Spec-o-Scope: Cache Probing at Cache Speed

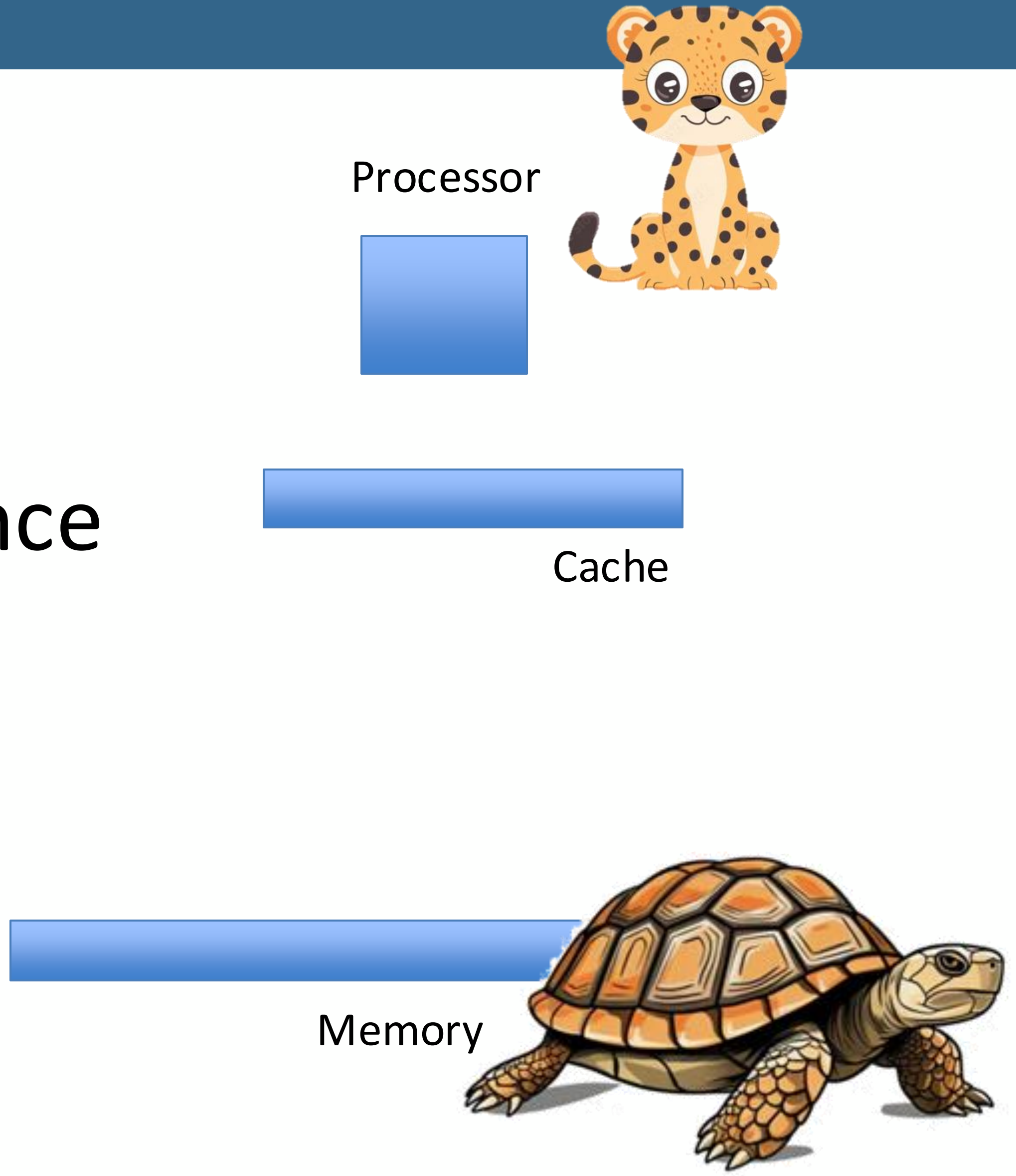
Gal Horowitz, Eyal Ronen, Yuval Yarom

TEL AVIV UNIVERSITY

RUHR UNIVERSITÄT BOCHUM RUB

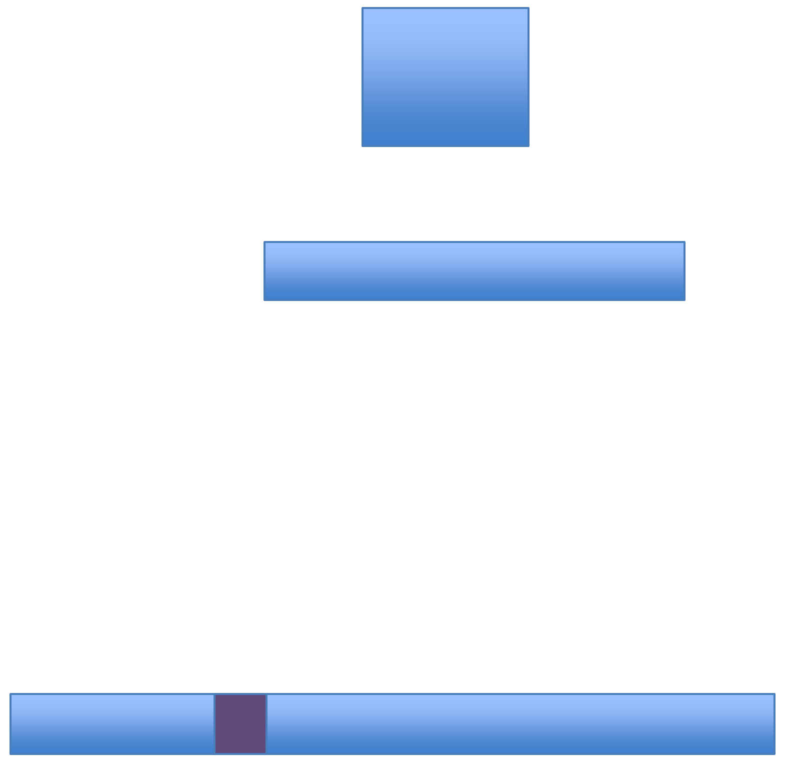
Memory Cache

- Memory is slow
- Cache: a small bank of fast memory. Exploit locality to improve performance
- Stores recently accessed data for quick future access



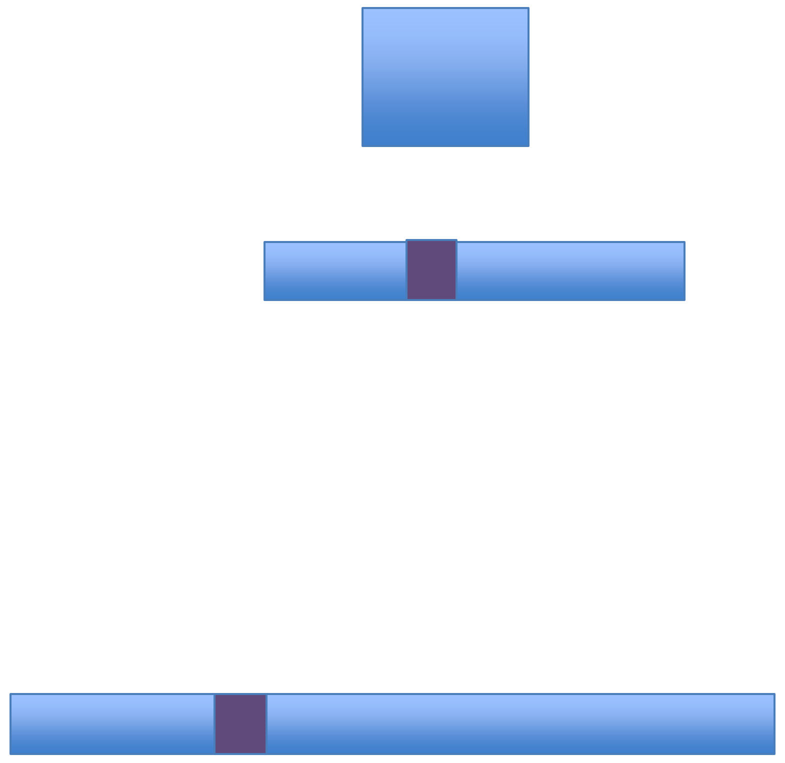
Cache operations

- Accessing memory brings it to the cache



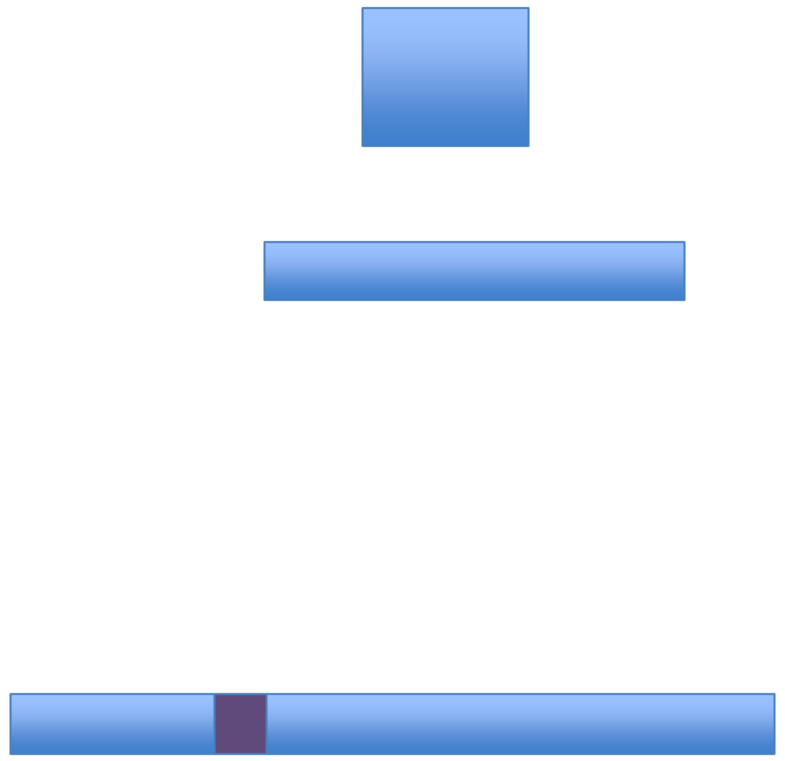
Cache operations

- Accessing memory brings it to the cache



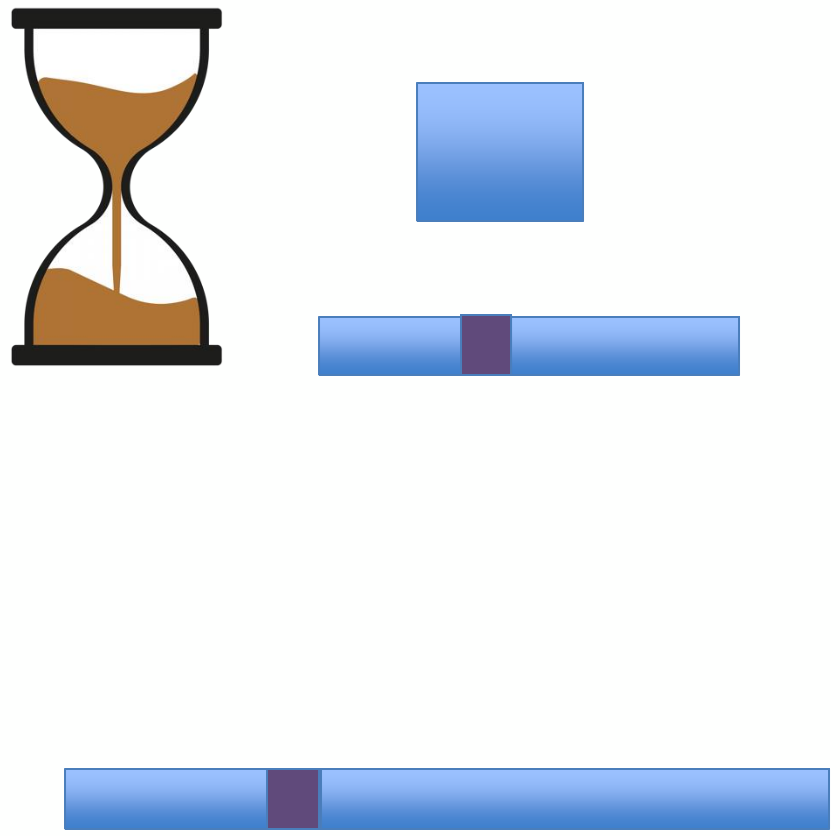
Cache operations

- Accessing memory brings it to the cache
- Flushing memory evicts it from the cache



Cache Side channel

- Measuring access time tells us whether a location is cached or not

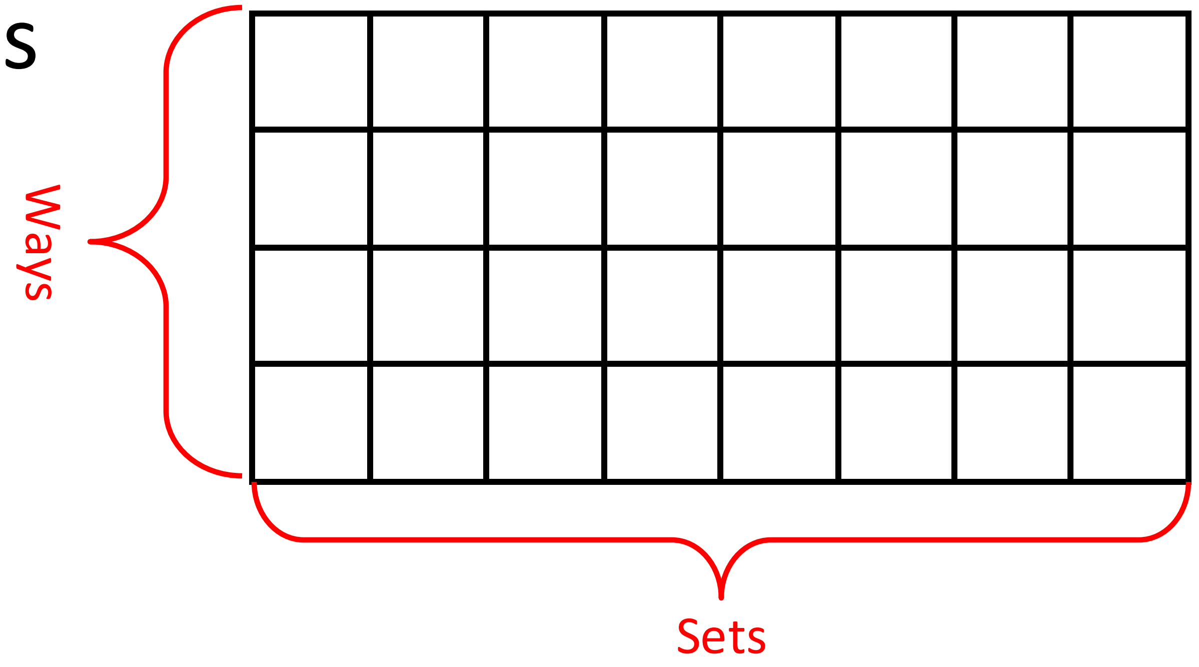


Cache construction

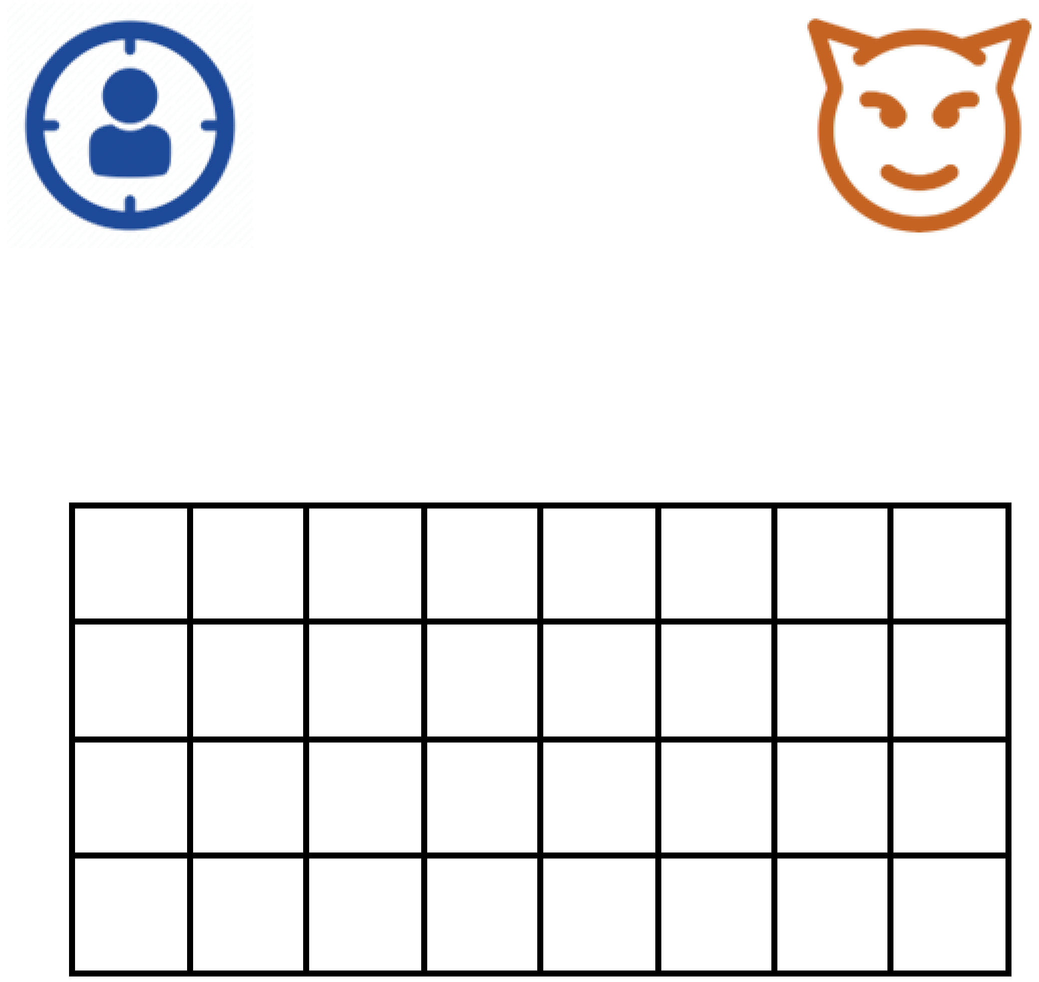
- Memory locations mapped to sets

- Each set can store multiple blocks

- Replacement policy decides where

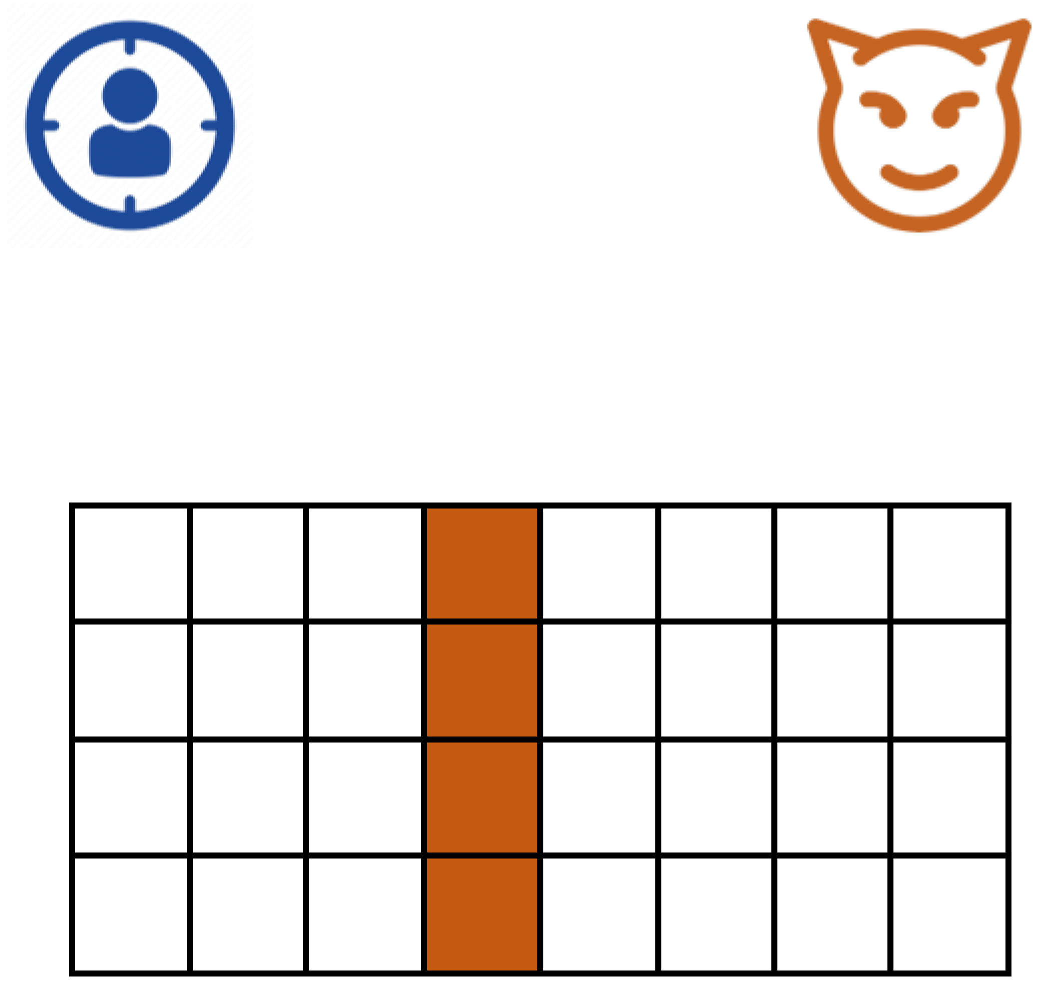


Prime+Probe



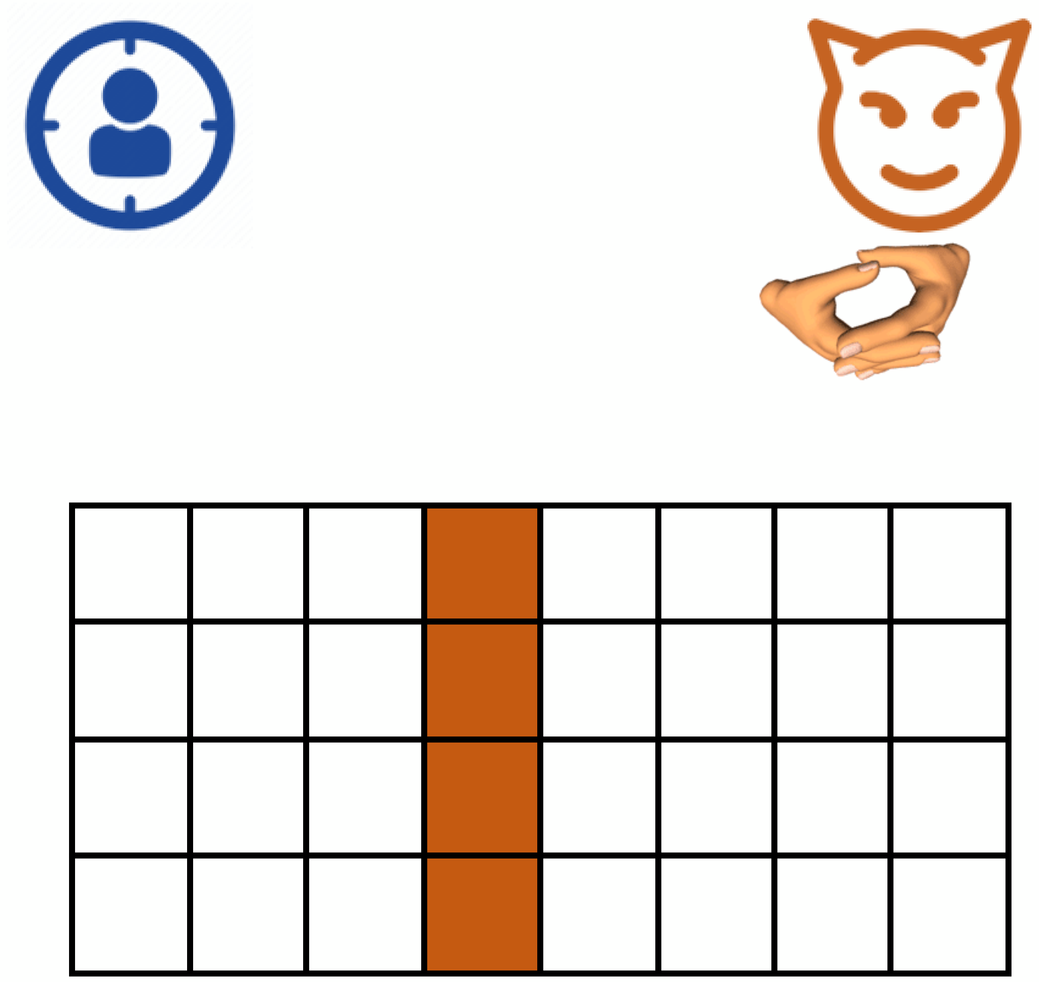
Prime+Probe

- Fill a cache set with data



Prime+Probe

- Fill a cache set with data
- Wait a bit

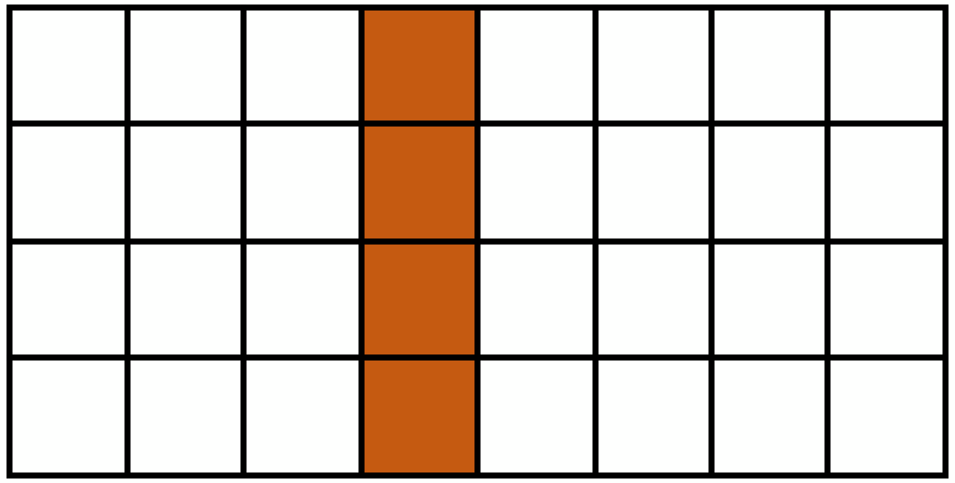


Prime+Probe

- Fill a cache set with data

- Wait a bit

- Measure access time to data

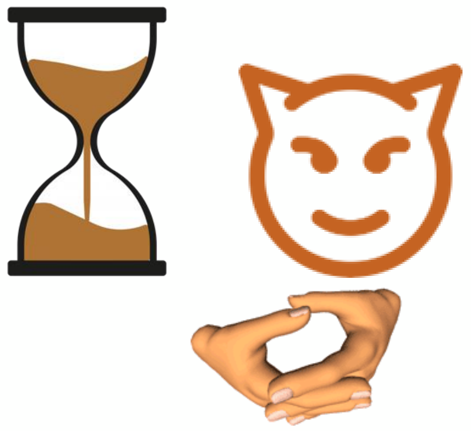


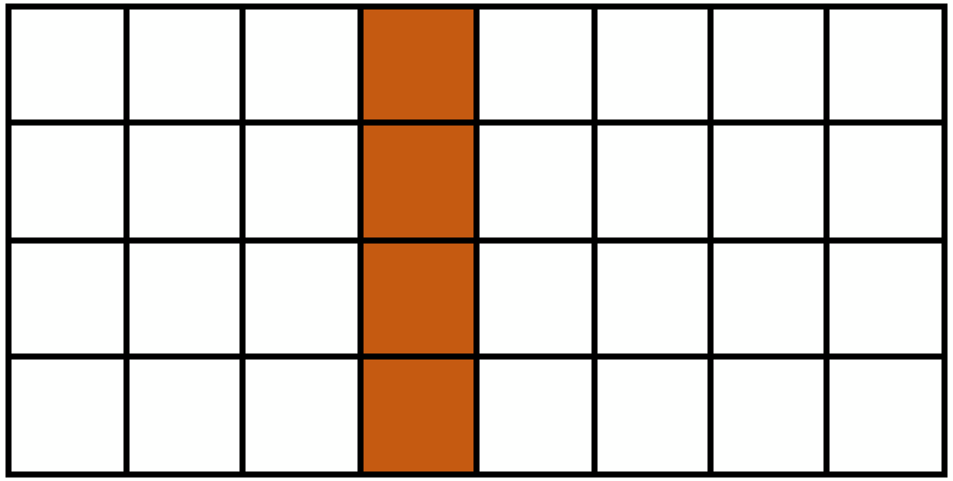
Prime+Probe

- Fill a cache set with data

- Wait a bit

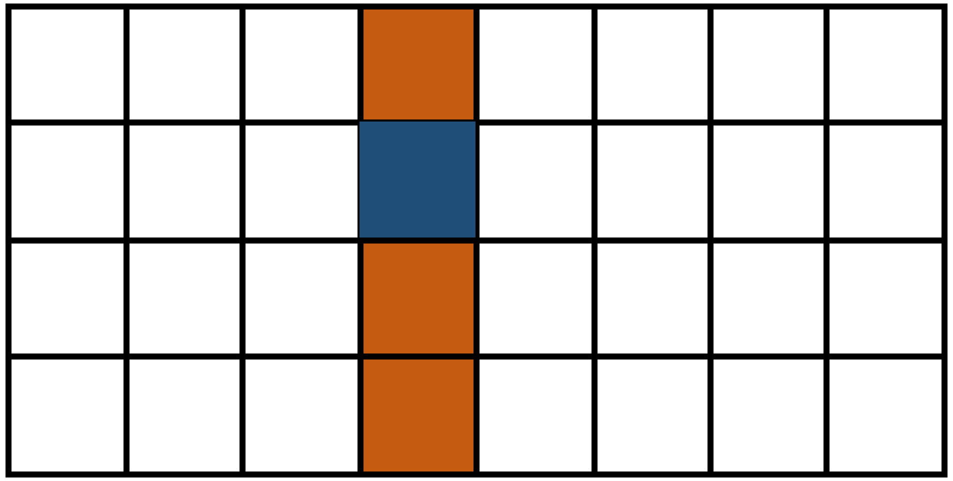
- Measure access time to data





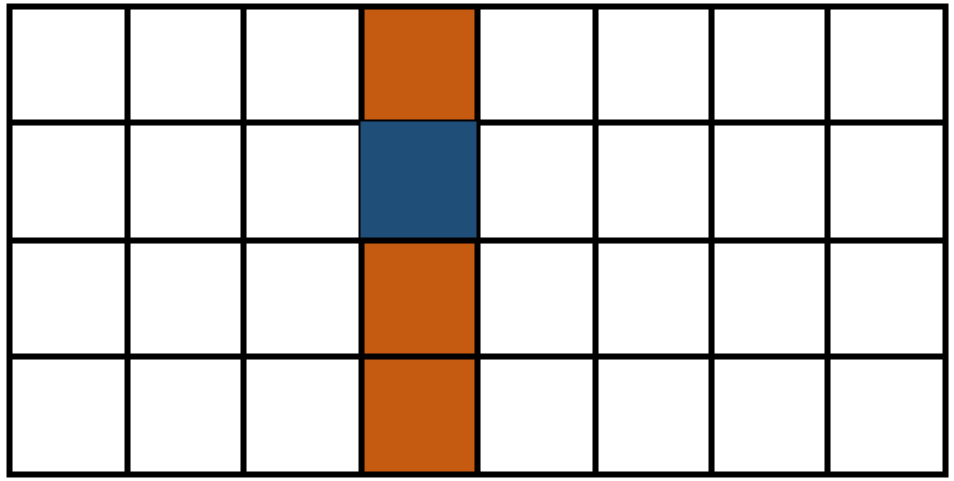
Prime+Probe

- Fill a cache set with data
- Wait a bit
- Measure access time to data



Prime+Probe

- Fill a cache set with data
- Wait a bit
- Measure access time to data
- Can monitor other programs!



Probe Rate

- *How fast can we probe the cache?*

- Limited temporal resolution
  - Thousands of cycles

- Prime+Scope (CCS 2021) – 70 cycles

Prime+Scope (CCS 2021)

- Carefully arrange data in different cache levels

- Victim access evicts LLC line

$\longrightarrow$ Line also evicted from L1

- “Scoping”: Repeatedly measure access time in L1

Prime+Scope code

```
uint32_t scope(char * address) {
  uint32_t start = rdtscp();
  char t = *address;
  uint32_t end = rdtscp();
  return end - start;
}
```

5 cycles

30 cycles

5 cycles

30 cycles

Prime+Scope code

5 cycles

```
uint32_t scope(char * address) {
  uint32_t start = rdtscp();
  char t = *address;
  uint32_t end = rdtscp();
  return end - start;
}
```

30 cycles

5 cycles

30 cycles

Measuring time is an order of magnitude slower than a cache access

Micro-architectural Weird Gates

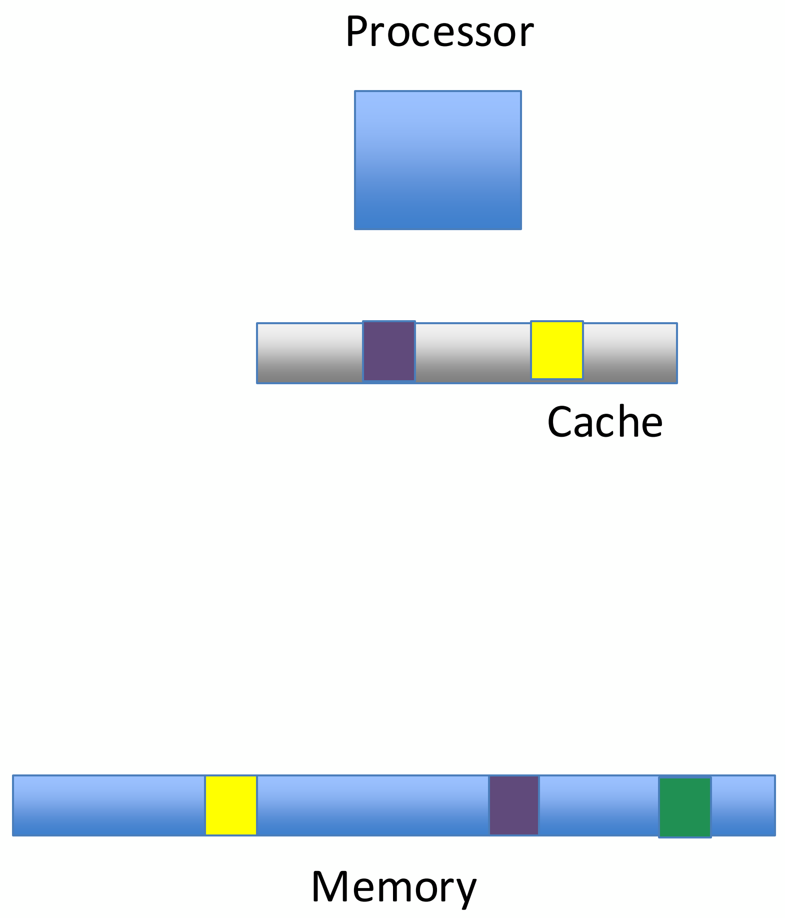
Micro-architectural Weird Gates

- Recent works perform computation using transient execution

- Interesting applications
  - Obfuscation
  - Amplification
  - Decoupling cache probe from measurement

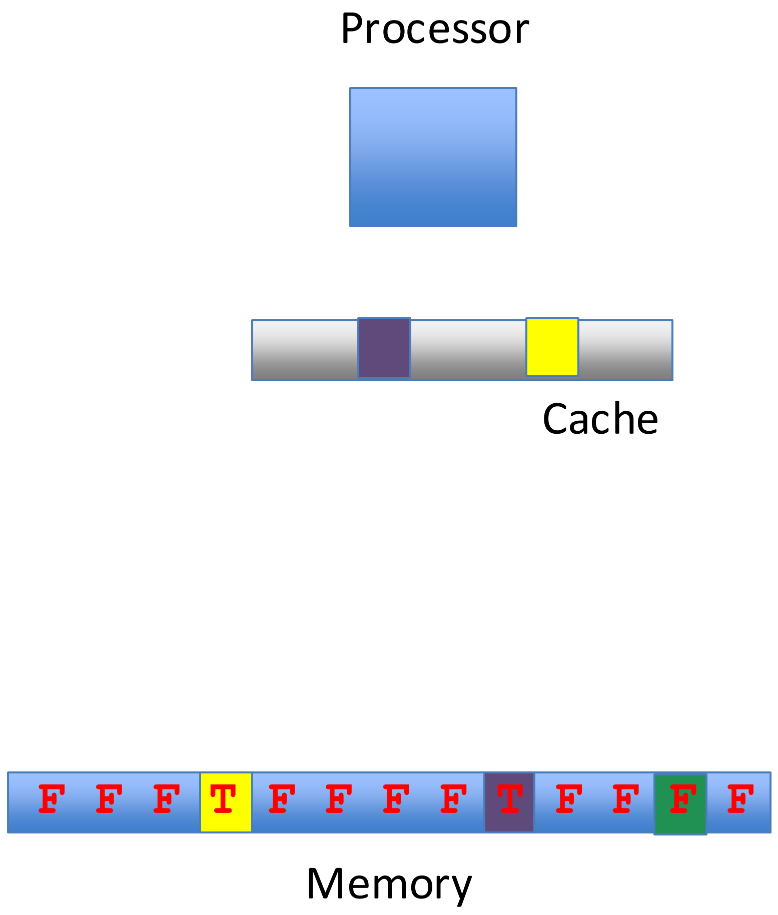
Logical State of Cache

- Associate a logical value with memory addresses
  - TRUE – address is cached
  - FALSE – address is not cached



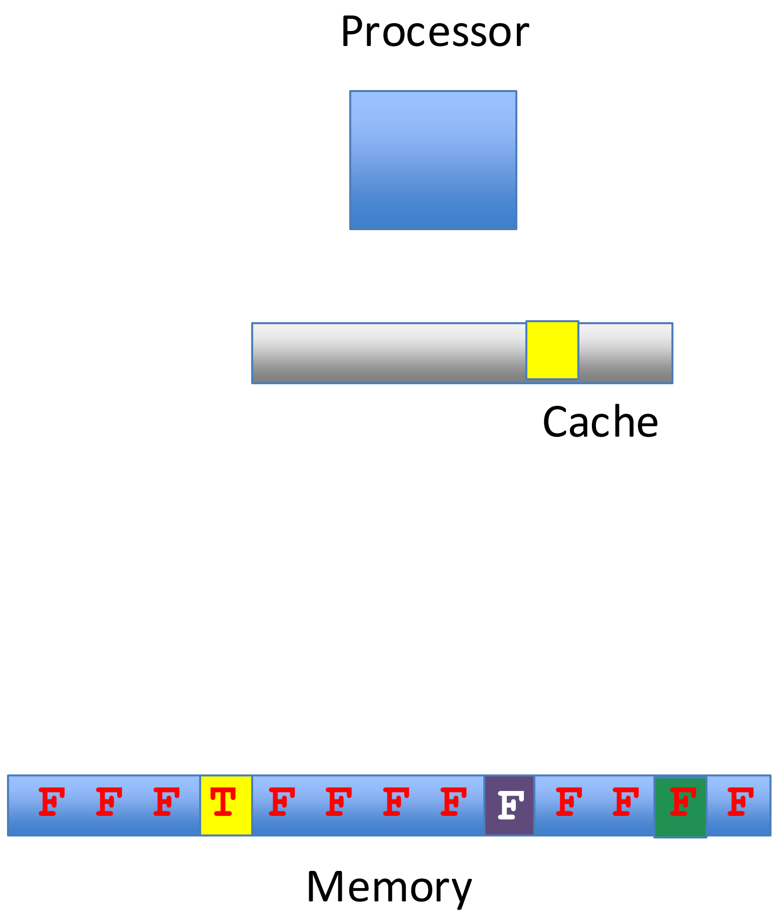
Logical State of Cache

- Associate a logical value with memory addresses
  - TRUE – address is cached
  - FALSE – address is not cached



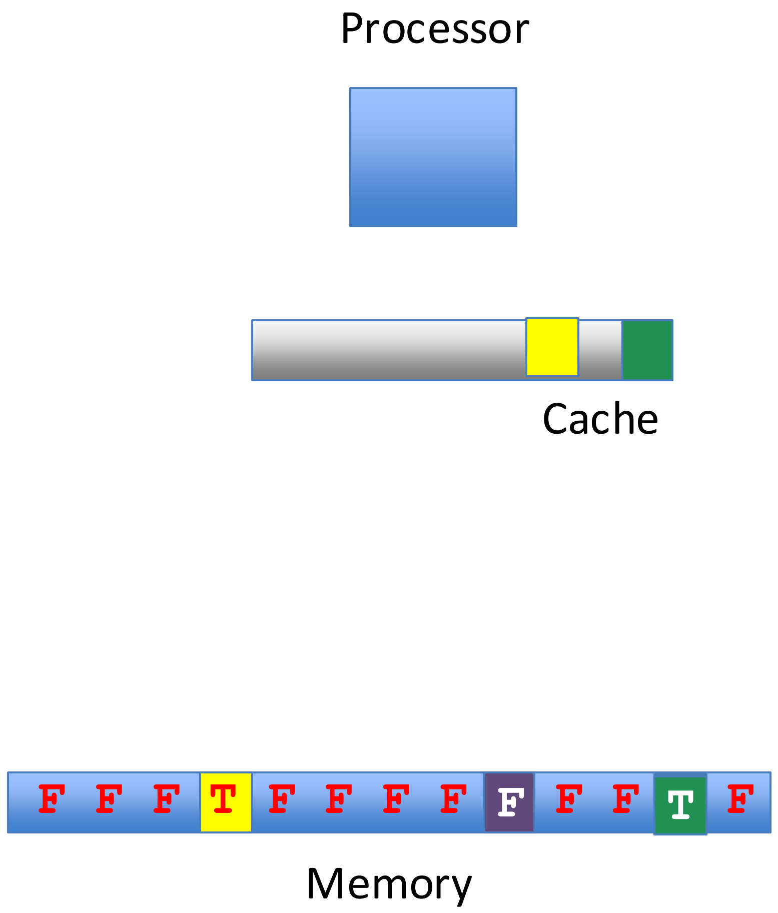
Logical State of Cache

- Associate a logical value with memory addresses
  - TRUE – address is cached
  - FALSE – address is not cached
- Flushing sets a value to FALSE



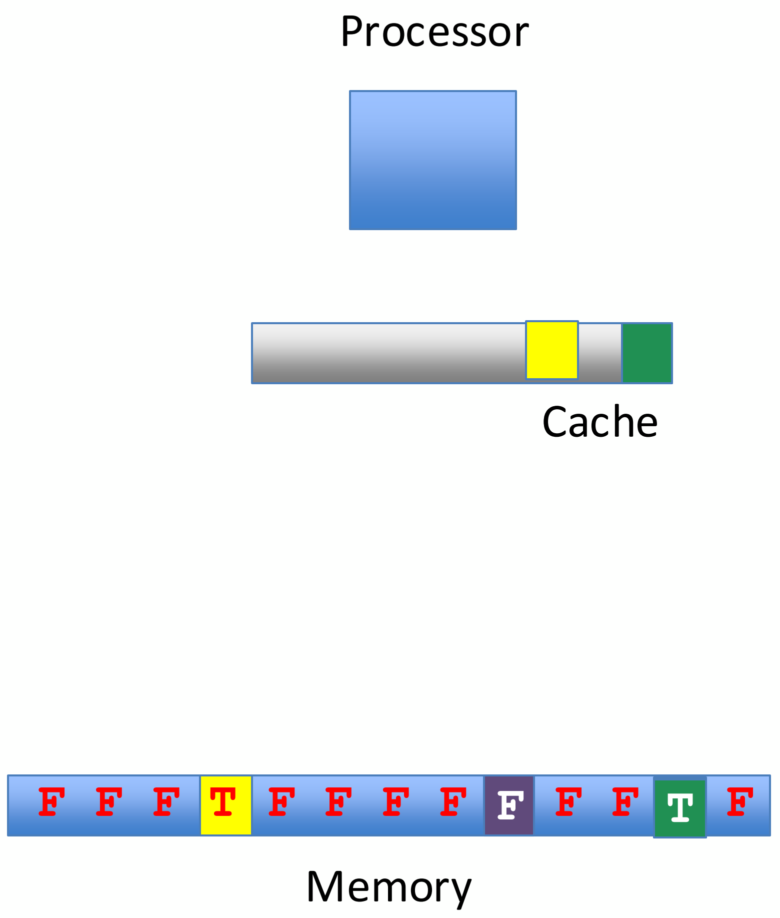
Logical State of Cache

- Associate a logical value with memory addresses
    - TRUE – address is cached
    - FALSE – address is not cached
- Flushing sets a value to FALSE
- Accessing memory sets a value to TRUE (may also set another to FALSE)



Logical State of Cache

- Associate a logical value with memory addresses
  - TRUE – address is cached
  - FALSE – address is not cached
- Flushing sets a value to FALSE
- Accessing memory sets a value to TRUE (may also set another to FALSE)
- Measuring access time observes value (and set to TRUE)



Conditional access

```
if (*in == 0)
    return;
out += 0;
out += 0;
a = *out
```

- What is the cache state of **`*out`** after execution?

Conditional access

```
if (*in == 0)
    return;
out += 0;
out += 0;
a = *out
```

- What is the cache state of `*out` after execution?
- TRUE if `*in != 0`.

Conditional access

```
if (*in == 0)
    return;
out += 0;
out += 0;
a = *out
```

- What is the cache state of `*out` after execution?
- TRUE if `*in != 0`.
- What if `*in == 0`?

Conditional access

```
if (*in == 0)
    return;
out += 0;
out += 0;
a = *out
```

- What is the cache state of **`*out`** after execution?
- TRUE if **`*in != 0`**.
- What if **`*in == 0`**?

Assume `*in == 0`

Speculative execution

Assume

```
*in == 0
```

```
if (*in == 0)
        return;
out += 0;
out += 0;
a = *out
```

Speculative execution

Assume
`*in == 0`

```
if (*in == 0)
        return;
out += 0;
out += 0;
a = *out
```

- Evaluation of branch conditions can take time

Speculative execution

Assume
`*in == 0`

```
if (*in == 0)
        return;
out += 0;
out += 0;
a = *out
```

- Evaluation of branch conditions can take time
- The CPU predicts future execution
  - Correct prediction – win
  - Incorrect prediction – rollback

Speculative execution

Assume
`*in == 0`

```
if (*in == 0)
        return;
out += 0;
out += 0;
a = *out
```

- Evaluation of branch conditions can take time
- The CPU predicts future execution
  - Correct prediction – win
  - Incorrect prediction – rollback
  - **Microarchitectural state remains**

Speculative execution

Assume
`*in == 0`

```
if (*in == 0)
        return;
out += 0;
out += 0;
a = *out
```

May be executed even if `*in == 0`

- Evaluation of branch conditions can take time
- The CPU predicts future execution
  - Correct prediction – win
  - Incorrect prediction – rollback
  - **Microarchitectural state remains**

Speculative execution

Assume
`*in == 0`

```
if (*in == 0)
        return;
out += 0;
out += 0;
a = *out
```

May be executed even if `*in == 0`

- Evaluation of branch conditions can take time

dicts future

ediction – win

rediction – rollback

  - **Microarchitectural state remains**

Assume branch mispredicted

Conditional Speculative Execution

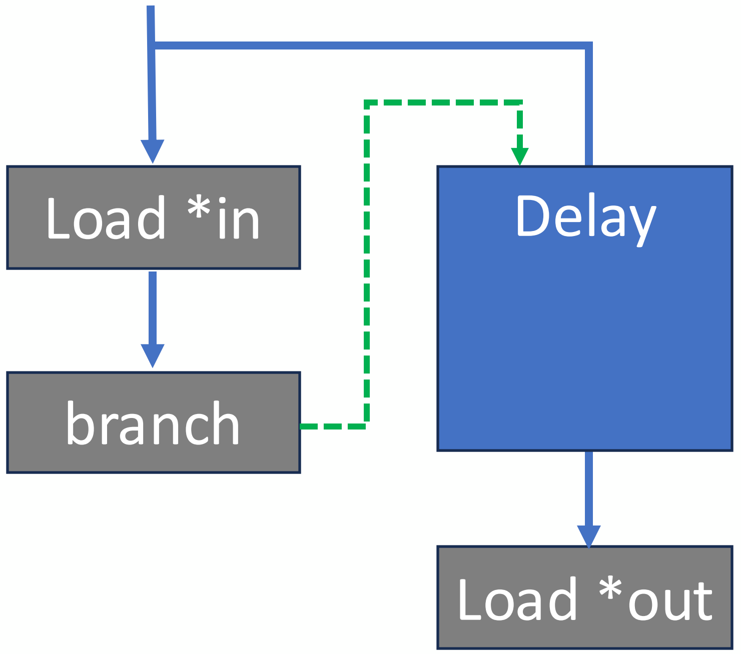
Assume
`*in == 0`
Branch mispredicted

```
if (*in == 0)
    return;
out += 0;
out += 0;
a = *out
```

Conditional Speculative Execution

Assume
`*in == 0`
Branch mispredicted

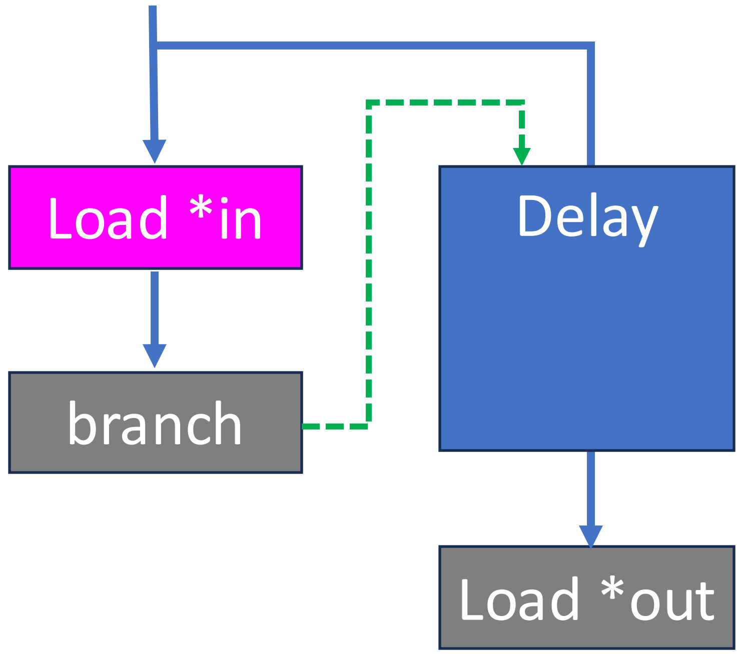
```
if (*in == 0)
        return;
out += 0;
out += 0;
a = *out
```



Conditional Speculative Execution

Assume
`*in == 0`
Branch mispredicted

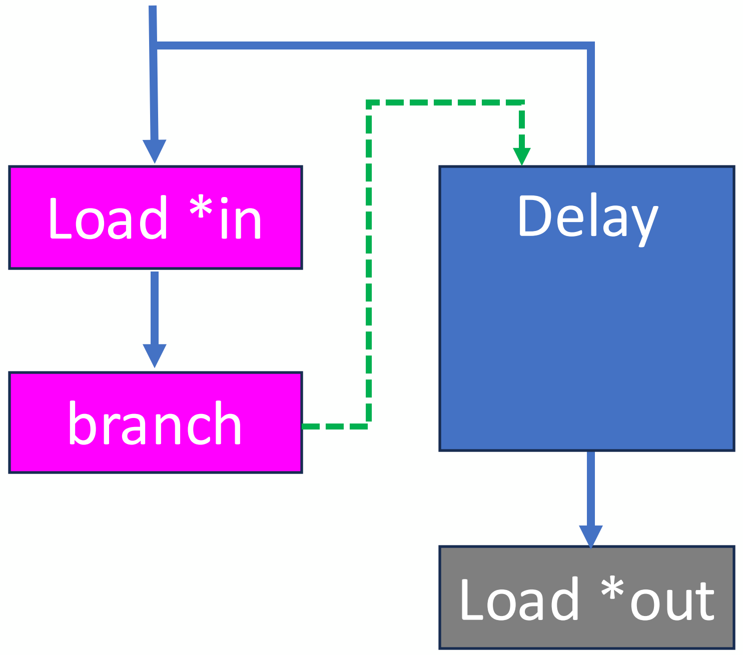
```
if (*in == 0)
        return;
out += 0;
out += 0;
a = *out
```



Conditional Speculative Execution

Assume
`*in == 0`
Branch mispredicted

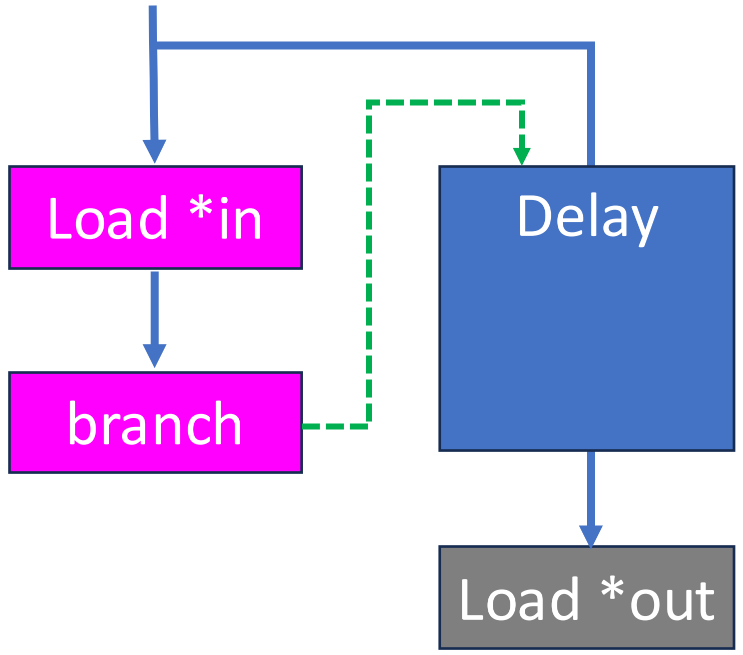
```
if (*in == 0)
        return;
out += 0;
out += 0;
a = *out
```



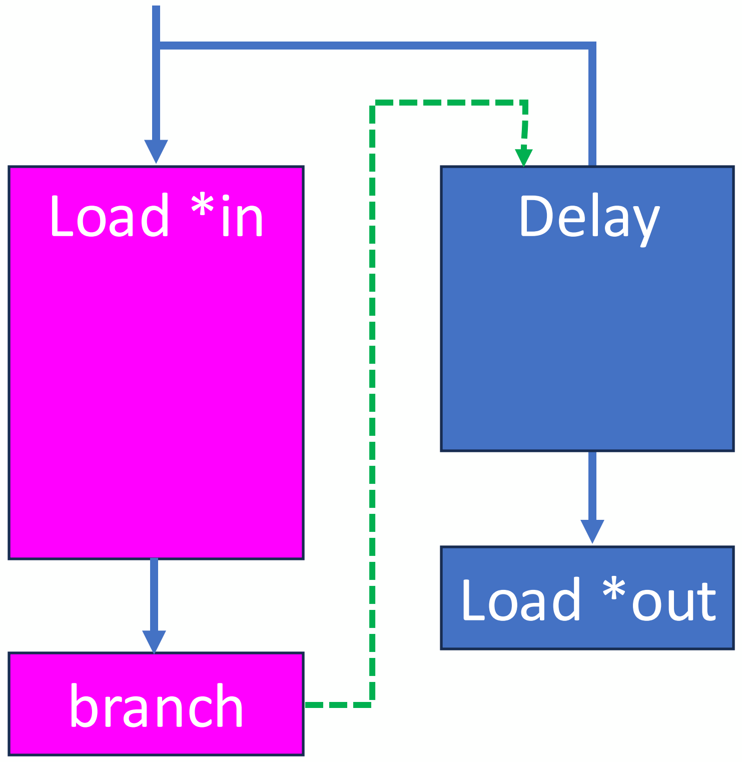
Conditional Speculative Execution

Assume
`*in == 0`
Branch mispredicted

```
if (*in == 0)
        return;
out += 0;
out += 0;
a = *out
```



*in cached



*in not cached

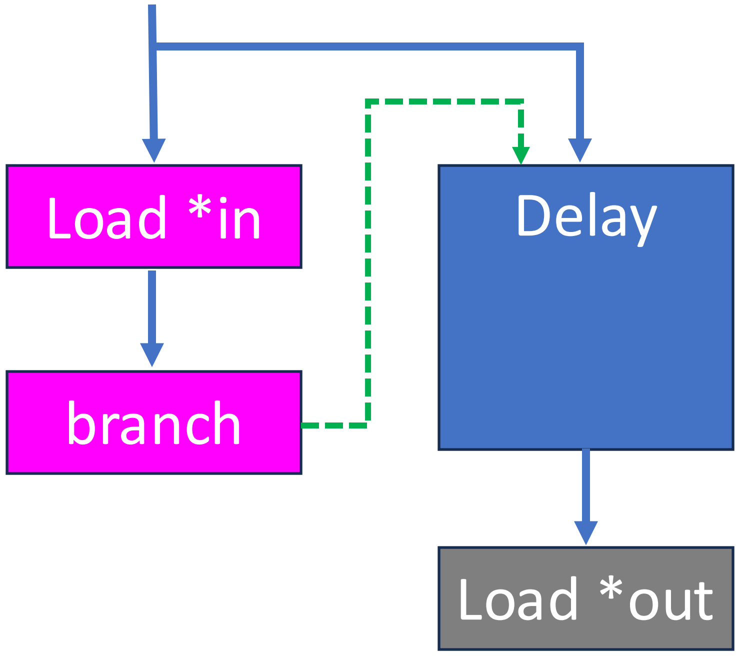
Weird NOT gate

Assume
`*in == 0`
Branch mispredicted

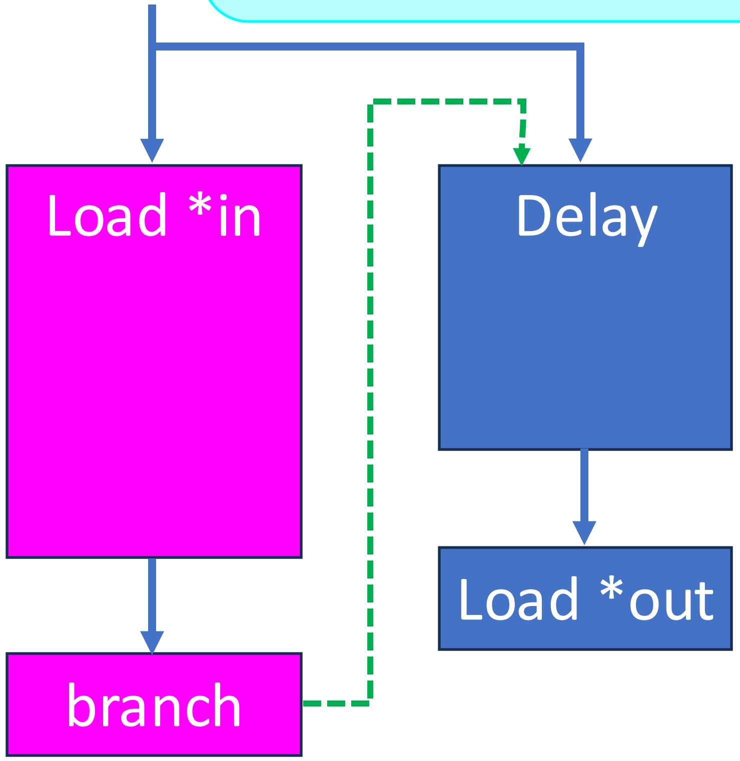
```
if (*in == 0)
    return;
out += 0;
out += 0;
a = *out
```

*in	*out
TRUE	FALSE
FALSE	TRUE

out ← NOT(in)



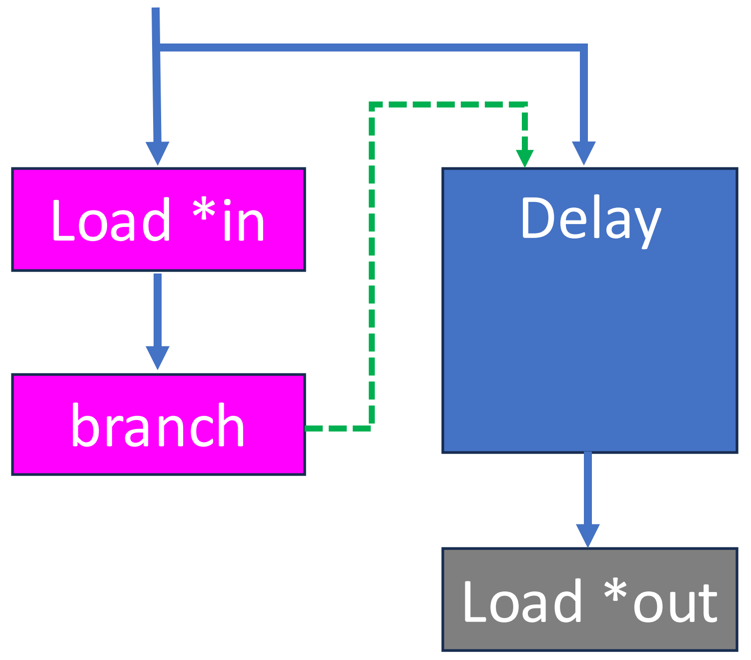
*in cached



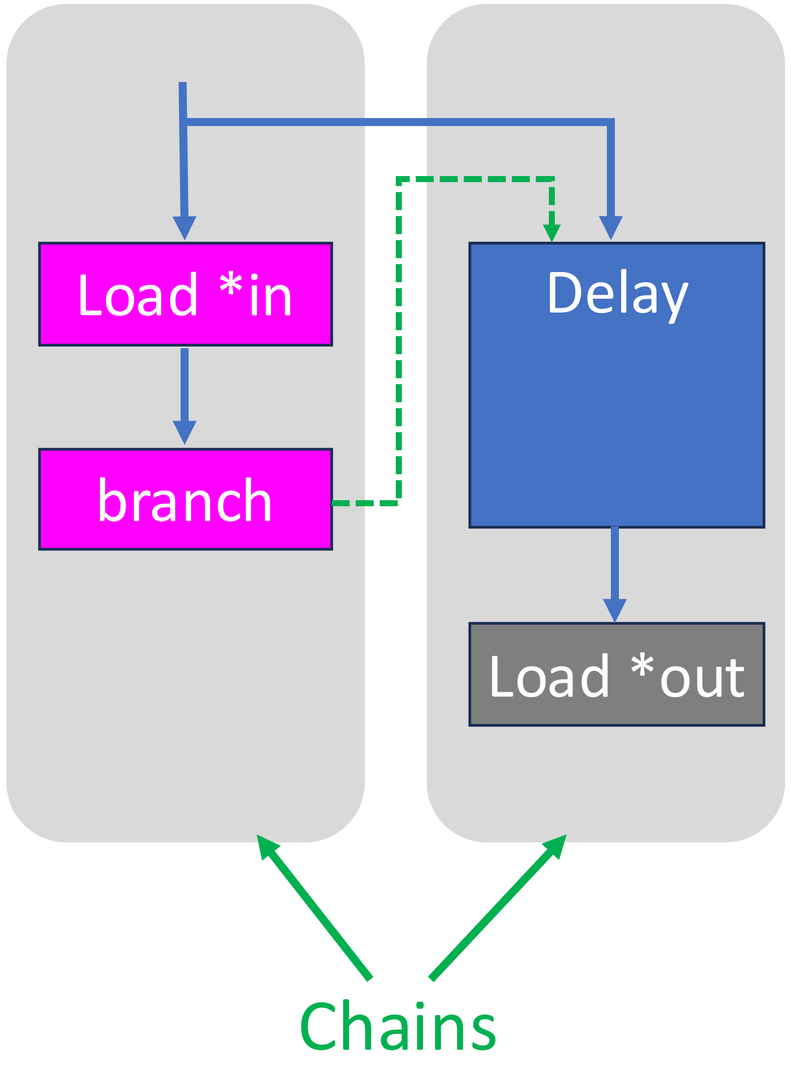
*in not cached

Thinking about this

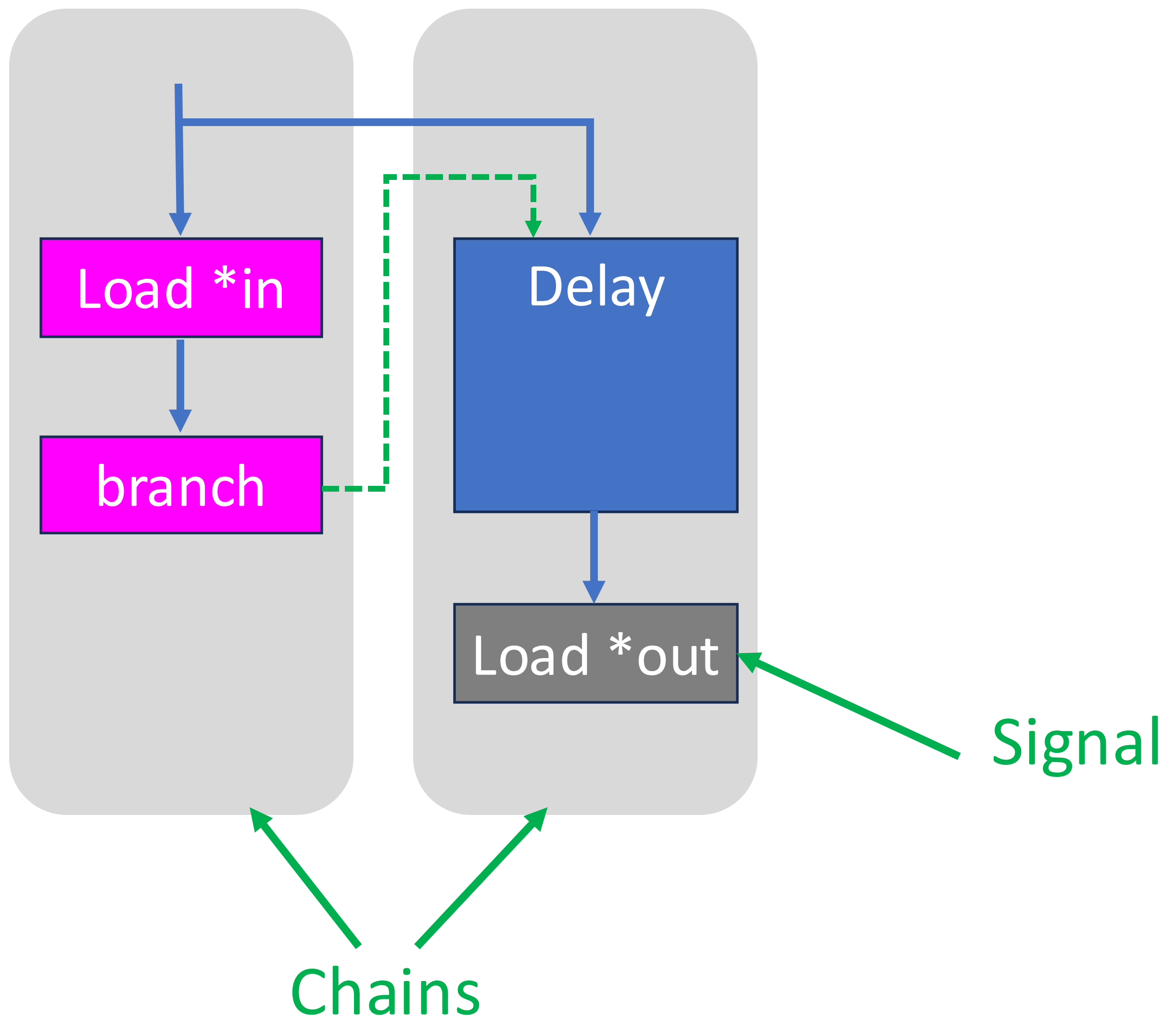
Thinking about this



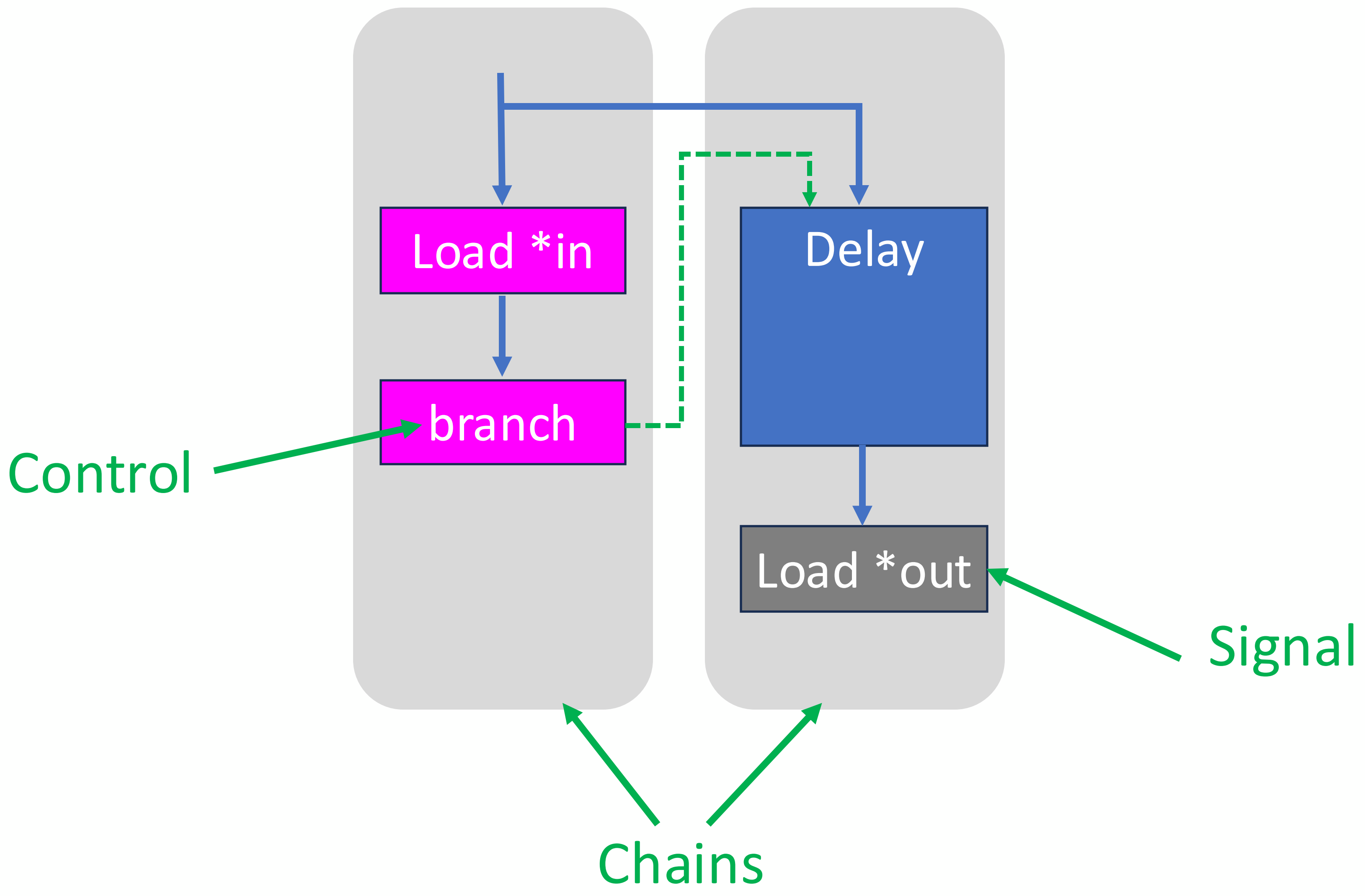
Thinking about this



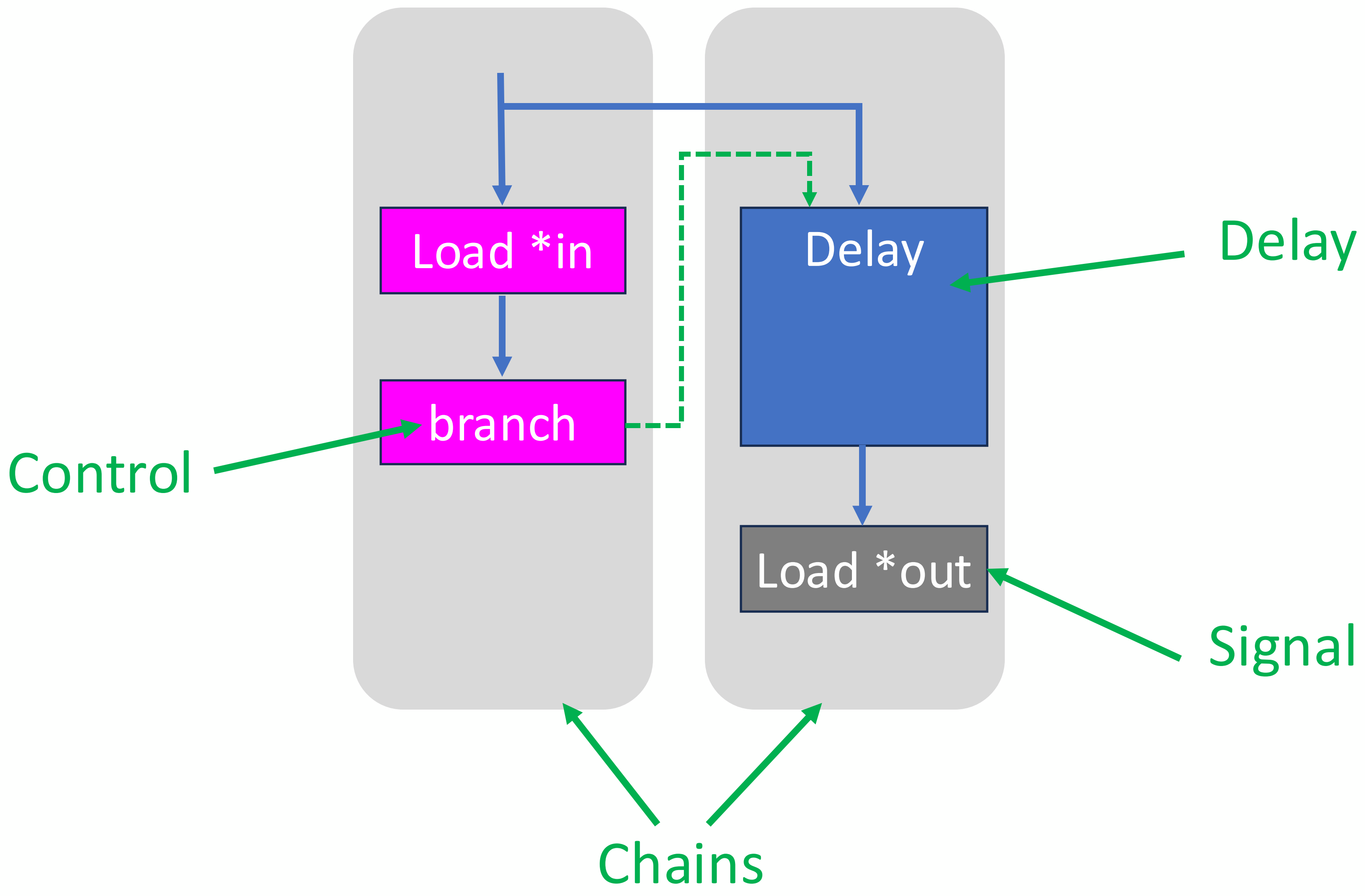
Thinking about this



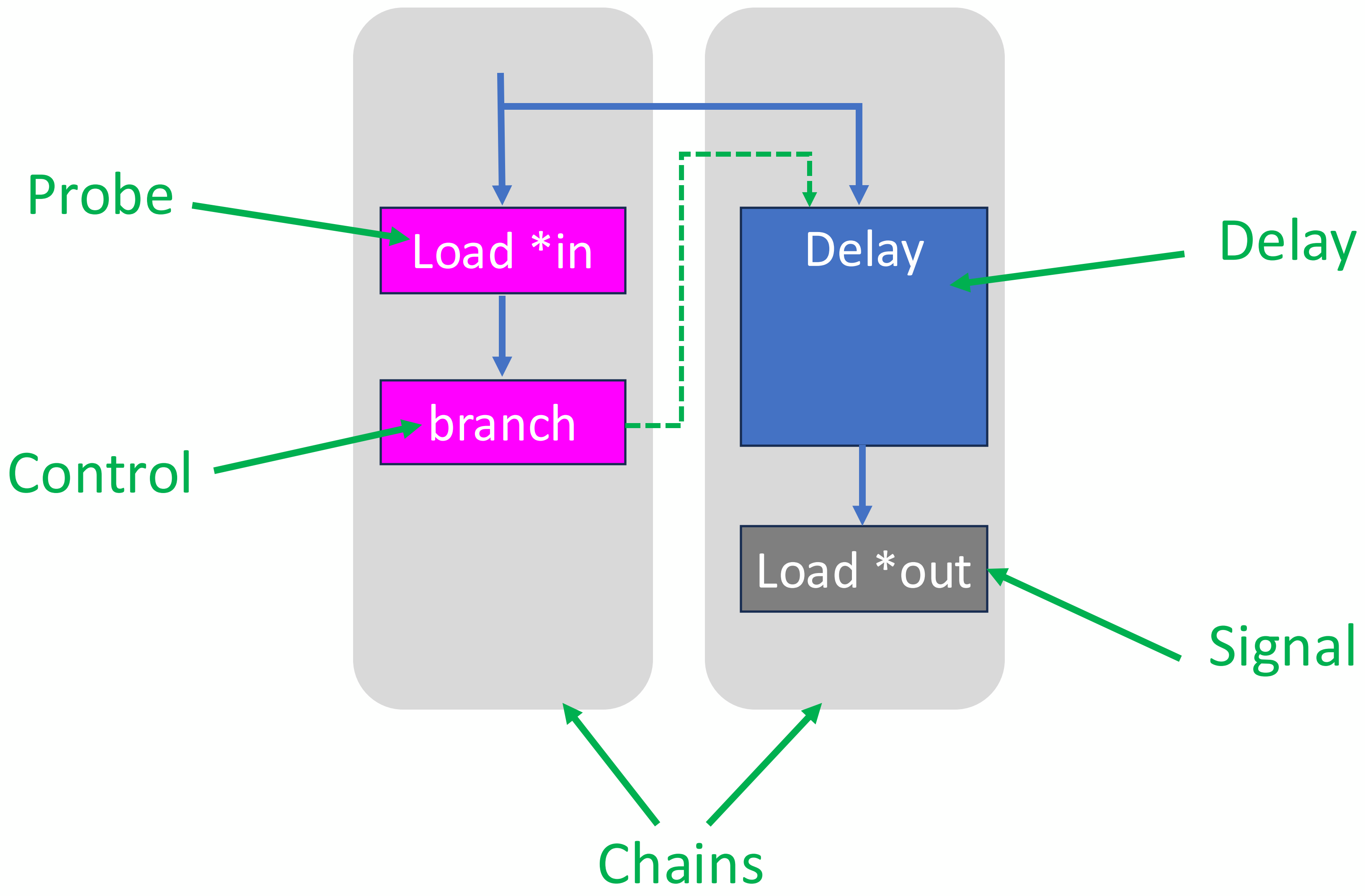
Thinking about this



Thinking about this



Thinking about this



Weird Prime+Scope

- Gates of Time (USENIX 2023) decouple Prime+Probe from time measurement: “Prime+Store”
  - Probe using NAND gate and store in cache
  - Measure cache state later

- **First Step:** Can apply to Prime+Scope

Using Prime+Store

- Using optimized gate construction: 48 cycles.
- 48 < 70
- Still very slow.

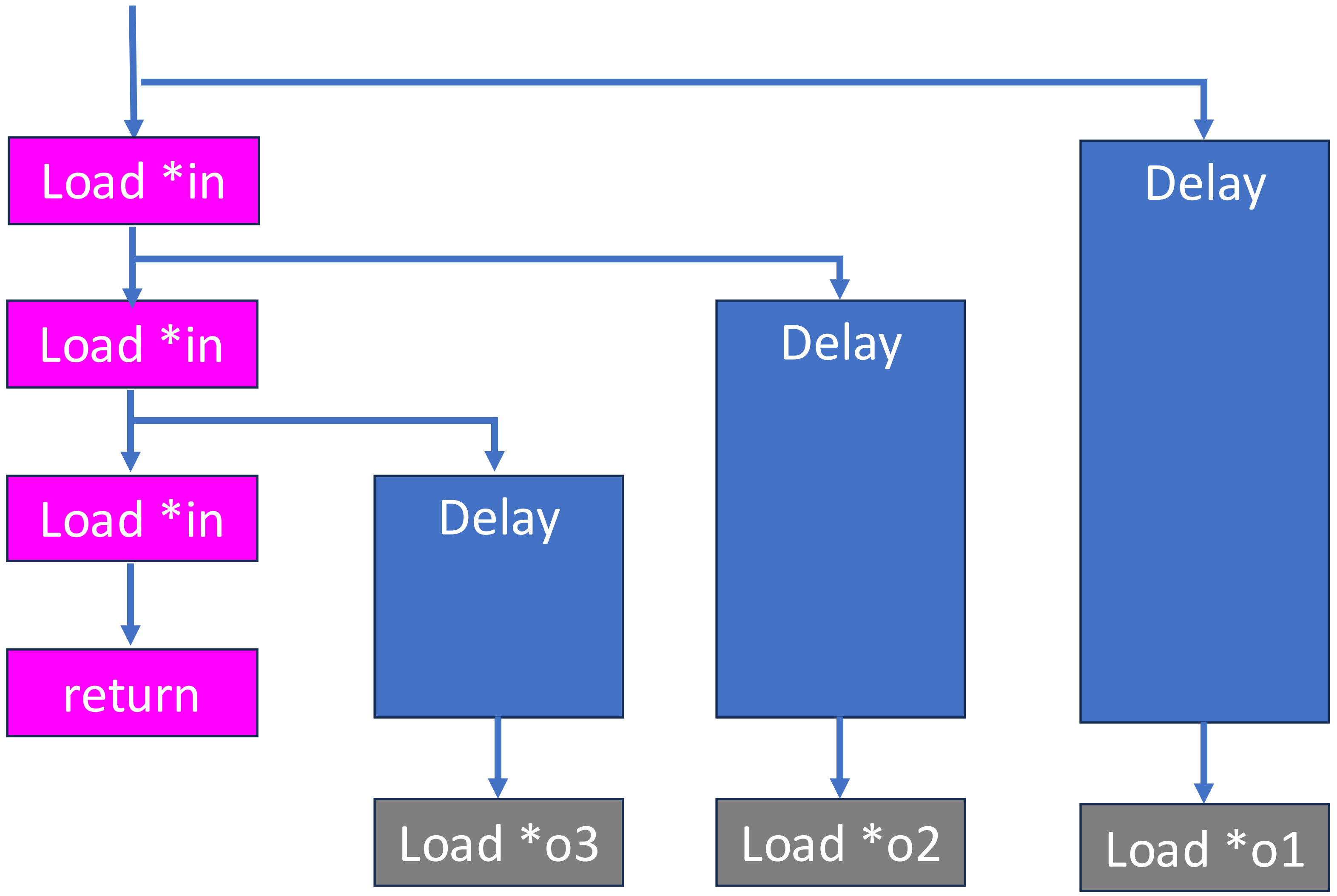
Multiple Probes

- Large overhead per scope
  - Mostly unavoidable
  - Misspeculation alone costs 19 cycles

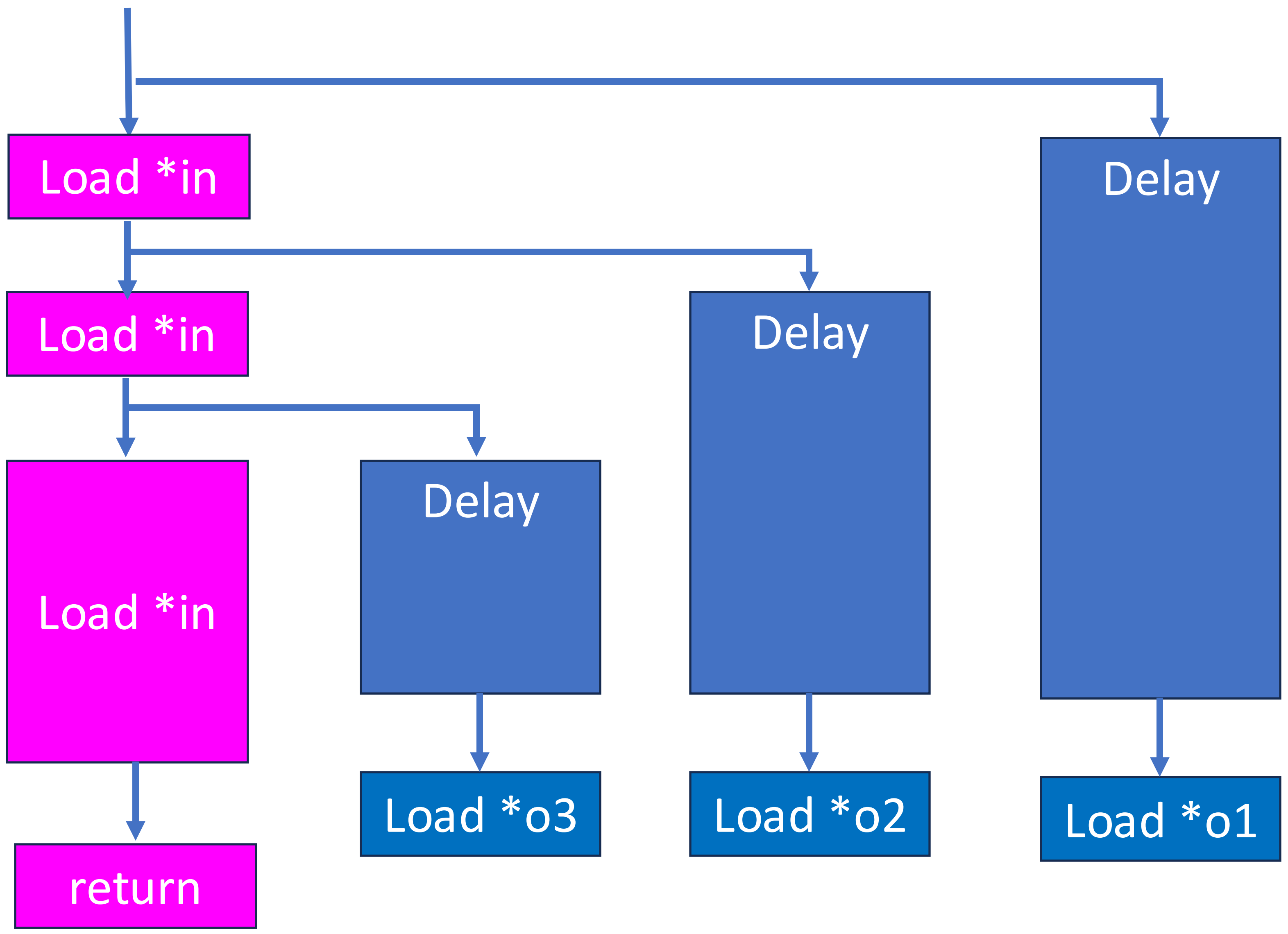
- Can we amortize?
  - Want multiple probes per gate

Tapped multi-probed gates

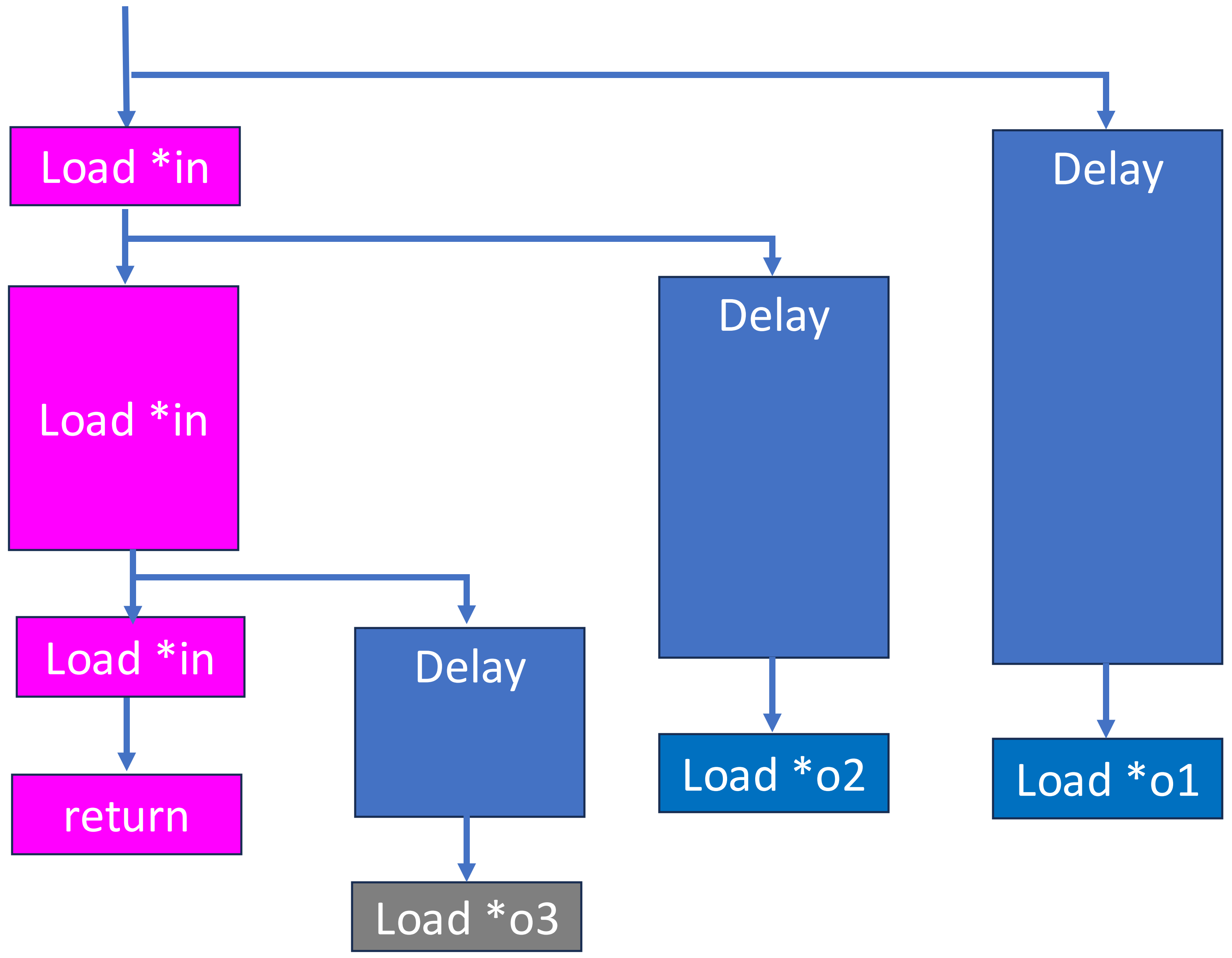
Tapped multi-probed gates



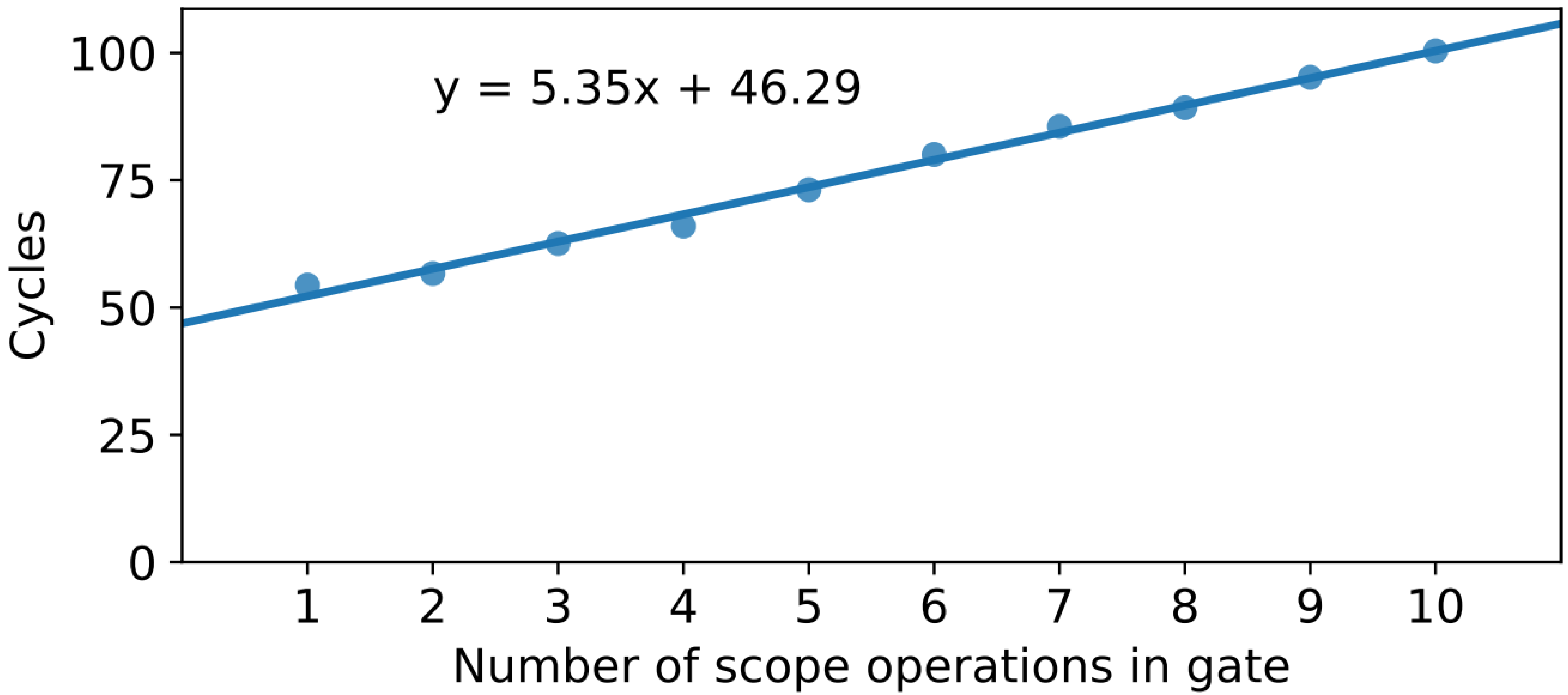
Tapped multi-probed gates



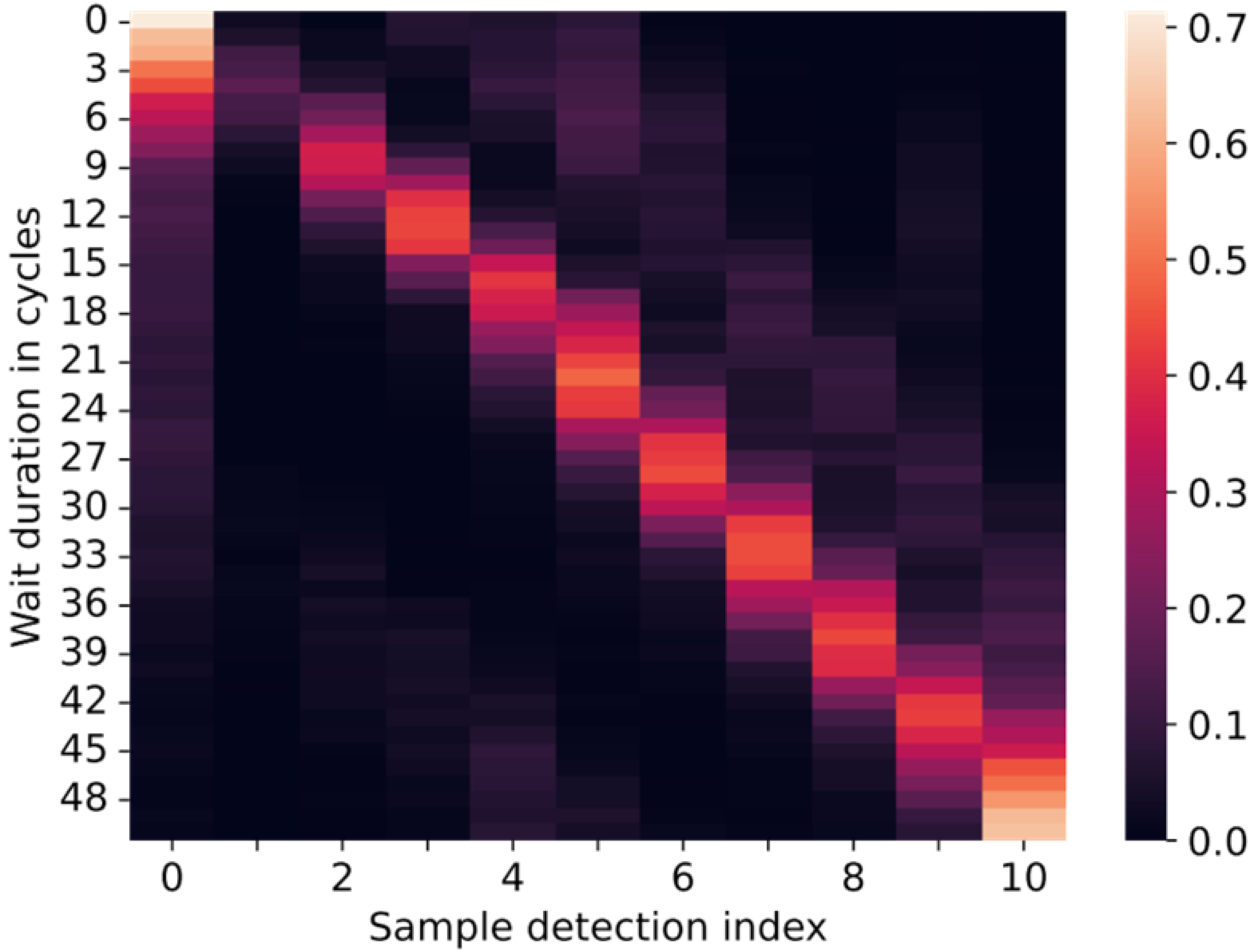
Tapped multi-probed gates



Gate operation time



Gate resolution



Results

Results

- For short runs, 5 cycles resolution

Results

- For short runs, 5 cycles resolution
- Sustained 10 cycles/probe, albeit non-uniform

Results

- For short runs, 5 cycles resolution
- Sustained 10 cycles/probe, albeit non-uniform
- Techniques for handling non-uniform probing

Attacking S-Box based AES

- S-Box based implementations are harder to attack
  - Only 4 cache lines, and more accesses

Attacking S-Box based AES

- S-Box based implementations are harder to attack
  - Only 4 cache lines, and more accesses

- Need to distinguish precise AES round

Attacking S-Box based AES

- S-Box based implementations are harder to attack
  - Only 4 cache lines, and more accesses
- Need to distinguish precise AES round
- Prior works need either
  - Non-trivial OS control
  - Utilize the deprecated Intel TSX
  - Modify the original code

Attacking S-Box based AES

- Full key recovery of AES128 using Spec-o-Scope

- Requires ≈10,000 traces
- Collected in less than 3 seconds

Summary

Summary

- Identify time measurement as rate limit

Summary

- Identify time measurement as rate limit
- Multi-probe weird gates allow to sample faster

Summary

- Identify time measurement as rate limit
- Multi-probe weird gates allow to sample faster
- Attacking S-Box AES is possible