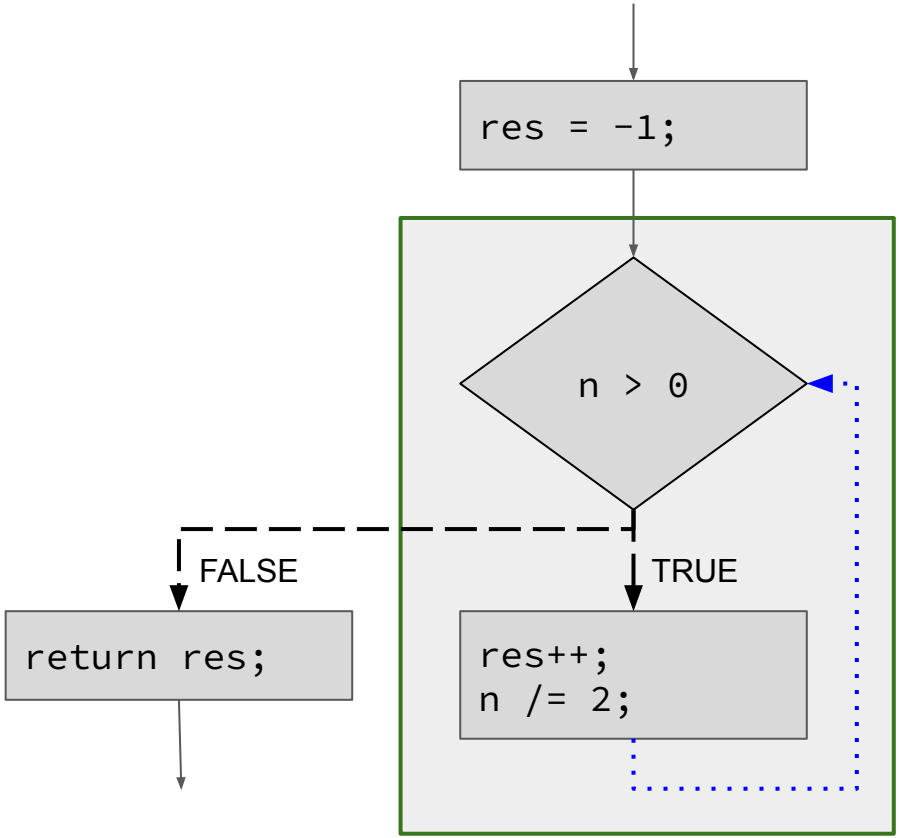
[1] *The microarchitecture of Intel, AMD, and VIA CPUs*, Agner Fog

CPU's Branch Predictor

```
int log2(int n)
{
    int res = -1;
    while (n > 0) {
        res++;
        n /= 2;
    }
    return res;
}
```

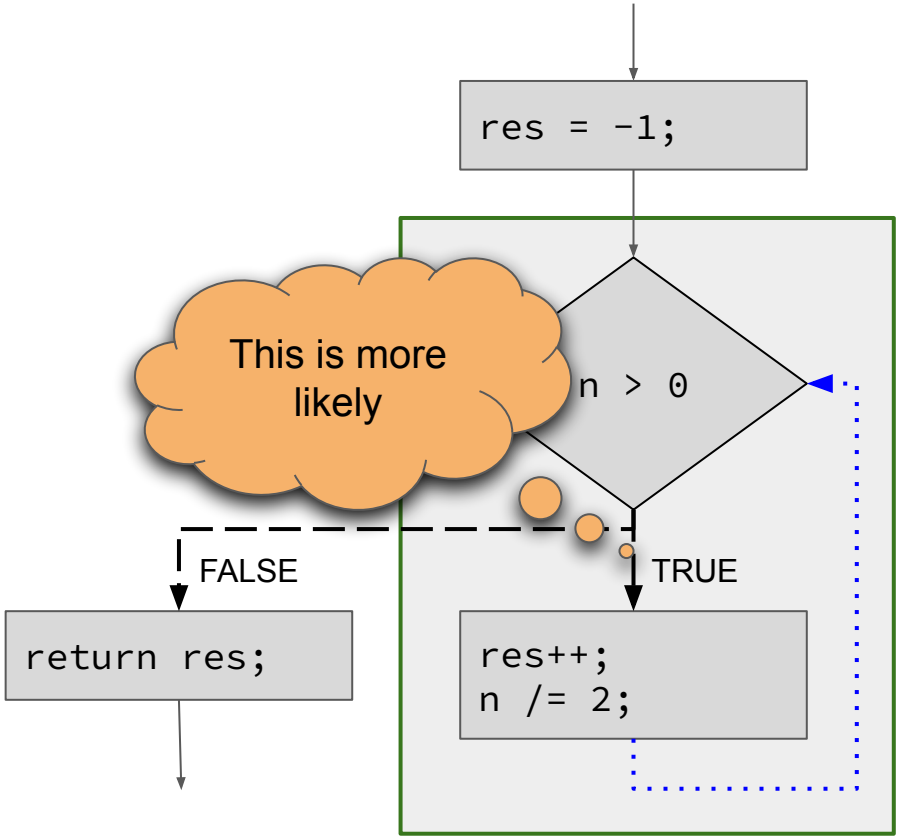
CPU's Branch Predictor

```
int log2(int n)
{
    int res = -1;
    while (n > 0) {
        res++;
        n /= 2;
    }
    return res;
}
```



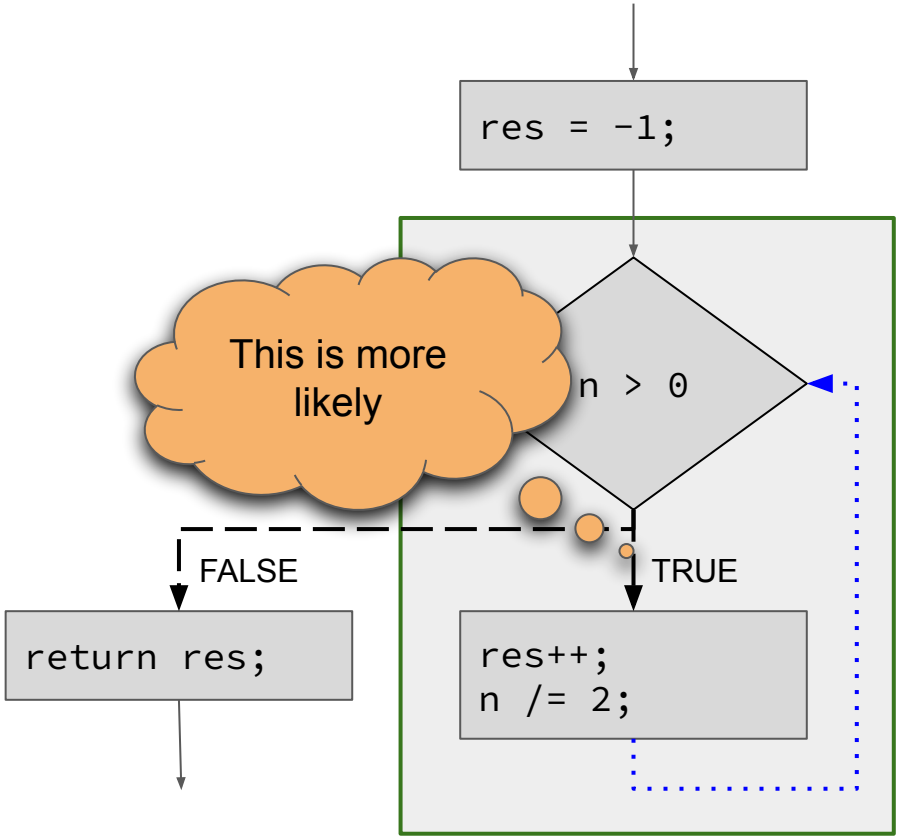
CPU's Branch Predictor

```
int log2(int n)
{
    int res = -1;
    while (n > 0) {
        res++;
        n /= 2;
    }
    return res;
}
```



CPU's Branch Predictor

The CPU doesn't know the structure of the program, but it still makes a good guess!



Does `[[likely]]` affect CPU branch predictor?

- **Branch hints:** encode prediction inside the machine code?
- x86-64 (Intel, AMD):
  - Historic branch hints: 0x2E (branch not taken), 0x3E (branch taken)
  - Existed only in Pentium 4, now reserved for future use
    - 0x3E is already repurposed for Control-flow Enforcement Technology (CET)
  - You're probably not writing C++20 code for Pentium 4 :-)
- Arm (Arm7, AArch32, AArch64):
  - None
- POWER (Power, PowerPC):
  - Branch hints exist, but not used in power64 compilation.
  - Usage discouraged "unless the static prediction [...] is highly likely to be correct."

Does `[[likely]]` affect CPU branch predictor?

- **Cold branch:** When a branch is encountered for the first time
- Intel:
  - Forward conditional branches: Predict NOT TAKEN.
  - Backward conditional branches: Predict TAKEN.
- AMD:
  - Always predict NOT TAKEN
- **Just the first time** => minimal performance impact
  - The CPU remembers the branch by its memory address
  - It can drop cold predictions - but they're cold, so they don't affect performance
- Indirect branches (switch/case jump table, call by pointer) are more complicated

Does `[[likely]]` affect CPU branch predictor?

- In summary:
  - **Not really**

Branch Predictor

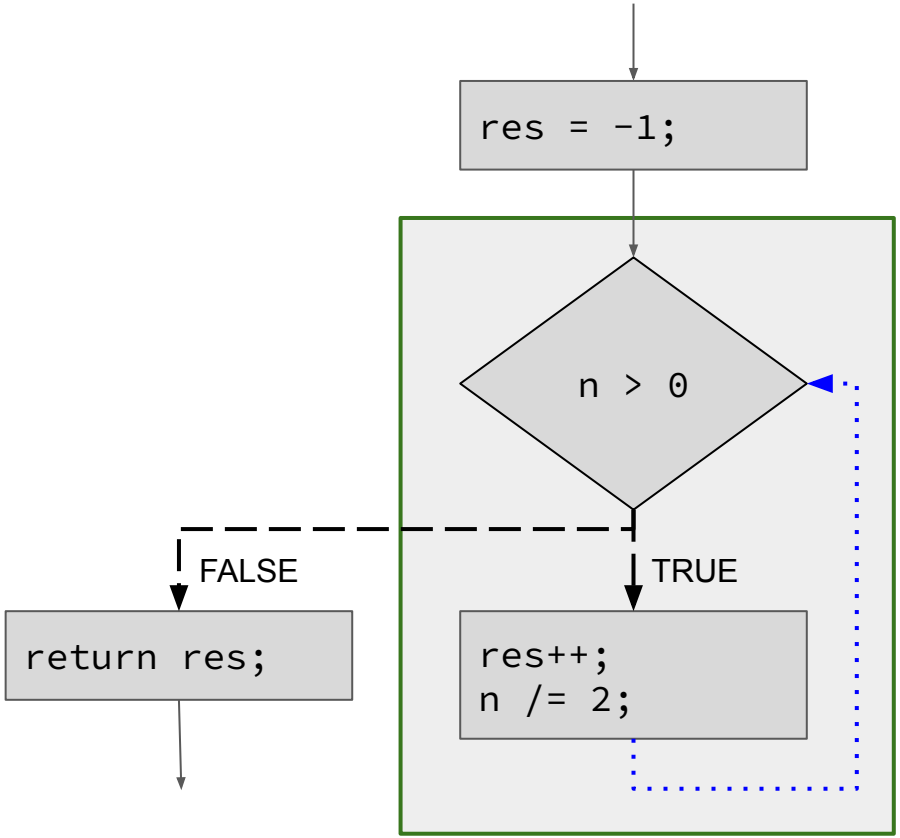
Which branch predictor does `[[likely]]` affect?

❌ CPU

- Compiler

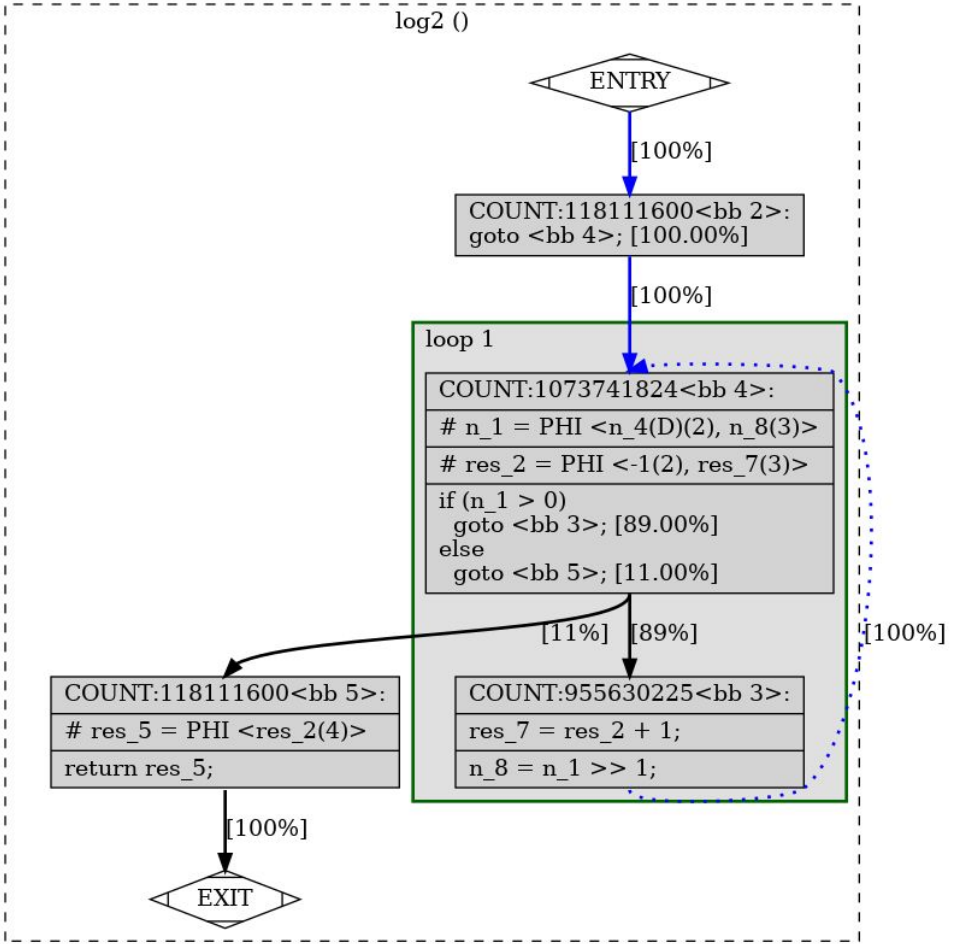
Compiler's Predictor

```
int log2(int n)
{
    int res = -1;
    while (n > 0) {
        res++;
        n /= 2;
    }
    return res;
}
```



Compiler's Predictor

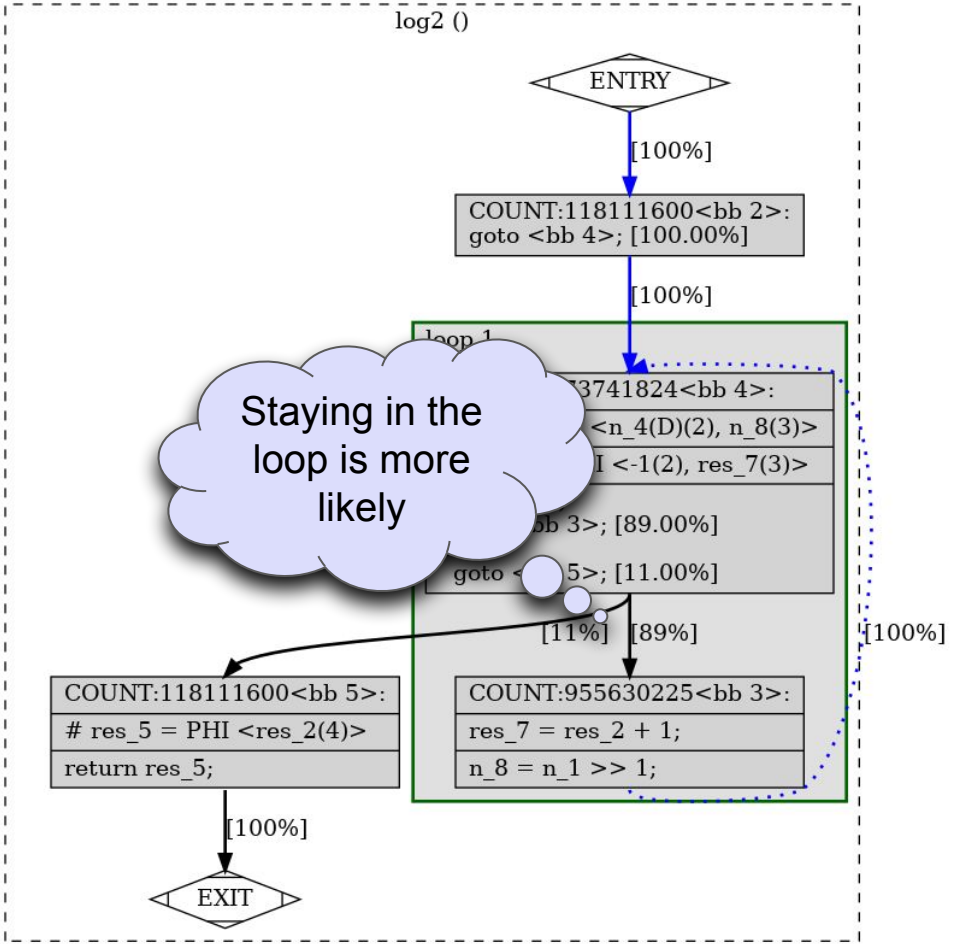
```
int log2(int n)
{
    int res = -1;
    while (n > 0) {
        res++;
        n /= 2;
    }
    return res;
}
```




```
g++ -fdump-tree-all-graph snippet.cpp -o snippet -O3
dot ./snippet.cpp.048t.profile_estimate.dot -Tpng > estimate.png
```

Compiler's Predictor

```
int log2(int n)
{
    int res = -1;
    while (n > 0) {
        res++;
        n /= 2;
    }
    return res;
}
```




```
g++ -fdump-tree-all-graph snippet.cpp -o snippet -O3
dot ./snippet.cpp.048t.profile_estimate.dot -Tpng > estimate.png
```

The effect of adding [[likely]]

```
int log2(int n)
{
    int res = -1;
    while (n > 0) {
        res++;
        n /= 2;
    }
    return res;
}
```

```
int log2(int):
    mov eax, -1
    testedi, edi
    jle .L4
    .p2align ...
.L3:
    add eax, 1
    sar edi
    jne .L3
    ret
    .p2align ...
.L4:
    ret
```

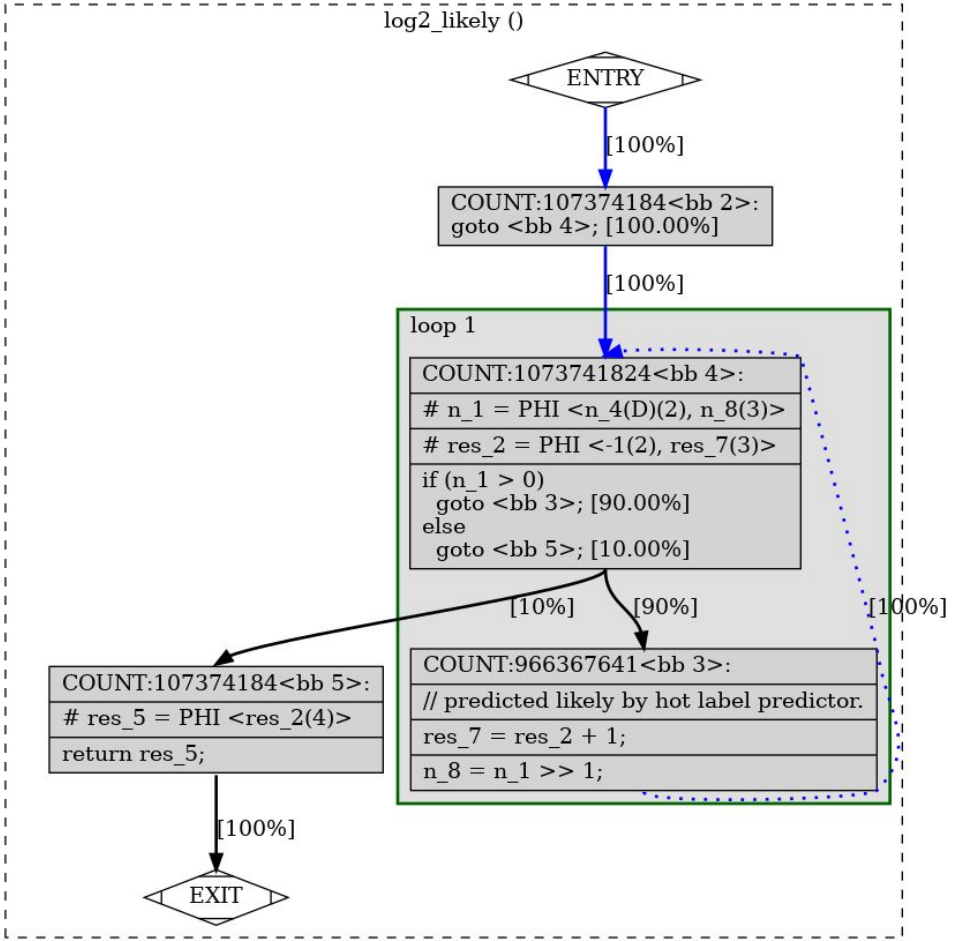
The effect of adding [[likely]]

```
int log2_likely(int n)
{
    int res = -1;
    while (n > 0) { [[likely]]
        res++;
        n /= 2;
    }
    return res;
}
```

```
int log2_likely(int):
    mov eax, -1
    testedi, edi
    jle .L4
    .p2align ...
.L3:
    add eax, 1
    sar edi
    jne .L3
    ret
    .p2align ...
.L4:
    ret
```

Compiler's Predictor

```
int log2(int n)
{
    int res = -1;
    while (n > 0) { [[likely]]
        res++;
        n /= 2;
    }
    return res;
}
```

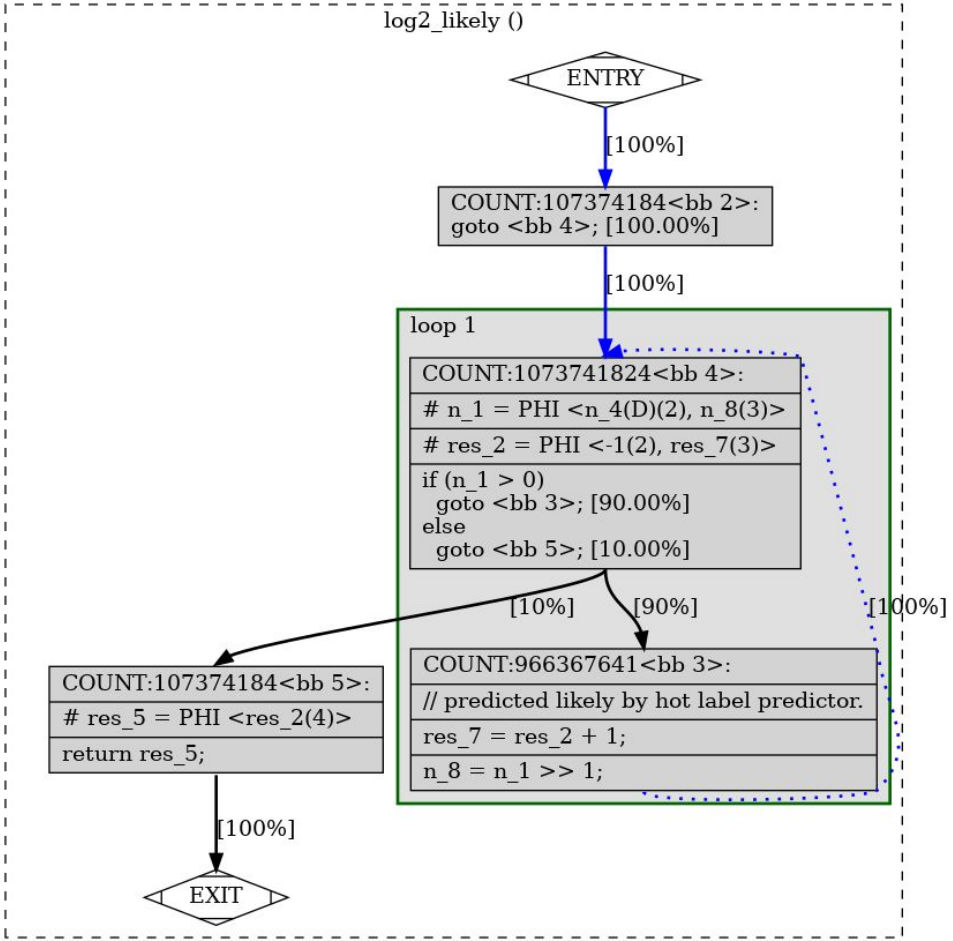


Compiler's Predictor

- Adding [[likely]] for the loop body is useless
- But it could be interesting to add [[unlikely]] there…

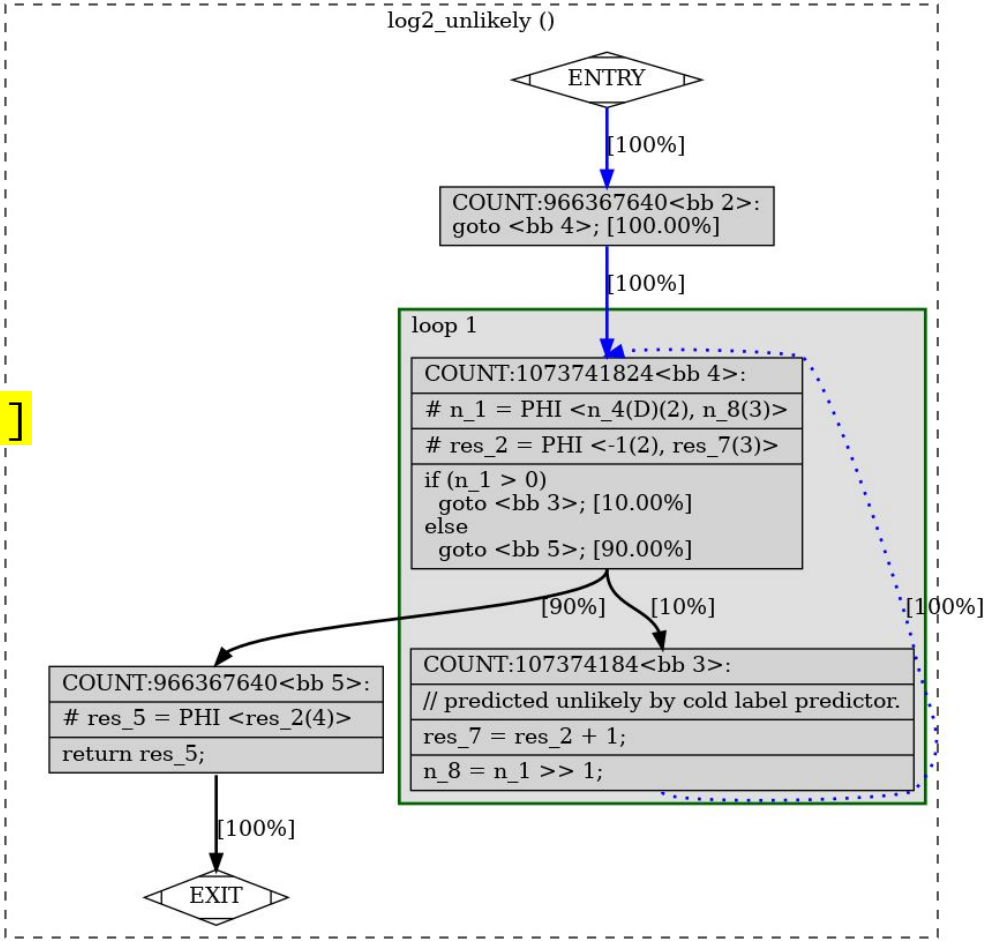
Compiler's Predictor

```
int log2(int n)
{
    int res = -1;
    while (n > 0) { [[likely]]
        res++;
        n /= 2;
    }
    return res;
}
```



Compiler's Predictor

```
int log2(int n)
{
    int res = -1;
    while (n > 0) { [[unlikely]]
        res++;
        n /= 2;
    }
    return res;
}
```



[[likely]] vs. [[unlikely]]

```
int log2_unlikely(int n)
{
    int res = -1;
    while (n > 0) { [[unlikely]]
        res++;
        n /= 2;
    }
    return res;
}
```

```
int log2_unlikely(int):
    mov eax, -1
    testedi, edi
    jg  .L3
    ret
    .p2align ...
.L3:
    add eax, 1
    sar edi
    jne .L3
    ret
```

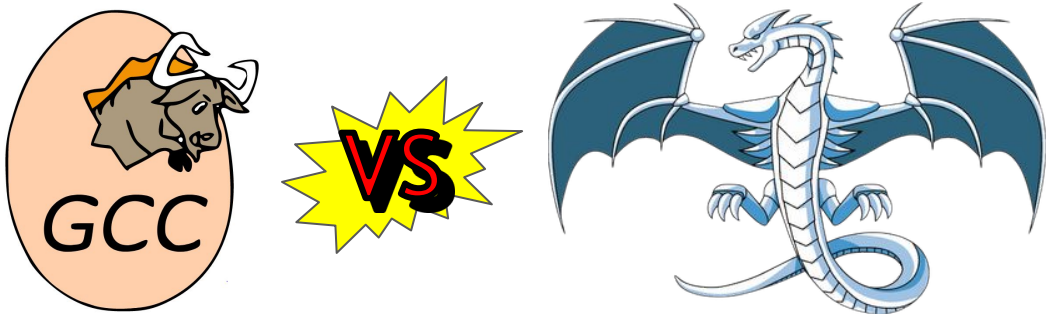
[[likely]] vs. [[unlikely]] - side by side

```
int log2_likely(int):
    mov eax, -1
    testedi, edi
    jle .L4
    .p2align ...

.L3:
    add eax, 1
    sar edi
    jne .L3
    ret
    .p2align ...
.L4:
    ret
```

```
int log2_unlikely(int):
    mov eax, -1
    testedi, edi
    jg  .L3
    ret
    .p2align ...
.L3:
    add eax, 1
    sar edi
    jne .L3
    ret
```

GCC vs Clang

- The `log2` examples are from GCC
- GCC has a visual view of the estimations
- Clang outputs the same assembly for both *likely* and *unlikely* for many other examples that we tried
- We'll focus on GCC for the rest of the talk



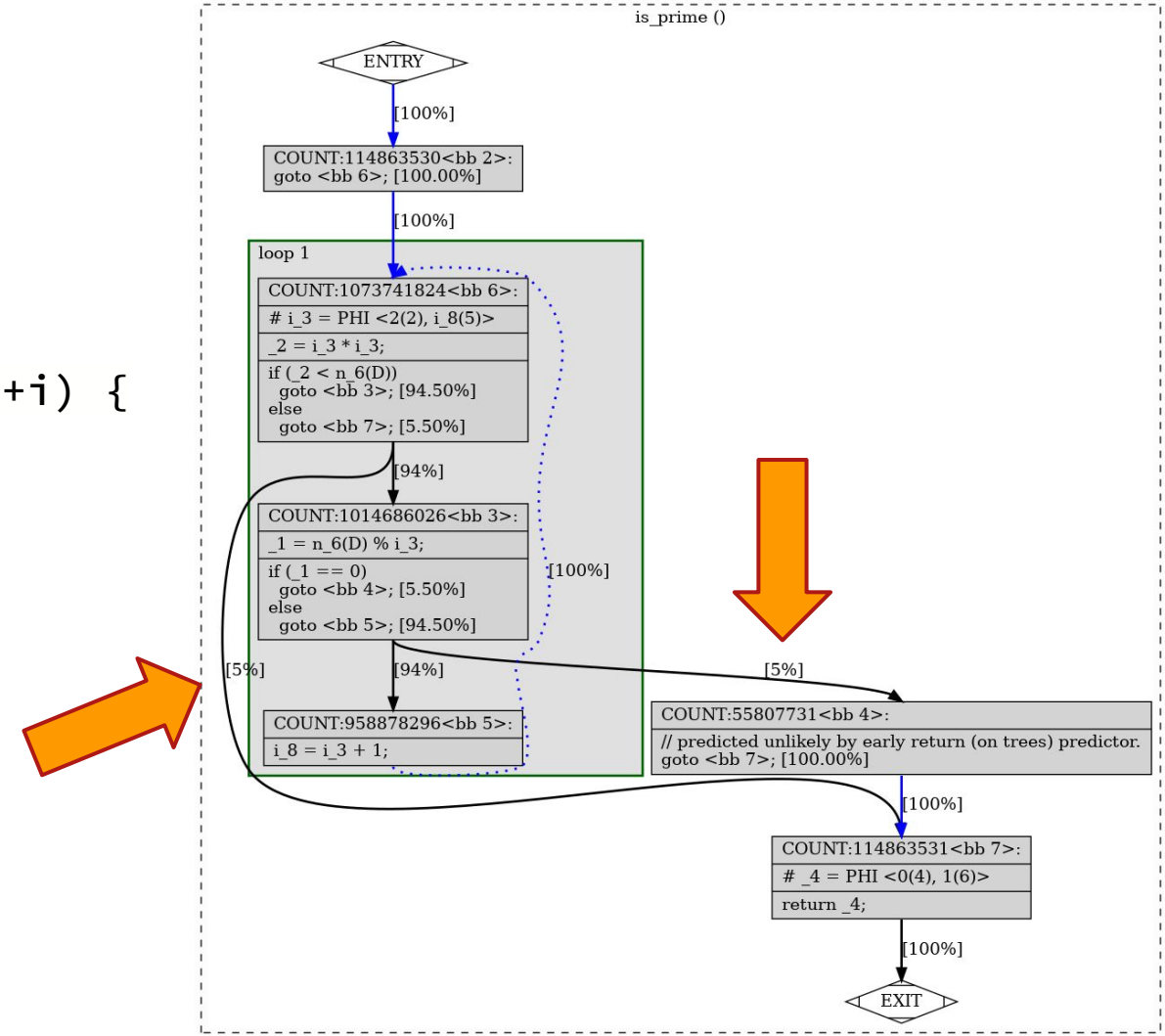
More examples

- gcc 12.1
- x86-64 (Intel/AMD)
- `--std=c++20 -O3`
- Compare code with & without [[likely]]/[[unlikely]]

https://godbolt.org/z/eKj61rMbj

Early return

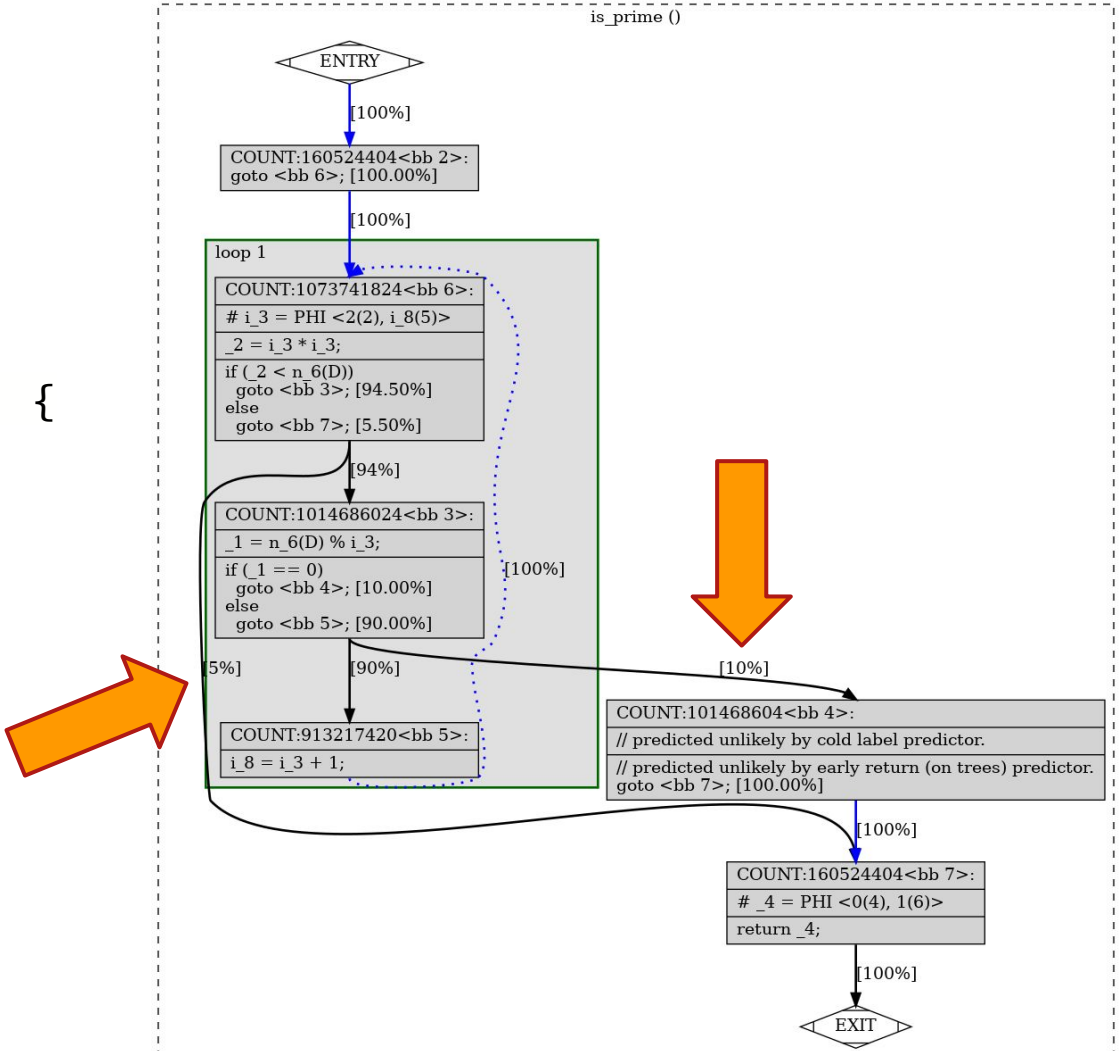
```
bool is_prime(int n)
{
  for (int i=2; i*i<n; ++i) {
    if (n % i == 0) {
      return false;
    }
  }
  return true;
}
```



* This function is incorrect for n <= 1

Early return

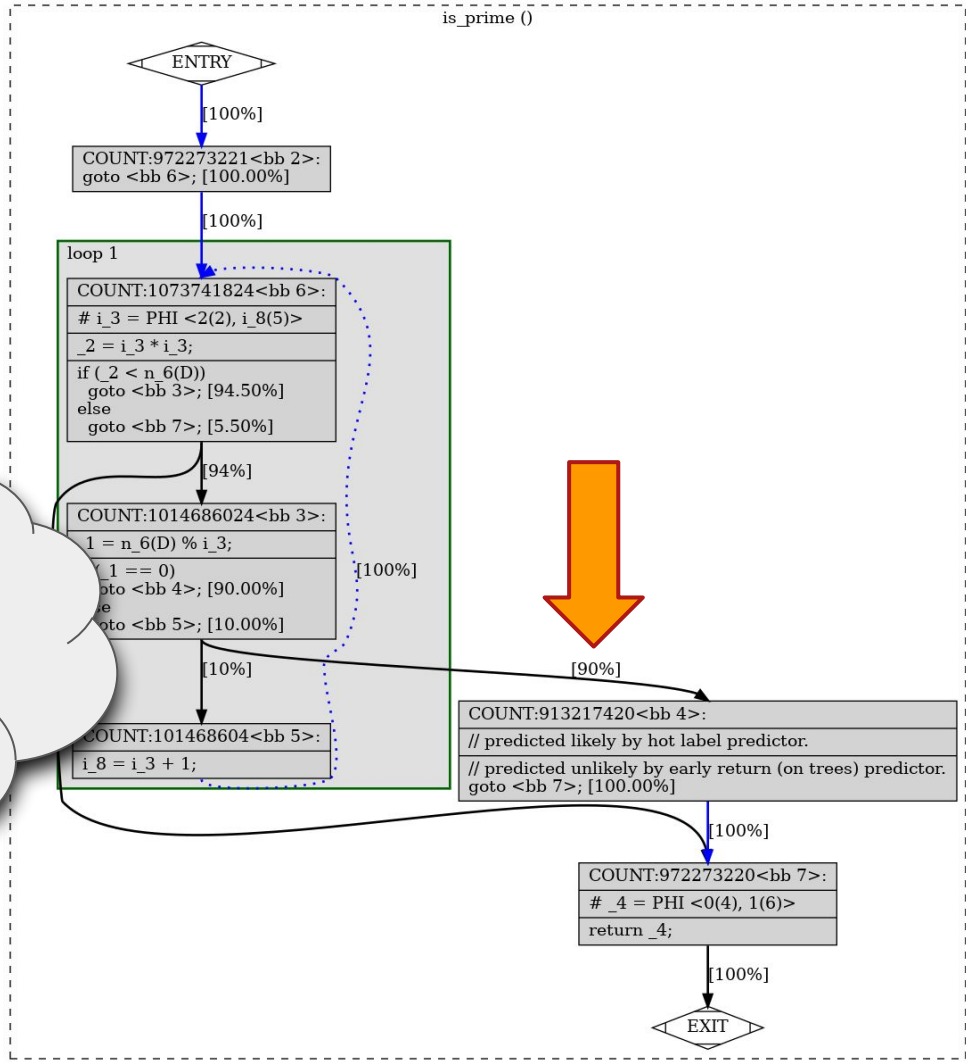
```
bool is_prime(int n)
{
  for (int i=2; i*i<n; ++i) {
    if (n % i == 0) {
      [[unlikely]]
      return false;
    }
  }
  return true;
}
```



Early return

```
bool is_prime(int n)
{
  for (int i=2; i*i<n; ++i) {
    if (n % i == 0) {
      [[likely]]
      retur
    }
  }
  return
}
```

Which version is better?
[[likely]] or [[unlikely]]



Exceptions

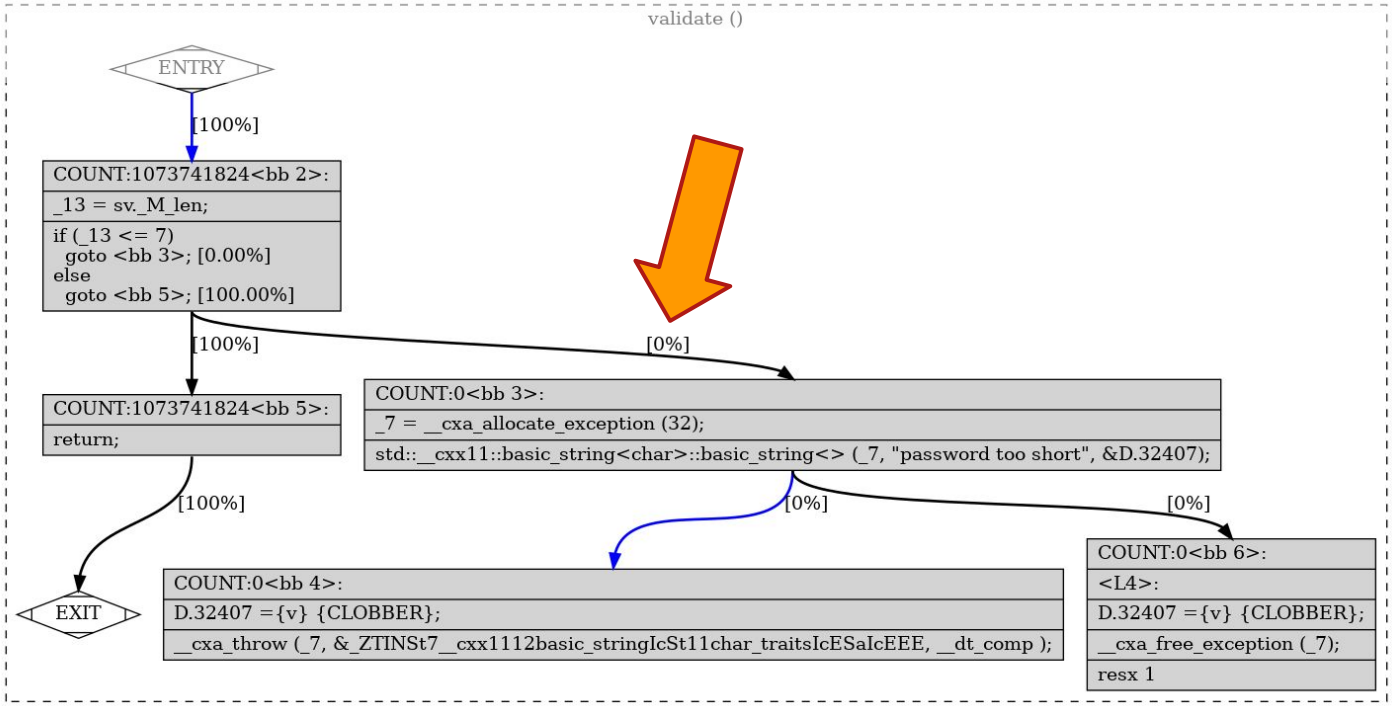
Photo by Muhammad Daudy on Unsplash

Exceptions

```
void validate(string_view sv)
{
  if (sv.size() < 8) { throw string("password too short"); }
}
```

Exceptions

```
void validate(string_view sv)
{
  if (sv.size() < 8) { throw string("password too short"); }
}
```



Exceptions

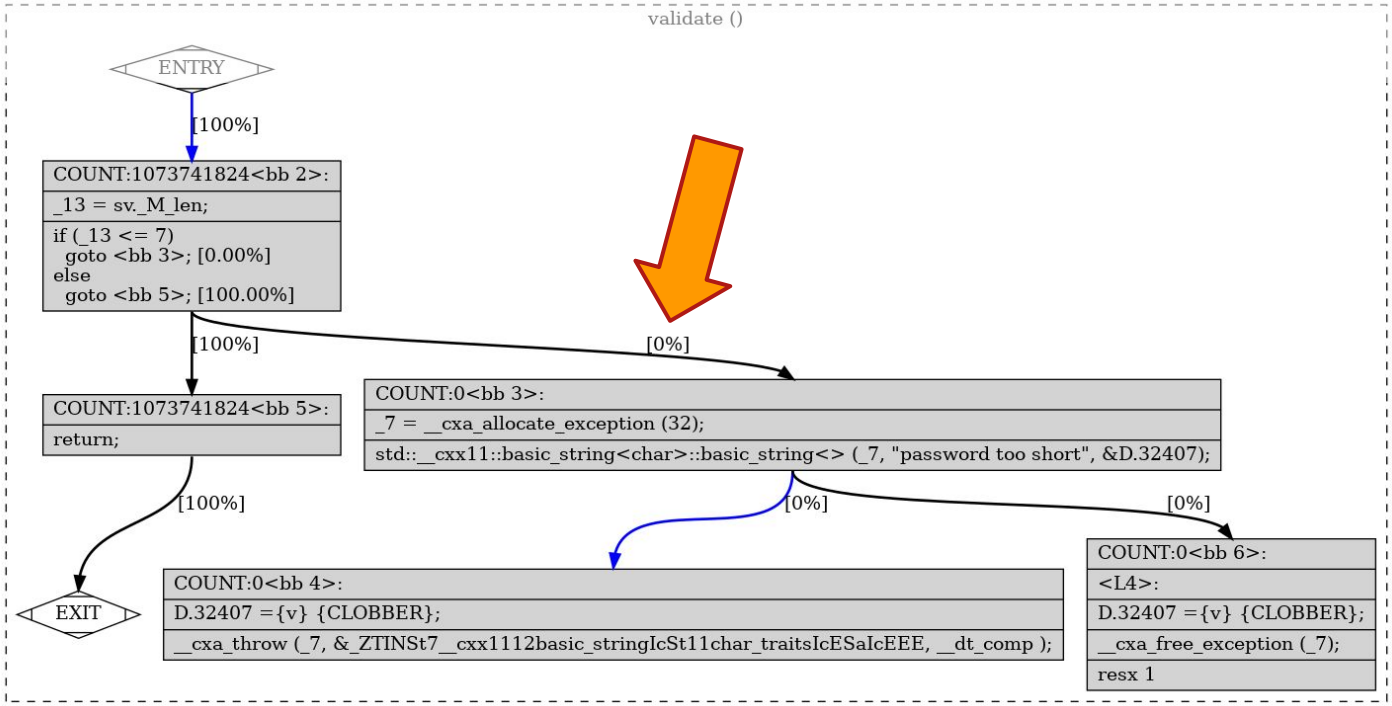
```
void validate(string_view sv)
{
  if (sv.size() < 8) { throw string("password too short"); }
}
```

```
validate(std::basic_string_view<char, std::char_traits<char> >):
        cmp     rdi, 7
        jbe     .L8
        ret
validate(std::basic_string_view<char, std::char_traits<char> >) [clone .cold]:
.L8:
        push    rbp
        mov     edi, 32
        push    rbx
        push    rax
        call    __cxa_allocate_exception
        ...
```

} if (size <= 7) ...

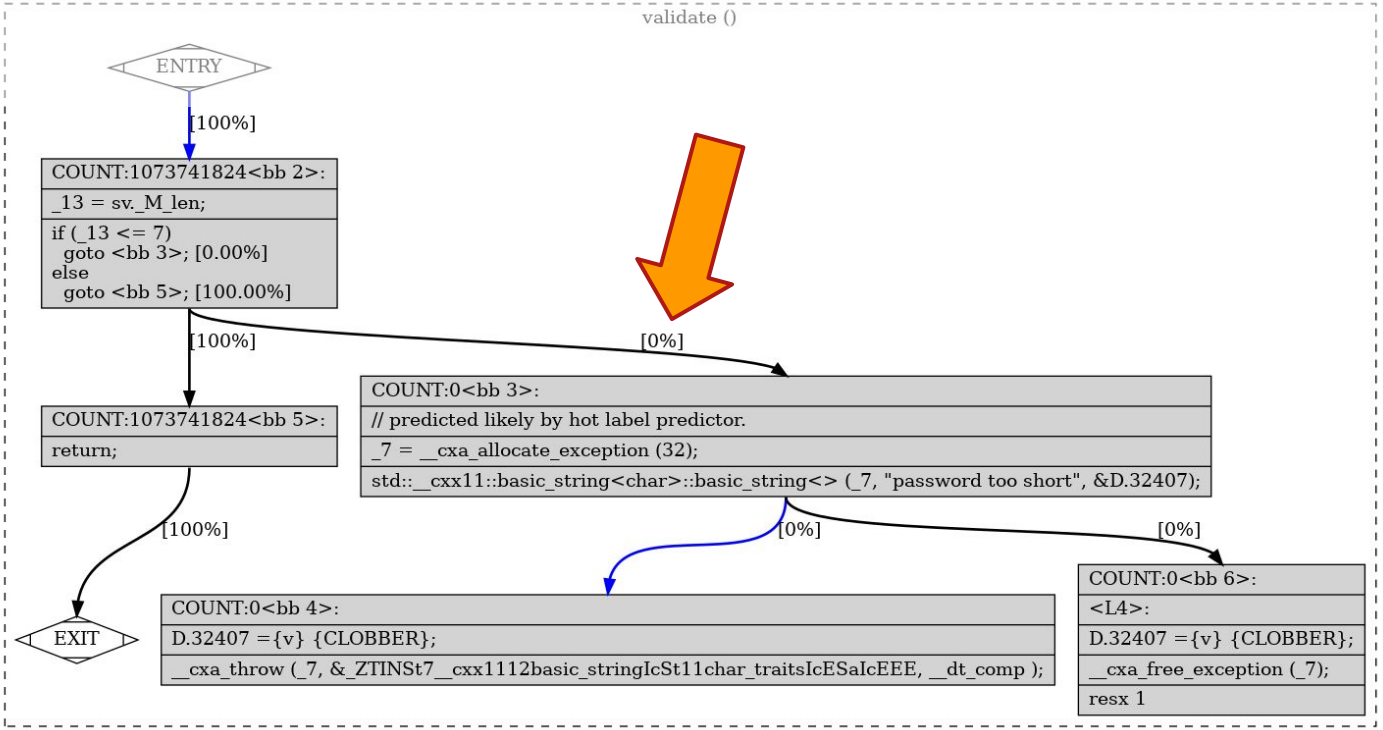
Exceptions

```
void validate(string_view sv)
{
  if (sv.size() < 8) { throw string("password too short"); }
}
```



Exceptions

```
void validate(string_view sv)
{
  if (sv.size() < 8) { [[likely]] throw string("password too short"); }
}
```



Some real code… (almost)

```
int update_vsyscall_tz(void)
{
    if (unlikely(vvar_data == NULL))
        return -ENOMEM;

    vvar
    vvar
    return 0;
}
```

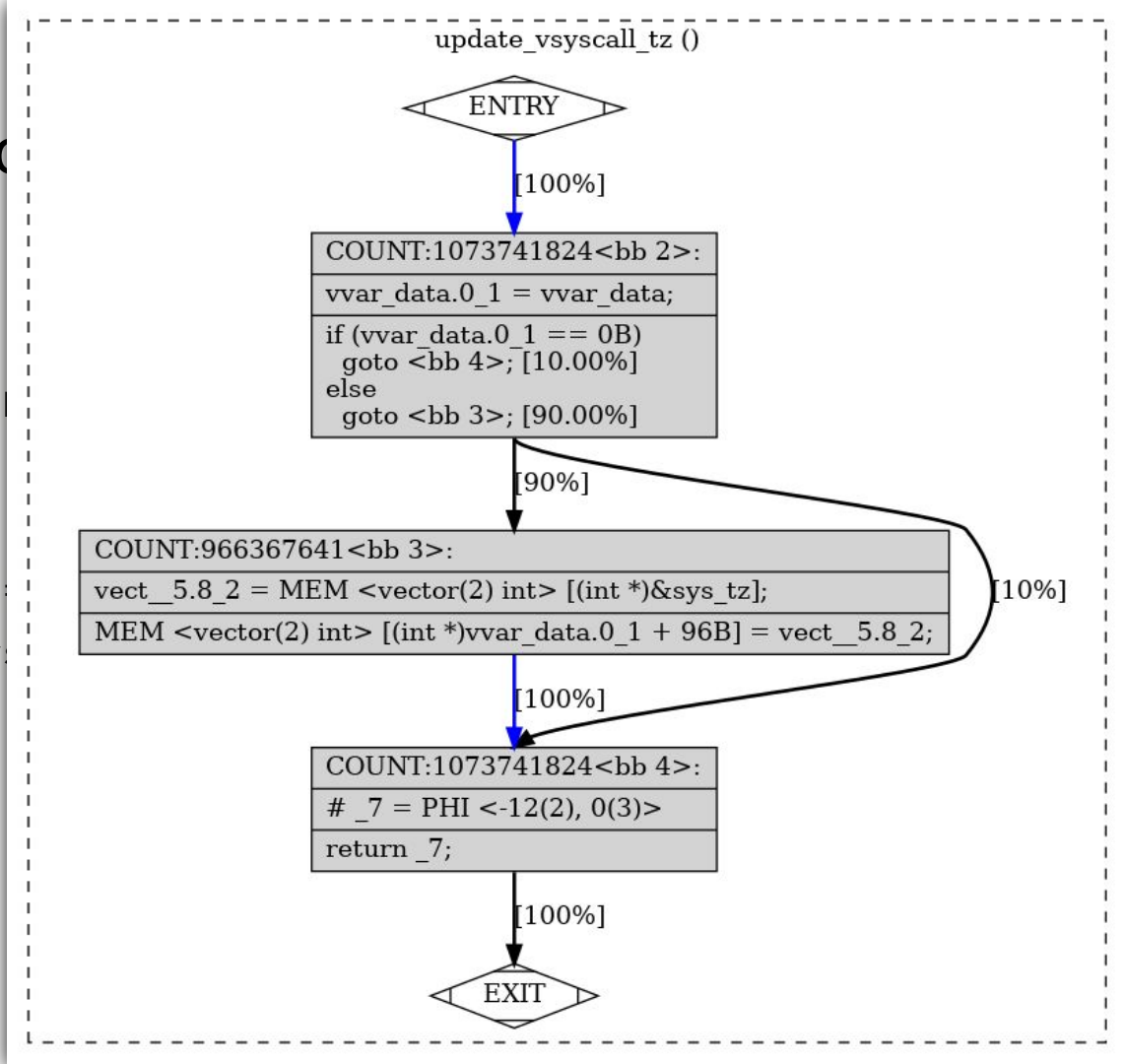
```
#define unlikely(x) __builtin_expect((x),0)
```

[[unlikely]]

https://livegrep.com/search/linux?q=if.*unlikely&fold_case=auto&regex=true&context=true

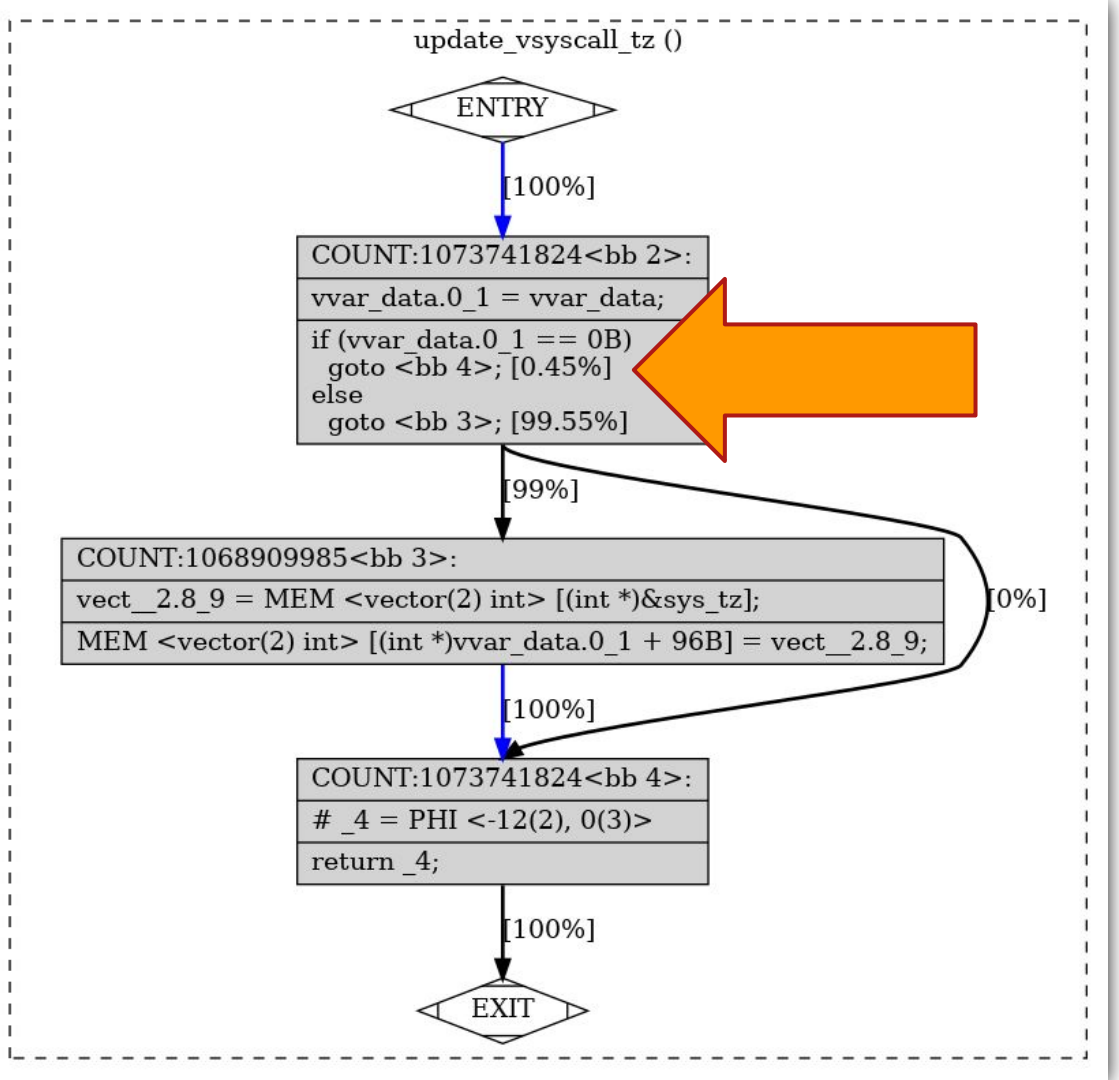
Some real code… (almo

```
int update_vsyscall_tz(void)
{
    if (unlikely(vvar_data == 
        return -ENOMEM;

    vvar_data->tz_minuteswest 
    vvar_data->tz_dsttime = sy
    return 0;
}
```



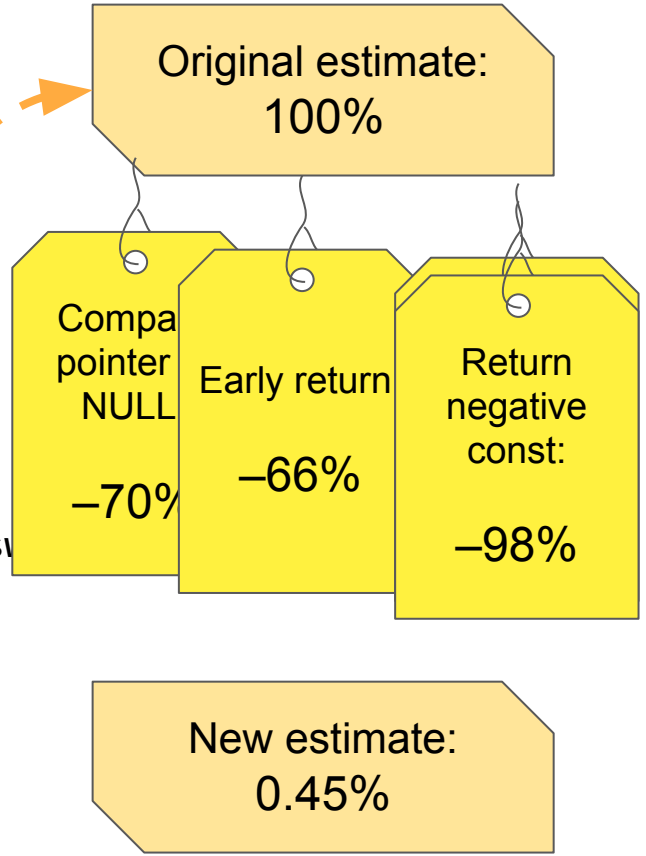
Some real code… (almo

```
int update_vsyscall_tz(void)
{
    if (unlikely(vvar_data ==
        return -ENOMEM;

    vvar_data->tz_minuteswest =
    vvar_data->tz_dsttime = sy
    return 0;
}
```



Some real code… (almost)

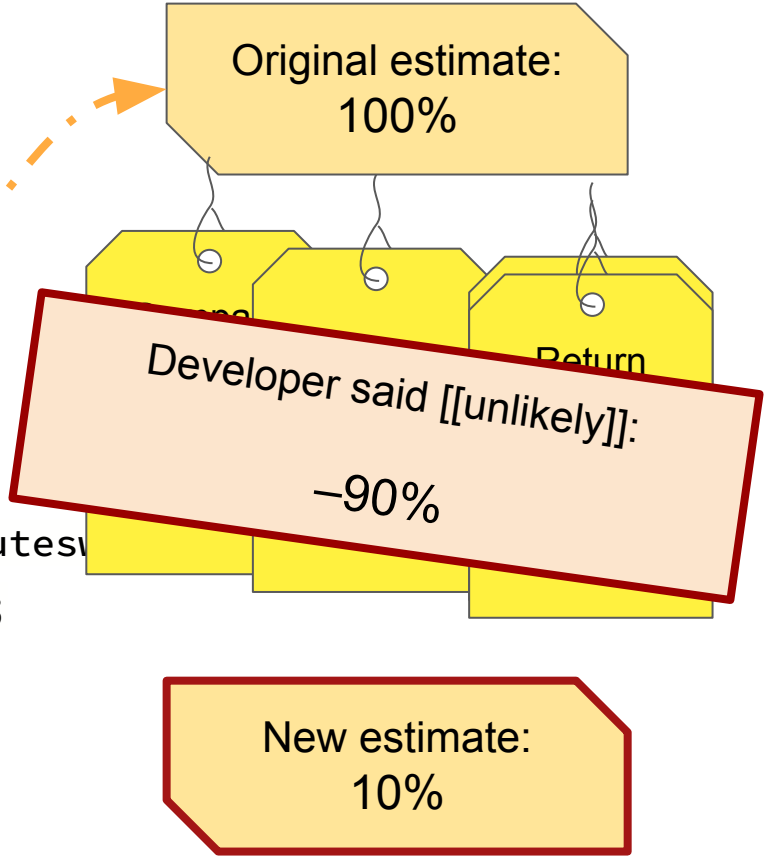
```
int update_vsyscall_tz(void)
{
    if (unlikely(vvar_data == NULL))
        return -ENOMEM;

    vvar_data->tz_minuteswest = sys_tz.tz_minutesw
    vvar_data->tz_dsttime = sys_tz.tz_dsttime;
    return 0;
}
```



http://gcc.gnu.org/git/?p=gcc.git;a=blob;f=gcc/predict.def
http://gcc.gnu.org/git/?p=gcc.git;a=blob;f=gcc/predict.cc
https://godbolt.org/z/snnsq7W63

Some real code… (almost)

```
int update_vsyscall_tz(void)
{
    if (unlikely(vvar_data == NULL))
        return -ENOMEM;

    vvar_data->tz_minuteswest = sys_tz.tz_minutesw
    vvar_data->tz_dsttime = sys_tz.tz_dsttime;
    return 0;
}
```



http://gcc.gnu.org/git/?p=gcc.git;a=blob;f=gcc/predict.def
http://gcc.gnu.org/git/?p=gcc.git;a=blob;f=gcc/predict.cc
https://godbolt.org/z/PcMcPeq9q

Compiler's Predictor

- Complicated (and good!) heuristics
- [[likely]]/[[unlikely]] is a crude override mechanism
- If you really care, use PGO
    - Make sure to run it on representative inputs!

BENCHMARKS

Photo by Agê Barros on Unsplash

Benchmarking is hard

- Micro-benchmarking is mega-hard
- But we're here to talk about performance, so let's try anyway!
- Compare `is_prime()` versions
- AWS EC2
  - Very stable measurements
  - Cheap instance: t2.micro
  - Dedicated node: c5.9xlarge
- Ubuntu 22.04
- GCC 11.2
- `g++ --std=c++20 -O2`

`-O2` is often used in practice

- Is optimisation level -O3 dangerous in g++? (2012)
- gcc optimization flag -O3 makes code slower than -O2 (2015)
- Why -O3? (2015)
- -O2 vs -O3 compiler optimization level (2018)
- Bubble sort slower with -O3 than -O2 with GCC (2021)

Conclusions apply to `-O3` as well

The two versions

```
bool is_prime_likely(int n)
{
    for (int i = 2; i * i < n; ++i) {
        if (n % i == 0) {
            [[likely]]
            return false;
        }
    }
    return true;
}
```

```
bool is_prime_unlikely(int n)
{
    for (int i = 2; i * i < n; ++i) {
        if (n % i == 0) {
            [[unlikely]]
            return false;
        }
    }
    return true;
}
```

Similar to compiler's default estimation, but explicit

Benchmark is_prime

```
bool is_prime_likely() {...}
bool is_prime_unlikely() {...}
int main() {
    vector<int> numbers = ...;
    auto benchmark = ...;
    benchmark(is_prime_likely);
    benchmark(is_prime_unlikely);
}
```

```
auto benchmark =
[&numbers](auto f, auto remark) {
    const auto start =
        chrono::high_resolution_clock::now();
    for (int n : numbers) {
        sink = f(n);
    }
    const chrono::duration<double> diff =
        chrono::high_resolution_clock::now()
        - start;
    cout << "Time: " << diff.count()
         << " sec " << remark << endl;
};
```

Based on https://en.cppreference.com/w/cpp/language/attributes/likely

Benchmark is_prime

The results:

```
Time: 6.619 sec ([[likely]])
Time: 5.987 sec ([[unlikely]])
```

`[[likely]] is slower by 10%!`

* For this input distribution

Benchmark is_prime

- Are we sure about these results?
- “Producing Wrong Data Without Doing Anything Obviously Wrong” [Mytkowicz et al., ASPLOS ‘09]
- Code location can affect measurements
- Let’s try again…

Benchmark is_prime: attempt #2

```
bool is_prime_likely() {...}
bool is_prime_unlikely() {...}
int main() {
    vector<int> numbers = random();
    benchmark(is_prime_likely);
    benchmark(is_prime_unlikely);
}

Time: 6.619 sec ([[likely]])
Time: 5.987 sec ([[unlikely]])
[[likely]] slower by 10%
```

Benchmark is_prime: attempt #2

```
bool is_prime_likely() {...}
bool is_prime_unlikely() {...}
int main() {
    vector<int> numbers = random();
    benchmark(is_prime_likely);
    benchmark(is_prime_unlikely);
}
```

```
Time: 6.619 sec ([[likely]])
Time: 5.987 sec ([[unlikely]])
[[likely]] slower by 10%
```

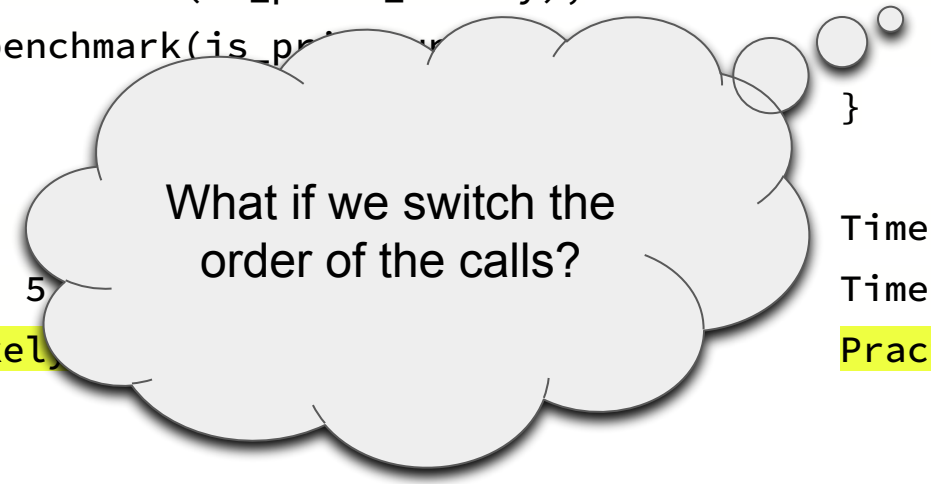
```
bool is_prime_unlikely() {...}
bool is_prime_likely() {...}
int main() {
    vector<int> numbers = random();
    benchmark(is_prime_likely);
    benchmark(is_prime_unlikely);
}
```

```
Time: 6.622 sec ([[likely]])
Time: 6.621 sec ([[unlikely]])
Practically the same
```

Benchmark is_prime: attempt #2

```
bool is_prime_likely() {...}
bool is_prime_unlikely() {...}
int main() {
    vector<int> numbers = random();
    benchmark(is_prime_likely);
    benchmark(is_pr
}

Time:
Time: 5
[[likel
```

```
bool is_prime_unlikely() {...}
bool is_prime_likely() {...}
int main() {
    vector<int> numbers = random();
    benchmark(is_prime_likely);
    benchmark(is_prime_unlikely);
}

Time: 6.622 sec ([[likely]])
Time: 6.621 sec ([[unlikely]])
Practically the same
```



Benchmark is_prime: attempt #2

- Measuring is hard
- But reading assembly is “easy”
- Let’s compare…

```
is_prime_likely(int):
        cmp     edi, 4
        jle     .L7
        xor     eax, eax
        mov     ecx, 2
        test    dil, 1
        jne     .L5
        ret
.L5:
        add     ecx, 1
        mov     eax, ecx
        imul    eax, ecx
        cmp     eax, edi
        jge     .L7
        mov     eax, edi
        cdq
        idiv    ecx
        test    edx, edx
        jne     .L5
        xor     eax, eax
        ret
.L7:
        mov     eax, 1
        ret
```

```
is_prime_unlikely(int):
        cmp     edi, 4
        jle     .L28
        xor     eax, eax
        mov     ecx, 2
        test    dil, 1
        jne     .L26
        ret
.L26:
        add     ecx, 1
        mov     eax, ecx
        imul    eax, ecx
        cmp     edi, eax
        jle     .L28
        mov     eax, edi
        cdq
        idiv    ecx
        test    edx, edx
        jne     .L26
        xor     eax, eax
        ret
.L28:
        mov     eax, 1
        ret
```

```
is_prime_likely(int):
        cmp     edi, 4
        jle     .L7
        xor     eax, eax
        mov     ecx, 2
        test    dil, 1
        jne     .L5
        ret
.L5:
        add     ecx
        mov     e
        imul
        cmp
        jge     .L
        mov     eax,
        cdq
        idiv    ecx
.
        mov     eax, 1
        ret
```

```
is_prime_unlikely(int):
        cmp     edi, 4
        jle     .L28
        xor     eax, eax
        mov     ecx, 2
        test    dil, 1
        jne     .L26
        ret
        dd      ecx, 1
                eax, ecx
                eax, ecx
                edi, eax
.L28:
        mov     eax, 1
        ret
```

They're practically the same!

"Producing Wrong Data …" strikes again!

Same code, 10% timing difference!

GCC Bug 106716 - Identical Code Folding (-fipa-icf) confuses between functions with different [[likely]] attributes

Benchmark is_prime: attempt #3

likely.cpp

```
bool is_prime_likely() {...}

int main() {
    vector<int> numbers = random();
    benchmark(is_prime_likely);
}
```

unlikely.cpp

```
bool is_prime_unlikely() {...}

int main() {
    vector<int> numbers = random();
    benchmark(is_prime_unlikely);
}
```

Benchmark is_prime: attempt #3

likely.cpp

```
bool is_prime_likely() {...}

int main() {
    vector<int> numbers = random();
    benchmark(is_prime_likely);
}
```

Time: 6.021 sec ([[likely]])

unlikely.cpp

```
bool is_prime_unlikely() {...}

int main() {
    vector<int> numbers = random();
    benchmark(is_prime_unlikely);
}
```

Benchmark is_prime: attempt #3

likely.cpp

```
bool is_prime_likely() {...}

int main() {
    vector<int> numbers = random();
    benchmark(is_prime_likely);
}

Time: 6.021 sec ([[likely]])
```

unlikely.cpp

```
bool is_prime_unlikely() {...}

int main() {
    vector<int> numbers = random();
    benchmark(is_prime_unlikely);
}

Time: 6.608 sec ([[unlikely]])
```

Benchmark is_prime: attempt #3

likely.cpp

```
bool is_prime_likely() {...}

int main() {
    vector<int> numbers = random();
    ben
}
```

Time: 6.021 sec ([[likely]])

unlikely.cpp

```
bool is_prime_unlikely() {...}

int main() {
    vector<int> numbers = random();
    benchmark(is_prime_unlikely);
}
```

Time: 6.608 sec ([[unlikely]])

Opposite of our previous results!

[[likely]] faster by 10%

Always measure under real circumstances!

Benchmark is_prime: analysis

- What's causing the difference?
  - It must be the code layout!
- We can use `perf` tool
- We'll need a dedicated node

Benchmark is_prime: analysis

likely.cpp

```
          5  context-switches #    1.156 /sec
          0  cpu-migrations   #    0.000 /sec
      26291  page-faults      #    6.079 K/sec
14872265690  cycles           #    3.439 GHz
18405442107  instructions     #    1.24  insn
                                  per cycle
 3644663340  branches         # 842.723 M/sec
   15401643  branch-misses    #    0.42% of
                                  all branches

4.361551776  seconds time elapsed
4.329149000  seconds user
0.032254000  seconds sys
```

unlikely.cpp

```
          6  context-switches #    1.241 /sec
          0  cpu-migrations   #    0.000 /sec
      26288  page-faults      #    5.437 K/sec
16628990662  cycles           #    3.439 GHz
18422072401  instructions     #    1.11  insn
                                  per cycle
 3636038732  branches         # 752.001 M/sec
   15457991  branch-misses    #    0.43% of
                                  all branches

4.875721549  seconds time elapsed
4.835235000  seconds user
0.040281000  seconds sys
```

```
sudo perf stat ./main
```

Benchmark is_prime: analysis

likely.cpp

```
          5  context-switches #    1.156 /sec
          0  cpu-migrations   #      000 /sec
      26291  page-faults      #      079 K/sec
14872265690  cycles
18405442107  instructions

 3644663340  branches
   15401643  branch-misses

4.3
4.3
0.032254000  seconds sys
```

unlikely.cpp

```
          6  context-switches #    1.241 /sec
             cpu-migrations   #    0.000 /sec
       262   age-faults       #    5.437 K/sec
             les              #    3.439 GHz
             ctions           #    1.11  insn
                               #  752.001 M/sec
             -misses          #    0.43% of
                                   all branches

4.8757215   seconds time elapsed
4.83523500  seconds user
0.040281000  seconds sys
```

Practically the same

We’re not looking at code layout stats!

sudo perf stat ./main

Benchmark is_prime: I-Cache analysis

likely.cpp

```
   886516 L1-icache-load-misses
  4451553 L1-dcache-load-misses
          # 2.17% of all L1-dcache accesses
204730378 L1-dcache-loads

4.333807067 seconds time elapsed
4.285502000 seconds user
0.047974000 seconds sys
```

unlikely.cpp

```
   920408 L1-icache-load-misses
  4485583 L1-dcache-load-misses
          # 2.19% of all L1-dcache accesses
204882705 L1-dcache-loads

4.826222626 seconds time elapsed
4.777984000 seconds user
0.048020000 seconds sys
```

```
sudo perf stat -e L1-icache-load-misses,L1-dcache-load-misses,L1-dcache-loads ./main
```

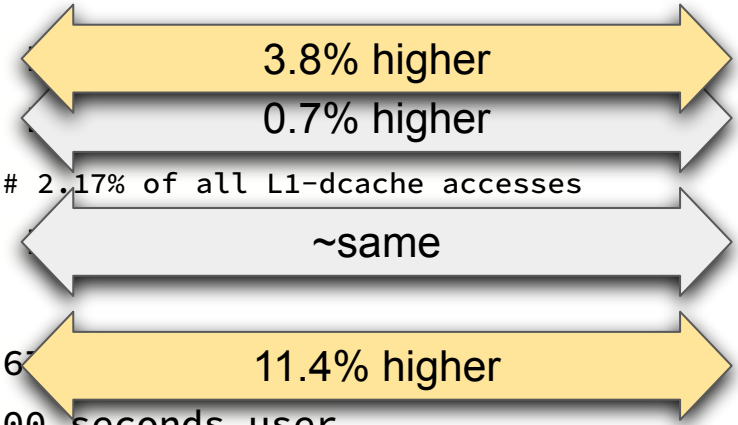
Benchmark is_prime: I-Cache analysis

likely.cpp

```
   886516      L1-icache-load-misses
  4451553      L1-dcache-load-misses
               # 2.17% of all L1-dcache accesses
204730378      L1-dcache-loads

4.33380706 seconds time elapsed
4.285502000 seconds user
0.047974000 seconds sys
```

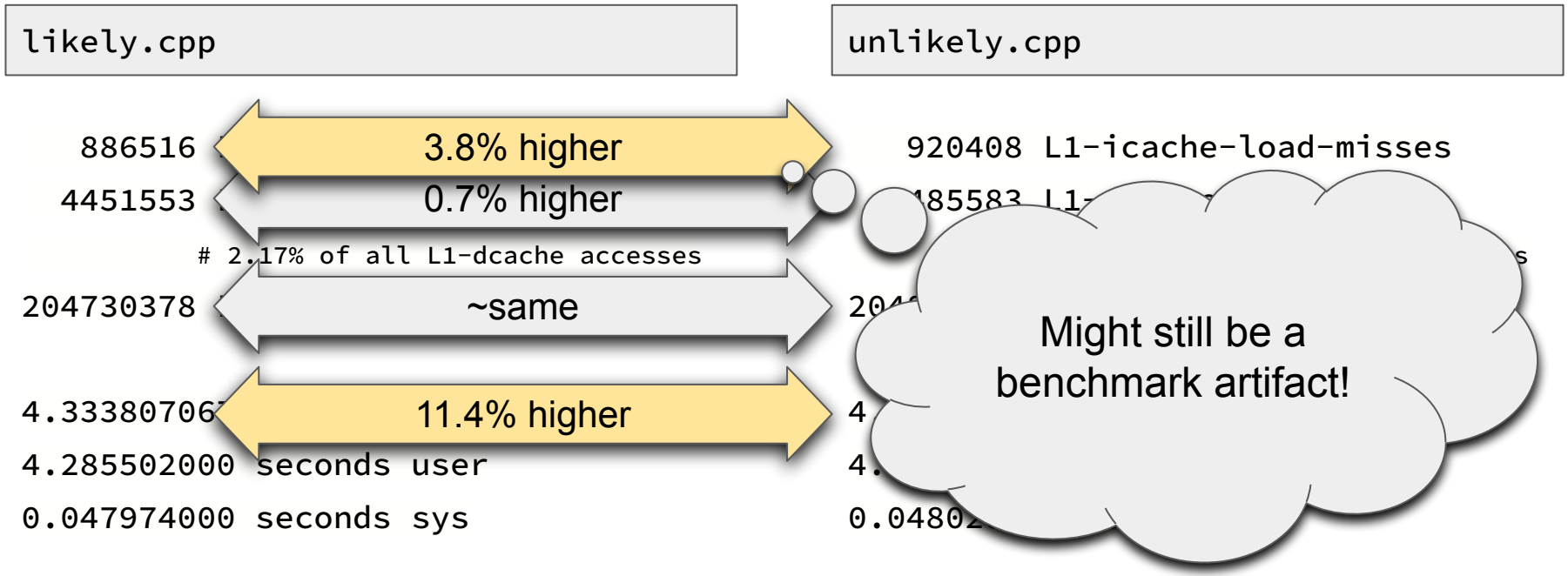
unlikely.cpp

```
   920408      L1-icache-load-misses
  4485583      L1-dcache-load-misses
               # 2.19% of all L1-dcache accesses
204882705      L1-dcache-loads

4.826222626 seconds time elapsed
4.777984000 seconds user
0.048020000 seconds sys
```




```
sudo perf stat -e L1-icache-load-misses,L1-dcache-load-misses,L1-dcache-loads ./main
```

Benchmark is_prime: I-Cache analysis




```
sudo perf stat -e L1-icache-load-misses,L1-dcache-load-misses,L1-dcache-loads ./main
```

SUMMARY

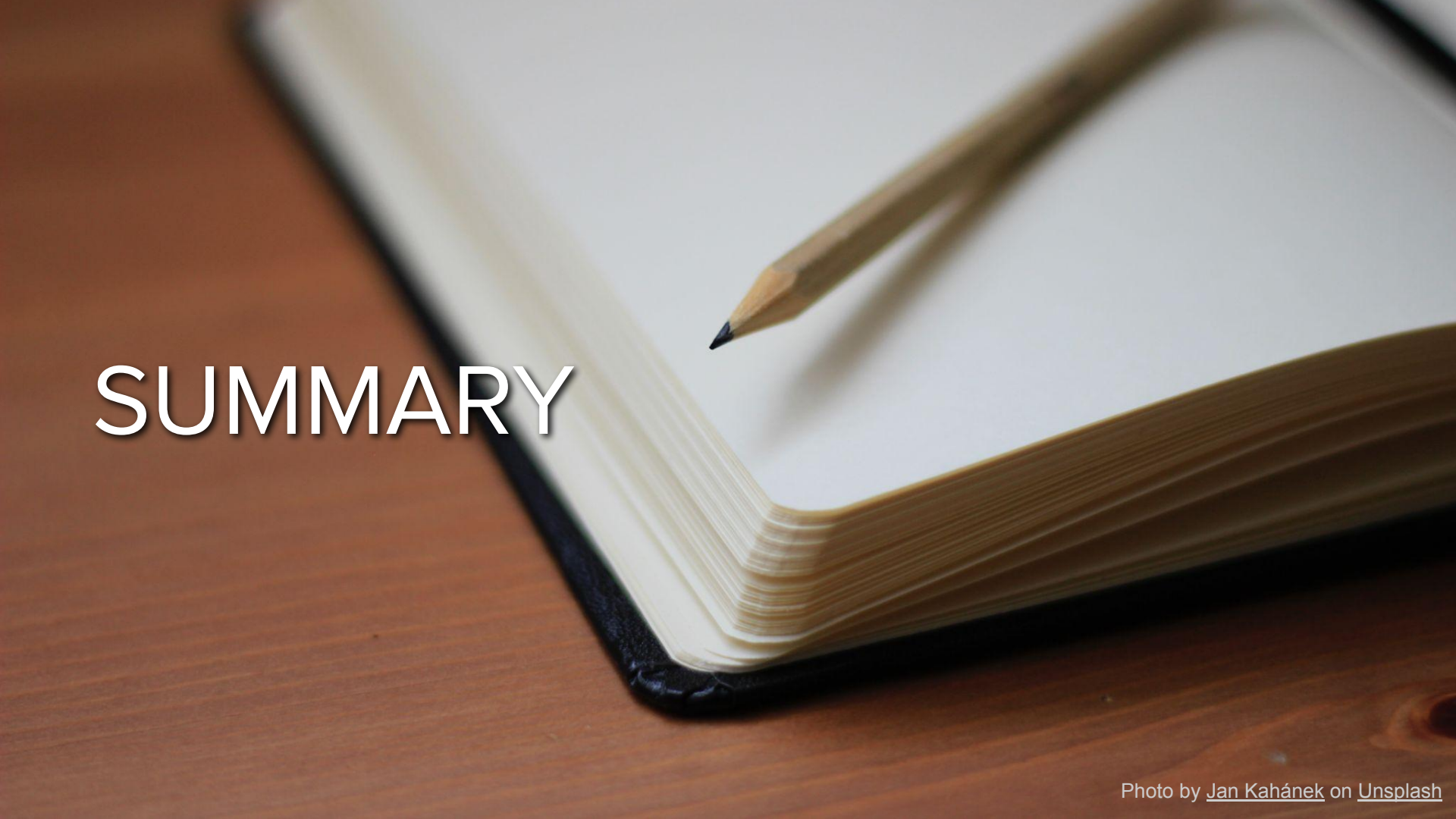


Photo by Jan Kahánek on Unsplash

Potential Benefits of Likely and Unlikely

Affects compiler branch predictor =>

May affect code layout =>

May improve Instructions Cache

However, it doesn’t actually affect CPU’s branch predictor! *
(Which is based on actual runtime input)

* may affect cold calls, but this shouldn’t have any significant effect on performance

Is it worth the hassle?

In most simple cases, the compiler branch predictor sees the same
⇒ thus no effect at all

In some cases it is just ignored (competes with other optimizations)

It is easy to have bad [[likely]] and [[unlikely]] as code changes
(attributes may remain in wrong place) - pessimization

See also: Don’t use the [[likely]] or [[unlikely]] attributes, by Aaron Ballman

Bottom Line

Be careful with what you ask for!
The same way it may help, it can also hurt performance.

Consider using [[likely]] and [[unlikely]] only in cases where the call ratio between the branches would be significantly different than what the compiler would assume

And only in cases where it should matter for performance (critical paths)

Take a look at the assembly and do your own benchmarks

Looking forward: Our suggestions

- Compilers:
  - gcc: Fix “identical” code folding bug (Bug 106716)
  - gcc: [[likely]]/[[unlikely]] should not completely override estimations
  - Clang: …
- Language: Maybe it should be a function expression?
  - Like the old built-in
  - Simpler syntax
  - Allows to specify things other than “code path reached”
- Language: Support for specific user % estimation
  - See `__builtin_expect_with_probability`
  - Allow automatic code updates from PGO?

Tomer Vromen
Tomer.Vromen@dell.com
tomerv@gmail.com

Photo by Howie R on Unsplash

Amir Kirsh
amir.kirsh@incredibuild.com
kirshamir@gmail.com

Questions?

From the original proposal…

Case	No expect	Opposite expect	Regular expect
pugixml	126.0936 ms	128.6235 ms (+2.5299)	122.7716 ms (-3.322)
picohttpparser -march=haswell	1.7857 s	1.7763 s (-0.0094)	1.7119 s (-0.0738)
picohttpparser	3.5989 s	3.7150 s (+0.1161)	3.0369 s (-0.562)