Expecting the expected

Honoring user branch hints for code placement optimizations

Zhi Zhuang, Stan Kvasov, Vince Del Vecchio

Simple example

```
void foo(int cond) {
  if (cond)
    debug();
}
```

- Let’s assume debug() is **cold** code
- Default block order is poor

```
  beqz  a0, .LBB0_2
  tail  debug
.LBB0_2:
  ret
```

RiscV used because behavior shows through to asm. On x86 and many others, LLVM IR optimizer behaves the same but ISel/back end transforms hide some issues in our simple examples.

Simple example with __builtin_expect

```
void foo(int cond) {
  if (cond)
    debug();
}
```

- Let's assume debug() is **cold** code
- Default block order is poor

```
  beqz  a0, .LBB0_2
  tail  debug
.LBB0_2:
  ret
```

RiscV used because behavior shows through to asm. On x86 and many others, LLVM IR optimizer behaves the same but ISel/back end transforms hide some issues in our simple examples.

```
void foo_expect(int cond) {
  if (__builtin_expect(cond, 0))
    debug();
}
```

- builtin_expect says: cond is likely 0
  - Changes branch weights in IR

```
  bnez  a0, .LBB0_2
  ret
.LBB0_2:
  tail  debug
```

Hot path is straight

Branch weights are functionally equivalent to probabilities.
We often discuss probabilities, which are easier to reason about.

MEDIATEK

More complicated example

```
void foo_expect_prob(char ai, char bi, char ci) {
  char a = 2 * ai, b = 2 * bi, c = 2 * ci;
  if (__builtin_expect_with_probability(((a == 2) || (b == 2) || (c == 0)),
                                         1, 0))
    debug();
}
```

- __builtin_expect_with_probability(x, 1, 0) says: x is ~0% likely to be 1
  - Our team implemented this builtin in LLVM in 2020, matching gcc (released in LLVM 11)
- Useful if:
  - Profile data might be hard to collect
  - Want more nuance than default 0.05% probability (10% / 25% / 0.0000001% likely)
  - We use this to control outlining – place debug() in cold / slower memory

More complicated example – unexpected!

```
void foo_expect_prob(char ai, char bi, char ci) {
  char a = 2 * ai, b = 2 * bi, c = 2 * ci;
  if (__builtin_expect_with_probability(((a == 2) || (b == 2) || (c == 0)),
                                          1, 0))
    debug();
}
```

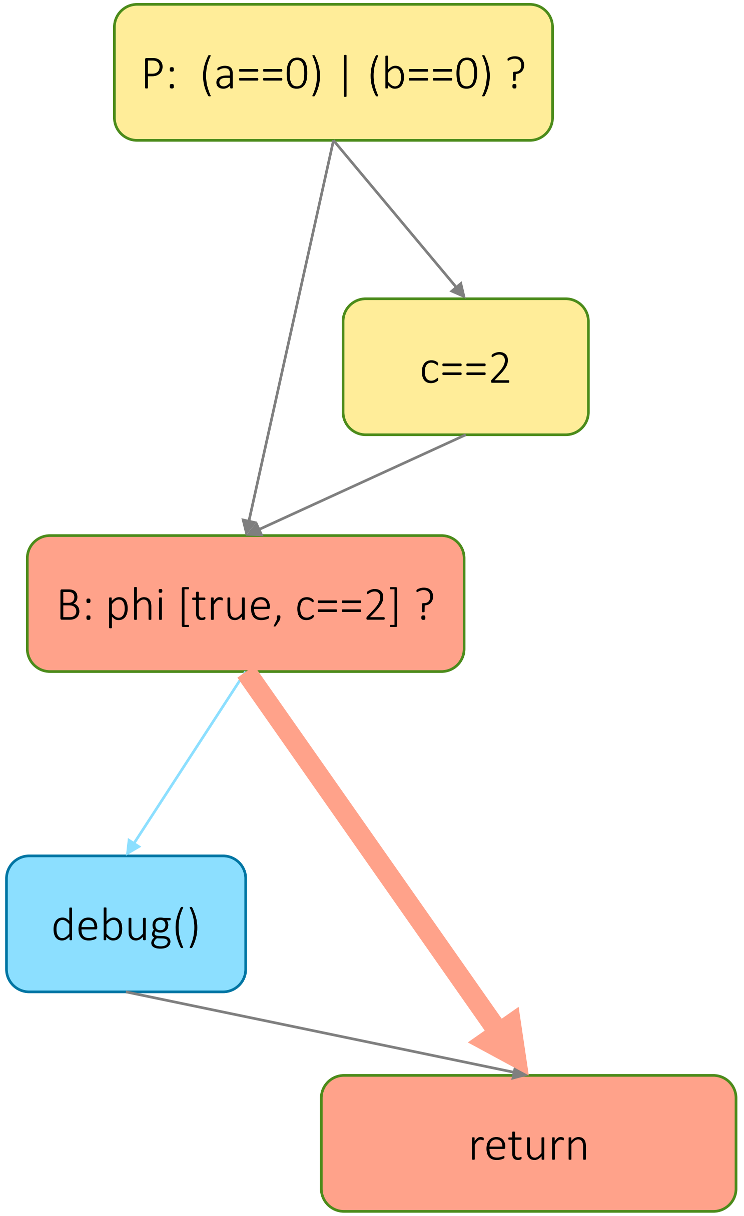
```
  andi  a0, a0, 127         andi  a1, a2, 127
  addi  a0, a0, -1          snez  a1, a1
  snez  a0, a0              and   a0, a0, a1
  andi  a1, a1, 127
  addi  a1, a1, -1          bnez  a0, .LBB0_2
  snez  a1, a1              tail  debug
  and   a0, a0, a1        .LBB0_2:
                            ret
```

No longer falls through to return!

MEDIATEK

What happened? SimplifyCFG

- We compute
  `%cond = (a == 0) || (b == 0) || (c == 2)`
  using control flow
- Then we branch on %cond in B
- SimplifyCFG removes control flow, makes
  `(a == 0) | (b == 0) | (c == 2)`
  - Halfway done in this picture



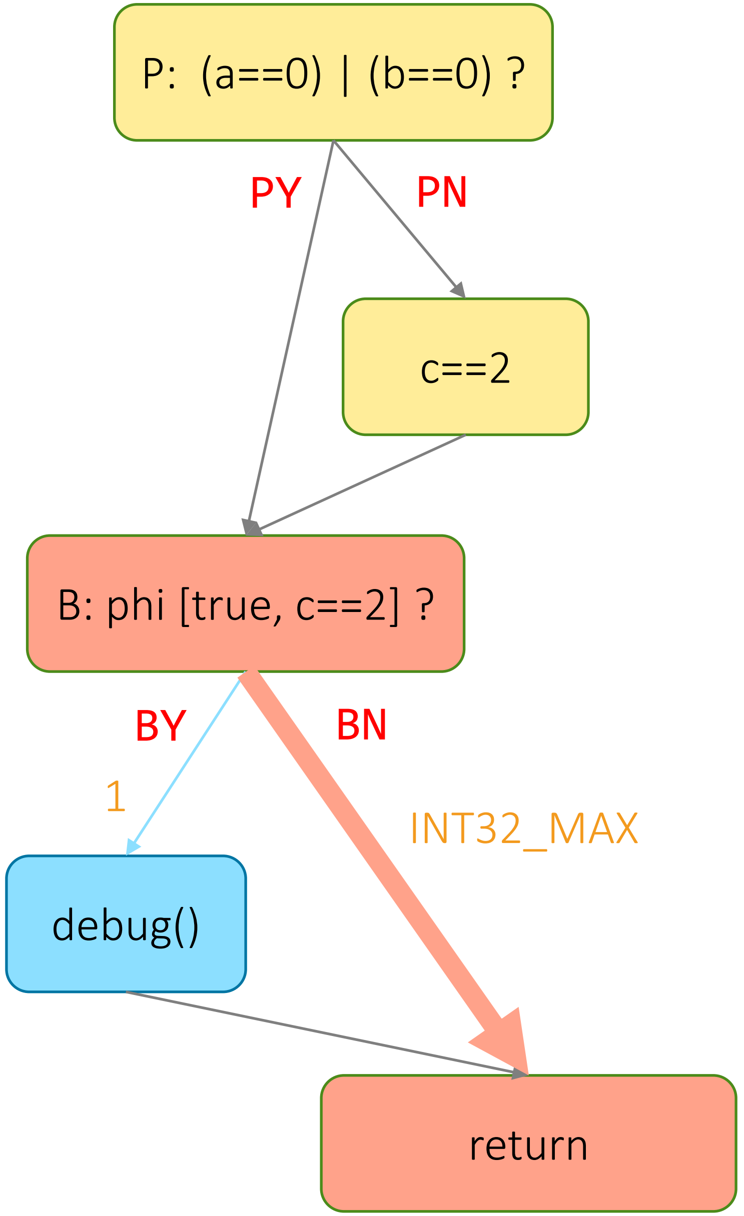
MEDIATEK

What happened? SimplifyCFG

- We compute
  `%cond = (a == 0) || (b == 0) || (c == 2)`
  using control flow
- Then we branch on %cond in B
- SimplifyCFG removes control flow, makes
  `(a == 0) | (b == 0) | (c == 2)`
  - Halfway done in this picture

- B has branch weights (from builtin)
- P has none, implicitly equal (50%)

- Since PY implies BY,
  SimplifyCFG threads those branches

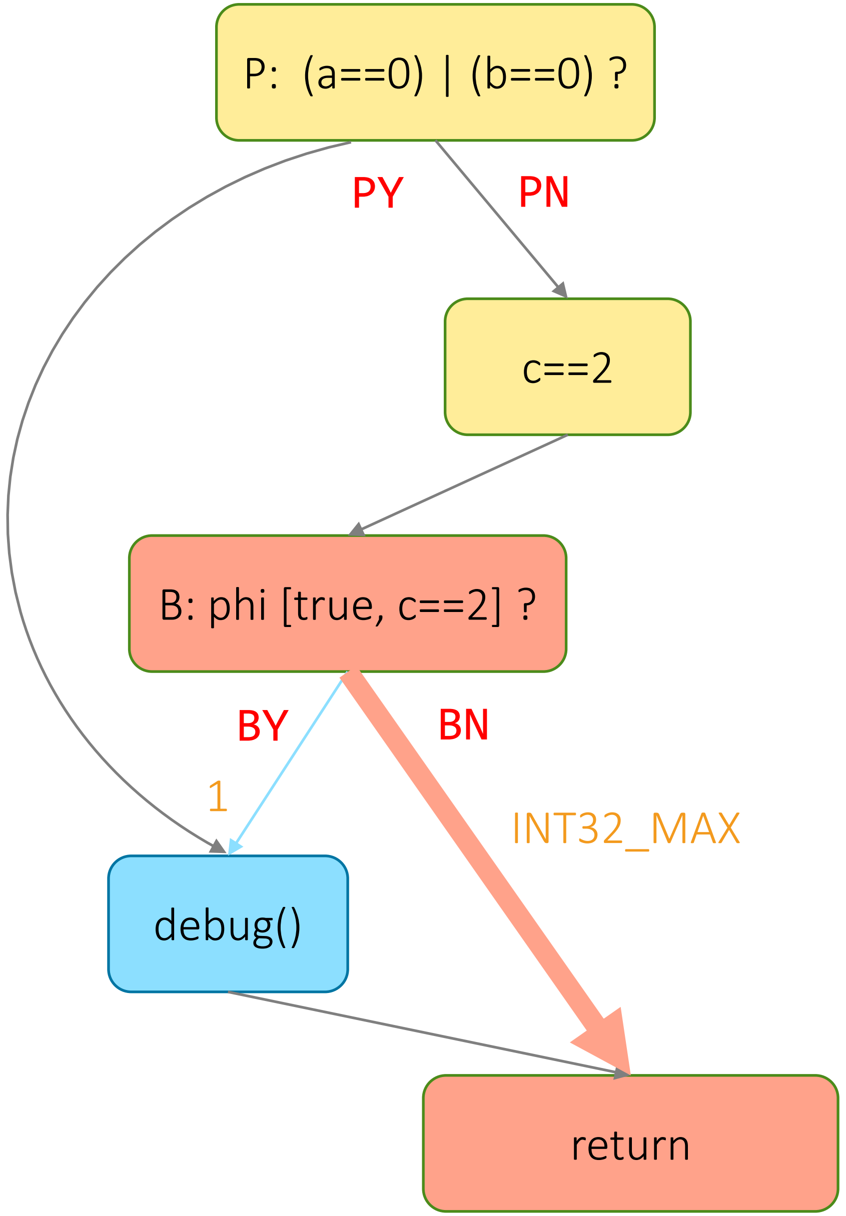


What happened? SimplifyCFG

- We compute
  `%cond = (a == 0) || (b == 0) || (c == 2)`
  using control flow
- Then we branch on %cond in B
- SimplifyCFG removes control flow, makes
  `(a == 0) | (b == 0) | (c == 2)`
  – Halfway done in this picture

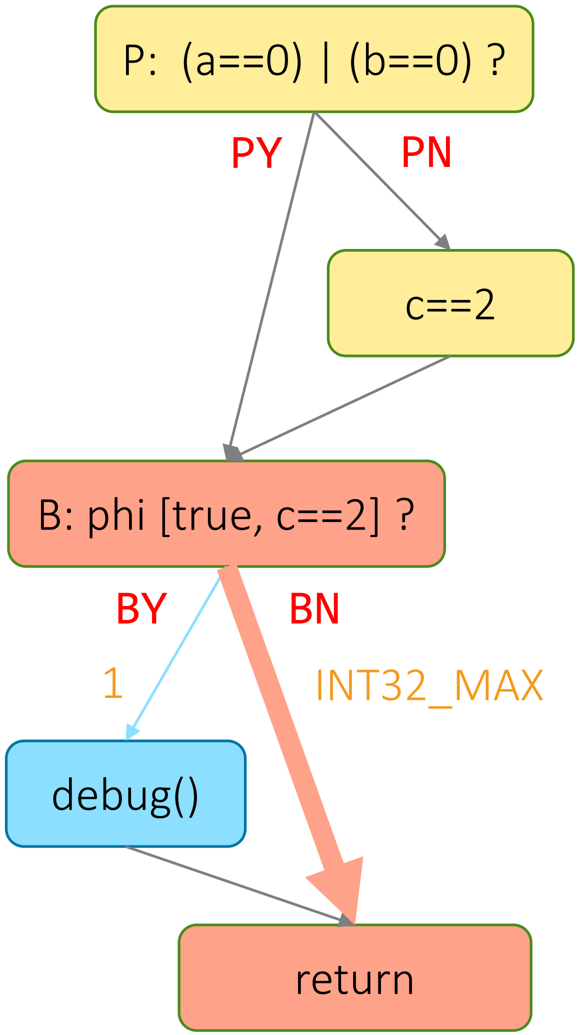
- B has branch weights (from builtin)
- P has none, implicitly equal (50%)

- Since PY implies BY,
  SimplifyCFG threads those branches



MEDIATEK

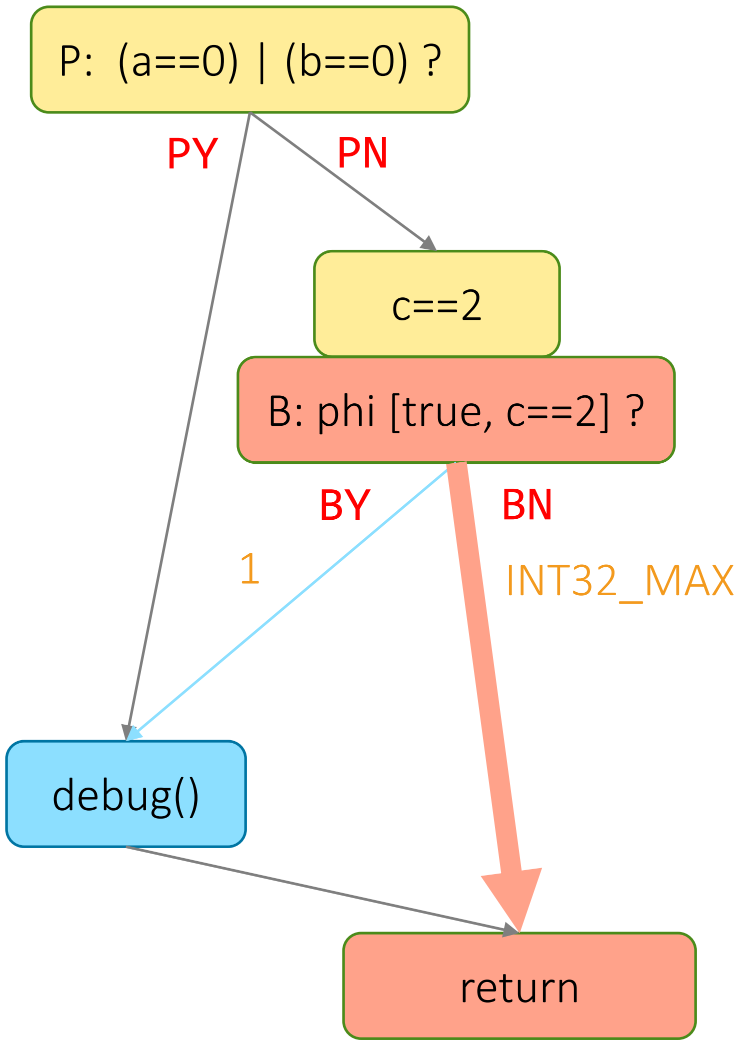
SimplifyCFG not meeting expectations



Before

- Before: ~0 chance to call (PY)
- After: ~50% chance to call: P(PY) + P(PN) * P(BY)

- Before: internally inconsistent
  - PY (50%) always goes to BY (0%)
- If P(BY) is really ~0, P(PY) must be ~0 or less



After

MEDIATEK

When can this happen? Any incomplete branch weight data

Using __builtin_expect_with_probability

```
void foo_expect_prob(char ai, char bi, char ci) {
  char a = 2 * ai, b = 2 * bi, c = 2 * ci;
  if (__builtin_expect_with_probability(
        ((a == 2) || (b == 2) || (c == 0)), 1, 0))
    debug();
}
```

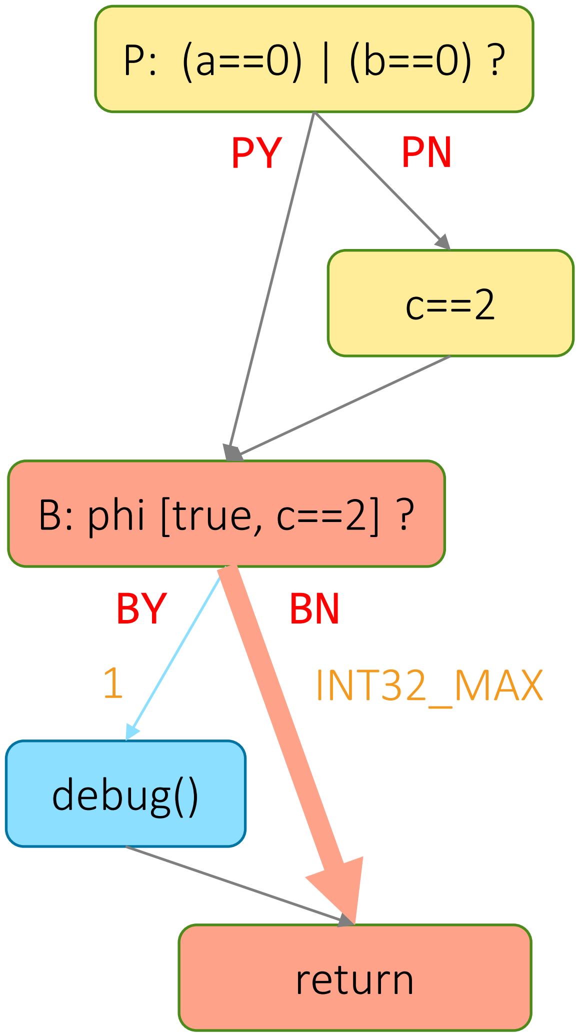
Using __builtin_expect

```
int cmp(int *a, int *b) {
  return *a == 2 || *b == 2;
}

void foo_expect(int a, int b) {
  if (__builtin_expect(cmp(&a, &b), 0))
    debug();
}
```

Using sample-based profiling

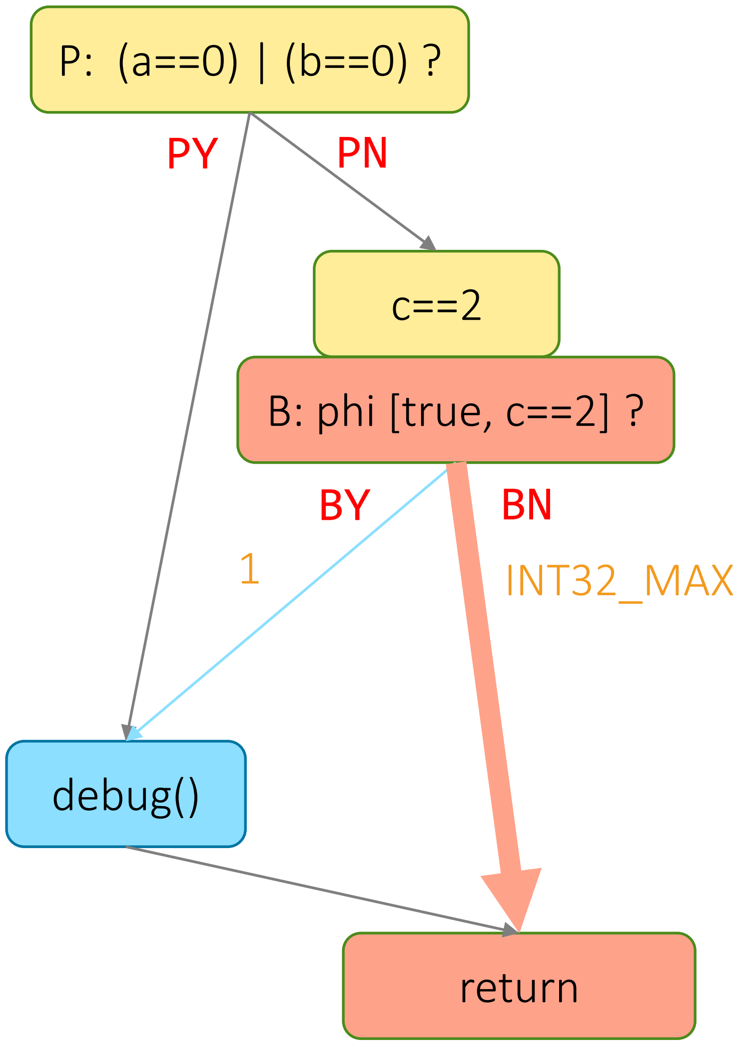
```
volatile int G;
__attribute__((noinline))
void func(int a, int b, int c) {
  if (a || b || c) {
    int i;
    for (i = 0; i < 1000; i++)
      G++;
  }
}

int main() {
  for (int j = 0; j < 10000; j++) {
    int i;
    for (i = 0; i < 100; i++) {
      func(i, 2*i, 3*i);
    }
  }
  return 0;
}
```

A local solution



Before

- “Backward propagation” of weights from B to P
  - Like JumpThreading, sometimes
  - Like LowerExpectIntrinsic, for some cases
- When P(PY) > P(BY), adjust P(PY)
  - Incomplete data - must guess!
  - (a) Set P(PY) = P(BY)
    - Ignores PN->BY
  - (b) Set P (PY) = P(PY) * P(BY)
  - (c) Other?



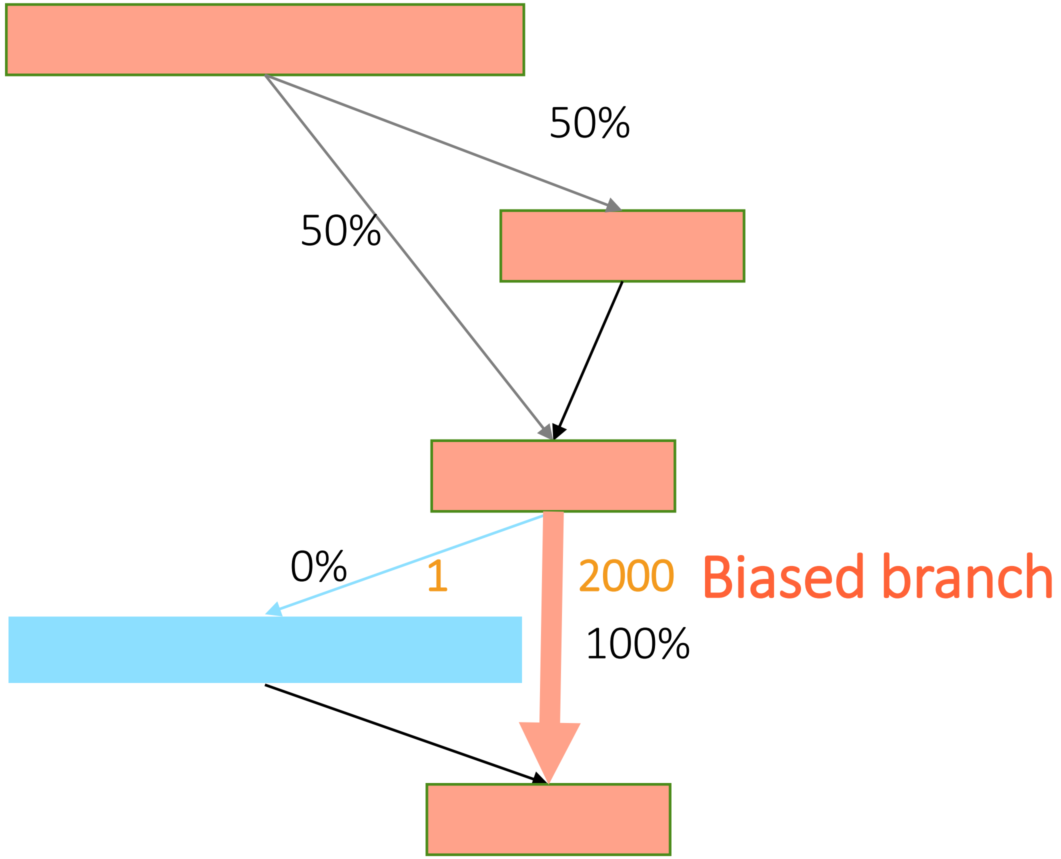
After

What else?

- We wondered if there were other such cases
- Created a tool Expectify to find out

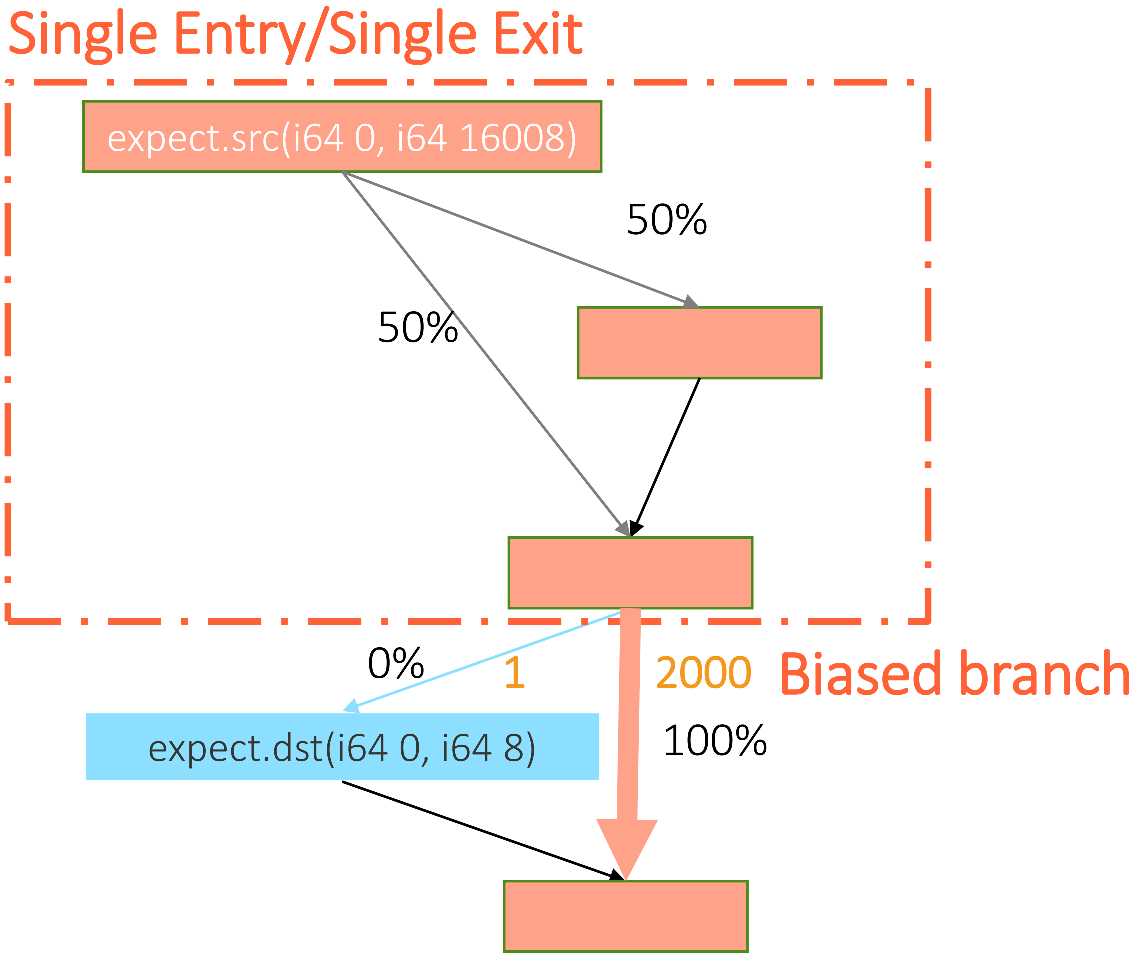
Expectify: Identify passes which skew branch probabilities

1. Selectively marks branches as unlikely



Expectify: Identify passes which skew branch probabilities

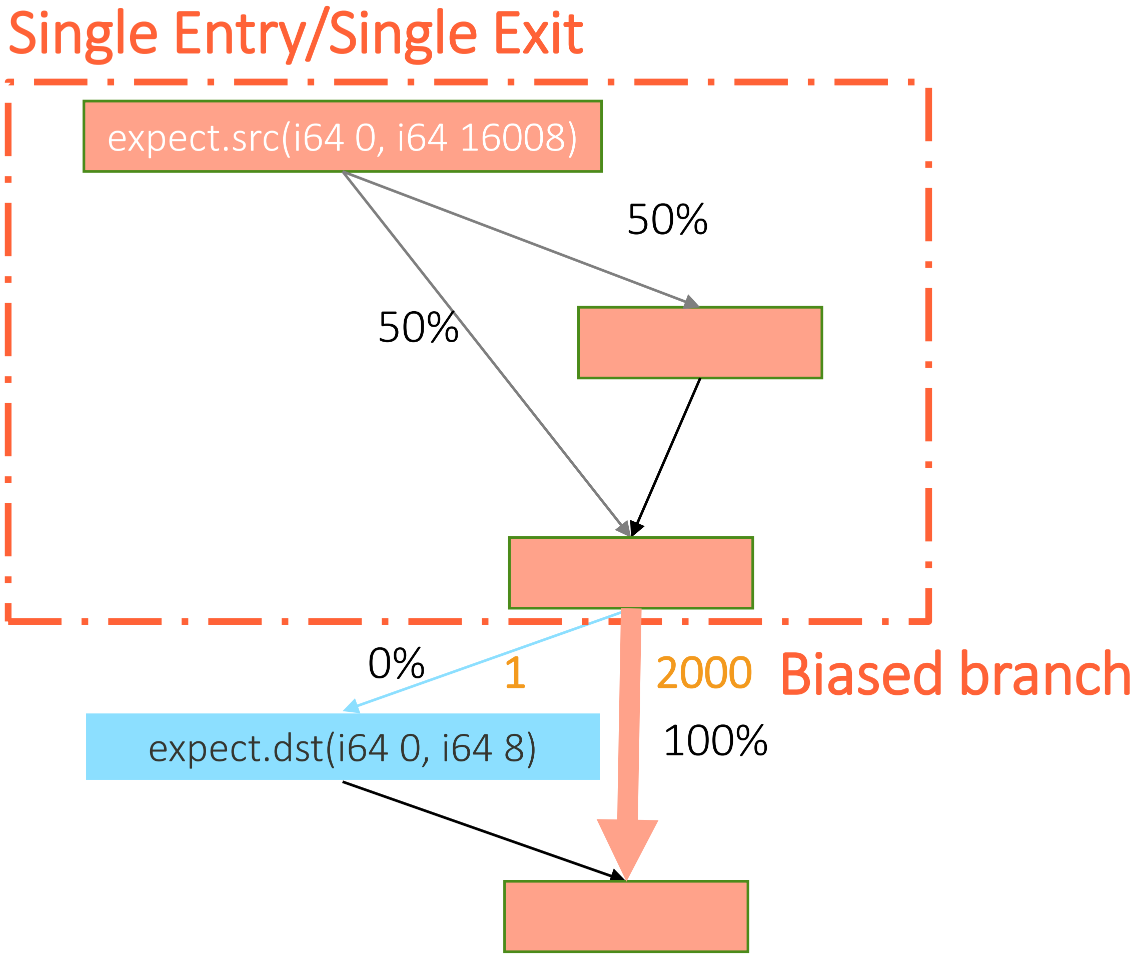
1. Selectively marks branches as unlikely
2. Instruments IR with:
   - expect.src(ID, SrcBlockFreq)
   - expect.dst(ID, DstBlockFreq)
   - Freqs found using BFI (BlockFrequencyInfo)
   - Pinit(src->dst) = DstBlockFreq/SrcBlockFreq



Pinit = 8/16008 = 0.05%

Expectify: Identify passes which skew branch probabilities

1. Selectively marks branches as unlikely
2. Instruments IR with:
   - expect.src(ID, SrcBlockFreq)
   - expect.dst(ID, DstBlockFreq)
   - Freqs found using BFI (BlockFrequencyInfo)
   - Pinit(src->dst) = DstBlockFreq/SrcBlockFreq
3. Compare Pn from later passes to Pinit
   - **Pn > 0.5 (likelihood inversion)**
   - If Pn/Pinit > N (probability skewing)

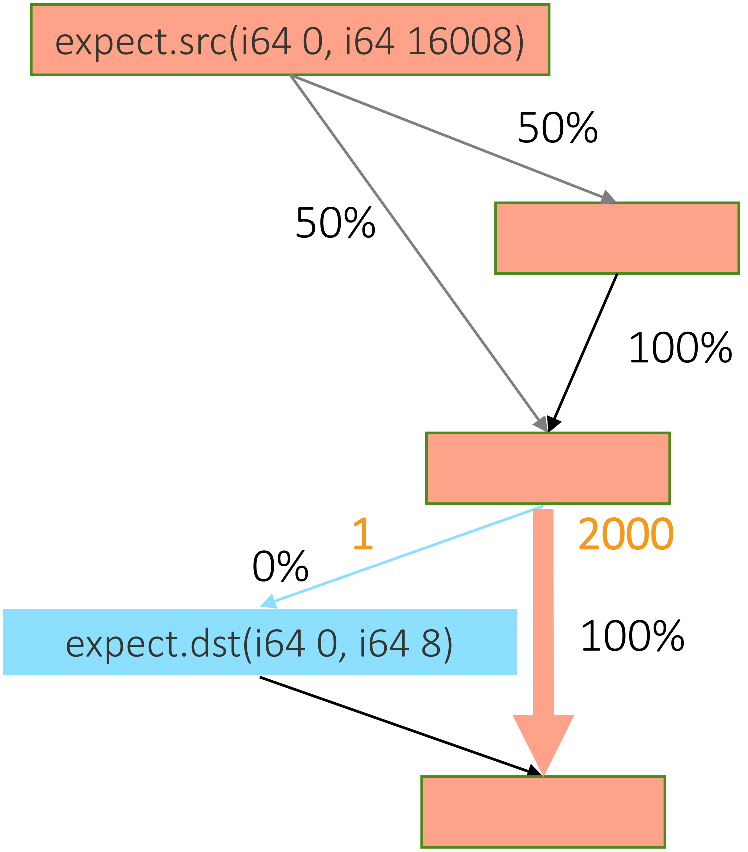


Pinit = 8/16008 = 0.05%
Pn = 8/16008 = 0.05%
Pn/Pinit = 1 ✓

MEDIATEK

Expectify: working on our original example

Before SimplifyCFG

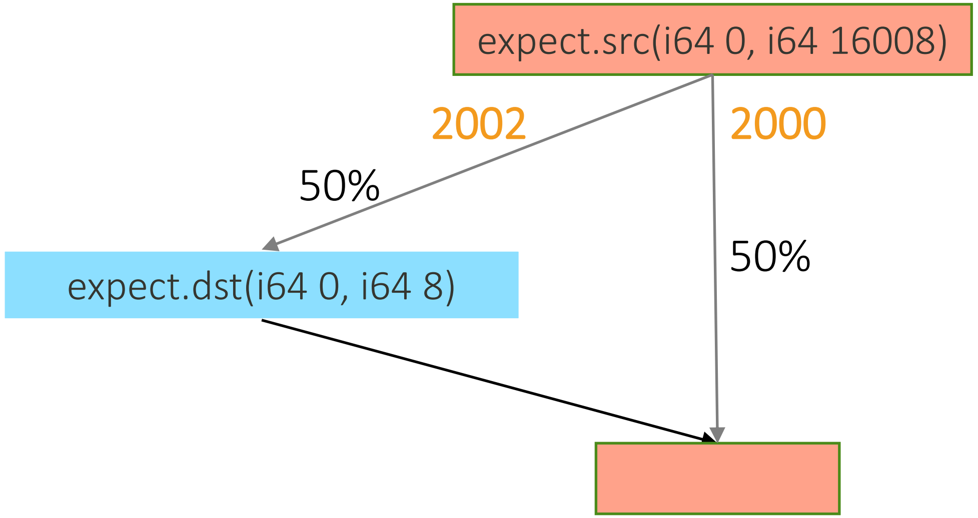


Pinit = 8/16008 = 0.05%

Pn = 8/16008 = 0.05%

Pn/Pinit = 1 ✓

After SimplifyCFG



Pinit = 8/16008 = 0.05%

Pn = 2002/4002 > 50%

Pn/Pinit = 1001 > 100 ✗

Expectify: Work In Progress

- Used Csmith + Expectify to identify passes which invert branch likelihood:
  - JumpThreading
  - SimplifyCFG
- Common symptoms:
  - Thread edges over blocks with branch weights
  - JumpThreading: Backward propagates weights for limited set of cases
  - SimplifyCFG: Makes no attempt to backward propagate weights
- Work in progress:
  - Common/refactor backward propagation in both passes
  - Iterate with Expectify to find new bugs
  - Reduced to 0 failures in a small batch of Csmith tests (100)

	% failing Csmith tests w/ Expectify
Initial	6%
Current	0%

Conclusions

- Currently __builtin_expect etc can give unexpected code placement
- LLVM can throw away valid branch weights
  – In both SimplifyCFG / JumpThreading when threading a jump
  – But behavior is also not consistent within/between those passes!
- Expectify: a new tool to find these issues so they can be fixed
  – We are working both on standardizing code behavior and reusing code between passes
  – We'll post fixes & Expectify after more iterations

MEDIATEK

everyday genius

We’re hiring!

See us or email vince.delvecchio@mediatek.com or stan.kvasov@mediatek.com for info!