Deferring branches to speed up symbolic execution

Eric Lu, Eddie Kohler

Motivating observation

- Optimized code patterns can slow down symbolic execution!
- Can we **undo** those optimizations in the symbolic executor to improve its performance?
- Example: **hash table lookup**

Hash table example

- Chained hash table containing *concrete* values
- `find_key` is used in lookup:
```
uint32_t h = hash(key);
...
find_key(table->bucket[h % N], h, key);
```
- In normal execution, `l->hash == hash` is fast
- But suppose key is a *symbolic* string. What happens?

```
typedef struct node {
  uint32_t hash;
  uint8_t *key;
  struct node *next;
} node;

node *find_key(node *l, uint32_t h,
               uint8_t *key) {
  while (l) {
    if (l->hash == h) {
      if (strcmp(l->key, key) == 0) {
        return l;
      }
    }
    l = l->next;
  }
  return NULL;
}
```

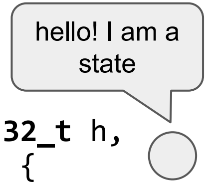
Hash table example

- Chained hash table containing *concrete* values
- `find_key` is used in lookup:

```
uint32_t h = hash(key);
...
find_key(table->bucket[h % N], h, key);
```

- In normal execution, `l->hash == hash` is fast
- But suppose key is a *symbolic* string. What happens?

```
typedef struct node {
  uint32_t hash;
  uint8_t *key;
  struct node *next;
} node;

node *find_key(node *l, uint32_t h,
               uint8_t *key) {
  while (l) {
    if (l->hash == h) {
      if (strcmp(l->key, key) == 0) {
        return l;
      }
    }
    l = l->next;
  }
  return NULL;
}
```



Hash table example

- Chained hash table containing *concrete* values
- `find_key` is used in lookup:

```
uint32_t h = hash(key);
...
find_key(table->bucket[h % N], h, key);
```

- In normal execution, `l->hash == hash` is fast
- But suppose key is a *symbolic* string. What happens?

```
typedef struct node {
  uint32_t hash;
  uint8_t *key;
  struct node *next;
} node;

node *find_key(node *l, uint32_t h,
                uint8_t *key) {
  while (l) {
    if (l->hash == h) {
      if (strcmp(l->key, key) == 0) {
        return l;
      }
    }
    l = l->next;
  }
  return NULL;
}
```

h is symbolic!

Hash table example

- Chained hash table containing *concrete* values
- `find_key` is used in lookup:

```
uint32_t h = hash(key);
...
find_key(table->bucket[h % N], h, key);
```

- In normal execution, `l->hash == hash` is fast
- But suppose key is a *symbolic* string. What happens?

```
typedef struct node {
  uint32_t hash;
  uint8_t *key;
  struct node *next;
} node;

node *find_key(node *l, uint32_t h,
                     uint8_t *key) {
  while (l) {
    if (l->hash == h) {
      if (strcmp(l->key, key) == 0) {
        return l;
      }
    }
    l = l->next;
  }
  return NULL;
}
```

an expensive fork! must find hash preimage

`l->hash == h`

`l->hash != h`

Hash table example

- Chained hash table containing *concrete* values
- `find_key` is used in lookup:

```
uint32_t h = hash(key);
...
find_key(table->bucket[h % N], h, key);
```

- In normal execution, `l->hash == hash` is fast
- But suppose key is a *symbolic* string. What happens?

```
typedef struct node {
  uint32_t hash;
  uint8_t *key;
  struct node *next;
} node;

node *find_key(node *l, uint32_t h,
                 uint8_t *key) {
  while (l) {
    if (l->hash == h) {
      if (strcmp(l->key, key) == 0) {
        return l;
      }
    }
    l = l->next;
  }
  return NULL;
}
```

an expensive fork! must find hash preimage

more forking in strcmp (or on a symbolic strcmp return value)

`l->hash == h ∧ keys match`

`l->hash != h`

`l->hash == h ∧ keys don’t match`

many states reach end of loop

Hash table example

- Chained hash table containing *concrete* values

- `find_key` is used in lookup:
```
uint32_t h = hash(key);
...
find_key(table->bucket[h % N], h, key);
```

- In normal execution, `l->hash == hash` is fast

- But suppose key is a *symbolic* string. What happens?

```
typedef struct node {
  uint32_t hash;
  uint8_t *key;
  struct node *next;
} node;

node *find_key(node *l, uint32_t h,
                  uint8_t *key) {
  while (l) {
    if (l->hash == h) {
      if (strcmp(l->key, key) == 0) {
        return l;
      }
    }
    l = l->next;
  }
  return NULL;
}
```

l->hash != h

l->hash == h ∧ keys don't match

l->hash == h ∧ keys match

states continue in next loop iteration

Hash table example

- Chained hash table containing *concrete* values
- `find_key` is used in lookup:
  ```
  uint32_t h = hash(key);
  ...
  find_key(table->bucket[h % N], h, key);
  ```
- In normal execution, `l->hash == hash` is fast
- But suppose key is a *symbolic* string. What happens?
- **How do we undo the optimization in this case?**

```
typedef struct node {
  uint32_t hash;
  uint8_t *key;
  struct node *next;
} node;

node *find_key(node *l, uint32_t h,
               uint8_t *key) {
  while (l) {
    if (l->hash == h) {
      if (strcmp(l->key, key) == 0) {
        return l;
      }
    }
    l = l->next;
  }
  return NULL;
}
```

Undoing the optimization

- We can **defer** the hash equality check until execution reaches the next condition
- Turn the **short-circuit &&** into **&**
- Avoid an expensive solver call and eliminate one of the generated states

```
typedef struct node {
  uint32_t hash;
  uint8_t *key;
  struct node *next;
} node;

node *find_key(node *l, uint32_t h,
                 uint8_t *key) {
  while (l) {
    if ((l->hash == hash) &
        (strcmp(l->key, key) == 0)) {
      return l;
    }
    l = l->next;
  }
  return NULL;
}
```

Undoing the optimization

- We can **defer** the hash equality check until execution reaches the next condition
- Turn the **short-circuit &&** into **&**
- Avoid an expensive solver call and eliminate one of the generated states

```
typedef struct node {
  uint32_t hash;
  uint8_t *key;
  struct node *next;
} node;

node *find_key(node *l, uint32_t h,
                         uint8_t *key) {
  while (l) {
    if ((l->hash == hash) &
        (strcmp(l->key, key) == 0)) {
      return l;
    }
    l = l->next;
  }
  return NULL;
}
```

l->hash == h ∧ keys match

l->hash != hash ∨ keys don’t match

Undoing the optimization

- With length-1 `l` and length-8 `key`
- Custom `eq`

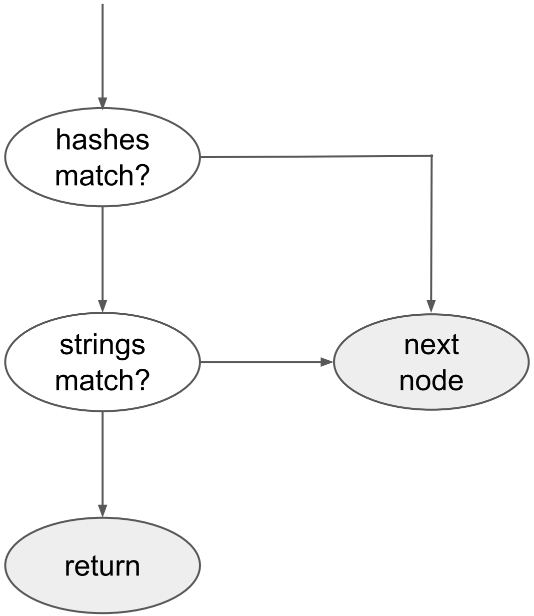
```
int eq(uint8_t *s1, uint8_t *s2) {
  return *((uint64_t *) s1) ==
         *((uint64_t *) s2);
}
```

- Version with **&&**:
  timeout after 1 hour (trying to solve the hash preimage)
- Version with **&**:
  finishes in 47 ms with 2 paths explored
- Version with a **simpler hash** (XOR all characters):
  finishes in 42 ms with 3 paths explored

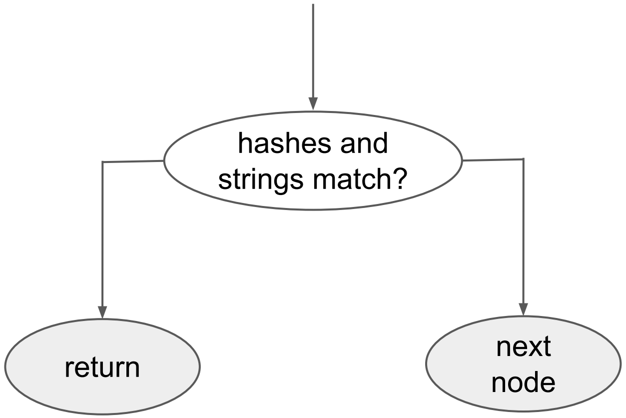
```
typedef struct node {
  uint32_t hash;
  uint8_t *key;
  struct node *next;
} node;

node *find_key(node *l, uint32_t h,
                    uint8_t *key) {
  while (l) {
    if ((l->hash == hash) &
        eq(l->key, key)) {
      return l;
    }
    l = l->next;
  }
  return NULL;
}
```

The plan: run a different CFG

Treat this...

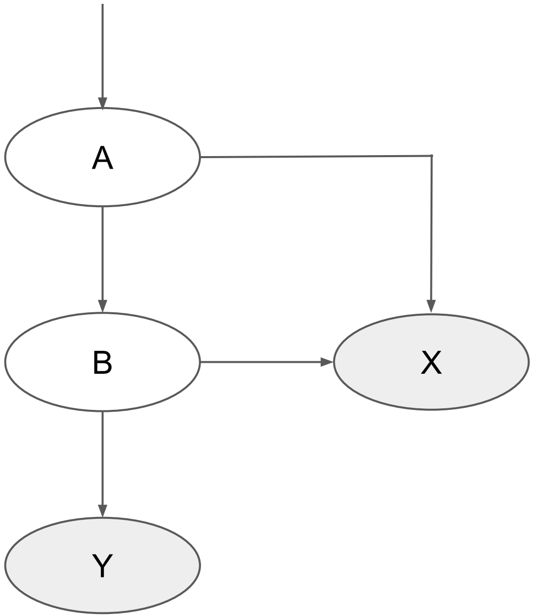


Like this!

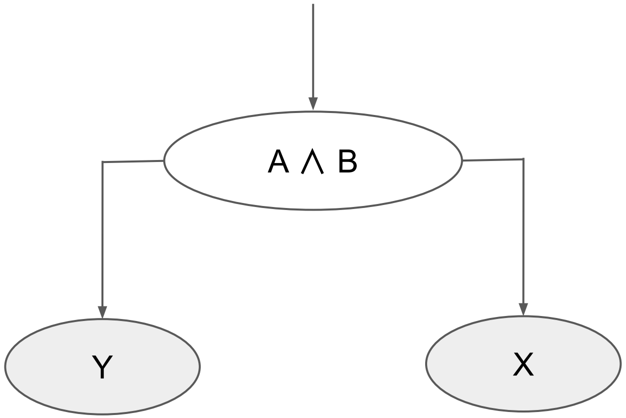


The plan: run a different CFG

Treat this...

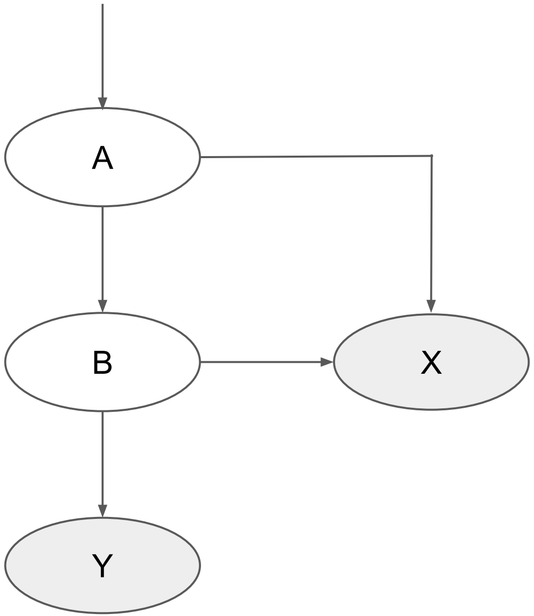


Like this!

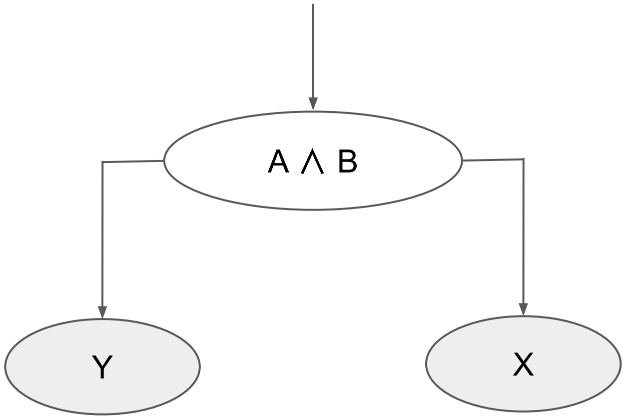


The plan: run a different CFG

Treat this...



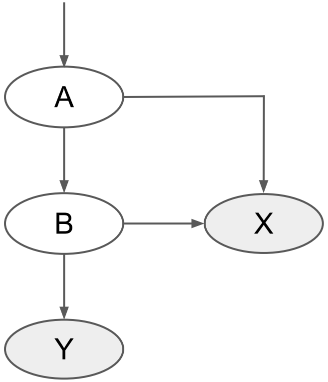
Like this!

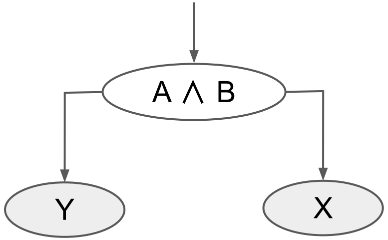


1. Turn two branches into one: fork less
2. Tradeoff: larger queries for fewer paths

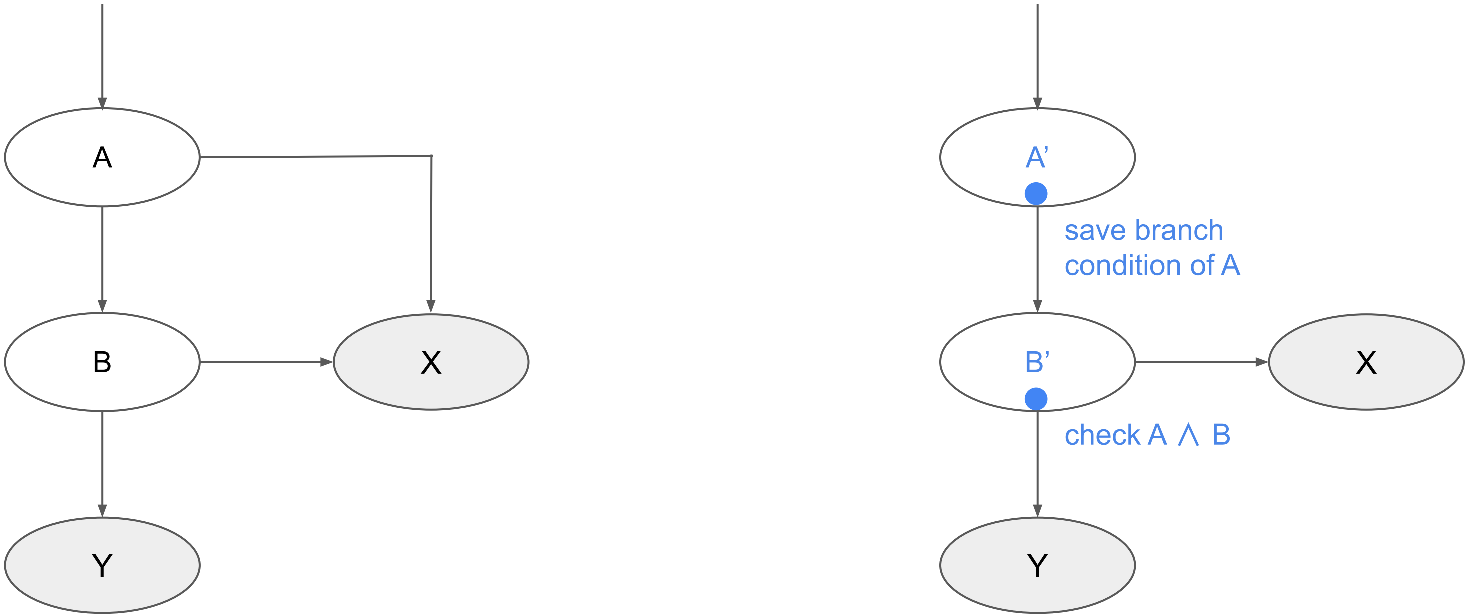
The plan: run a different CFG

- Compile time
  - Identify A && B pattern heuristically
  - Transform to execute as A ∧ B pattern
  - This transformation preserves semantics only when B doesn't modify observable state and can't cause an error (e.g., null pointer dereference)
  - Difficult to prove absence of errors statically, so rely on run time checks
- Run time
  - New **intrinsics** mark start and end of transformed A ∧ B pattern
  - On error in transformed region, check against original A && B pattern before reporting



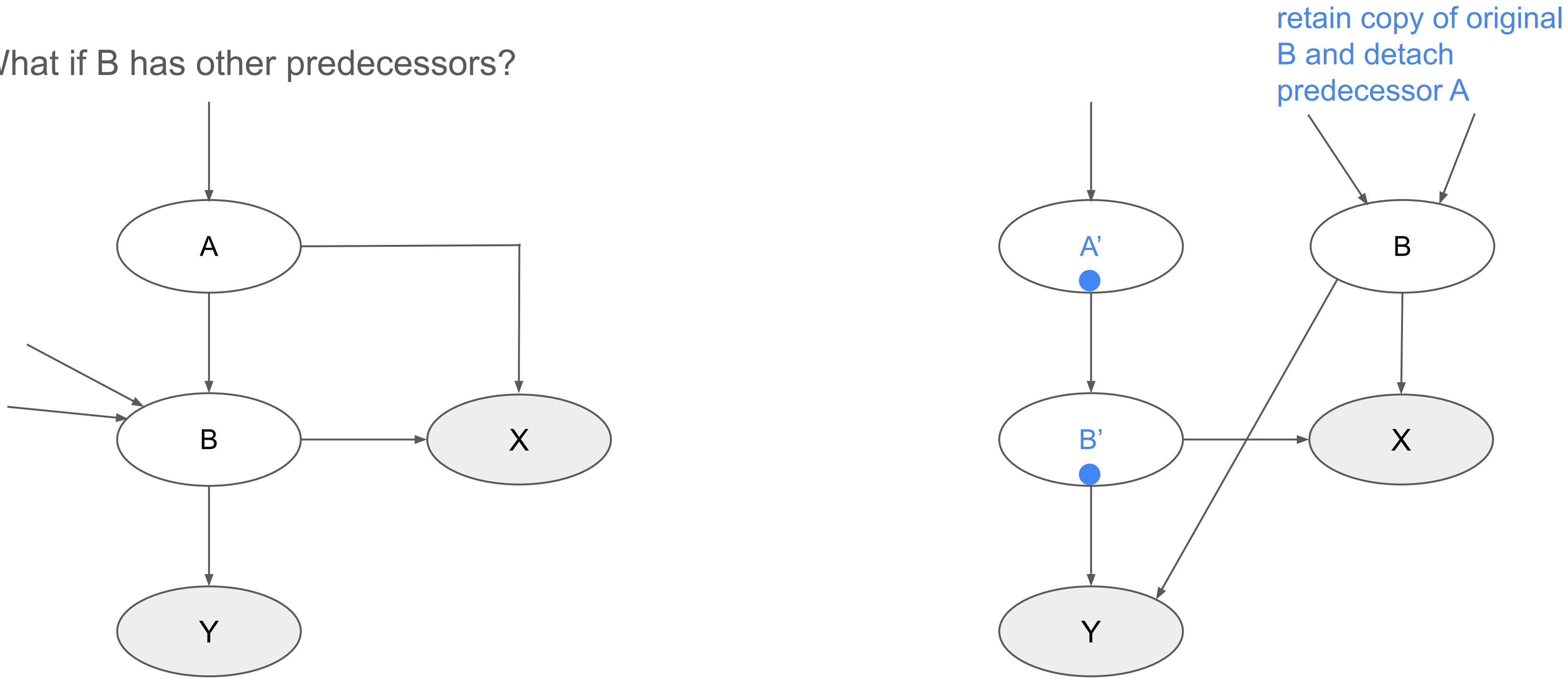


Compile time: transforming short-circuit CFG fragments



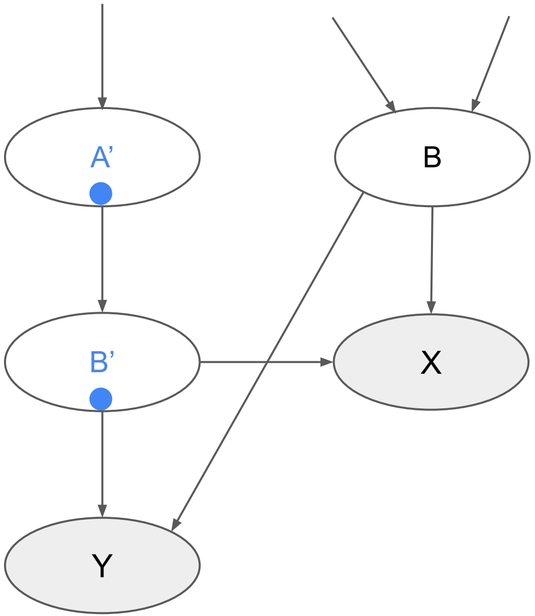
Compile time: transforming short-circuit CFG fragments

- What if B has other predecessors?



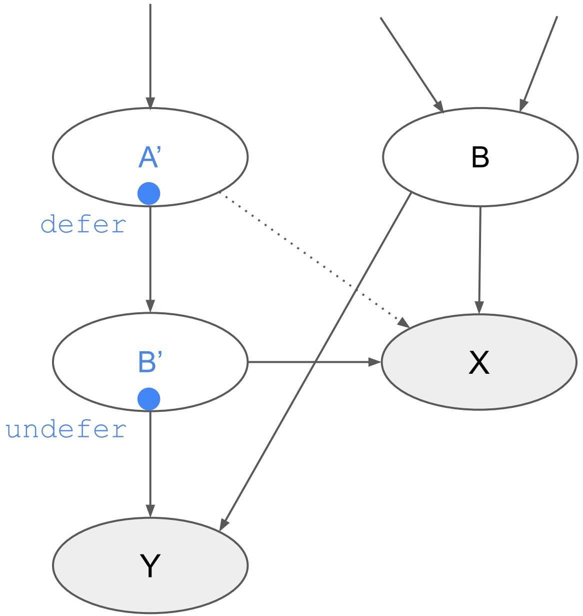
Compile time: transforming short-circuit CFG fragments

- What if we encounter an error in B' during deferral?
- Maybe the error is real and should be reported
- Or maybe the error is a transformation artifact: A would have branched to X, avoiding the error



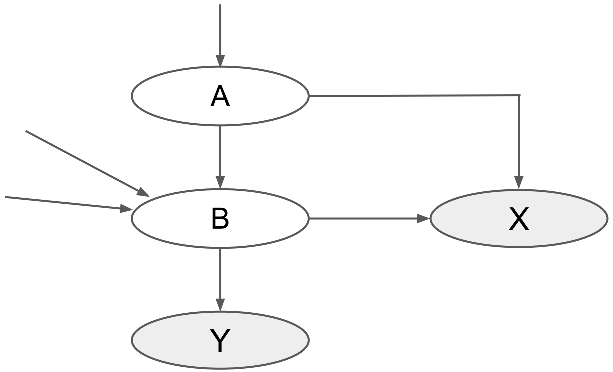
Compile time: transforming short-circuit CFG fragments

- Solution: use `defer` and `undefer` intrinsics
- If an error happens in B', fork on the deferred branch condition A
- Resulting states where A is infeasible should have gone to X in the first place
    - Jump directly to X
- Represented in CFG as untaken branch A'→X

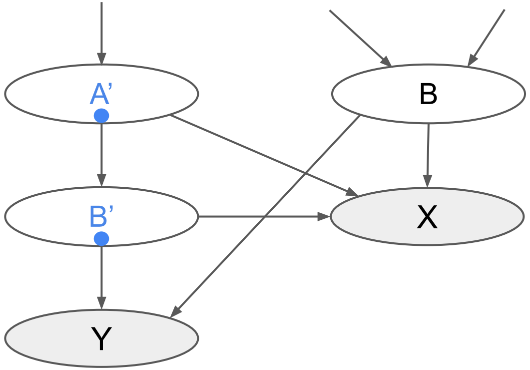


LLVM code example

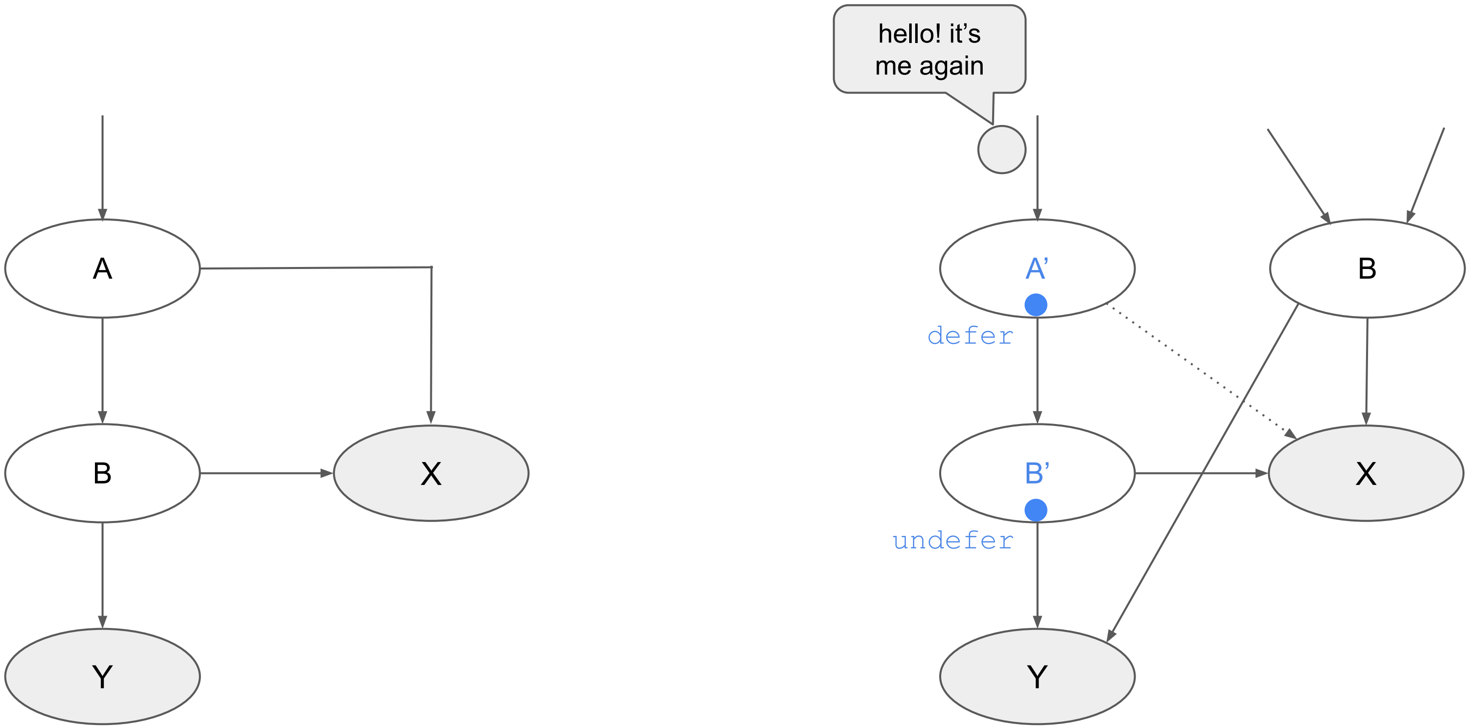
```
block.A:
  ...
  %condA = ...
  br i1 %condA, label %block.B, label %block.X

block.B:
  ...
  %condB = ...
  br i1 %condB, label %block.Y, label %block.X

block.X: ...
block.Y: ...
```



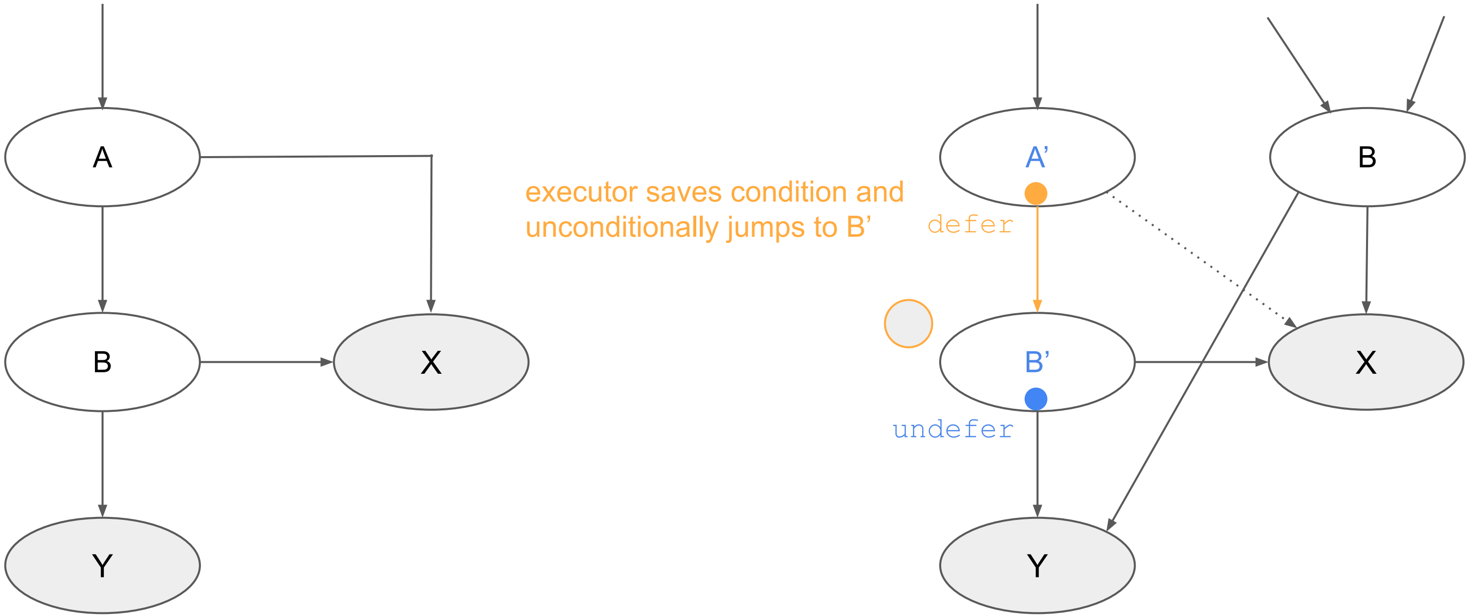

```
block.A:
  ...
  %condA = ...
  call void %klee_defer_next_branch(i32 0)
  br i1 %condA, label %block.B.undefer, label %block.X

block.B.undefer:
  ...
  %condB = ...
  call void %klee_undefer_next_branch(i32 0)
  br i1 %condB, label %block.Y, label %block.X

block.B: ...
block.X: ...
block.Y: ...
```



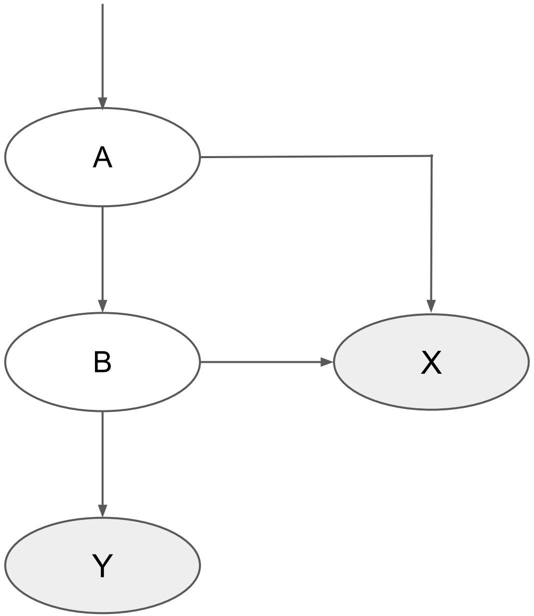
Run time: executing the intrinsics

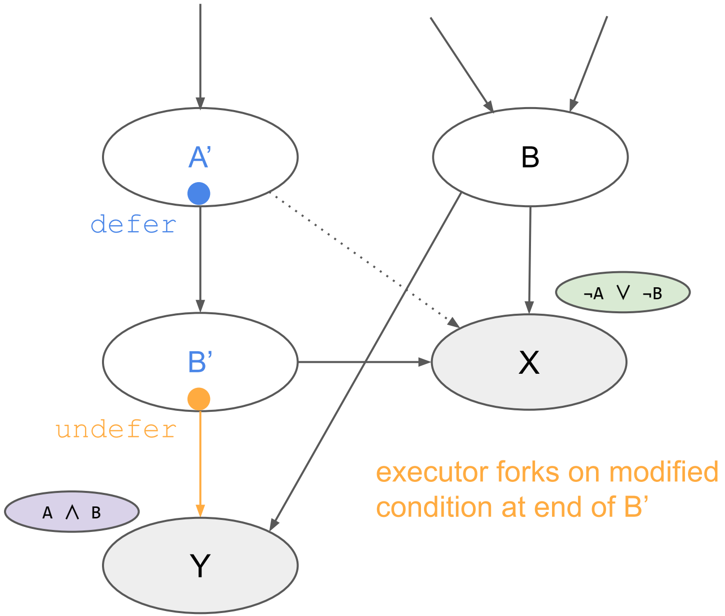


Run time: executing the intrinsics

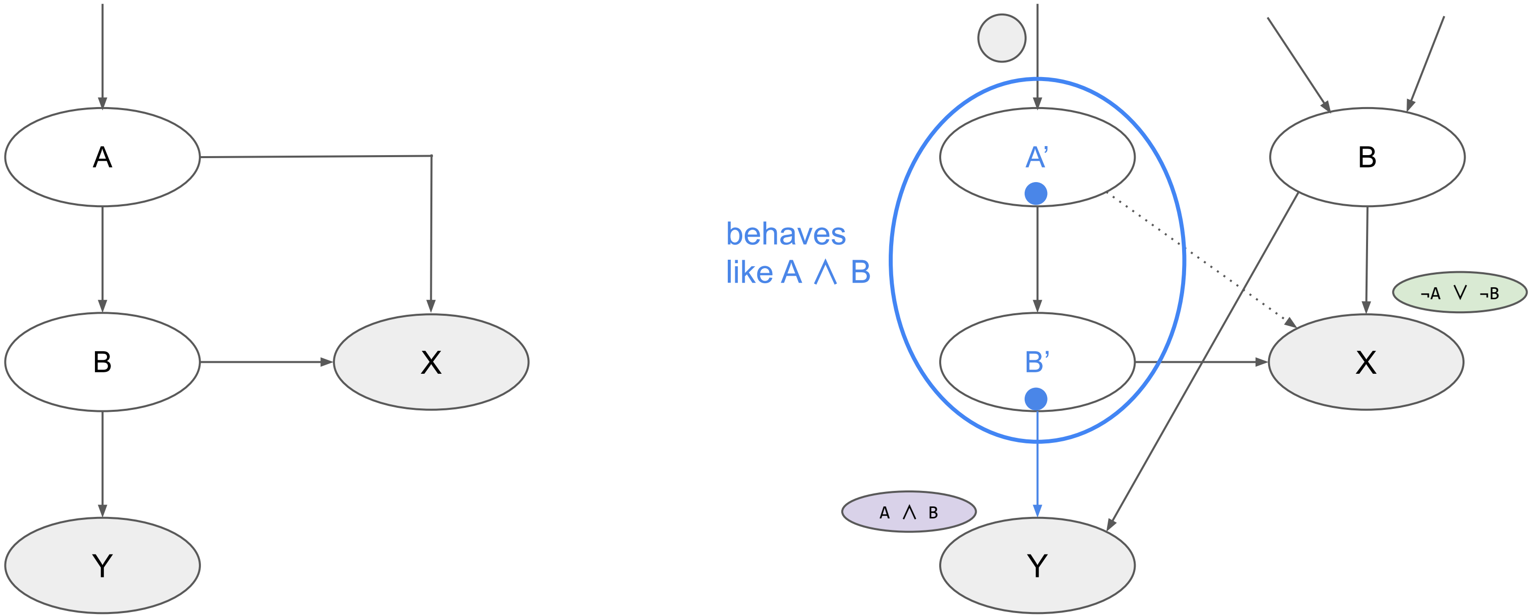


Run time: executing the intrinsics



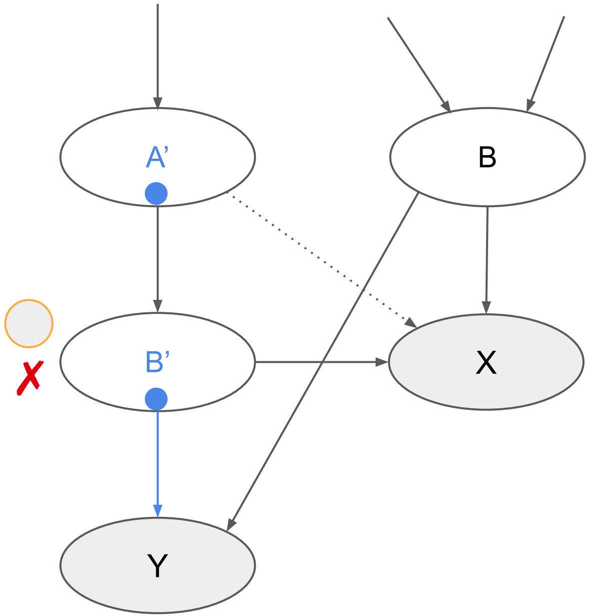


Run time: executing the intrinsics



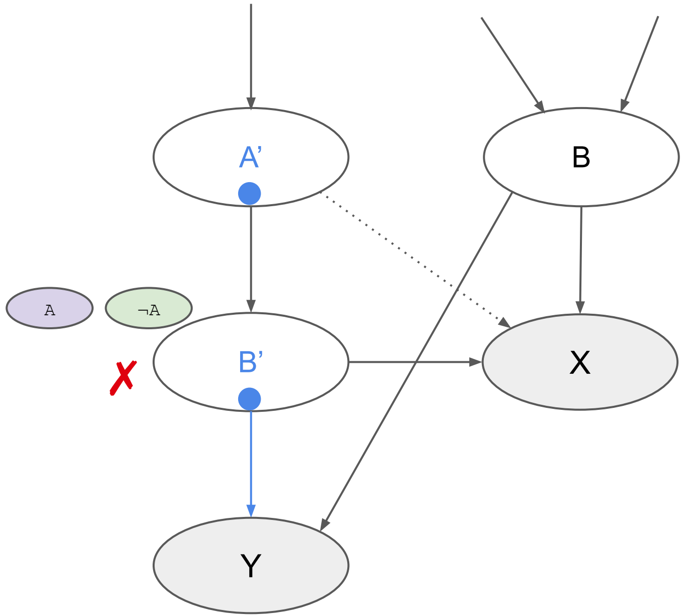
Reverting on errors during deferral

- Suppose state encounters error in B’



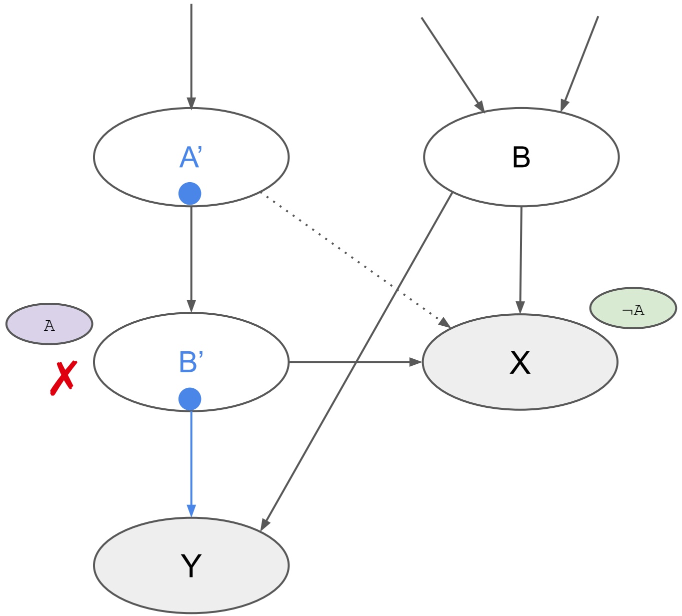
Reverting on errors during deferral

- Suppose state encounters error in B’
- Fork on the deferred branch condition



Reverting on errors during deferral

- Suppose state encounters error in B’
- Fork on the deferred branch condition
- States where A is feasible report a real bug
- States where A is infeasible should have gone to X in the first place
    - Jump directly to X, as without deferral



How does it do?

How does it do? Hash table example

Recall the example. With a **length-1** `l` and length-8 `key` as before:

- Version with **&&** and **branch deferral**:
  finishes in 44 ms with 2 paths explored
- Version with **&**:
  finishes in 47 ms with 2 paths explored
- Version with a **simpler hash** (XOR all characters):
  finishes in 42 ms with 3 paths explored

Branch deferral performs comparably to version with **&**!

```
typedef struct node {
  uint32_t hash;
  uint8_t *key;
  struct node *next;
} node;

node *find_key(node *l, uint32_t h,
               uint8_t *key) {
  while (l) {
    if (l->hash == h) {
      if (strcmp(l->key, key) == 0) {
        return l;
      }
    }
    l = l->next;
  }
  return NULL;
}
```

How does it do? Hash table example

Recall the example. With a **length-8** `l` and length-8 `key` on each node:

- Version with **&&** and **branch deferral**:
  finishes in 82 ms with 9 paths explored
- Version with **&**:
  finishes in 96 ms with 9 paths explored
- Version with a **simpler hash** (XOR all characters):
  finishes in 119 ms with 17 paths explored

Branch deferral performs comparably to version with **&**!

```
typedef struct node {
  uint32_t hash;
  uint8_t *key;
  struct node *next;
} node;

node *find_key(node *l, uint32_t h,
               uint8_t *key) {
  while (l) {
    if (l->hash == h) {
      if (strcmp(l->key, key) == 0) {
        return l;
      }
    }
    l = l->next;
  }
  return NULL;
}
```

How does it do? SQLite

- sqlite-amalgamation-3450100
- 1 hour maximum time
- 30 second solver timeout
- solver: STP with MiniSat
- search: random-path with nurs:covnew
- div-by-zero and overshift checks disabled
- optimizations off!
- 392872 total instructions
  - transformation applied in both cases
  - deferral disabled via disabling intrinsics
- 40% more coverage!

Deferral	on	off
# Covered instructions	18,672	13,356
# Completed paths (# generated tests)	229 (81)	57 (47)
# Solver queries	65,270	51,021
Solver time (s)	3,308	3,207
# Instructions executed	458,089,686	296,394,468

That’s all for now!

Work in progress. We’d like to ask for feedback!

That’s all for now!

Work in progress. We’d like to ask for feedback!

- We’re currently transforming wherever possible. Transforming at some sites may hurt performance. Where is this likely?
- Measuring which sites most affect performance: how?
- Implementation is not robust to optimization.
- Programs/benchmarks to try?

Conclusion

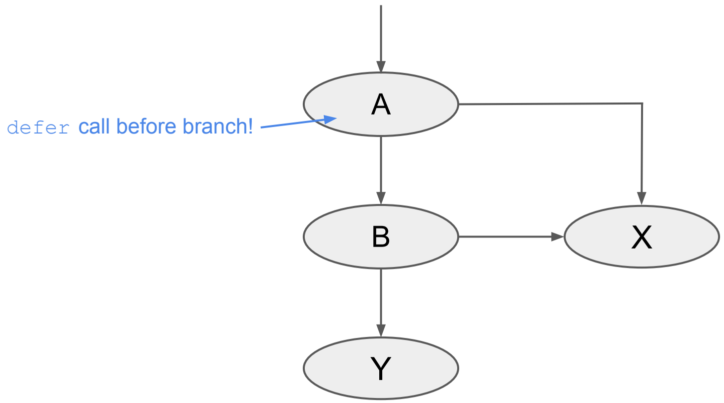
- We presented **branch deferral**, an optimization that modifies execution of short-circuit CFGs to reduce forking.
- Branch deferral helps on microbenchmarks and sqlite.

Hash function (similar to `full_name_hash`)

```
#define GOLDEN_RATIO_64 0x61C8864680B583EBull
uint32_t hash(uint8_t *s) {
  uint64_t x = 0;
  uint64_t y = 5381;
  for (int i = 0; i < 8; i++) {
    x ^= s[i];
    y ^= x;
    x = (x << 7) | (x >> 25);
    x += y;
    y = (y << 20) | (y >> 12);
    y *= 9;
  }
  y ^= x * GOLDEN_RATIO_64;
  y *= GOLDEN_RATIO_64;
  return y >> 32;
}
```

KLEE implementation: speculating on the branch

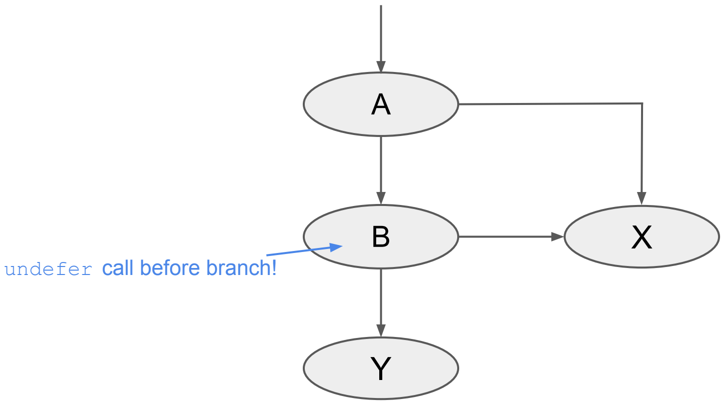
- `klee_defer_next_branch` sets a flag on the execution state
- Upon next branch, store the condition and transfer unconditionally to B




```
case Instruction::Br: {
  BranchInst *bi = cast<BranchInst>(i);

  ...

  if (state.deferNext != -1) {

    state.deferredConstraints.emplace_back(
      cond, ...);

    transferToBasicBlock(
      bi->getSuccessor(1 - state.deferNext),
      bi->getParent(), state);

    state.deferNext = -1;

    break;
  }

  ...
```

KLEE implementation: handling the deferred condition

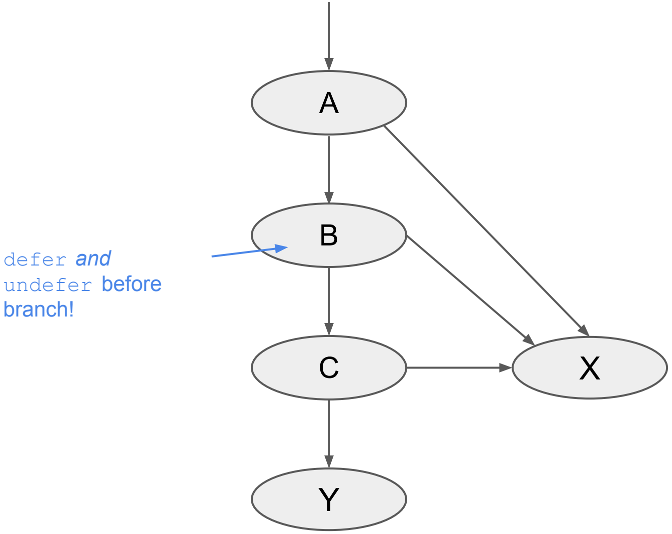
- `klee_undefer_next_branch` also sets a flag on the execution state
- Upon next branch, pop the deferred condition and modify branch condition




```
case Instruction::Br: {
  BranchInst *bi = cast<BranchInst>(i);

  ...

  if (state.undeferNext != -1) {
    auto record = state.deferredConstraints.back();

    cond = /* compute branch condition */

    state.deferredConstraints.pop_back();

    state.undeferNext = -1;
  }

  ...

  Executor::StatePair branches = fork(state, cond, ...);

  ...
```

KLEE implementation: handling the deferred condition

- Handle chained short-circuits by handling `undefer` before `defer`.




```
case Instruction::Br: {
  BranchInst *bi = cast<BranchInst>(i);

  ...

  if (state.undeferNext != -1) {
    ...
  }

  if (state.deferNext != -1) {
    ...

    break;
  }

  ...

  Executor::StatePair branches = fork(state, cond, ...);

  ...
```

KLEE implementation: reverting on errors during deferral

- If an error is encountered in B block during deferral, we must check whether it is actually feasible
- Fork on original deferred condition
- States satisfying the condition have encountered a real bug
- States not satisfying the condition should have gone to X in the first place

```
void Executor::terminateStateOnProgramError(...) {

  if (state.deferredConstraints.size() != 0) {

    auto record = state.deferredConstraints.back();

    state.deferredConstraints.pop_back();

    Executor::StatePair branches = fork(
      state, record.cond, ...);

    if (branches.first) {
      terminateStateOnError(*branches.first, ...);
    }

    if (branches.second) {
      /* transfer to X block */
    }

  }

  ...
```