# Speculative Execution and Cache Attacks

Chitchanok Chuengsatiansup

Hasso Plattner Institute, University of Potsdam, Germany

29 June – 3 July 2026

Cyber in Saint-Malo, France

# Computer Revolution

# Computer Revolution

| | | Processor speed | Memory latency |
|---|---|---|---|
| 1977 |  | 1 MHz | 500 ns |

# Computer Revolution

| | | Processor speed | Memory latency |
|---|---|---|---|
| 1977 |  | 1 MHz | 500 ns |
| 2020 |  | 20 × 3700 MHz | 50 ns |

# Bridging Speed Gap

# Bridging Speed Gap

- Utilize locality

# Bridging Speed Gap

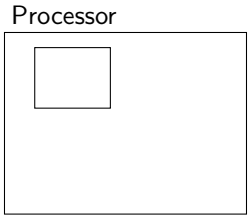
- Utilize locality
  - spatial: divide memory into lines

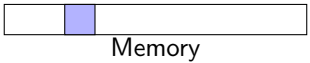
# Bridging Speed Gap

- Utilize locality
  - spatial: divide memory into lines
  - temporal: store recently used lines in cache

# Bridging Speed Gap

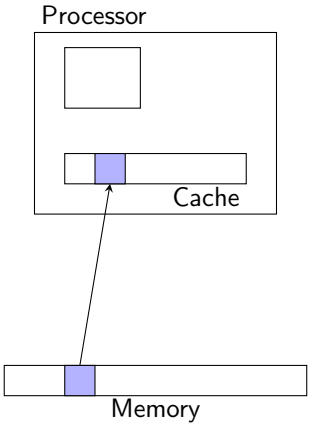
- Utilize locality
  - spatial: divide memory into lines
  - temporal: store recently used lines in cache



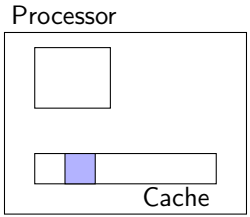
# Bridging Speed Gap

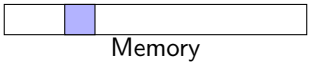
- Utilize locality
  - spatial: divide memory into lines
  - temporal: store recently used lines in cache



# Bridging Speed Gap

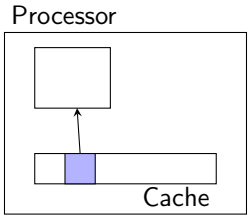
- Utilize locality
  - spatial: divide memory into lines
  - temporal: store recently used lines in cache
- Check if data is in cache

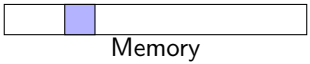


# Bridging Speed Gap

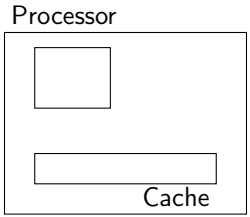
- Utilize locality
  - spatial: divide memory into lines
  - temporal: store recently used lines in cache
- Check if data is in cache
  - cache hit: serve data from cache

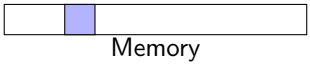


# Bridging Speed Gap

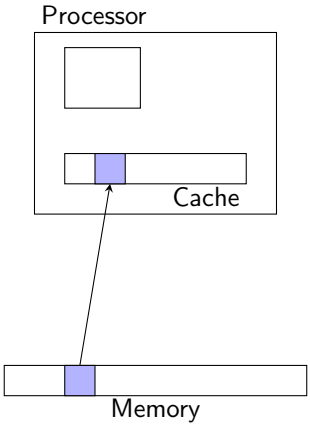
- Utilize locality
  - spatial: divide memory into lines
  - temporal: store recently used lines in cache
- Check if data is in cache
  - cache hit: serve data from cache
  - cache miss: get data from memory to cache



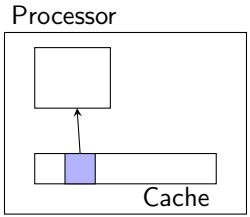
# Bridging Speed Gap

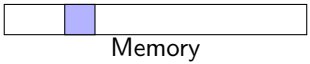
- Utilize locality
  - spatial: divide memory into lines
  - temporal: store recently used lines in cache
- Check if data is in cache
  - cache hit: serve data from cache
  - cache miss: get data from memory to cache



# Bridging Speed Gap

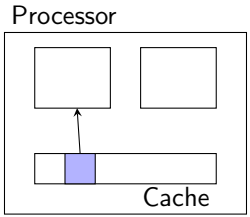
- Utilize locality
  - spatial: divide memory into lines
  - temporal: store recently used lines in cache
- Check if data is in cache
  - cache hit: serve data from cache
  - cache miss: get data from memory to cache

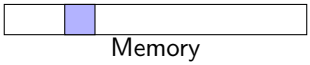


# Bridging Speed Gap

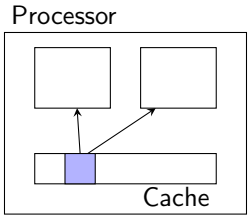
- Utilize locality
  - spatial: divide memory into lines
  - temporal: store recently used lines in cache
- Check if data is in cache
  - cache hit: serve data from cache
  - cache miss: get data from memory to cache
- Share cache
  - improve performance of multi-core processors

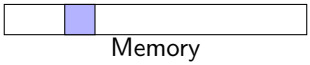


# Bridging Speed Gap

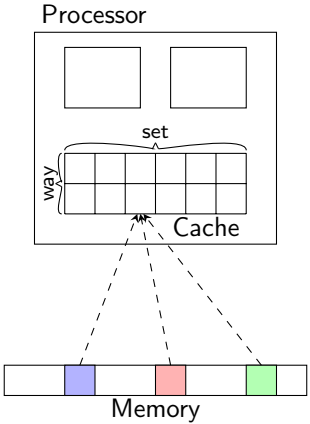
- Utilize locality
  - spatial: divide memory into lines
  - temporal: store recently used lines in cache
- Check if data is in cache
  - cache hit: serve data from cache
  - cache miss: get data from memory to cache
- Share cache
  - improve performance of multi-core processors



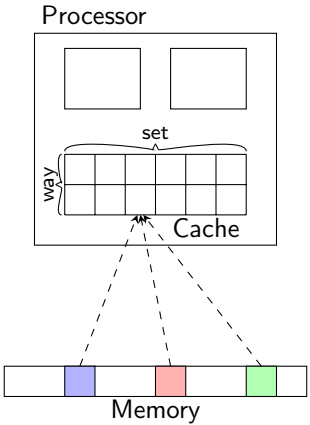
# Set-Associative Caches

- Memory lines map to cache sets



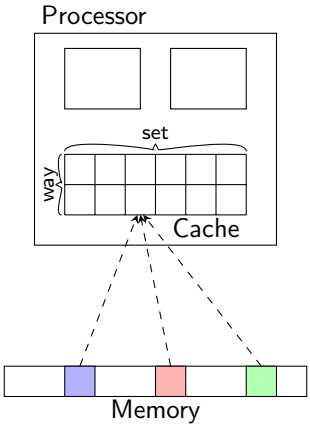
# Set-Associative Caches

- Memory lines map to cache sets
- Each line maps to one particular set and can be stored in any of its ways



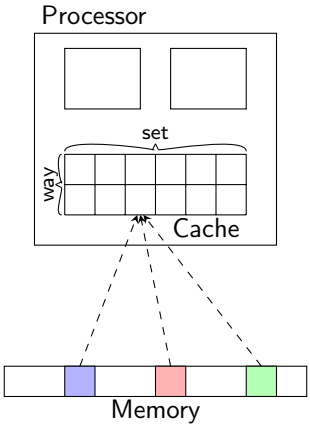
# Set-Associative Caches

- Memory lines map to cache sets
- Each line maps to one particular set and can be stored in any of its ways
- Multiple lines map to the same set



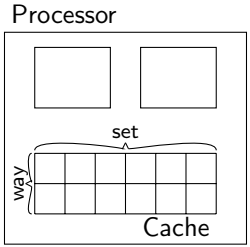
# Set-Associative Caches

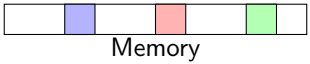
- Memory lines map to cache sets
- Each line maps to one particular set and can be stored in any of its ways
- Multiple lines map to the same set
- When cache miss and that set is full, one of the lines in that set is evicted



# Set-Associative Caches

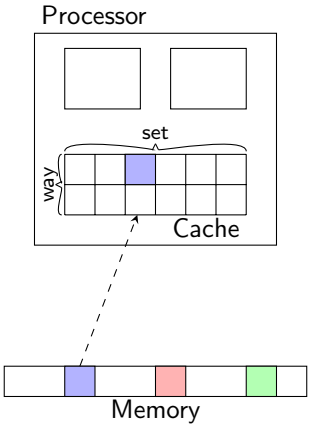
- Memory lines map to cache sets
- Each line maps to one particular set and can be stored in any of its ways
- Multiple lines map to the same set
- When cache miss and that set is full, one of the lines in that set is evicted



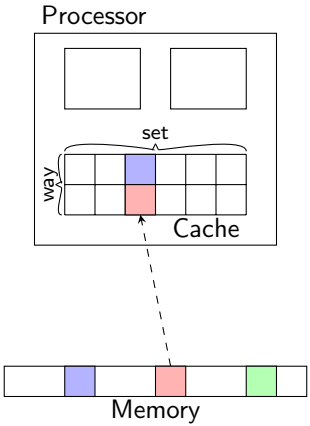
# Set-Associative Caches

- Memory lines map to cache sets
- Each line maps to one particular set and can be stored in any of its ways
- Multiple lines map to the same set
- When cache miss and that set is full, one of the lines in that set is evicted



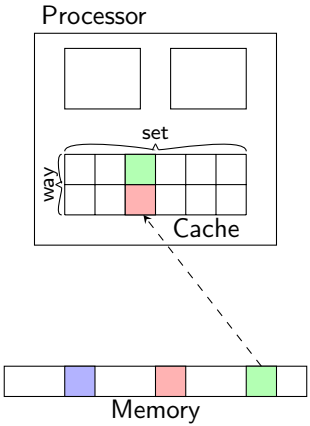
# Set-Associative Caches

- Memory lines map to cache sets
- Each line maps to one particular set and can be stored in any of its ways
- Multiple lines map to the same set
- When cache miss and that set is full, one of the lines in that set is evicted



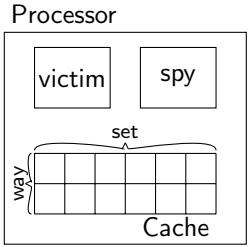
# Set-Associative Caches

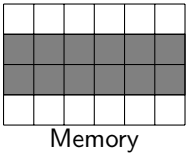
- Memory lines map to cache sets
- Each line maps to one particular set and can be stored in any of its ways
- Multiple lines map to the same set
- When cache miss and that set is full, one of the lines in that set is evicted



# Prime + Probe [OST05, Per05]

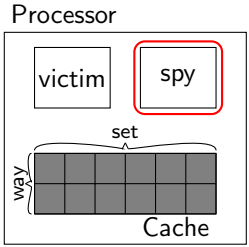
- (Allocate cache-sized memory buffer)
- **Prime**: access all lines in buffer
  → fill cache with attacker's data
- Wait (victim executes)
  → may evict some cache lines
- **Probe**: measure time to access buffer
  - slow → cache line evicted
  - fast → no victim access

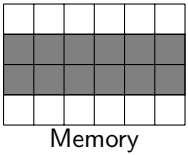


# Prime + Probe [OST05, Per05]

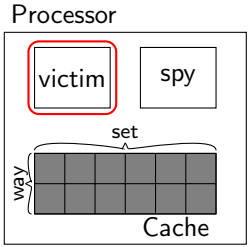
- (Allocate cache-sized memory buffer)
- **Prime**: access all lines in buffer
  → fill cache with attacker's data
- Wait (victim executes)
  → may evict some cache lines
- **Probe**: measure time to access buffer
  - slow → cache line evicted
  - fast → no victim access

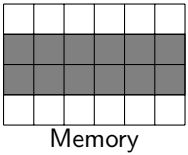


# Prime + Probe [OST05, Per05]

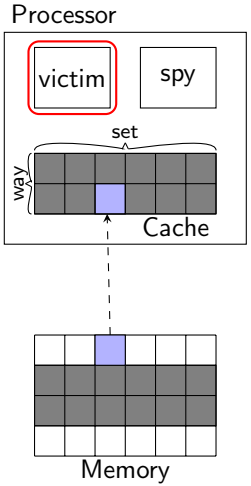
- (Allocate cache-sized memory buffer)
- **Prime**: access all lines in buffer
  → fill cache with attacker's data
- Wait (victim executes)
  → may evict some cache lines
- **Probe**: measure time to access buffer
  - slow → cache line evicted
  - fast → no victim access



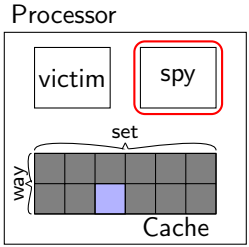
# Prime + Probe [OST05, Per05]

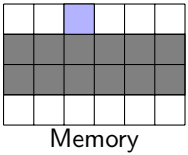
- (Allocate cache-sized memory buffer)
- **Prime**: access all lines in buffer
  → fill cache with attacker's data
- Wait (victim executes)
  → may evict some cache lines
- **Probe**: measure time to access buffer
  - slow → cache line evicted
  - fast → no victim access



# Prime + Probe [OST05, Per05]

- (Allocate cache-sized memory buffer)
- **Prime**: access all lines in buffer
  → fill cache with attacker's data
- Wait (victim executes)
  → may evict some cache lines
- **Probe**: measure time to access buffer
  - slow → cache line evicted
  - fast → no victim access



# Logical State of Cache

# Logical State of Cache

- Associate a logical state with memory addresses

# Logical State of Cache

- Associate a logical state with memory addresses
  - TRUE: address is cached
  - FALSE: address is **not** cached

# Logical State of Cache

- Associate a logical state with memory addresses
  - TRUE: address is cached
  - FALSE: address is **not** cached
- Flushing sets a state to FALSE

# Logical State of Cache

- Associate a logical state with memory addresses
  - TRUE: address is cached
  - FALSE: address is **not** cached
- Flushing sets a state to FALSE
- Accessing memory sets a state to TRUE

# Logical State of Cache

- Associate a logical state with memory addresses
  - TRUE: address is cached
  - FALSE: address is **not** cached
- Flushing sets a state to FALSE
- Accessing memory sets a state to TRUE
  - may also set another to FALSE

# Logical State of Cache

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

# Conditional Access

# Conditional Access

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

```
if (*in == 99)
  return
a = *out
```

# Conditional Access

- What is the cache state of `*out` after execution?
- TRUE if `*in != 99`

```
if (*in == 99)
  return
a = *out
```

# Conditional Access

- What is the cache state of `*out` after execution?
- TRUE if `*in != 99`
- What if `*in == 99`?

```
if (*in == 99)
  return
a = *out
```

# Speculative Execution

# Speculative Execution

- Evaluation of branch conditions can take time

# Speculative Execution

- Evaluation of branch conditions can take time
- The CPU predicts future execution

# Speculative Execution

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

# Speculative Execution

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

# Speculative Execution

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

# Speculative Execution

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

```
if (index < array.length){
  x = array[index];
  y = array2[x * 4096];
}
```

# Speculative Execution

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

```
if (index < array.length){
    x = array[index];
    y = array2[x * 4096];
}
```

can execute even condition is `FALSE`

# Conditional Speculative Execution

# Conditional Speculative Execution

- Speculation terminates when condition is resolved

# Conditional Speculative Execution

- Speculation terminates when condition is resolved

(What if `*in == 99`?)

```
if (*in == 99)
  return
a = *out
```

# Conditional Speculative Execution

- Speculation terminates when condition is resolved

  (What if `*in == 99`?)
- `*in` in cache
  - Condition resolves fast
  - `*out` is **not** accessed

```
if (*in == 99)
  return
a = *out
```

# Conditional Speculative Execution

- Speculation terminates when condition is resolved

  (What if `*in == 99`?)
- `*in` in cache
  - Condition resolves fast
  - `*out` is **not** accessed
- `*in` **not** in cache
  - Condition resolves slow
  - `*out` is accessed

```
if (*in == 99)
  return
a = *out
```

# Conditional Speculative Execution

- Speculation terminates when condition is resolved

(What if `*in == 99`?)

- `*in` in cache
  - Condition resolves fast
  - `*out` is **not** accessed
- `*in` **not** in cache
  - Condition resolves slow
  - `*out` is accessed

```
if (*in == 99)
  return
a = *out
```

| `*in` | `*out` |
|---|---|
| TRUE | FALSE |
| FALSE | TRUE |

# Conditional Speculative Execution

- Speculation terminates when condition is resolved

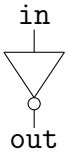
(What if `*in == 99`?)

- `*in` in cache
  - Condition resolves fast
  - `*out` is **not** accessed
- `*in` **not** in cache
  - Condition resolves slow
  - `*out` is accessed

```
if (*in == 99)
  return
a = *out
```

| *in | *out |
|---|---|
| TRUE | FALSE |
| FALSE | TRUE |

`out = NOT(in)`



# Other Functions

# Other Functions

```
if (*in1 == 99)
  return
if (*in2 == 99)
  return
a = *out
```

# Other Functions

```
if (*in1 == 99)
  return
if (*in2 == 99)
  return
a = *out
```

| *in1 | *in2 | *out |
|---|---|---|
| FALSE | FALSE | |
| FALSE | TRUE | |
| TRUE | FALSE | |
| TRUE | TRUE | |

# Other Functions

```
if (*in1 == 99)
  return
if (*in2 == 99)
  return
a = *out
```

| *in1 | *in2 | *out |
|---|---|---|
| FALSE | FALSE | TRUE |
| FALSE | TRUE | |
| TRUE | FALSE | |
| TRUE | TRUE | |

# Other Functions

```
if (*in1 == 99)
  return
if (*in2 == 99)
  return
a = *out
```

| *in1 | *in2 | *out |
|---|---|---|
| FALSE | FALSE | TRUE |
| FALSE | TRUE | FALSE |
| TRUE | FALSE | |
| TRUE | TRUE | |

# Other Functions

```
if (*in1 == 99)
  return
if (*in2 == 99)
  return
a = *out
```

| *in1 | *in2 | *out |
|---|---|---|
| FALSE | FALSE | TRUE |
| FALSE | TRUE | FALSE |
| TRUE | FALSE | FALSE |
| TRUE | TRUE | |

# Other Functions

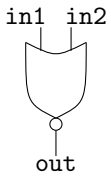
```
if (*in1 == 99)
  return
if (*in2 == 99)
  return
a = *out
```

| *in1 | *in2 | *out |
|---|---|---|
| FALSE | FALSE | TRUE |
| FALSE | TRUE | FALSE |
| TRUE | FALSE | FALSE |
| TRUE | TRUE | FALSE |

# Other Functions

```
if (*in1 == 99)
  return
if (*in2 == 99)
  return
a = *out
```

| *in1 | *in2 | *out |
|---|---|---|
| FALSE | FALSE | TRUE |
| FALSE | TRUE | FALSE |
| TRUE | FALSE | FALSE |
| TRUE | TRUE | FALSE |



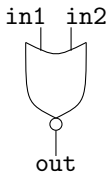

out = NOR(in1,in2)

# Other Functions

```
if (*in1 == 99)
  return
if (*in2 == 99)
  return
a = *out
```

```
if (*in1 + *in2 == 99)
  return
a = *out
```

| *in1 | *in2 | *out |
|---|---|---|
| FALSE | FALSE | TRUE |
| FALSE | TRUE | FALSE |
| TRUE | FALSE | FALSE |
| TRUE | TRUE | FALSE |



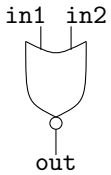

out = **NOR**(in1,in2)

# Other Functions

```
if (*in1 == 99)
  return
if (*in2 == 99)
  return
a = *out
```

```
if (*in1 + *in2 == 99)
  return
a = *out
```

| *in1 | *in2 | *out |
|---|---|---|
| FALSE | FALSE | TRUE |
| FALSE | TRUE | FALSE |
| TRUE | FALSE | FALSE |
| TRUE | TRUE | FALSE |




| *in1 | *in2 | *out |
|---|---|---|
| FALSE | FALSE | |
| FALSE | TRUE | |
| TRUE | FALSE | |
| TRUE | TRUE | |

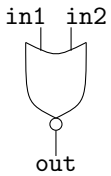
out = **NOR**(in1,in2)

# Other Functions

```
if (*in1 == 99)
  return
if (*in2 == 99)
  return
a = *out
```

```
if (*in1 + *in2 == 99)
  return
a = *out
```

| *in1 | *in2 | *out |
|---|---|---|
| FALSE | FALSE | TRUE |
| FALSE | TRUE | FALSE |
| TRUE | FALSE | FALSE |
| TRUE | TRUE | FALSE |




| *in1 | *in2 | *out |
|---|---|---|
| FALSE | FALSE | TRUE |
| FALSE | TRUE | |
| TRUE | FALSE | |
| TRUE | TRUE | |

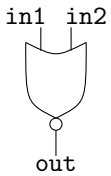
out = **NOR**(in1,in2)

# Other Functions

```
if (*in1 == 99)
  return
if (*in2 == 99)
  return
a = *out
```

```
if (*in1 + *in2 == 99)
  return
a = *out
```

| *in1 | *in2 | *out |
|---|---|---|
| FALSE | FALSE | TRUE |
| FALSE | TRUE | FALSE |
| TRUE | FALSE | FALSE |
| TRUE | TRUE | FALSE |




| *in1 | *in2 | *out |
|---|---|---|
| FALSE | FALSE | TRUE |
| FALSE | TRUE | TRUE |
| TRUE | FALSE | |
| TRUE | TRUE | |

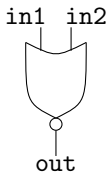
out = NOR(in1,in2)

# Other Functions

```
if (*in1 == 99)
  return
if (*in2 == 99)
  return
a = *out
```

```
if (*in1 + *in2 == 99)
  return
a = *out
```

| *in1 | *in2 | *out |
|---|---|---|
| FALSE | FALSE | TRUE |
| FALSE | TRUE | FALSE |
| TRUE | FALSE | FALSE |
| TRUE | TRUE | FALSE |




out = NOR(in1,in2)

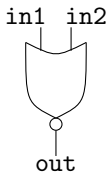
| *in1 | *in2 | *out |
|---|---|---|
| FALSE | FALSE | TRUE |
| FALSE | TRUE | TRUE |
| TRUE | FALSE | TRUE |
| TRUE | TRUE | |

# Other Functions

```
if (*in1 == 99)
  return
if (*in2 == 99)
  return
a = *out
```

```
if (*in1 + *in2 == 99)
  return
a = *out
```

| *in1 | *in2 | *out |
|---|---|---|
| FALSE | FALSE | TRUE |
| FALSE | TRUE | FALSE |
| TRUE | FALSE | FALSE |
| TRUE | TRUE | FALSE |



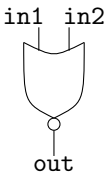

out = NOR(in1,in2)

| *in1 | *in2 | *out |
|---|---|---|
| FALSE | FALSE | TRUE |
| FALSE | TRUE | TRUE |
| TRUE | FALSE | TRUE |
| TRUE | TRUE | FALSE |

# Other Functions

```
if (*in1 == 99)
  return
if (*in2 == 99)
  return
a = *out
```

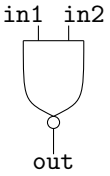
| *in1 | *in2 | *out |
|---|---|---|
| FALSE | FALSE | TRUE |
| FALSE | TRUE | FALSE |
| TRUE | FALSE | FALSE |
| TRUE | TRUE | FALSE |




out = **NOR**(in1,in2)

```
if (*in1 + *in2 == 99)
  return
a = *out
```

| *in1 | *in2 | *out |
|---|---|---|
| FALSE | FALSE | TRUE |
| FALSE | TRUE | TRUE |
| TRUE | FALSE | TRUE |
| TRUE | TRUE | FALSE |




out = **NAND**(in1,in2)

# Quiz Time

```
if (*in1 + *in2 == 99)
  return
if (*in2 + *in3 == 99)
  return
if (*in1 + *in3 == 99)
  return
a = *out
```

# Quiz Time

```
if (*in1 + *in2 == 99)
  return
if (*in2 + *in3 == 99)
  return
if (*in1 + *in3 == 99)
  return
a = *out
```

| *in1 | *in2 | *in3 | *out |
|---|---|---|---|
| FALSE | FALSE | FALSE | |
| FALSE | FALSE | TRUE | |
| FALSE | TRUE | FALSE | |
| FALSE | TRUE | TRUE | |
| TRUE | FALSE | FALSE | |
| TRUE | FALSE | TRUE | |
| TRUE | TRUE | FALSE | |
| TRUE | TRUE | TRUE | |

# Quiz Time

```
if (*in1 + *in2 == 99)
  return
if (*in2 + *in3 == 99)
  return
if (*in1 + *in3 == 99)
  return
a = *out
```

| *in1 | *in2 | *in3 | *out |
|---|---|---|---|
| FALSE | FALSE | FALSE | TRUE |
| FALSE | FALSE | TRUE | |
| FALSE | TRUE | FALSE | |
| FALSE | TRUE | TRUE | |
| TRUE | FALSE | FALSE | |
| TRUE | FALSE | TRUE | |
| TRUE | TRUE | FALSE | |
| TRUE | TRUE | TRUE | |

# Quiz Time

```
if (*in1 + *in2 == 99)
  return
if (*in2 + *in3 == 99)
  return
if (*in1 + *in3 == 99)
  return
a = *out
```

| *in1 | *in2 | *in3 | *out |
|---|---|---|---|
| FALSE | FALSE | FALSE | TRUE |
| FALSE | FALSE | TRUE | TRUE |
| FALSE | TRUE | FALSE | |
| FALSE | TRUE | TRUE | |
| TRUE | FALSE | FALSE | |
| TRUE | FALSE | TRUE | |
| TRUE | TRUE | FALSE | |
| TRUE | TRUE | TRUE | |

# Quiz Time

```
if (*in1 + *in2 == 99)
  return
if (*in2 + *in3 == 99)
  return
if (*in1 + *in3 == 99)
  return
a = *out
```

| *in1 | *in2 | *in3 | *out |
|---|---|---|---|
| FALSE | FALSE | FALSE | TRUE |
| FALSE | FALSE | TRUE | TRUE |
| FALSE | TRUE | FALSE | TRUE |
| FALSE | TRUE | TRUE | |
| TRUE | FALSE | FALSE | |
| TRUE | FALSE | TRUE | |
| TRUE | TRUE | FALSE | |
| TRUE | TRUE | TRUE | |

# Quiz Time

```
if (*in1 + *in2 == 99)
  return
if (*in2 + *in3 == 99)
  return
if (*in1 + *in3 == 99)
  return
a = *out
```

| *in1 | *in2 | *in3 | *out |
|---|---|---|---|
| FALSE | FALSE | FALSE | TRUE |
| FALSE | FALSE | TRUE | TRUE |
| FALSE | TRUE | FALSE | TRUE |
| FALSE | TRUE | TRUE | FALSE |
| TRUE | FALSE | FALSE | |
| TRUE | FALSE | TRUE | |
| TRUE | TRUE | FALSE | |
| TRUE | TRUE | TRUE | |

# Quiz Time

```
if (*in1 + *in2 == 99)
  return
if (*in2 + *in3 == 99)
  return
if (*in1 + *in3 == 99)
  return
a = *out
```

| *in1 | *in2 | *in3 | *out |
|---|---|---|---|
| FALSE | FALSE | FALSE | TRUE |
| FALSE | FALSE | TRUE | TRUE |
| FALSE | TRUE | FALSE | TRUE |
| FALSE | TRUE | TRUE | FALSE |
| TRUE | FALSE | FALSE | TRUE |
| TRUE | FALSE | TRUE | |
| TRUE | TRUE | FALSE | |
| TRUE | TRUE | TRUE | |

# Quiz Time

```
if (*in1 + *in2 == 99)
  return
if (*in2 + *in3 == 99)
  return
if (*in1 + *in3 == 99)
  return
a = *out
```

| *in1 | *in2 | *in3 | *out |
|---|---|---|---|
| FALSE | FALSE | FALSE | TRUE |
| FALSE | FALSE | TRUE | TRUE |
| FALSE | TRUE | FALSE | TRUE |
| FALSE | TRUE | TRUE | FALSE |
| TRUE | FALSE | FALSE | TRUE |
| TRUE | FALSE | TRUE | FALSE |
| TRUE | TRUE | FALSE | |
| TRUE | TRUE | TRUE | |

# Quiz Time

```
if (*in1 + *in2 == 99)
  return
if (*in2 + *in3 == 99)
  return
if (*in1 + *in3 == 99)
  return
a = *out
```

| *in1 | *in2 | *in3 | *out |
|---|---|---|---|
| FALSE | FALSE | FALSE | TRUE |
| FALSE | FALSE | TRUE | TRUE |
| FALSE | TRUE | FALSE | TRUE |
| FALSE | TRUE | TRUE | FALSE |
| TRUE | FALSE | FALSE | TRUE |
| TRUE | FALSE | TRUE | FALSE |
| TRUE | TRUE | FALSE | FALSE |
| TRUE | TRUE | TRUE | |

# Quiz Time

```
if (*in1 + *in2 == 99)
  return
if (*in2 + *in3 == 99)
  return
if (*in1 + *in3 == 99)
  return
a = *out
```

| *in1 | *in2 | *in3 | *out |
|---|---|---|---|
| FALSE | FALSE | FALSE | TRUE |
| FALSE | FALSE | TRUE | TRUE |
| FALSE | TRUE | FALSE | TRUE |
| FALSE | TRUE | TRUE | FALSE |
| TRUE | FALSE | FALSE | TRUE |
| TRUE | FALSE | TRUE | FALSE |
| TRUE | TRUE | FALSE | FALSE |
| TRUE | TRUE | TRUE | FALSE |

# Quiz Time

```
if (*in1 + *in2 == 99)
  return
if (*in2 + *in3 == 99)
  return
if (*in1 + *in3 == 99)
  return
a = *out
```

| *in1 | *in2 | *in3 | *out |
|---|---|---|---|
| FALSE | FALSE | FALSE | TRUE |
| FALSE | FALSE | TRUE | TRUE |
| FALSE | TRUE | FALSE | TRUE |
| FALSE | TRUE | TRUE | FALSE |
| TRUE | FALSE | FALSE | TRUE |
| TRUE | FALSE | TRUE | FALSE |
| TRUE | TRUE | FALSE | FALSE |
| TRUE | TRUE | TRUE | FALSE |

`out = NOT(major(in1,in2,in3))`

# Multiple Outputs

# Multiple Outputs

- Needed because gates destroy their inputs

# Multiple Outputs

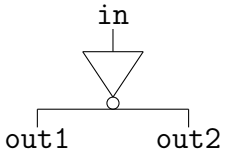
- Needed because gates destroy their inputs

```
if (*in == 99)
  return
a = *out1 + *out2
```

# Multiple Outputs

- Needed because gates destroy their inputs

```
if (*in == 99)
  return
a = *out1 + *out2
```



# Circuits

# Circuits

- 4-bit ALU
  - 1 258 gates, 84-95% accuracy

# Circuits

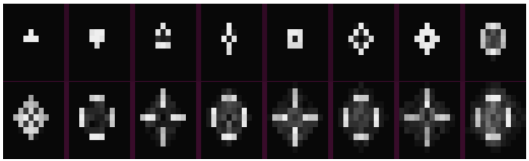
- 4-bit ALU
  - 1 258 gates, 84-95% accuracy
- SHA-1
  - one round: 2 208 gates, 95% accuracy

# Circuits

- 4-bit ALU
  - 1 258 gates, 84-95% accuracy
- SHA-1
  - one round: 2 208 gates, 95% accuracy
- Game of Life
  - 7 807 gates, 73% accuracy for one generation

# Circuits

- 4-bit ALU
  - 1 258 gates, 84-95% accuracy
- SHA-1
  - one round: 2 208 gates, 95% accuracy
- Game of Life
  - 7 807 gates, 73% accuracy for one generation

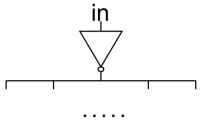


# Amplification

- NOT gate with large fan-out

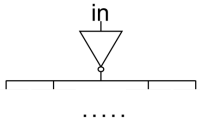
# Amplification

- NOT gate with large fan-out (e.g., Y = 8)



# Amplification

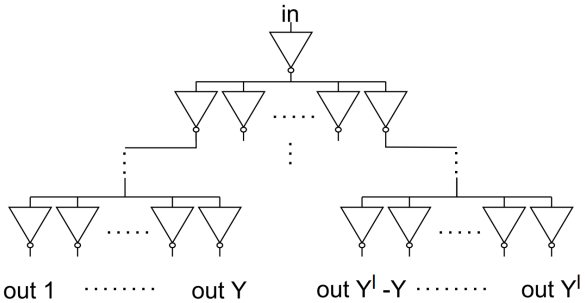
- NOT gate with large fan-out (e.g., Y = 8)




```
if (*in == 99)
  return
a = *out1 + *out2 + *out3 + *out4+ *out5 + *out6 + *out7 + *out8
```

# Amplification

- NOT gate with large fan-out (e.g., $Y = 8$)
  - $l = 2$ layers: 64
  - $l = 3$ layers: 512
  - $l = 4$ layers: 4 096

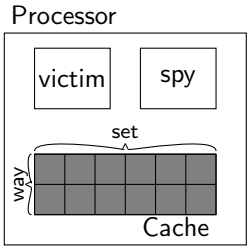


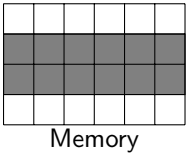

```
if (*in == 99)
  return
a = *out1 + *out2 + *out3 + *out4+ *out5 + *out6 + *out7 + *out8
```

# Prime+Store: High-Resolution Prime+Probe

# Prime+Store: High-Resolution Prime+Probe

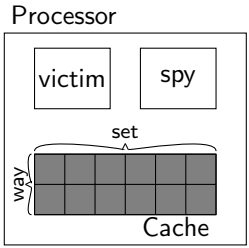
- Prime as in Prime+Probe

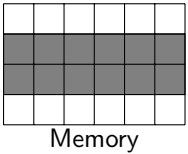


# Prime+Store: High-Resolution Prime+Probe

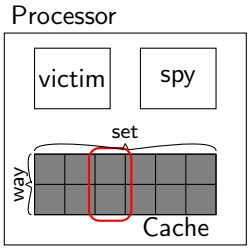
- Prime as in Prime+Probe
- Probe is basically a NAND gate

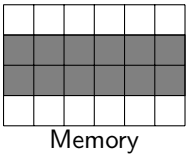


# Prime+Store: High-Resolution Prime+Probe

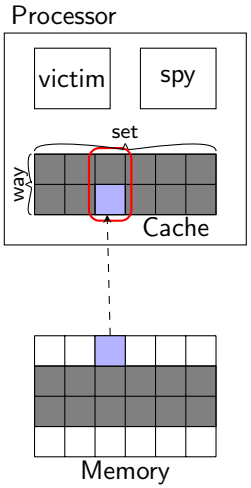
- Prime as in Prime+Probe
- Probe is basically a NAND gate
  - FALSE if all are cached



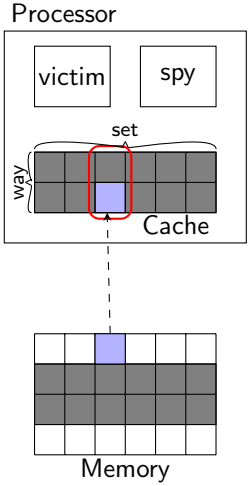
# Prime+Store: High-Resolution Prime+Probe

- Prime as in Prime+Probe
- Probe is basically a NAND gate
  - FALSE if all are cached
  - TRUE if some are not cached



# Prime+Store: High-Resolution Prime+Probe

- Prime as in Prime+Probe
- Probe is basically a NAND gate
  - FALSE if all are cached
  - TRUE if some are not cached
- Do multiple probes of the same cache set and store results.



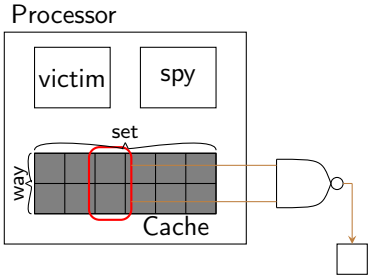
# Prime+Store: High-Resolution Prime+Probe (cont.)

# Prime+Store: High-Resolution Prime+Probe (cont.)

- Decouple probe from time measurement

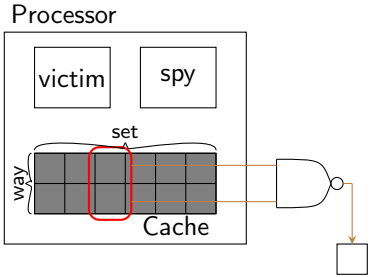
# Prime+Store: High-Resolution Prime+Probe (cont.)

- Decouple probe from time measurement



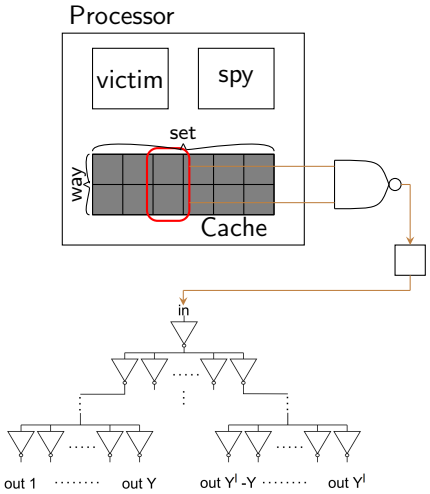
# Prime+Store: High-Resolution Prime+Probe (cont.)

- Decouple probe from time measurement
- Measure time with amplification tree



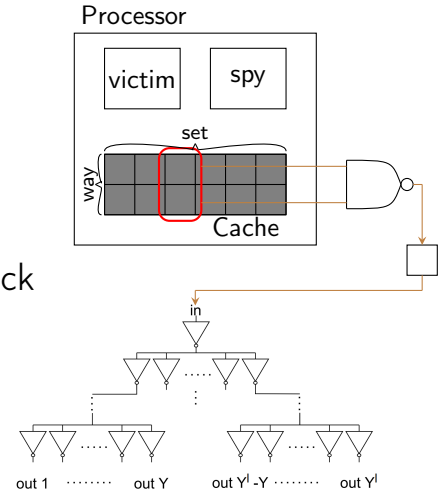
# Prime+Store: High-Resolution Prime+Probe (cont.)

- Decouple probe from time measurement
- Measure time with amplification tree



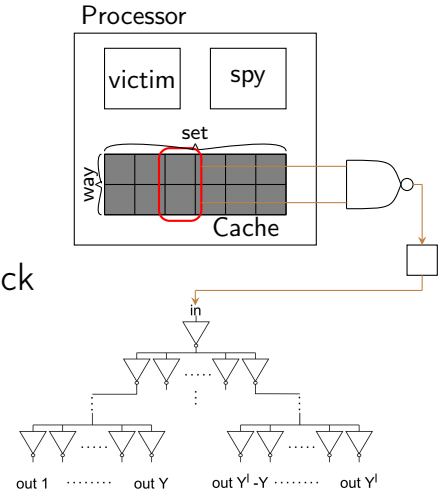
# Prime+Store: High-Resolution Prime+Probe (cont.)

- Decouple probe from time measurement
- Measure time with amplification tree
- Attack square-and-multiply with slow clock



# Prime+Store: High-Resolution Prime+Probe (cont.)

- Decouple probe from time measurement
- Measure time with amplification tree
- Attack square-and-multiply with slow clock
- Bypass countermeasure of reducing timer resolution



# Challenges in Finding Eviction Sets

# Challenges in Finding Eviction Sets

- Cache replacement policies

# Challenges in Finding Eviction Sets

- Cache replacement policies
  - e.g., LRU, pseudo-LRU, FIFO

# Challenges in Finding Eviction Sets

- Cache replacement policies
  - e.g., LRU, pseudo-LRU, FIFO
- Cache hierarchies

# Challenges in Finding Eviction Sets

- Cache replacement policies
  - e.g., LRU, pseudo-LRU, FIFO
- Cache hierarchies
  - e.g., inclusive, non-inclusive

# Challenges in Finding Eviction Sets

- Cache replacement policies
  - e.g., LRU, pseudo-LRU, FIFO
- Cache hierarchies
  - e.g., inclusive, non-inclusive
- Mapping memory blocks to cache sets

# Challenges in Finding Eviction Sets

- Cache replacement policies
  - e.g., LRU, pseudo-LRU, FIFO
- Cache hierarchies
  - e.g., inclusive, non-inclusive
- Mapping memory blocks to cache sets
  - e.g., use physical addresses to map to cache sets of LLC

# Challenges in Finding Eviction Sets

- Cache replacement policies
  - e.g., LRU, pseudo-LRU, FIFO
- Cache hierarchies
  - e.g., inclusive, non-inclusive
- Mapping memory blocks to cache sets
  - e.g., use physical addresses to map to cache sets of LLC
- Cache slicing

# Challenges in Finding Eviction Sets

- Cache replacement policies
  - e.g., LRU, pseudo-LRU, FIFO
- Cache hierarchies
  - e.g., inclusive, non-inclusive
- Mapping memory blocks to cache sets
  - e.g., use physical addresses to map to cache sets of LLC
- Cache slicing
  - LLC is partitioned into many slices

# Challenges in Finding Eviction Sets

- Cache replacement policies
  - e.g., LRU, pseudo-LRU, FIFO
- Cache hierarchies
  - e.g., inclusive, non-inclusive
- Mapping memory blocks to cache sets
  - e.g., use physical addresses to map to cache sets of LLC
- Cache slicing
  - LLC is partitioned into many slices
  - The slice is determined by undocumented function

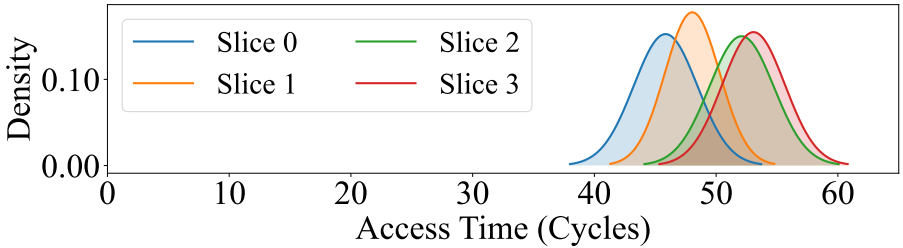
# Slice Access Latency

# Slice Access Latency

- Different LLC slices have different access time

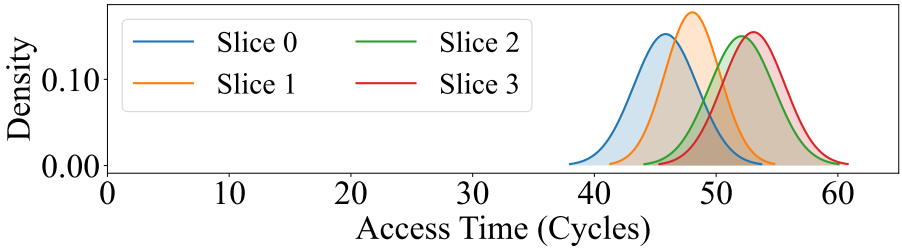
# Slice Access Latency

- Different LLC slices have different access time



# Slice Access Latency

- Different LLC slices have different access time
- Can we determine the slice index by the access time?



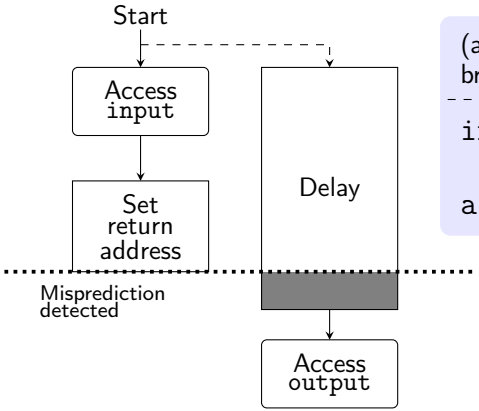
# Weird Gate Operation

# Weird Gate Operation

(assume `*in = 99`, branch mispredicted)

```
if (*in == 99)
  return
a = *out
```

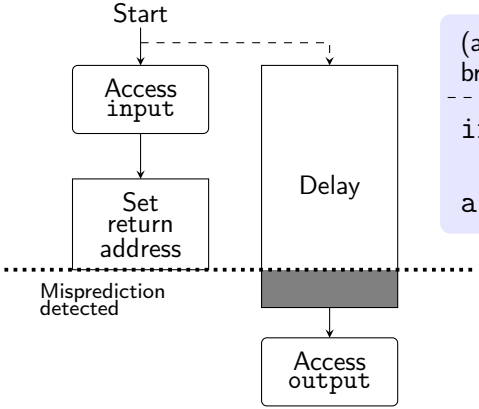
# Weird Gate Operation




(assume `*in = 99`, branch mispredicted)

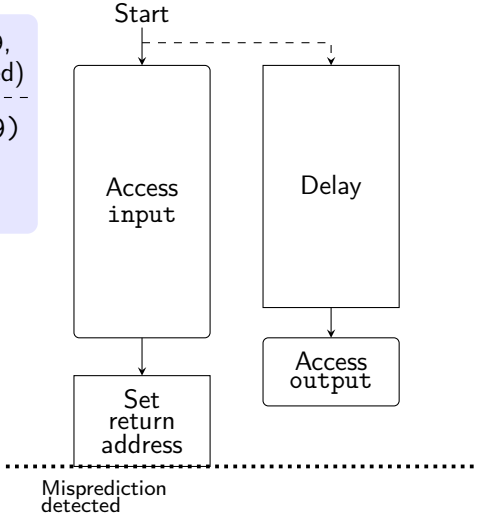
```
if (*in == 99)
  return
a = *out
```

# Weird Gate Operation

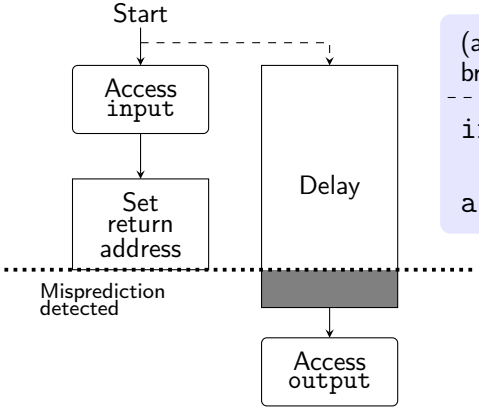



(assume *in = 99, branch mispredicted)

```
if (*in == 99)
  return
a = *out
```

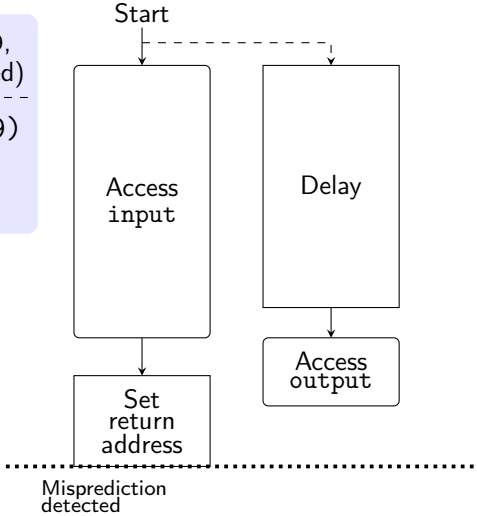


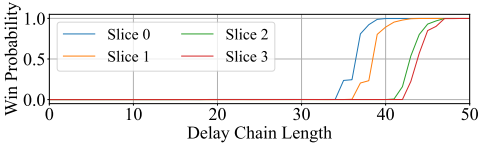
# Weird Gate Operation



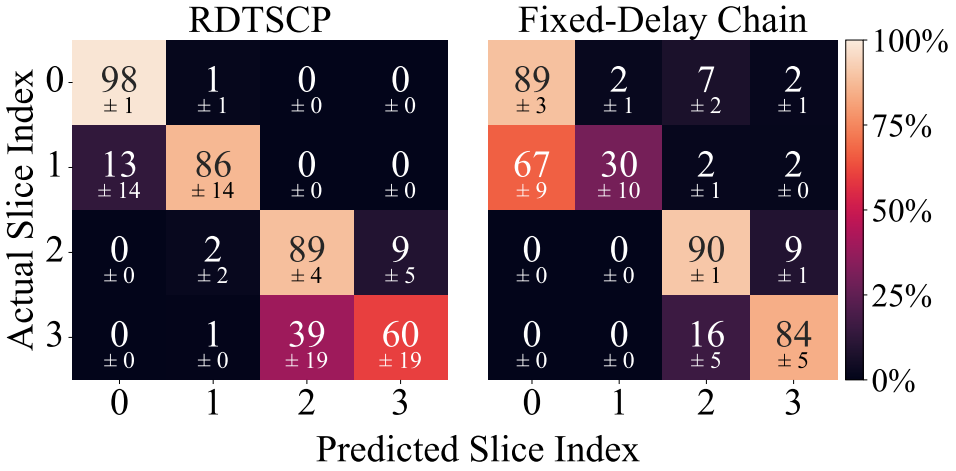

(assume *in = 99, branch mispredicted)

```
if (*in == 99)
   return
a = *out
```

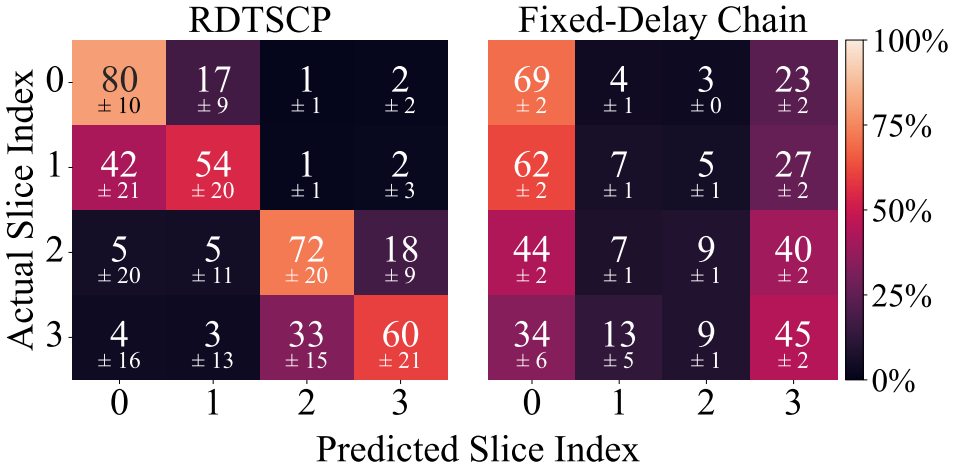


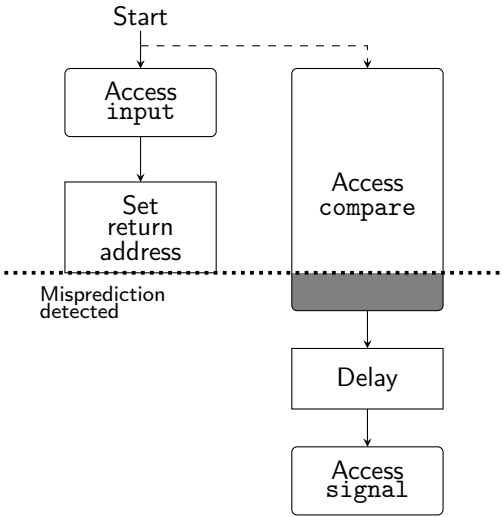
# Slice Classification: quiet system

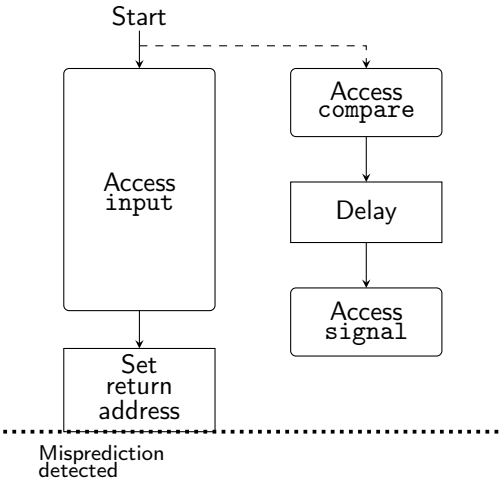


# Slice Classification: busy system

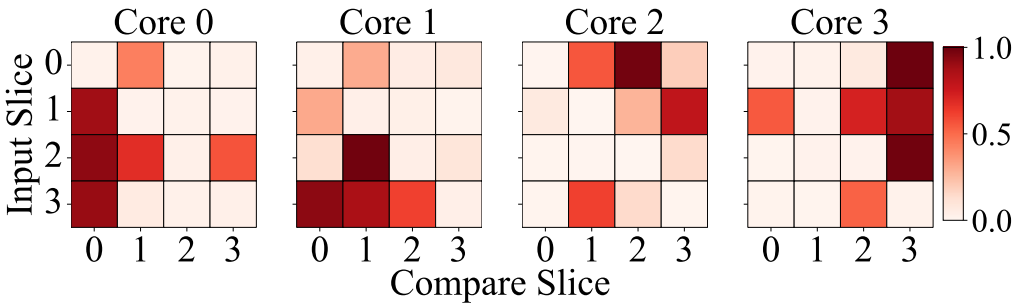


# Comparator Gate

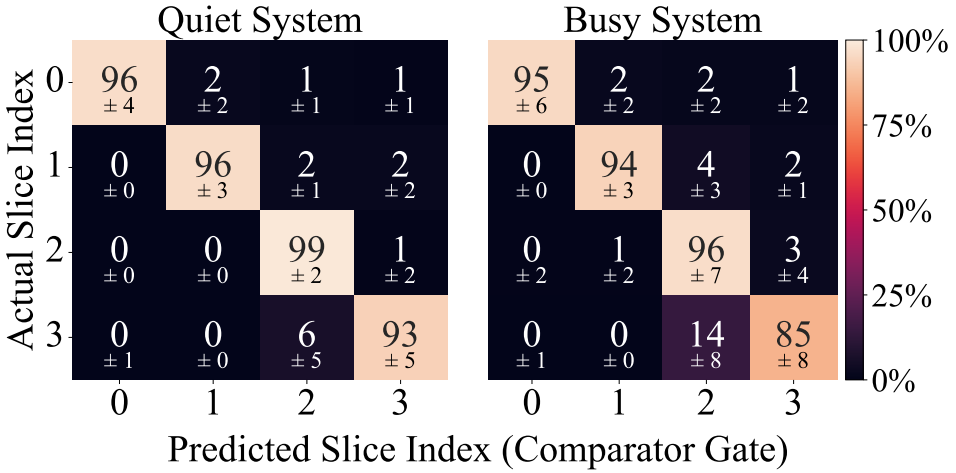


# Comparator Gate: signal access



# Slice Classification with Comparator Gate



# Summary

# Summary

- Conditional speculative execution allows constructing logic gates

# Summary

- Conditional speculative execution allows constructing logic gates
- Cache-state logic gates are Turing-complete

# Summary

- Conditional speculative execution allows constructing logic gates
- Cache-state logic gates are Turing-complete
- Nested large fan-out gates amplify cache access signal

# Summary

- Conditional speculative execution allows constructing logic gates
- Cache-state logic gates are Turing-complete
- Nested large fan-out gates amplify cache access signal
- Amplification enables detection with a slow clock

# Summary

- Conditional speculative execution allows constructing logic gates
- Cache-state logic gates are Turing-complete
- Nested large fan-out gates amplify cache access signal
- Amplification enables detection with a slow clock
- Prime+Store decouples sampling from clock

# Summary

- Conditional speculative execution allows constructing logic gates
- Cache-state logic gates are Turing-complete
- Nested large fan-out gates amplify cache access signal
- Amplification enables detection with a slow clock
- Prime+Store decouples sampling from clock
- Comparator gate allows more accurate slice index determination