Language Based Security (LBT)

Leaky Abstractions: When Hardware Undermines Language Security

Chiara Bodei, Gian-Luigi Ferrari

Lecture April 30 2025

When Hardware Breaks Language Guarantees

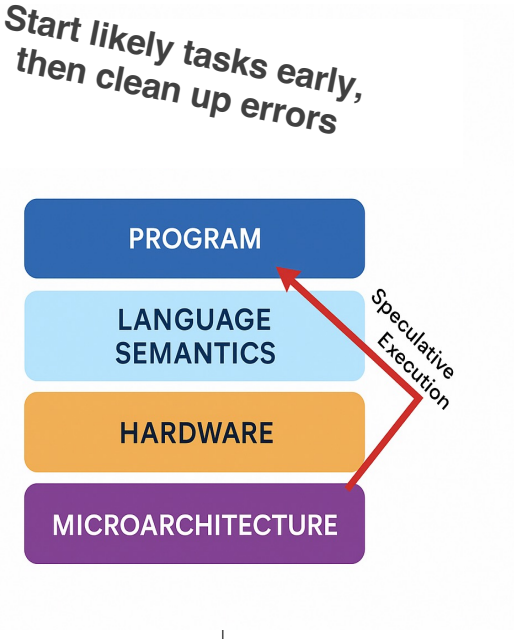
Language-based security aims to enforce properties like information flow control, non-interference, ecc.

- These guarantees rely on a semantic assumption:
  - the program does what it says — no more, no less

Speculative Execution

Speculative execution is a **CPU-level optimization**, where the processor predicts the outcome of a branch and executes instructions ahead of time to improve performance.

If the prediction is wrong, the processor discards the results and rolls back to the correct path



Speculative execution challenges semantic assumption:

- it can leak data without changing the program's output

- Not coding bugs, but microarchitectural side effects:
  - a semantic-level guarantee does not prevent hardware-level leakage

Speculative execution violates the semantic expectation
"if data is not assigned, it is not leaked"

Need for Extended Models in Language-Based Security

Speculative execution introduces security-relevant behavior that is not visible in standard program semantics.

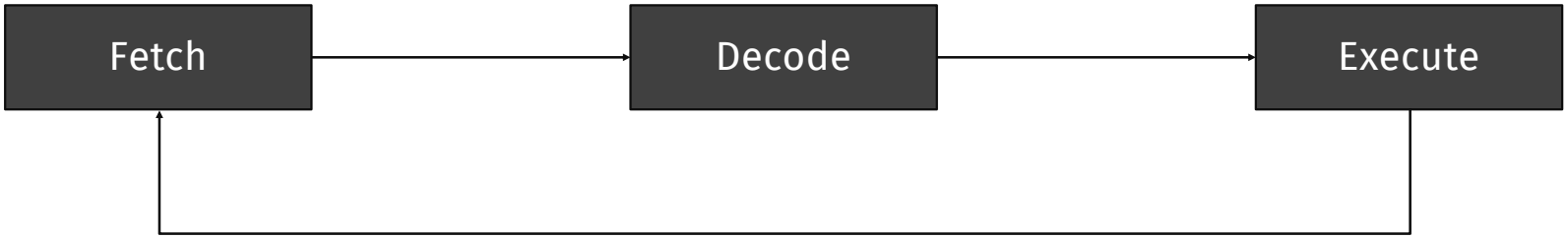
As a result, traditional notions like non-interference or constant-time execution — defined over “normal” execution traces — no longer suffice

To respond, the research community has developed extended models that explicitly incorporate speculative behaviors into the analysis

We will focus on secure compilation to account for speculative leakage

Computer architecture

Do you remember how processors work?



Problem:
not very efficient (≥1 cycle per stage)

No more easy gains from low-level physics!

Outline

- Memory Hierarchy and cache
- Cache Static Analysis
- Must Analysis
- May Analysis

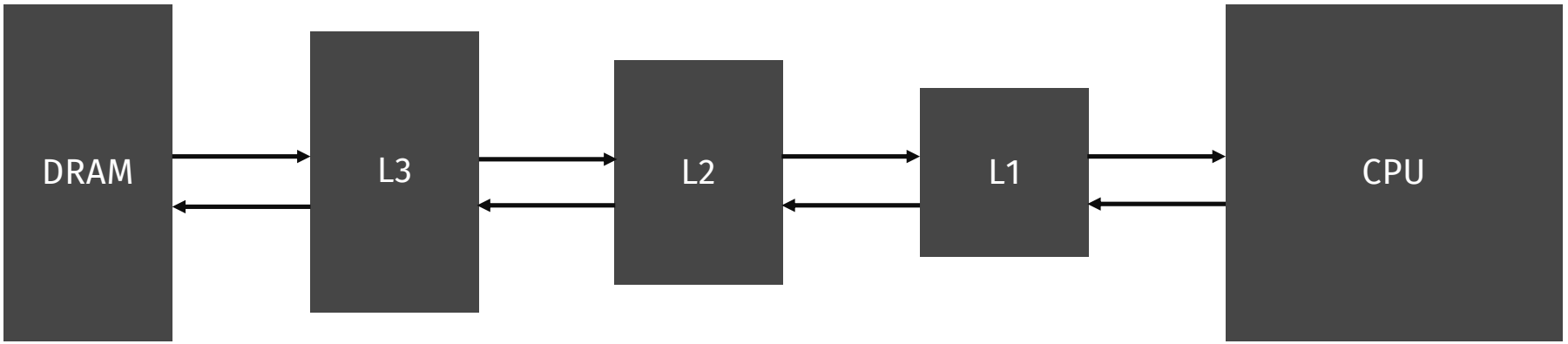
Computer architecture

- **First idea:** make cycles “shorter”, i.e., **increase** the processor’s clock
  - We can reach about 4 GHz (4 billion cycles per second)
  - **Very good**, but ~2004 we reached the peak: the clock speed has not increased
  - Also: power-consumption issues, too much heat, …
- People started thinking about **alternatives**!
  - **Idea:** make the average case faster
  - **Solutions:** pipelines, caches, speculation, multi-core, multi-thread, …

Caches

To bridge the **memory gap between the CPU and main memory speeds**

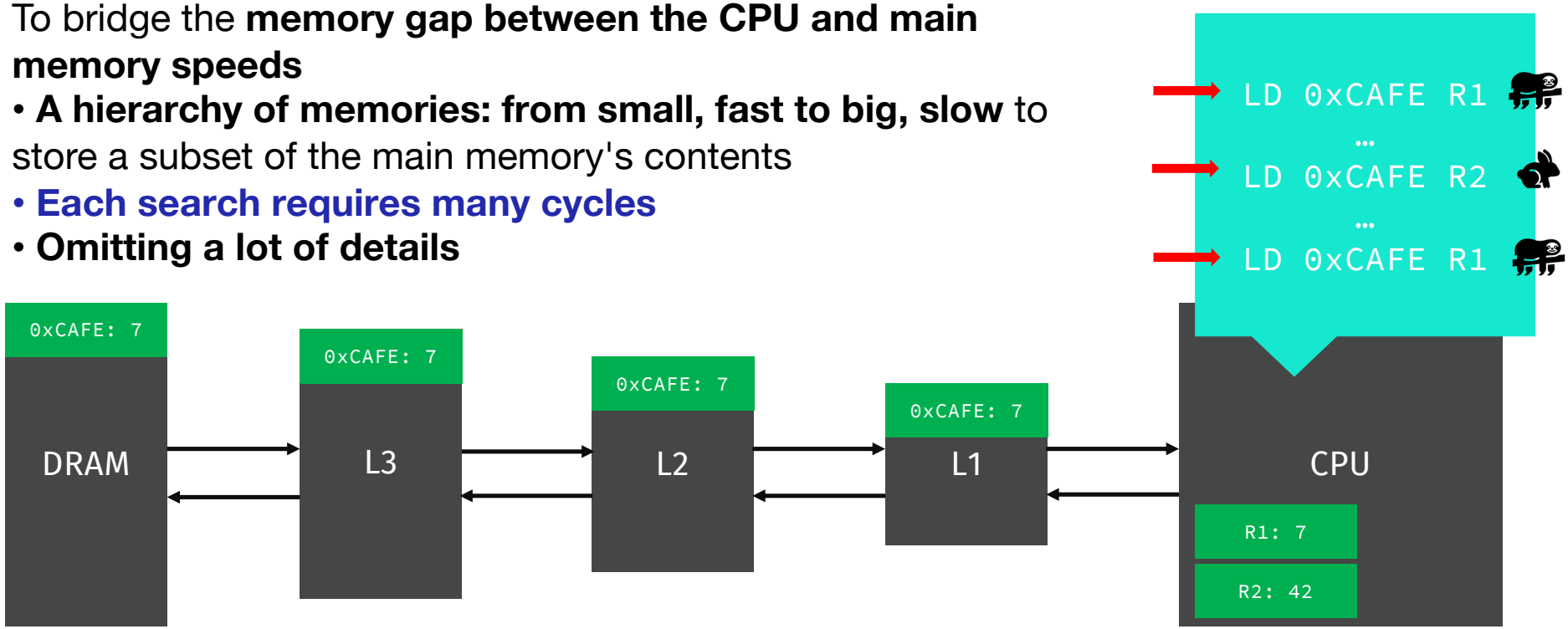
- **A hierarchy of memories: from small, fast to big, slow** to store a subset of the main memory's contents
- **Each search requires many cycles**
- **Omitting a lot of details**



Caches

To bridge the **memory gap between the CPU and main memory speeds**

- **A hierarchy of memories: from small, fast to big, slow** to store a subset of the main memory's contents
- **Each search requires many cycles**
- **Omitting a lot of details**



Caches are fast when data is already there, otherwise we need to wait for the DRAM

Why memory hierarchy?

CPUs have become much faster than main memory

Accessing memory is a major bottleneck

Solution: introduce faster intermediate storage levels (cache)

Cache hierarchy reduces average access time

Latency comparison

Component	Latency (approx.)
Registers	0.25–0.5 ns
L1 Cache	~1 ns
L2 Cache	~3–4 ns
L3 Cache	~10–15 ns
DRAM (Dynamic RAM)	~100 ns
SSD (solid-state drive)	~100 µs
HDD (Hard Drive)	~10 ms

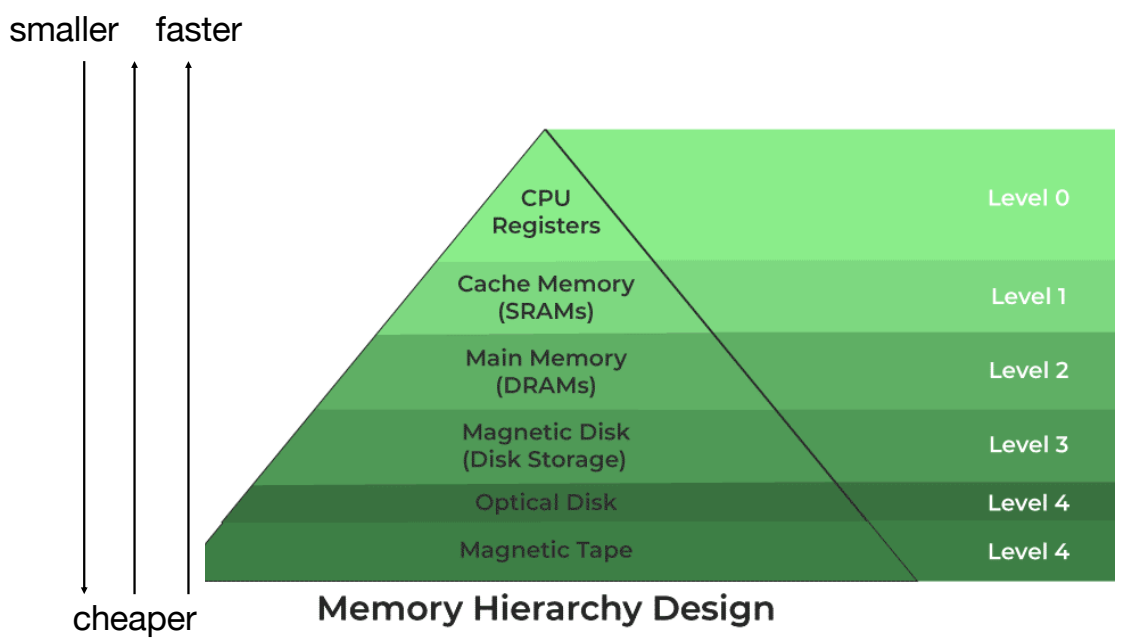
Memory hierarchy overview

Memory is organized in levels by:

- access time
- cost per byte
- capacity

Faster levels are smaller and closer to CPU

Slower levels provide large capacity



Why cache works: locality

The **principle of locality** works into two dimensions:

- **Temporal locality**: recently accessed data likely accessed again soon

- **Spatial locality**: nearby data likely accessed soon after

```
for (int i = 0; i < N; i++) sum += A[i];
```

Repeated access to sum

A[0]
A[1]
A[2]
A[3]
A[4]

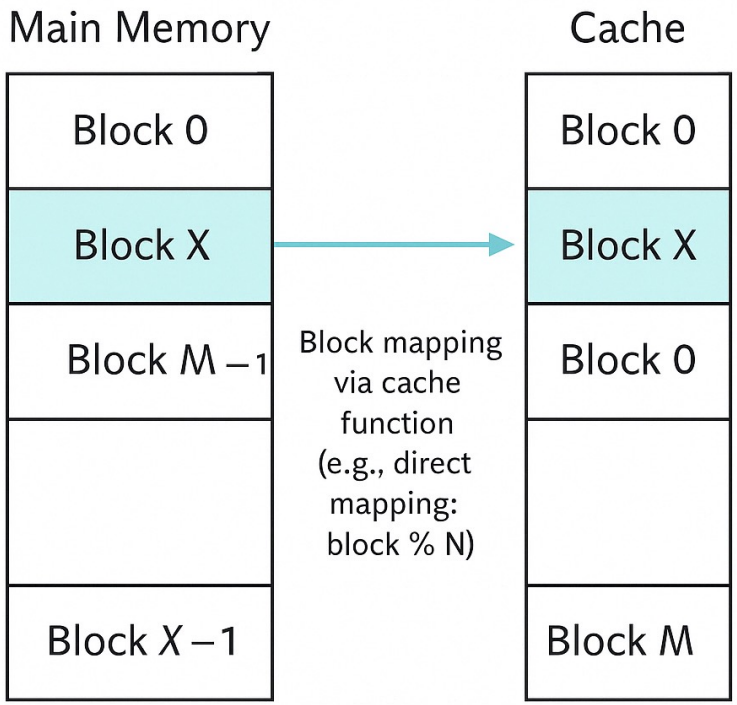
Memory blocks being read sequentially

Cache Structure and Memory Blocks

- Caches are organized in fixed-size blocks (**cache lines**), not individual addresses
- Cache stores memory blocks (lines) into cache lines
- Due to spatial locality, access to a memory location X implies the involvement of the entire block containing X

- A mapping function relates:

Main memory block numbers ⇔ Cache block numbers



Cache Structure and Memory Blocks

Whenever the processor generates an access to a memory location, it will first check the cache memory to see if it contains the desired data

- **Cache Hit** = required data is in the current level of cache: the data is returned and to the CPU, without involving the main memory
- **Cache Miss** = required data is not present in the current level of cache: the new block is brought from the main memory
  - if cache is full a block must be evicted and replaced

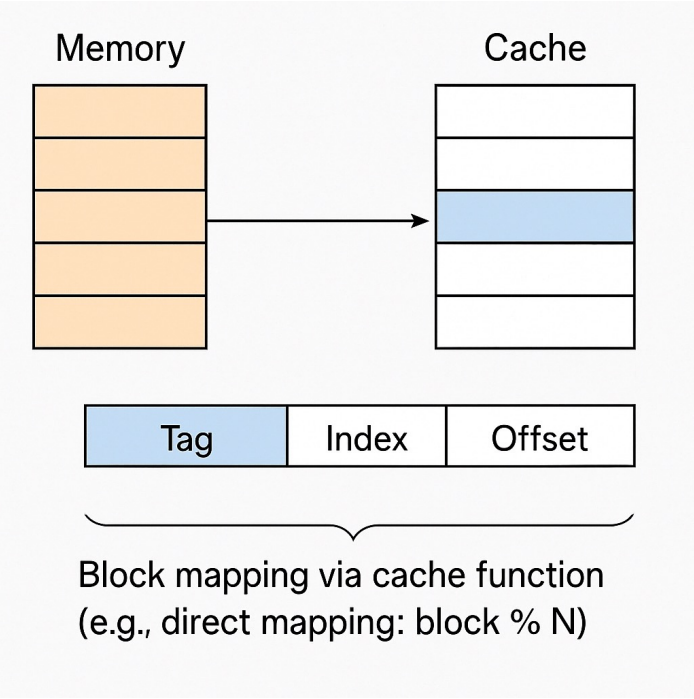
Cache write

In case of a **write** to cache operation there are two possible policies:

- **Write Back**: Update only cached copy. The location is marked as dirty to recall this misalignment. Only in the case of replacement, the entire block is written back

- **Write Through (Store Through)**: every write updates all levels of the hierarchy (cache levels and main memory) to keep them in sync. Worse performance but no writeback is needed

Identifying Blocks via Address Range



- Each main memory block maps to one unique cache line
- Multiple memory blocks → same cache line (conflict risk)
- Use a Tag to distinguish them.
- Memory addresses is split into: [Tag | Index | Offset]
  - Tag: identifies the block
  - Index: identifies the set/line
  - Offset: location inside block

Analogy: Airplane seats addressing [Class | Row | Position]

Mapping Techniques

To determine where blocks can be placed in the cache, there are three main methods:

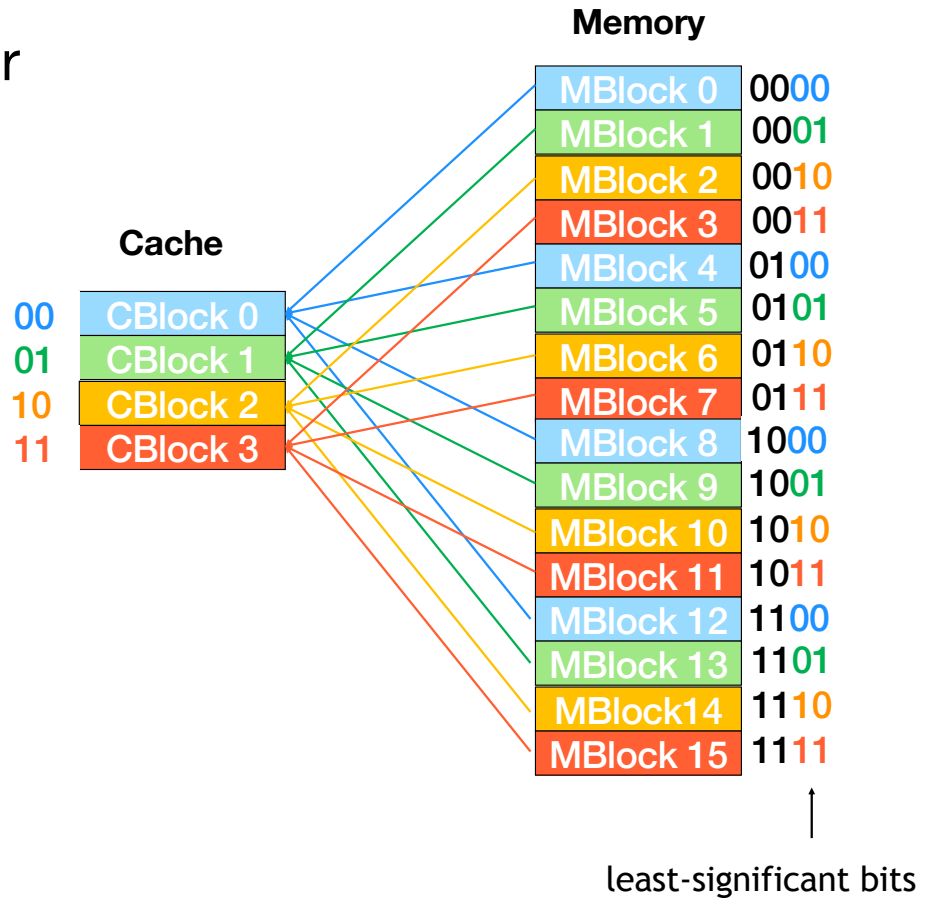
- Direct Mapping
- Fully Associative Mapping
- Set-Associative Mapping

Cache mappings are reminescent of hash table mappings

Direct Mapping

Cache resembles a hash table with one slot per bucket

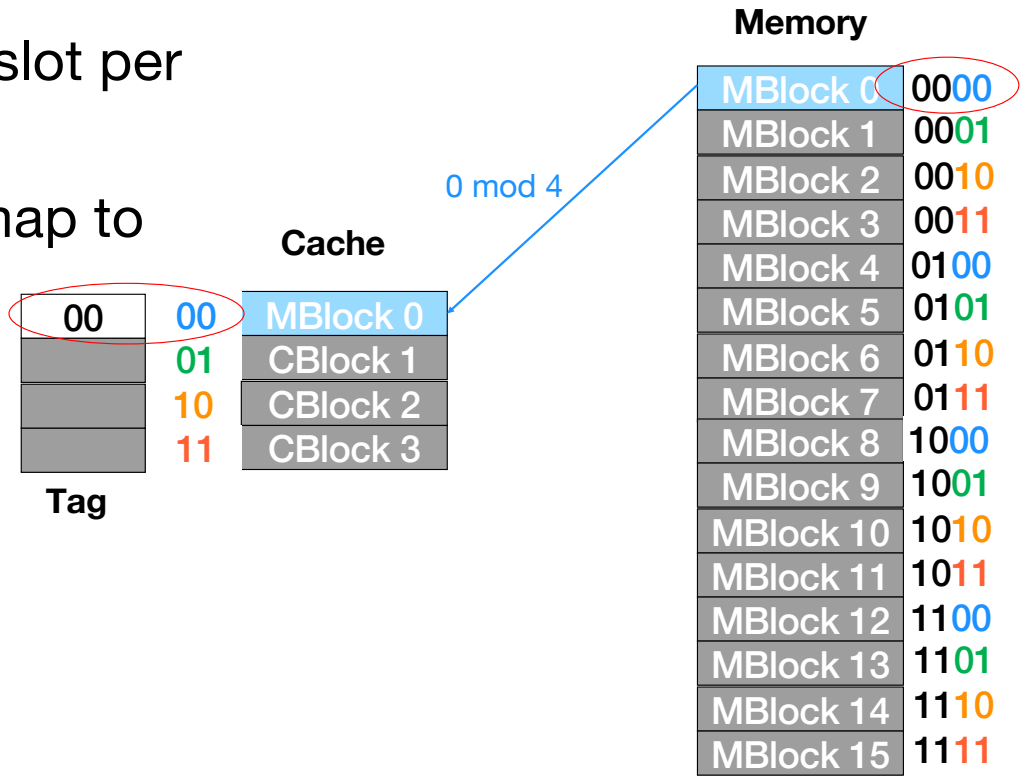
- Each main memory block will always map to the same cache block: *i mod n*



Direct Mapping

Cache resembles a hash table with one slot per bucket

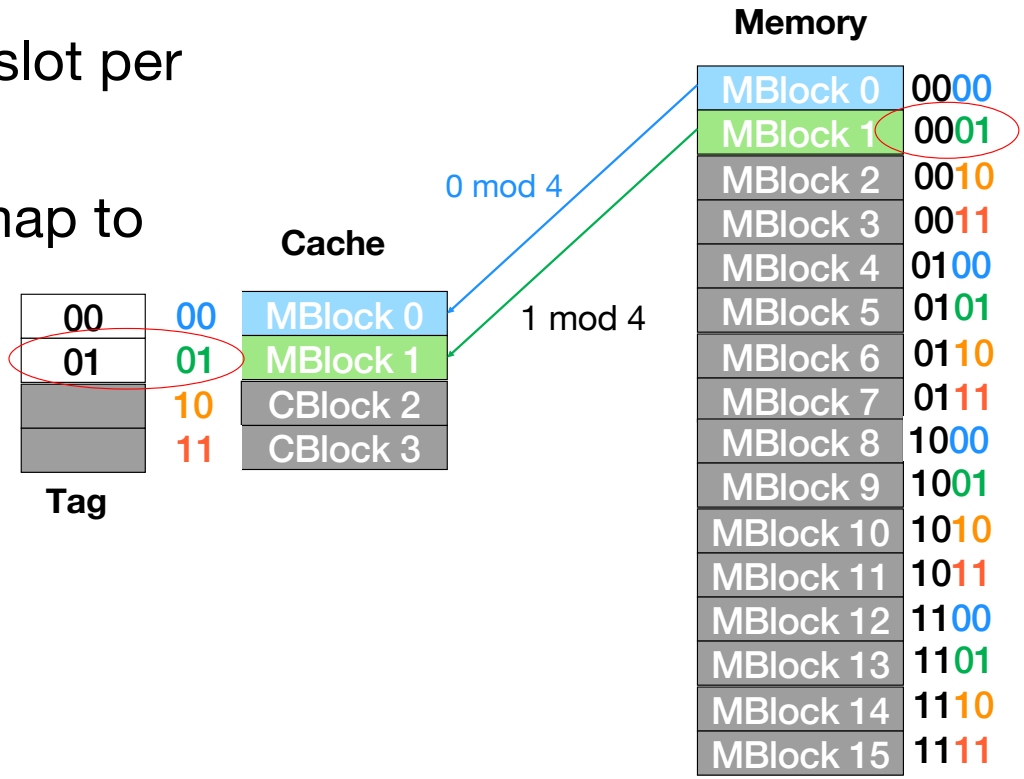
- Each main memory block will always map to the same cache block: *i mod n*



Direct Mapping

Cache resembles a hash table with one slot per bucket

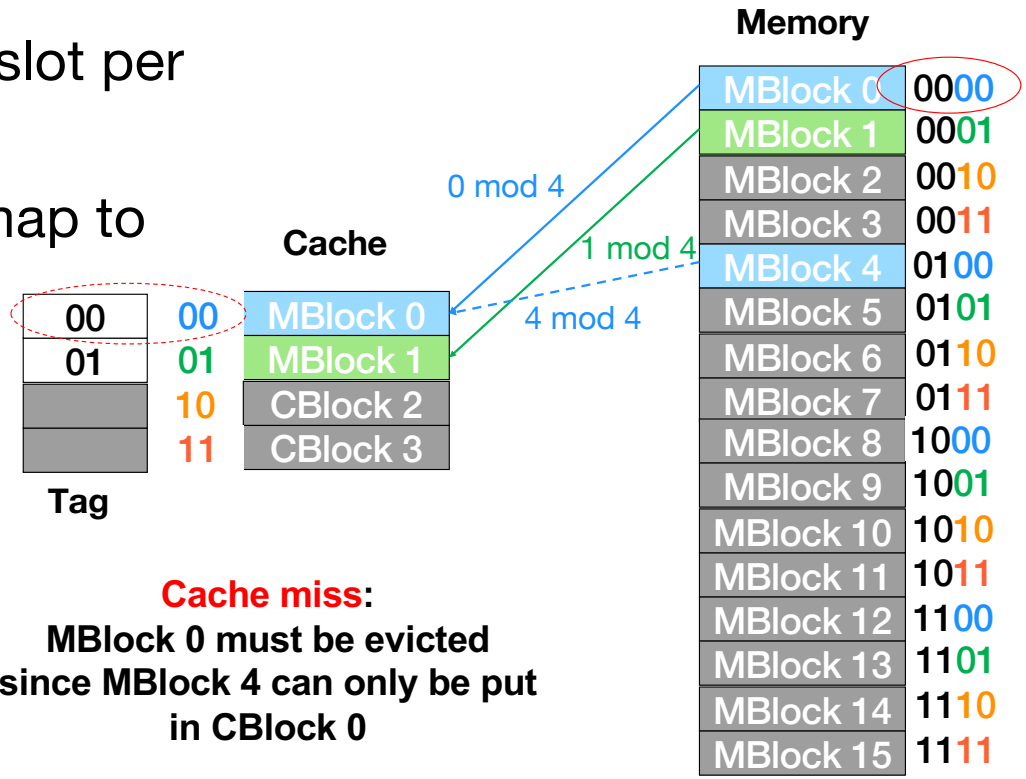
- Each main memory block will always map to the same cache block: *i mod n*



Direct Mapping

Cache resembles a hash table with one slot per bucket

- Each main memory block will always map to the same cache block: *i mod n*
- Collisions lead to evictions

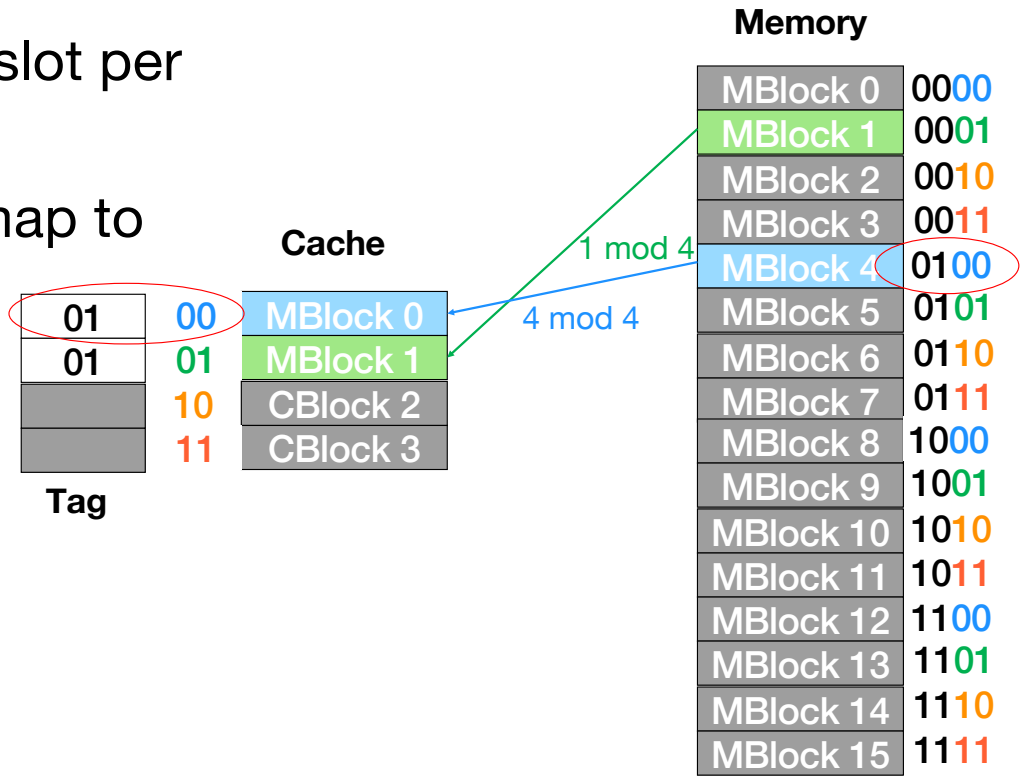


Cache miss:
MBlock 0 must be evicted since MBlock 4 can only be put in CBlock 0

Direct Mapping

Cache resembles a hash table with one slot per bucket

- Each main memory block will always map to the same cache block: *i mod n*
- Collisions lead to evictions



Direct Mapping

Pros:

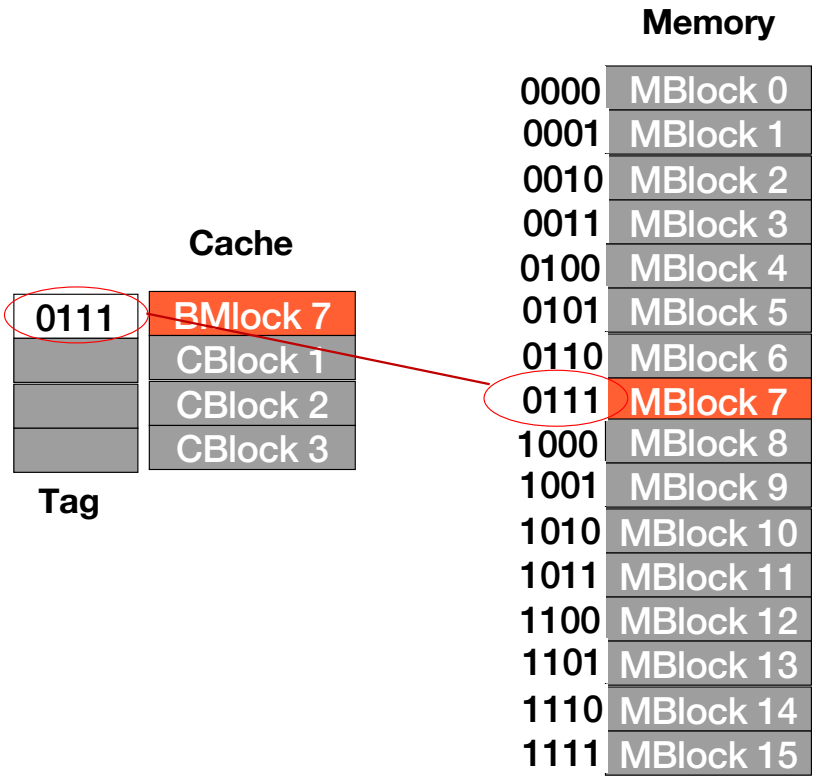
- indices and offsets easy to be computed with bit operators or simple arithmetic, because each memory address belongs in exactly one block
- no replacement algorithm is required

Cons:

- inefficient space utilization
- conflict between two or more memory addresses which map to a single cache block

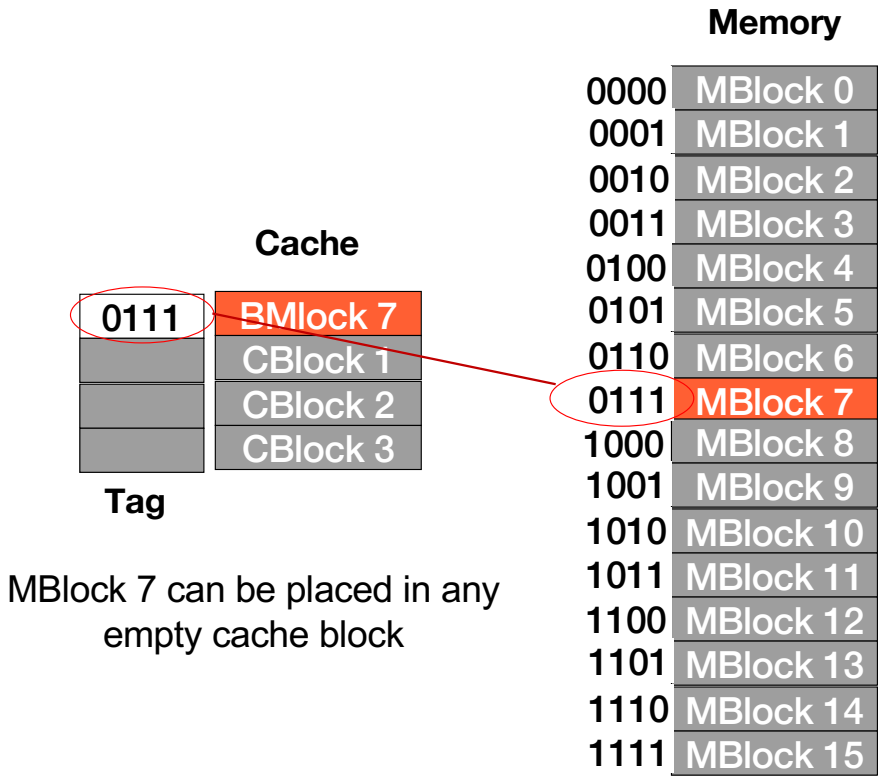
Fully Associative Mapping

- A Main Memory block can be stored in any cache block (any unused block suits)
- Since data could be anywhere in the cache, we must check the tag of every cache block



Fully Associative Mapping

- A Main Memory block can be stored in any cache block (any unused block suits)
- Since data could be anywhere in the cache, we must check the tag of every cache block
- The **entire address** must be used as the **tag**:
  - if the tag matches one of the cache tags: **cache hit**
  - else **cache miss**: we need room



Replacement strategy

In case all the blocks are already in use, we have to evict one block to make room for the new block: we need a **replacement strategy** to decide which block to replace

Optimal strategy: replace block used the farthest in the future

Impossible to predict!

Replacement strategy

Possible practical strategies:

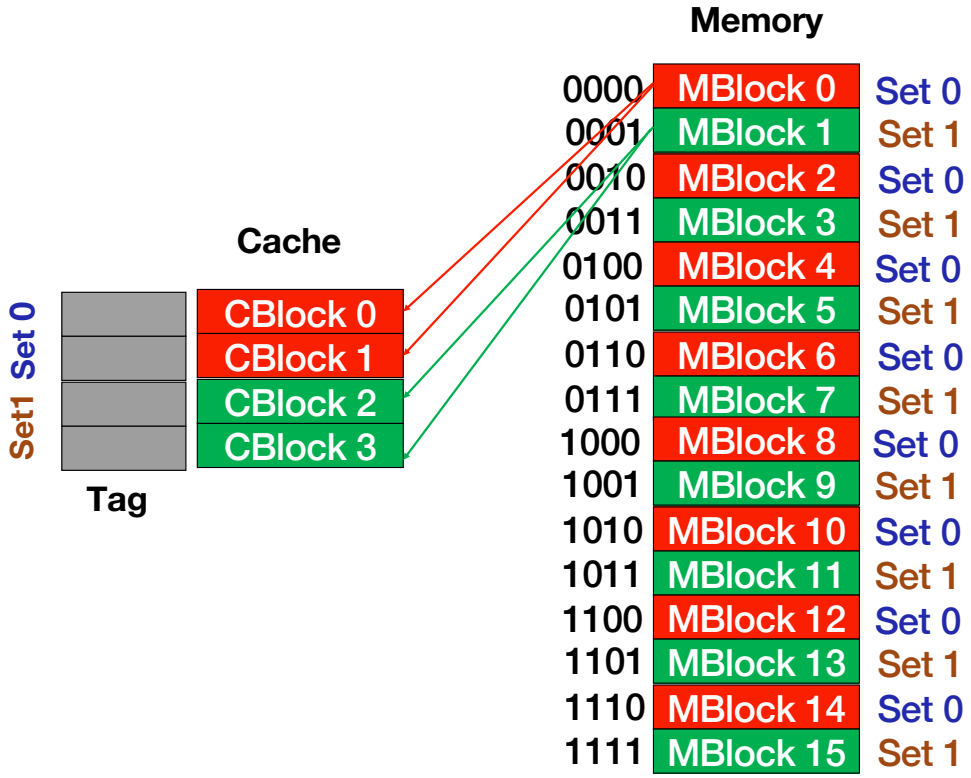
- **LRU** (Least Recently Used): discards least recently used items first, exploiting the locality principle, according to which if a block has not been used in a while, it will not be needed again anytime soon
- **LFU** (Least Frequently Used): similar to LRU, except that how many times a block was accessed is stored instead of how recently
- **FIFO** (First-In First-Out): the cache behaves like a queue
- **Random** replacement: randomly selects the item to be discarded
- …

K-Set Associative Mapping

Full associativity implies linear search time

An intermediate possibility is a **k-set-associative mapping**

- The cache is divided into **k** groups of blocks, called **sets**
- Each memory address maps to exactly one **set** in the cache, but data may be placed in any block within that set
- Search time for k-set associativity depends on **k** (the set dimension)

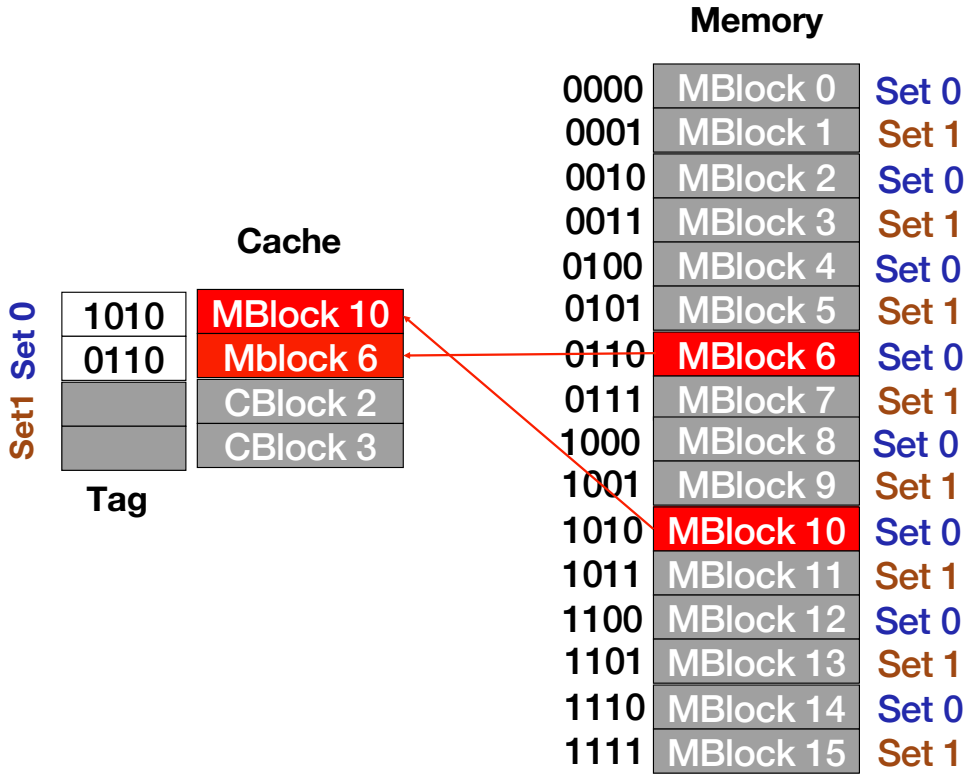


K-Set Associative Mapping

Full associativity implies linear search time

An intermediate possibility is a **k-set-associative mapping**

- The cache is divided into **k** groups of blocks, called **sets**
- Each memory address maps to exactly one **set** in the cache, but data may be placed in any block within that set
- Search time for k-set associativity depends on **k** (the set dimension)

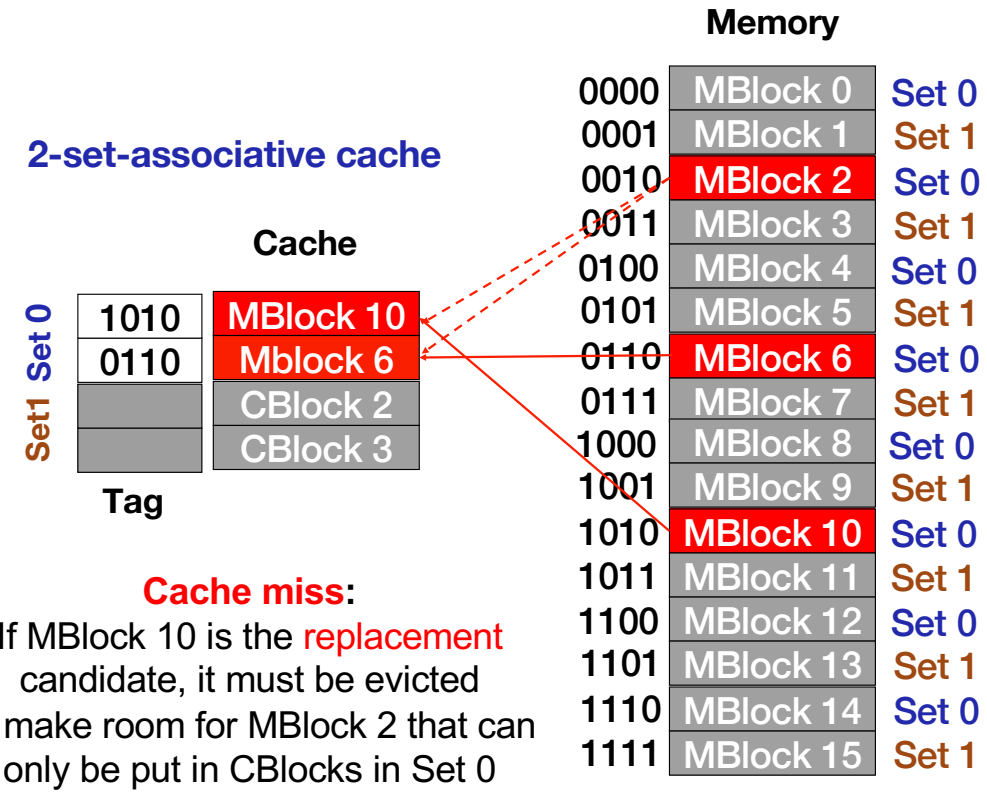


K-Set Associative Mapping

Full associativity implies linear search time

An intermediate possibility is a **k-set-associative mapping**

- The cache is divided into **k** groups of blocks, called **sets**
- Each memory address maps to exactly one **set** in the cache, but data may be placed in any block within that set
- Search time for k-set associativity depends on **k** (the set dimension)



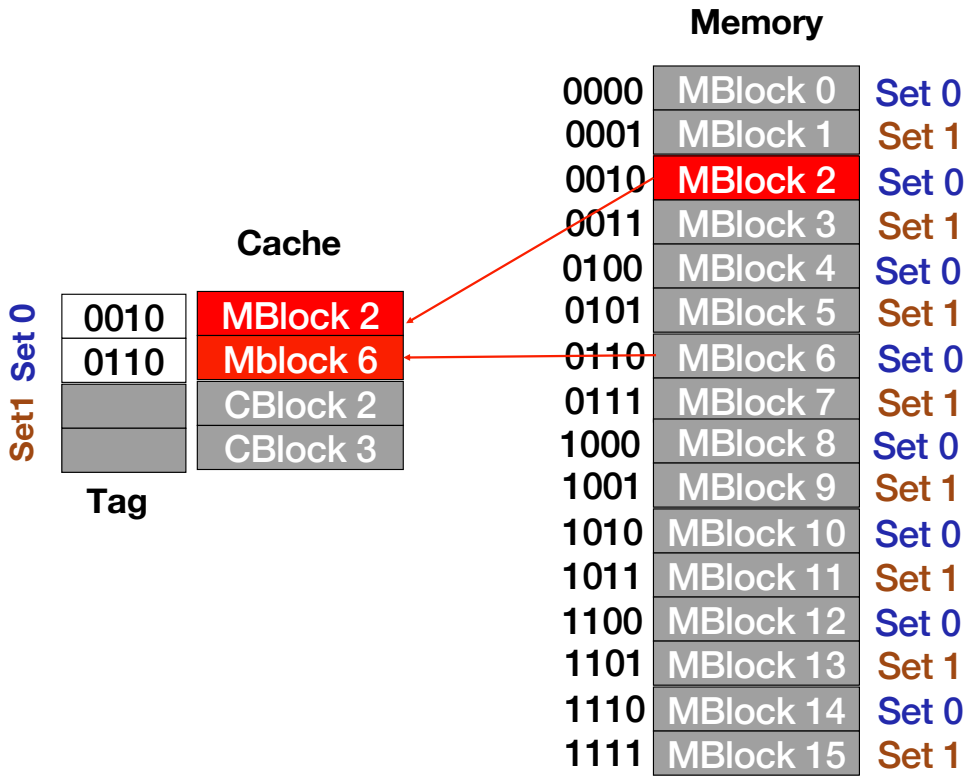
Cache miss:
If MBlock 10 is the replacement candidate, it must be evicted to make room for MBlock 2 that can only be put in CBlocks in Set 0

K-Set Associative Mapping

Full associativity implies **linear search time**

An intermediate possibility is a **k-set-associative mapping**

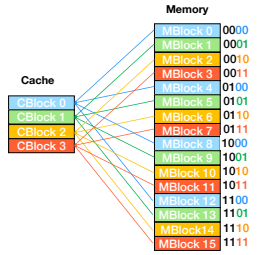
- The cache is divided into **k** groups of blocks, called sets
- Each memory address maps to exactly one **set** in the cache, but data may be placed in any block within that set
- Search time for k-set associativity depends on **k** (the set dimension)



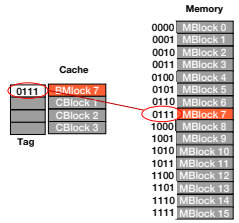
Mapping Techniques

All mappings can be classified as set-associative:

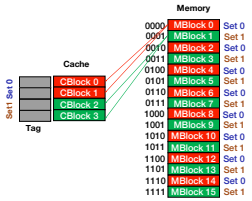
- **Direct Mapping**: only one line per set, i.e., **set size = 1**



- **Fully Associative Mapping**: only one cache, i.e., **set size = n** = 2^k



- **K-Set-Associative Mapping**: **set size = k**



- Set-Associative caches are compositions of fully associative caches
- Larger sets and higher associativity lead to fewer cache conflicts and lower miss rates, but they also increase the hardware cost

Outline

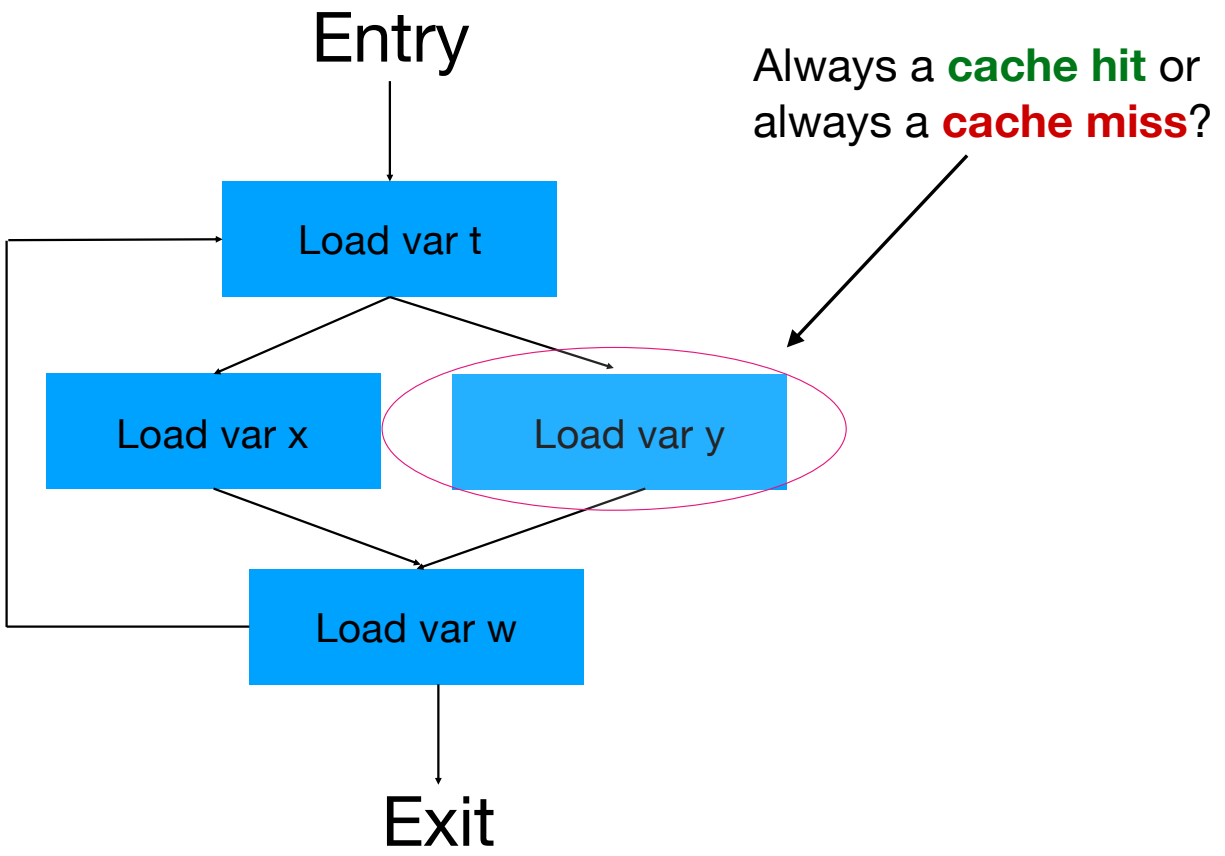
Memory Hierarchy and cache

Cache Static Analysis

Must Analysis

May Analysis

Cache Analysis



Static Cache Analysis

Static cache analysis characterizes a program's cache behavior
by determining in a **sound but approximate** way

- which memory accesses result in **cache hits** and
- which memory accesses result in **cache misses**
- which memory accesses result in "unclassified" or "unknown"

This information can be expolited in

- optimizing compilers,
- worst-case execution time analysis (important to bound, especially for real-time applications), and
- side-channel attack detection and mitigation

Static Cache Analysis

The first **classifying cache analysis** based on abstract interpretation (AI) was proposed by Ferdinand and Wilhelm at the end of the 1990s

- It provides **local guarantees for individual accesses**
  - **May analysis**: to over-approximate cache content and to predict **cache misses**
  - **Must analysis**: to under-approximate cache content and to predict **cache hits**

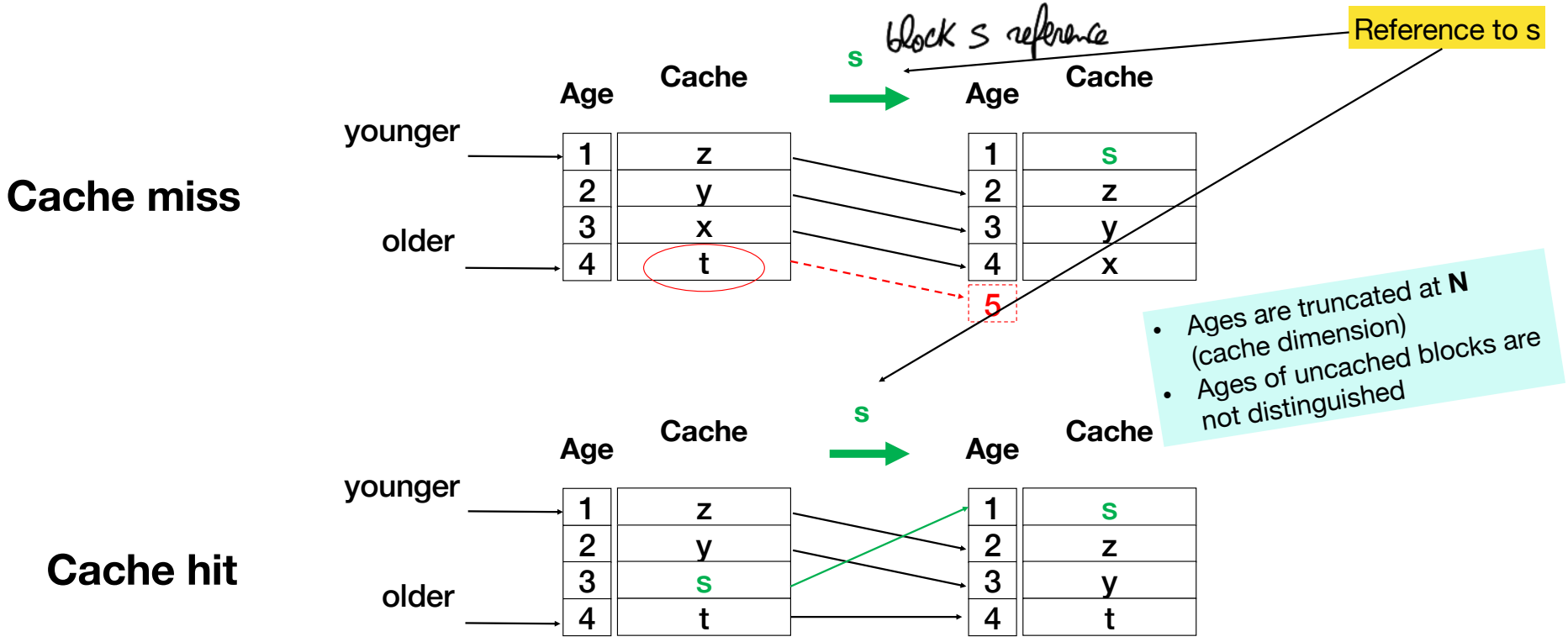
Static Cache Analysis

The overall approach works in two phases:

- An AI-based cache analysis computes **abstract cache states** at all program points as part of a **fixed-point solution** research problem: we do that to find solution with monotone fw
- These abstract cache states are **queried** in order to **classify memory accesses**
  - **always hit**: the memory reference will always result in a **cache hit**
  - **always miss**: the memory reference will always result in a **cache miss**

LRU concrete behaviour

LRU has a notion of **age** related to the position in the cache



LRU abstract behaviour: age of variables

- $V = \{v_1,\ldots,v_n\}$ is the set of variables used in the program:
- each v_i in V is mapped to a subset of cache lines
- Age(v) is the age of v in V and is a set of integers corresponding to ages of the lines it may reside (e.g., along all paths and for all inputs)
- If N is the number of cache lines, Age(v) ranges in [1,N+1] and

Age(v) =
- 1 means that v is the most recently used variable
- N means that v is the least recently used variable, and so the next candidate for eviction
- ≥ N means that v is outside the cache

- $S = \langle Age(v_1),\ldots, Age(v_n)\rangle$ tuple that represents the program's cache state

LRU abstract behaviour: states and updates

- **Abstract cache states** represent sets of concrete cache states and are bound to each CFG node, i.e., to each program point
- Only memory accesses are considered

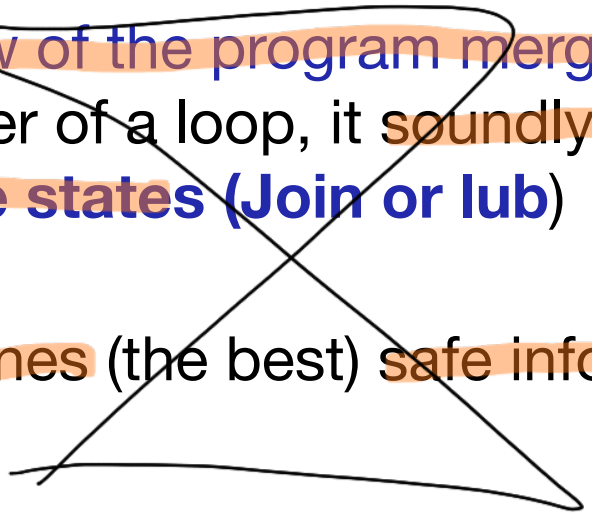
- Whenever the analysis encounters a memory access, it **updates the abstract cache state** in a way induced by the update that the processor would perform on the concrete cache states

- These abstract update functions that reflect the instruction's effect on the incoming abstract values are **Transfer functions**. They are are **monotone**, because information, once computed, is not lost again

LRU abstract behaviour: merge

- Whenever the control flow of the program merges, e.g. at the end of a conditional or at the header of a loop, it soundly **combines the incoming abstract cache states (Join or lub**)

- **Merge** operation determines (the best) safe information holding for all incoming paths



When we need to merge info we use a join

LRU abstract behaviour: partial order

As in the typical data-flow analyses, the **abstract domains** are **lattices**, i.e., partially ordered sets, where all subsets have least upper bounds

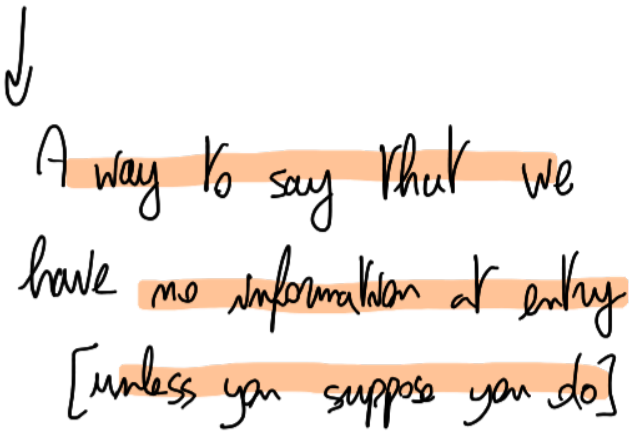
The **partial order**

- is imposed by the set inclusion order of the underlying concrete domain (sets of concrete cache states), i.e., $A \subseteq B$, if the corresponding concrete states are in the same relationship A is more informative than B
- reflects the relative information content of two lattice elements: elements lower in the lattice represent more information than information higher in the lattice, i.e., if $A \subseteq B$, then A contains no worse information than B

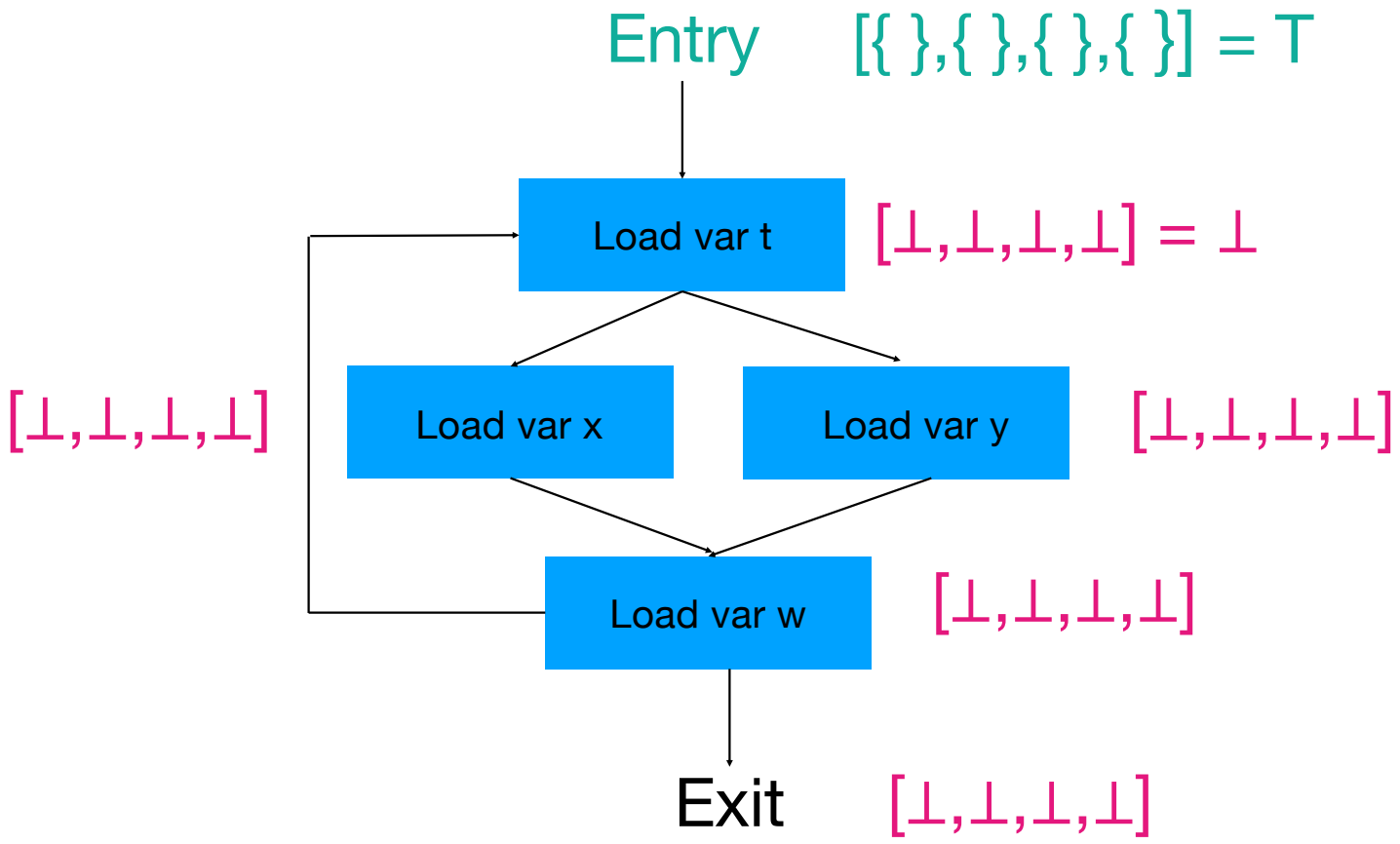
The lower in the lattice the more information

LRU abstract behaviour: partial order (cont.)

- **Bottom element ⊥**:
  - is the lowest element in the lattice of abstract cache states, and describes the empty set of concrete cache states
  - It is the initial analysis information at all program points except for program entry
- **Top element ⊤**:
  - is the highest lattice element, and described the set of all concrete cache states at program entry, in absence of any information about the cache contents
- For any **element A** of the lattice: ⊥ ⊆ A ⊆ ⊤



Cache Analysis



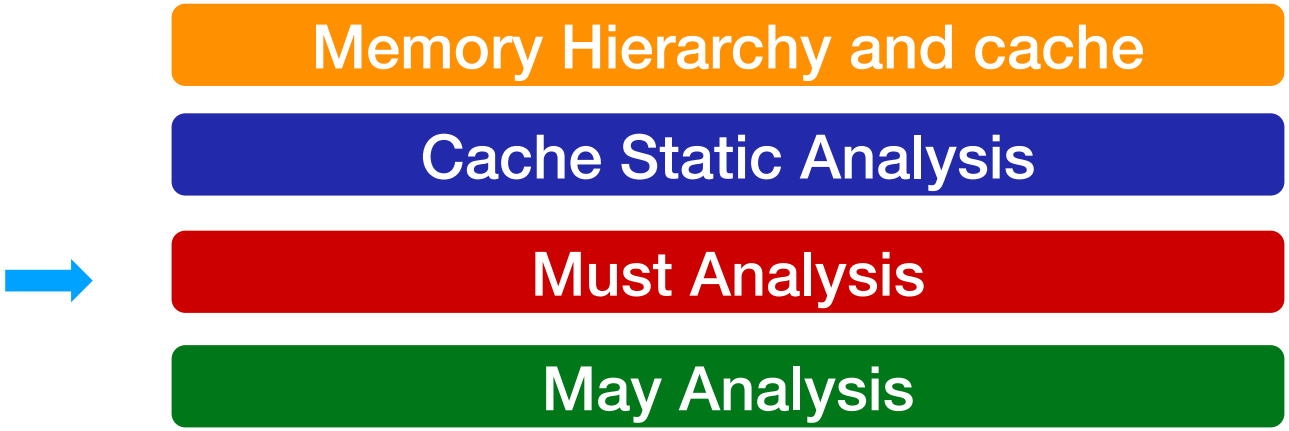
LRU abstract behaviour: equations

- In order to solve the analysis for a program, one can construct a system of recursive equations from its control flow graph

- A fixed-point iteration over these equations is guaranteed to terminate and deliver the **least fixed point** as solution

- Essential for termination is the finiteness of the lattice

Outline


Memory Hierarchy and cache
Cache Static Analysis
Must Analysis
May Analysis

Must Analysis

Must analysis

- computes, at each program point, the **common cache contents** in **all possible executions** leading to this program point [intersection as the join operation]
- can prove that a block **must be in the cache**

- An **abstract age** is associated with each memory block v in the **abstract state**, which is an **upper bound** of its ages in all the corresponding concrete states, i.e., an interval of ages enclosing the possible ages of the block during any program execution

Concrete Age(v) ≤ Abstract Age(v)

Must analysis: example

- Consider a program with 4 variables x, s, t and y, and Age in [1,4]
- The following **abstract state** describes the set of all concrete states in which **x, s**, **t**, and **y** occur: e.g.,
- {**s**,**t**} in position 3 means that the memory blocks **s** and **t** **must be** in the cache, with an **age of 3**, i.e.,
  not older (younger) than 3, e.g., not 4

Abstract age	Abstract state
1	{x}
2	{ }
3	{s,t}
4	{y}

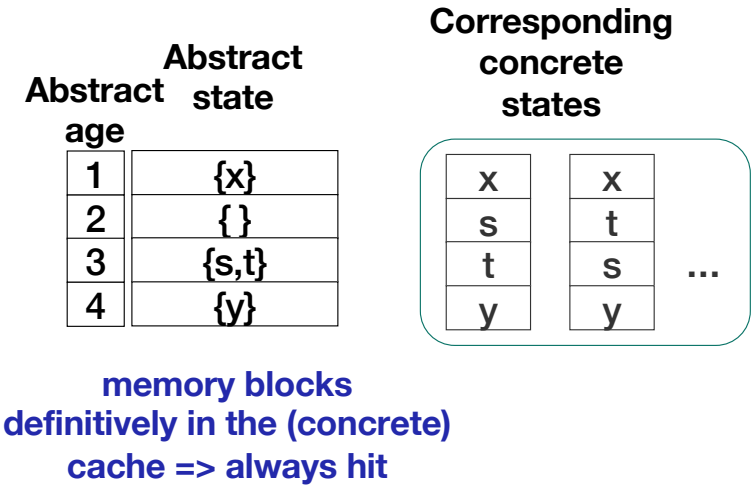
memory blocks definitively in the (concrete) cache => always hit

Must analysis: example

- Consider a program with 4 variables x, s, t and y, and Age in [1,4]
- The following **abstract state** describes the set of all concrete states in which x, s, t, and y occur: e.g.,
- {**s**,**t**} in position 3 means that the memory blocks **s** and **t** **must be** in the cache, with an **age of 3**, i.e.,
  not older (younger) than 3, e.g., not 4

Concrete Age(v) ≤ Abstract Age(v)

{1} = Concrete Age(x) ≤ Abstract Age(x) = 1 ≤ 4
{2,3} = Concrete Age(s) ≤ Abstract Age(s) = 3 ≤ 4
{2,3} = Concrete Age(t) ≤ Abstract Age(t) = 3 ≤ 4
{4} = Concrete Age(y) ≤ Abstract Age(y) = 4 ≤ 4

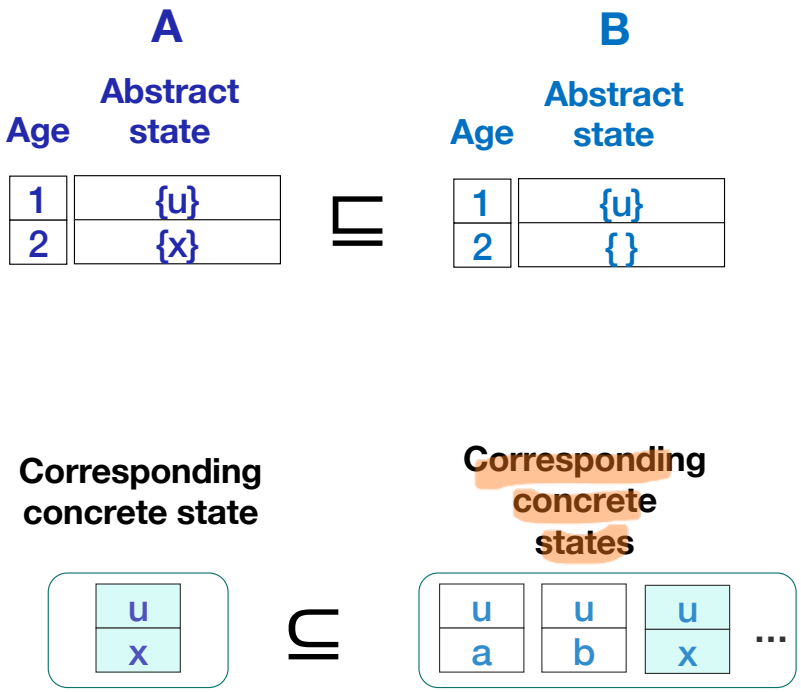


Must Analysis

To determine whether the memory access to **v is always a cache hit**, it suffices to know that **v is in the abstract cache state** reaching its program point:

Abstract Age(v) ≤ N ⇒ **Concrete Age(v)** ≤ Abstract Age(v) **≤ N definitive cache hit**
Abstract Age(v) > N ⇒ v may be outside of the cache (potential cache miss)

Must analysis: Order

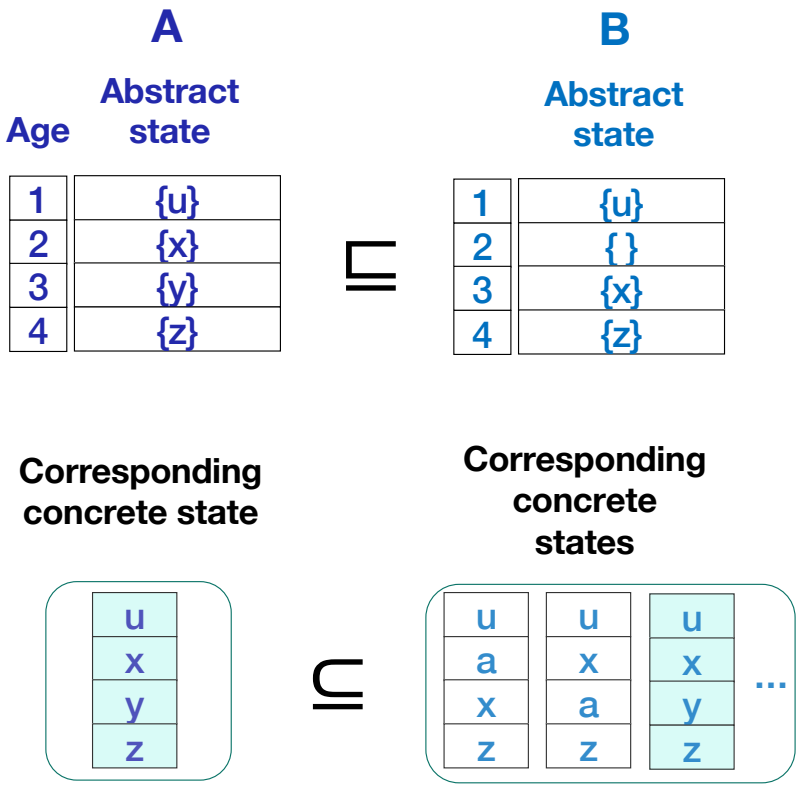


Recall that A ⊑ B iff A ⊔ B = B

- All the concrete cache states represented by A are also represented by B
- All the memory blocks known to be contained in the concrete cache states described by B, in the example just **u**, are also known to be contained in the concrete cache states described by A
- In addition A tells us that block **x** is guaranteed to be in the cache while B does not

Clearly, the abstract cache state A contains better information than B

Must analysis: Order (cont.)



- {2} = Concrete Age(x) ≤ Abstract Age(x) = 2
- {2,3} = Concrete Age(x) ≤ Abstract Age(x) = 3

- All the concrete cache states represented by A are also represented by B
- All the memory blocks known to be contained in the concrete cache states described by B, in the example **u**, **x** and **z,** are also known to be contained in the concrete cache states described by A
- In addition A tells us that

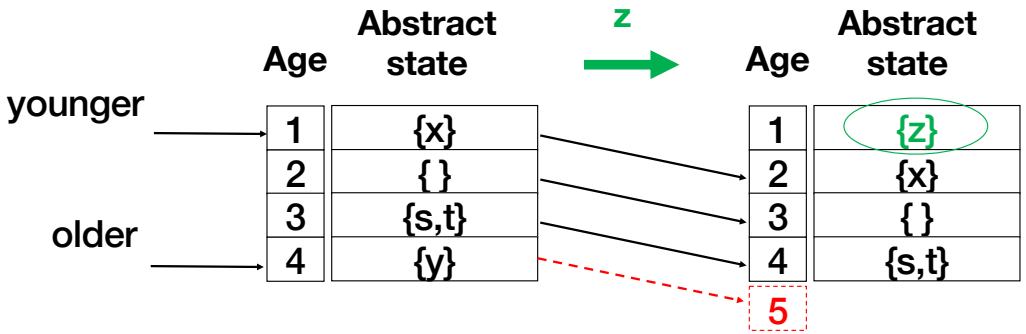
1. block **y** is guaranteed to be in the cache while B does not
2. the age upper bound estimated for block **x** in A is **smaller than** that in B: $Age_A(\mathbf{x}) = 2 < 3 = Age_B(\mathbf{x})$

Clearly, the abstract cache state A contains better information than B

Must analysis: update update on memory access

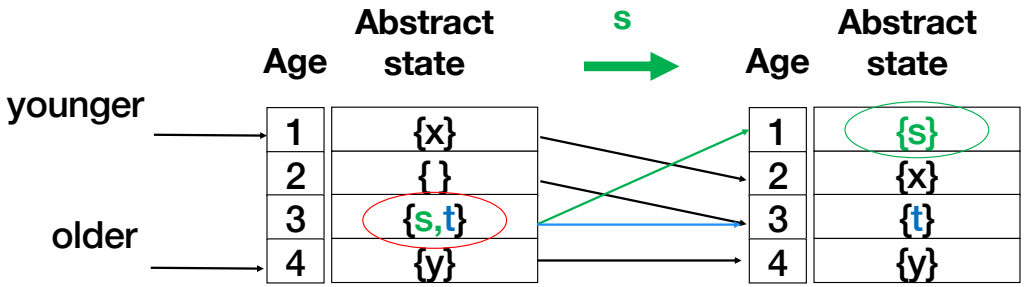
$S = \langle Age(x), Age(s), Age(t) \rangle$ $S' = \langle Age'(x), Age'(s), Age'(t) \rangle$

Potential Cache miss



We only focus on memory-related instructions

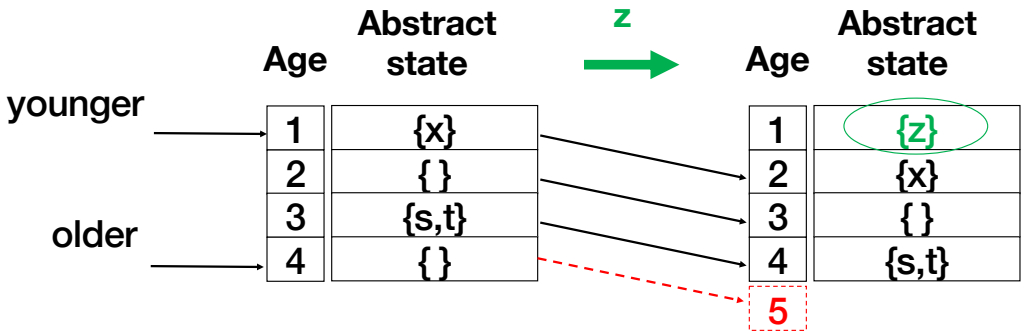
Definite Cache hit



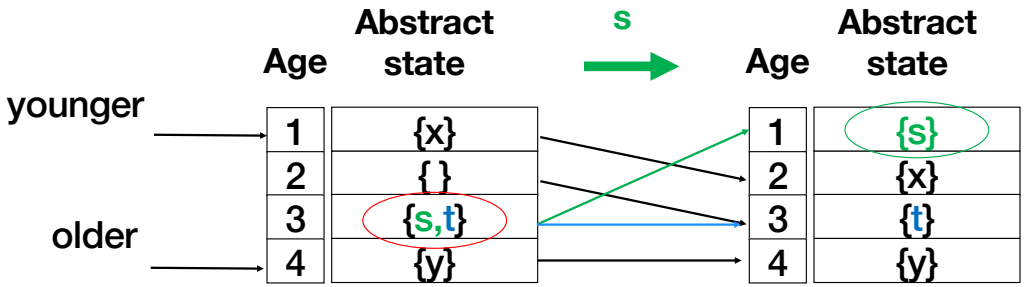
Must analysis: update update on memory access

$S = \langle Age(x), Age(s), Age(t) \rangle$ $S' = \langle Age'(x), Age'(s), Age'(t) \rangle$

Potential Cache miss



Definite Cache hit



Why does t not age in the second case?

We only focus on memory-related instructions

Transfer function

Transfer(S, inst(v))

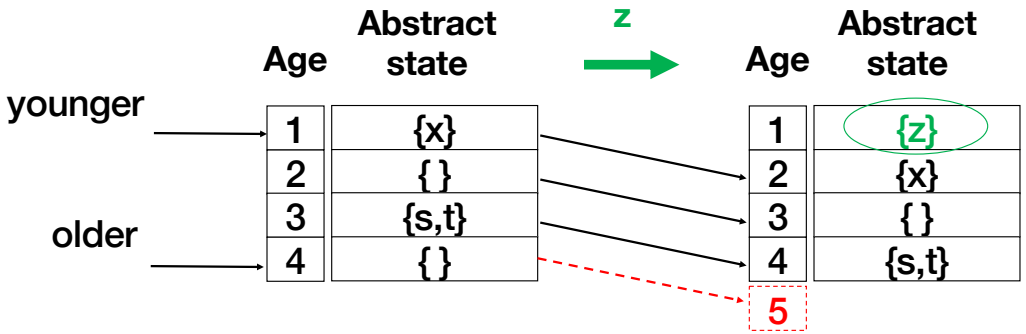
- $Age'(v) = 1$
- $Age(u) < Age(v) \Rightarrow Age'(u) = Age(u) + 1$
- $Age(u) \geq Age(v) \Rightarrow$
- $Age'(u) = Age(u)$

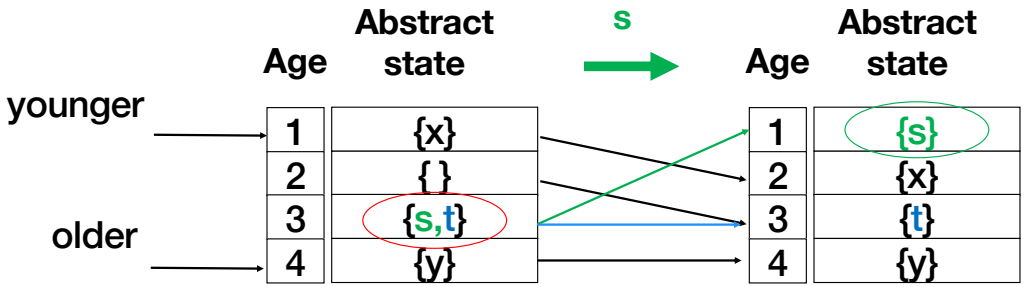
Must analysis: update update on memory access

Potential Cache miss

Definite Cache hit

$S = \langle Age(x), Age(s), Age(t) \rangle$ $S' = \langle Age'(x), Age'(s), Age'(t) \rangle$





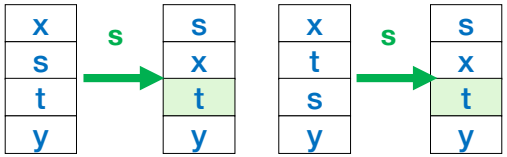
Why does t not age in the second case?
The abstract set state is an **upper bound**

We only focus on memory-related instructions

Transfer function

Transfer(S, inst(v))

- $Age'(v) = 1$
- $Age(u) < Age(v) \Rightarrow Age'(u) = Age(u) + 1$
- $Age(u) \geq Age(v) \Rightarrow$
- $Age'(u) = Age(u)$

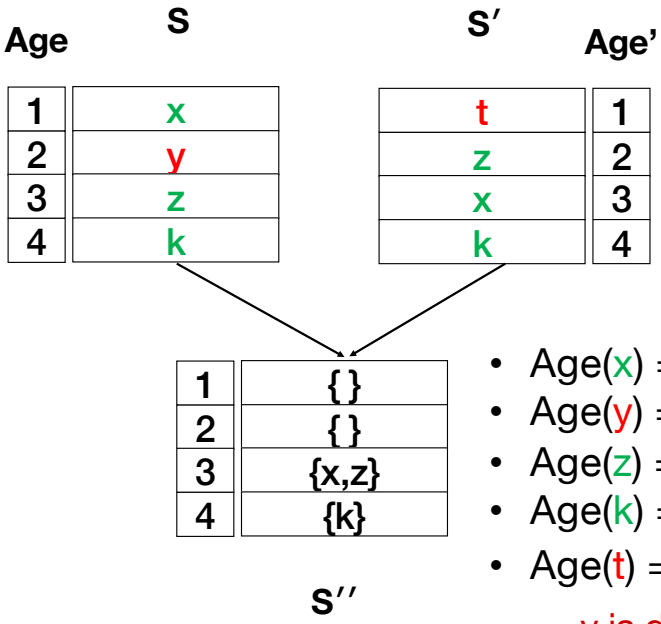


Must Analysis: join

How to combine information?

Intersection (+ maximal age)

It is a forward analysis



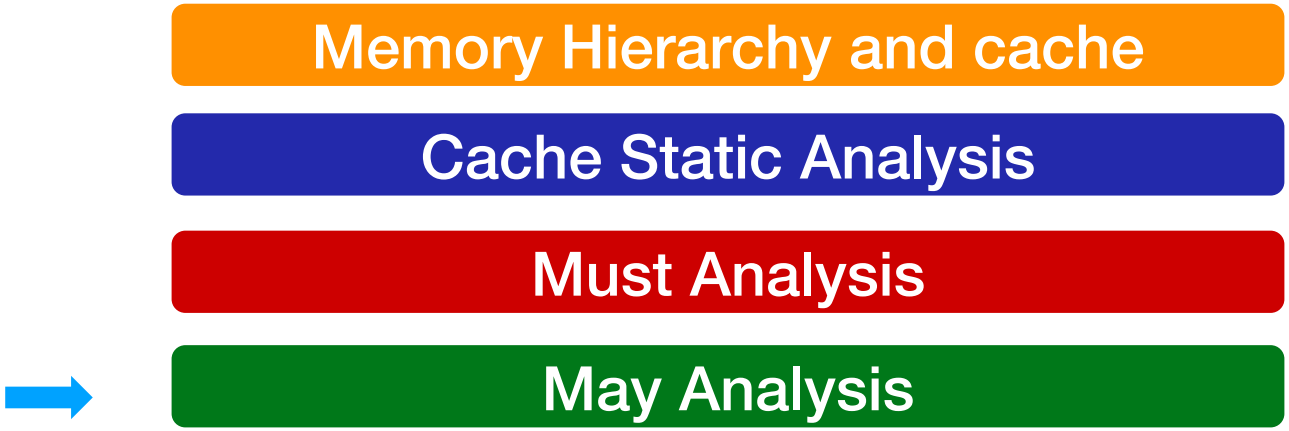
- Age(x) = 1, Age′(x) = 3 ⇒ Age′′(x) = 3
- Age(y) = 2, Age′(y) = 5 ⇒ Age′′(y) = 5
- Age(z) = 3, Age′(z) = 2 ⇒ Age′′(z) = 3
- Age(k) = 4, Age′(k) = 4 ⇒ Age′′(k) = 4
- Age(t) = 1, Age′(t) = 5 ⇒ Age′′(t) = 5

v is definitely in the cache only
if it is in the cache according to both states

Given two states

- S = ⟨Age(v_1), …,Age(v_n)⟩ and
- S′ = ⟨Age′(v_1), …,Age′(v_n)⟩, as follows:
- S′′ = S⊔S′ = ⟨**max**(Age(v_1),Age′(v_1)), …, **max**(Age(v_n),Age′(v_n))⟩

Outline


Memory Hierarchy and cache
Cache Static Analysis
Must Analysis
May Analysis

May Analysis

May analysis

- computes, at each program point, **all potentially cached contents in all possible executions** leading to this program point
- can prove that a block **may not be in the cache**

(is not)

- An age is associated with each memory block in the abstract state, which is an **lower bound** of its ages in all the corresponding concrete states

Concrete Age(v) ≥ Abstract Age(v)

May analysis: example

- Consider a program with 5 variables x, y, s, t, and u and
- The following abstract state describes the set of all concrete states in which x, y, s, t, and u occur
- {**s**,**t**} in position 3 means that the memory blocks **s** and **t may be** in the cache, with an **age of 3**, i.e.,
  not younger (older) than 3

Age	Abstract state
1	{x,y}
2	{ }
3	{s,t}
4	{u}

memory blocks definitively not in the (concrete) cache => always miss

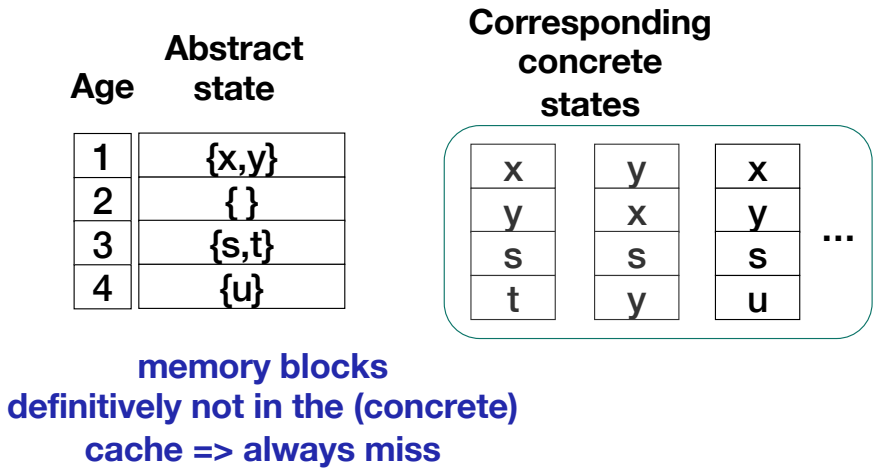
May analysis: example

- Consider a program with 5 variables x, y, s, t, and u and
- The following abstract state describes the set of all concrete states in which x, y, s, t, and u occur
- {**s**,**t**} in position 3 means that the memory blocks **s** and **t** **may be** in the cache, with an **age of 3**, i.e.,

  not younger (older) than 3

Concrete Age(v) ≥ Abstract Age(v)

Concrete Age(x) ≥ Abstract Age(x) = 1
Concrete Age(y) ≥ Abstract Age(y) = 1
Concrete Age(s) ≥ Abstract Age(s) = 3
Concrete Age(t) ≥ Abstract Age(t) = 3
Concrete Age(u) ≥ Abstract Age(u) = 4



memory blocks definitively not in the (concrete) cache => always miss

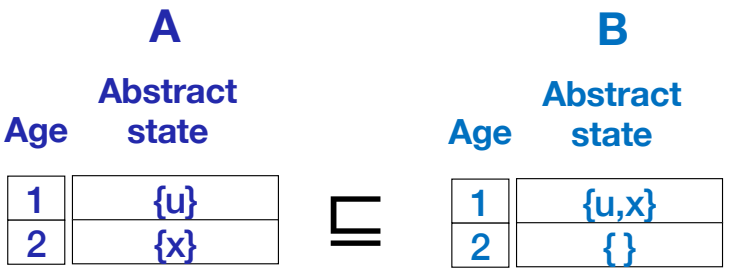
May Analysis

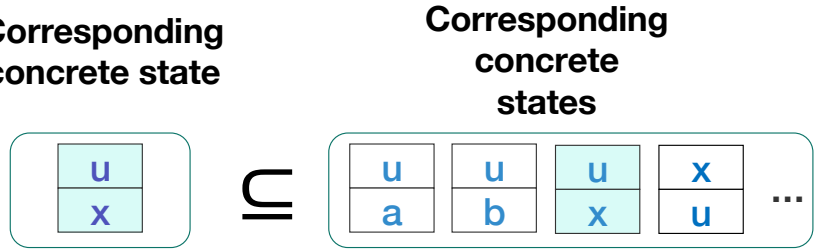
To determine whether the memory access to v **is always a cache miss**, it suffices to know that v **is not in the abstract cache state** reaching its program point

Abstract age(v) ≥ N ⇒ **Concrete age(v)** ≥ Abstract age(v) **≥ N (definite cache miss)**

Abstract age(v) < N ⇒ v may be inside the cache: **(potential cache hit)**

May analysis: Order





Recall that $A \sqsubseteq B$ iff $A \sqcup B = B$

- All the concrete cache states represented by A are also represented by B
- All the memory blocks known to be contained in the concrete cache states described by B, in the example just **u** and **x**, are also known to be contained in the concrete cache states described by A

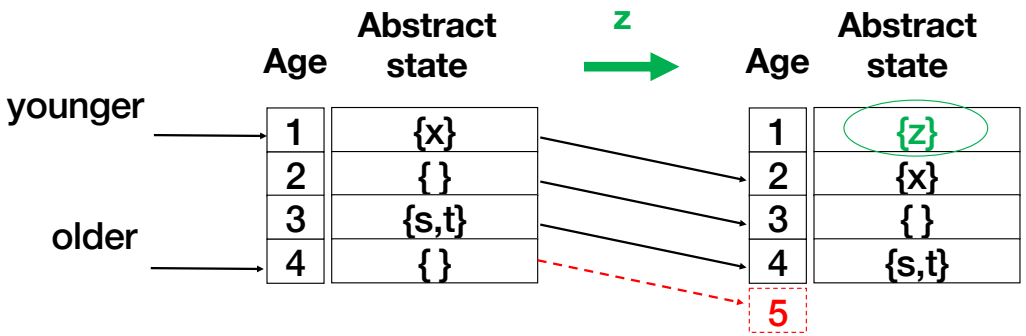
1. the age lower bound estimated for block **x** in A is **greater than** that in B: $Age_A(\mathbf{x}) = 2 > 1 = Age_B(\mathbf{x})$

Clearly, the abstract cache state A contains better information than B

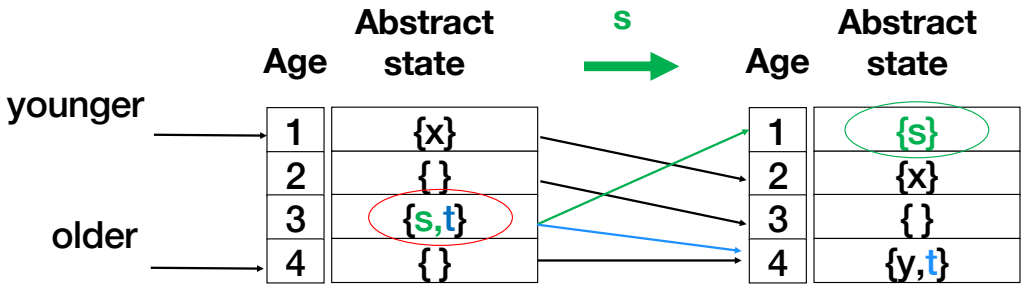
May analysis: update

Definite Cache miss

$S = \langle Age(x), Age(s), Age(t) \rangle$ $S' = \langle Age'(x), Age'(s), Age'(t) \rangle$



Potential Cache hit



Why does t not age in the second case?

The most recently used variable (v) occupies the youngest cache line

Transfer function

Transfer(S, inst(v))

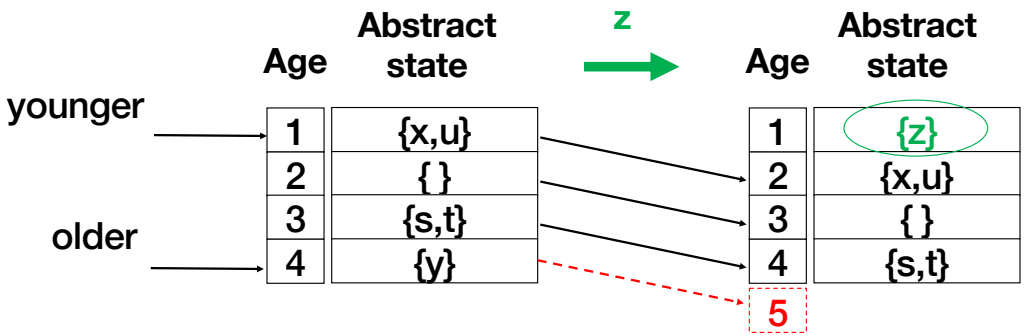
- $Age(v) = 1$
- $Age(u) \leq Age(v) \Rightarrow Age'(u) = Age(u) + 1$
- $Age(w) > Age(v) \Rightarrow Age'(w) = Age(w)$

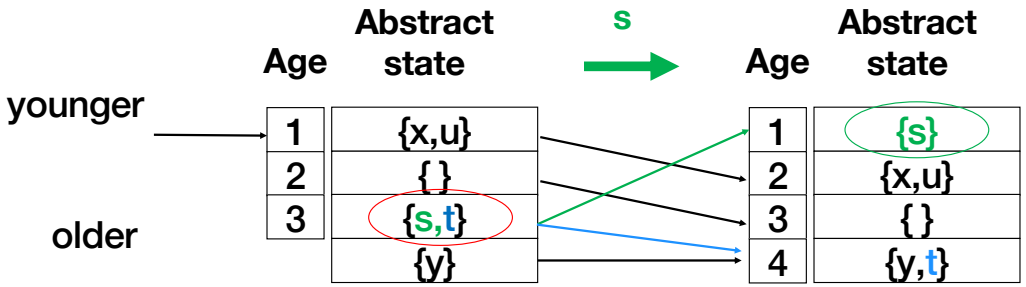
May analysis: update

Definite
Cache miss

Potential
Cache hit

$S = \langle Age(x), Age(s), Age(t) \rangle$ $S' = \langle Age'(x), Age'(s), Age'(t) \rangle$





Why does t not age in the second case?
The abstract set state is a **lower bound**:
the introduction of s implies a **potential** cache hit

The most recently used variable (v) occupies the youngest cache line

Transfer function

Transfer(S, inst(v))

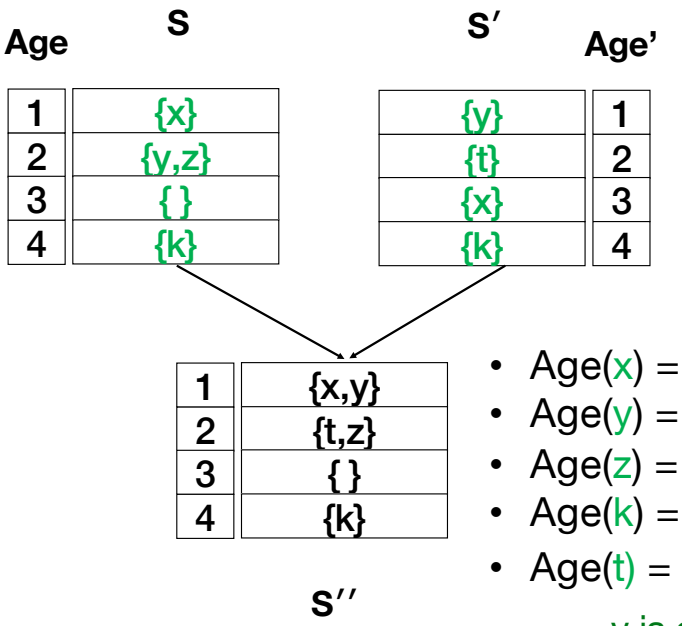
- $Age(v) = 1$
- $Age(u) \leq Age(v) \Rightarrow Age'(u) = Age(u) + 1$
- $Age(w) > Age(v) \Rightarrow Age'(w) = Age(w)$

May Analysis: join

How to combine information?

Union (+ minimal age)

It is a **forward** analysis



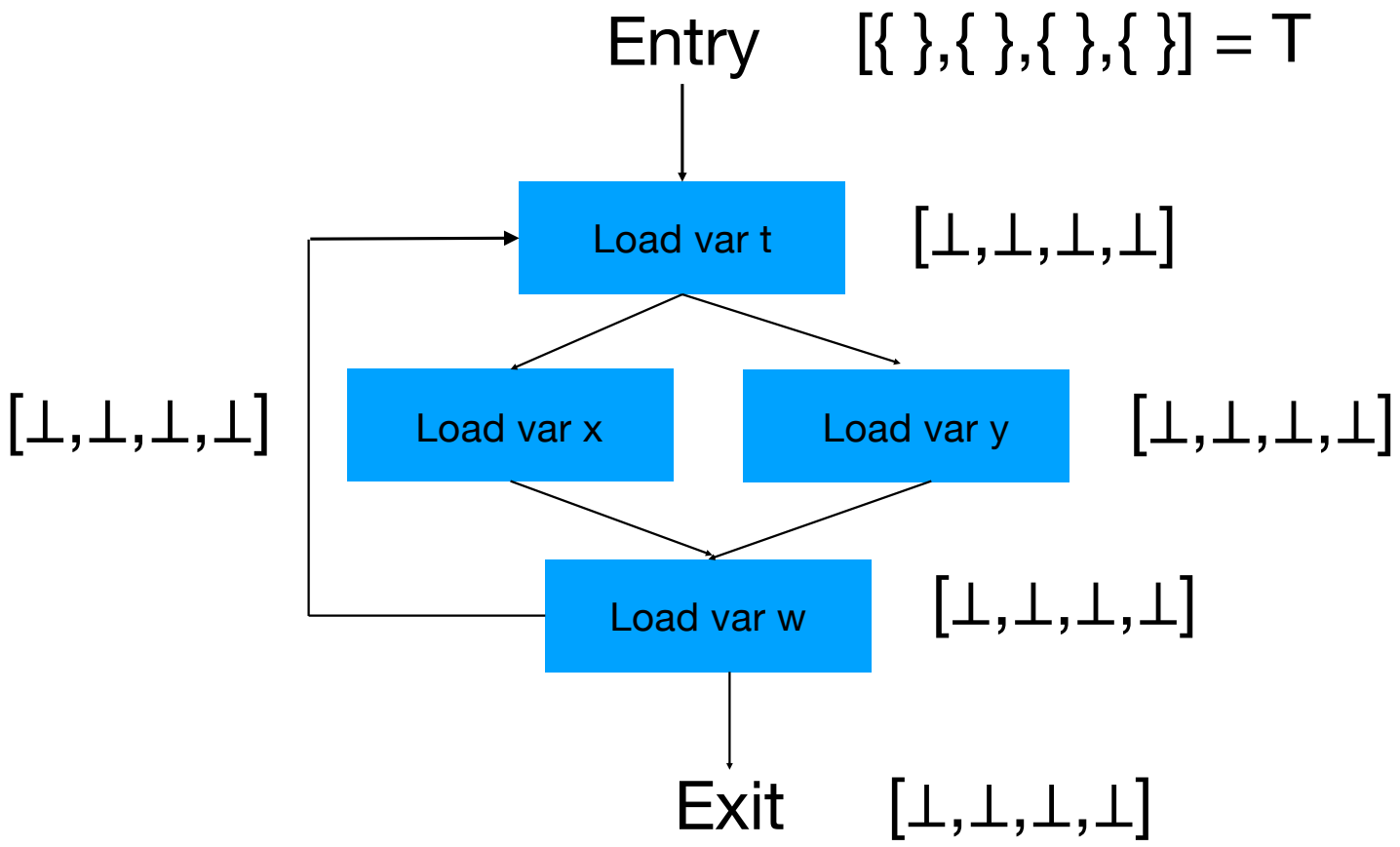
- Age(x) = 1, Age′(x) = 3 ⇒ Age′′(x) = 1
- Age(y) = 2, Age′(y) = 1 ⇒ Age′′(y) = 1
- Age(z) = 2, Age′(z) = 5 ⇒ Age′′(z) = 2
- Age(k) = 4, Age′(k) = 4 ⇒ Age′′(k) = 4
- Age(t) = 5, Age′(t) = 2 ⇒ Age′′(t) = 2

v is definitely not cached only
if it is not in the cache according to both states

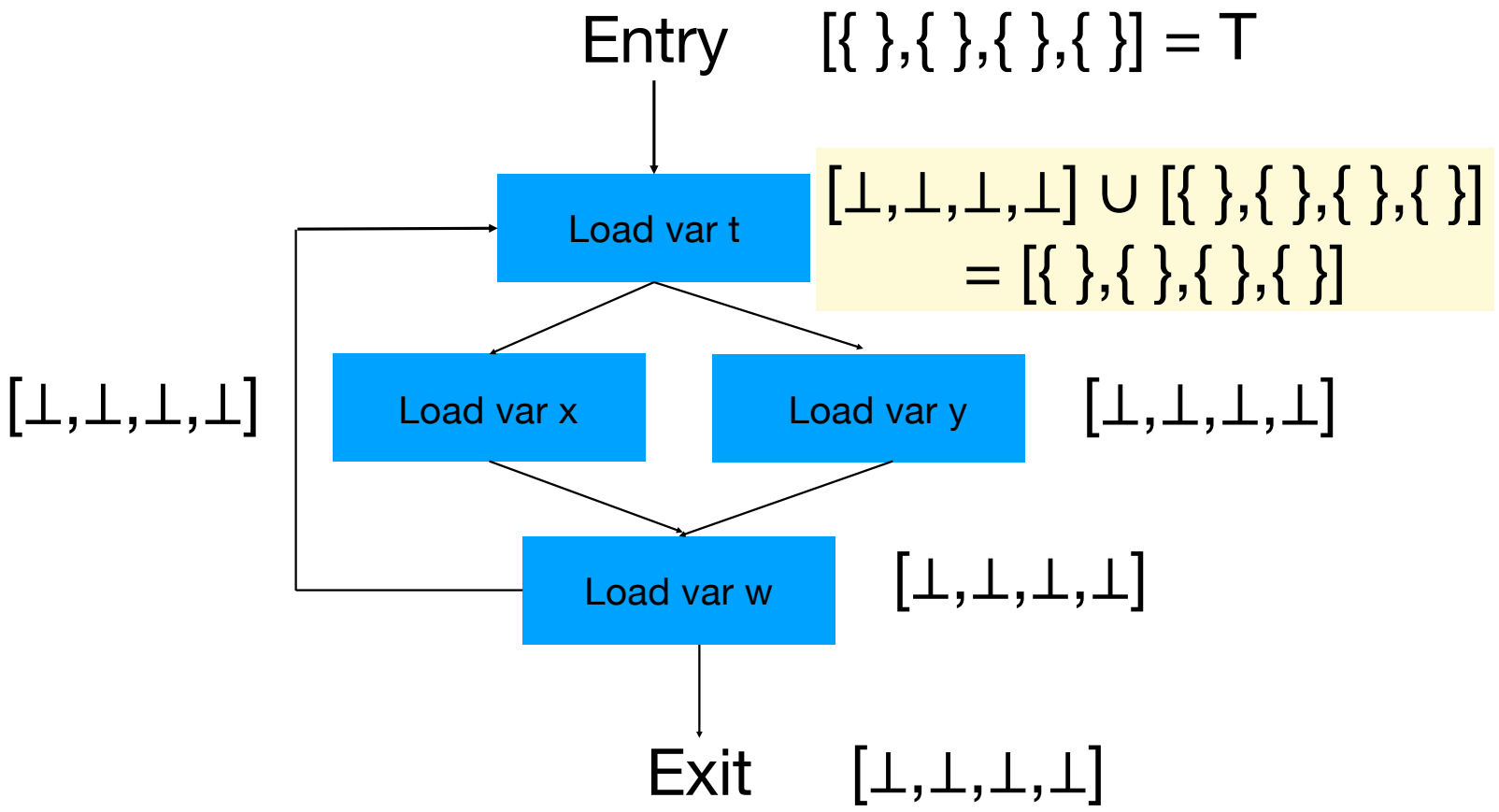
Given two states

- S = ⟨Age(v_1), …,Age(v_n)⟩ and
- S′ = ⟨Age′(v_1), …,Age′(v_n)⟩, as follows:
- S′′ = S⊔S′ = ⟨**min**(Age(v_1),Age′(v'_1)), …, **min**(Age(v_n),Age′(v_n))⟩

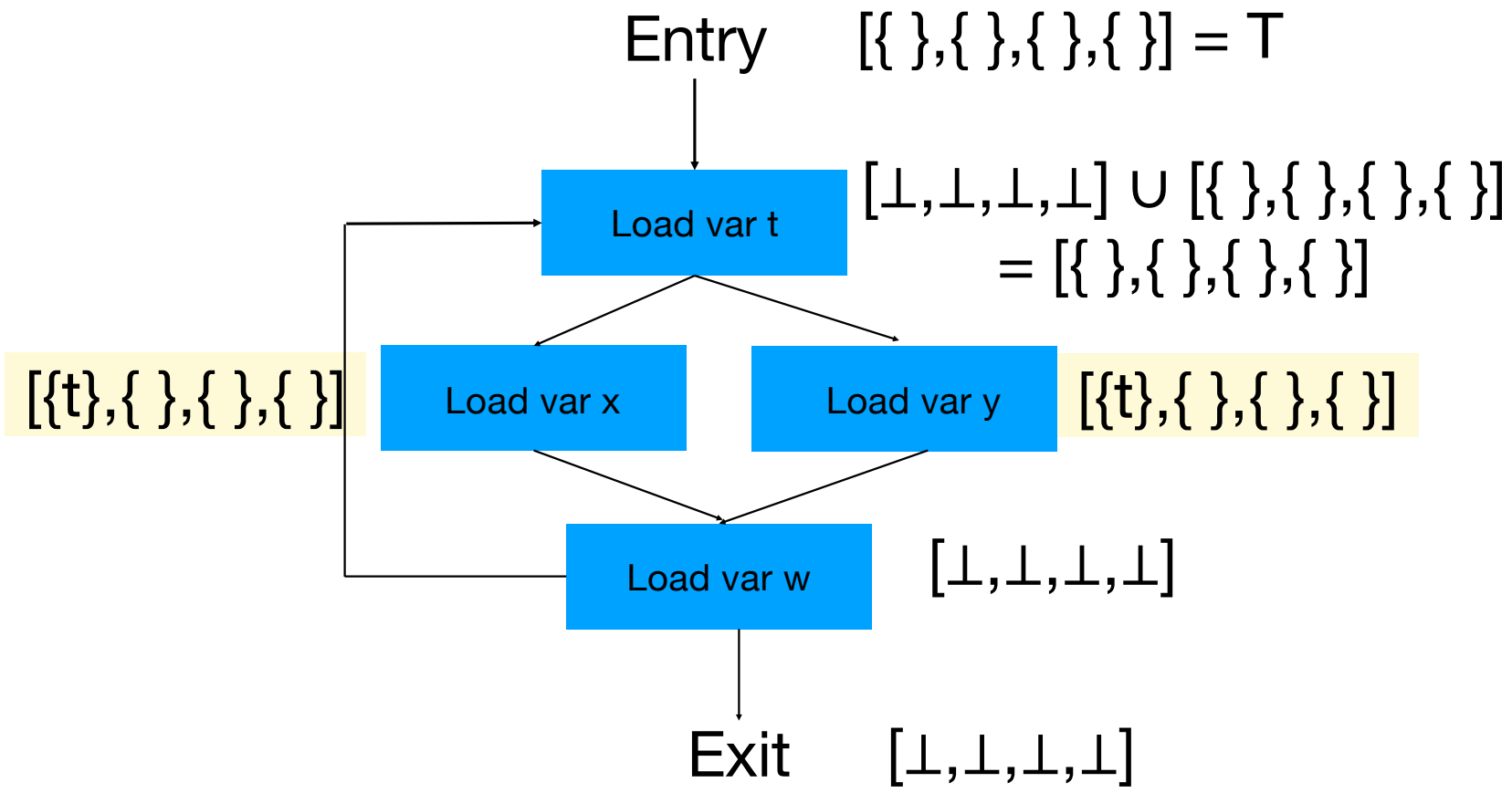
Back to Must Analysis: example



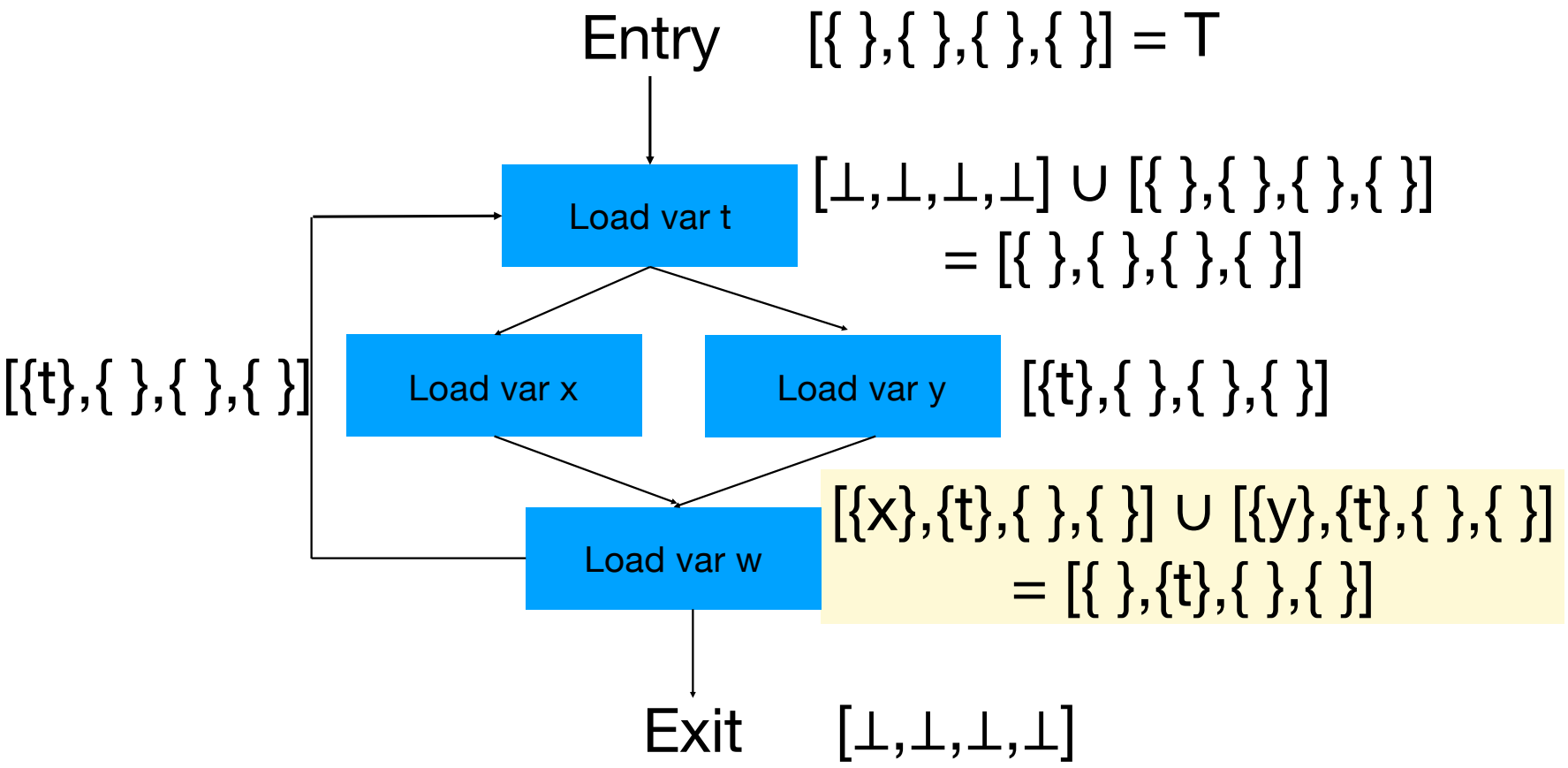
Back to Must Analysis: example



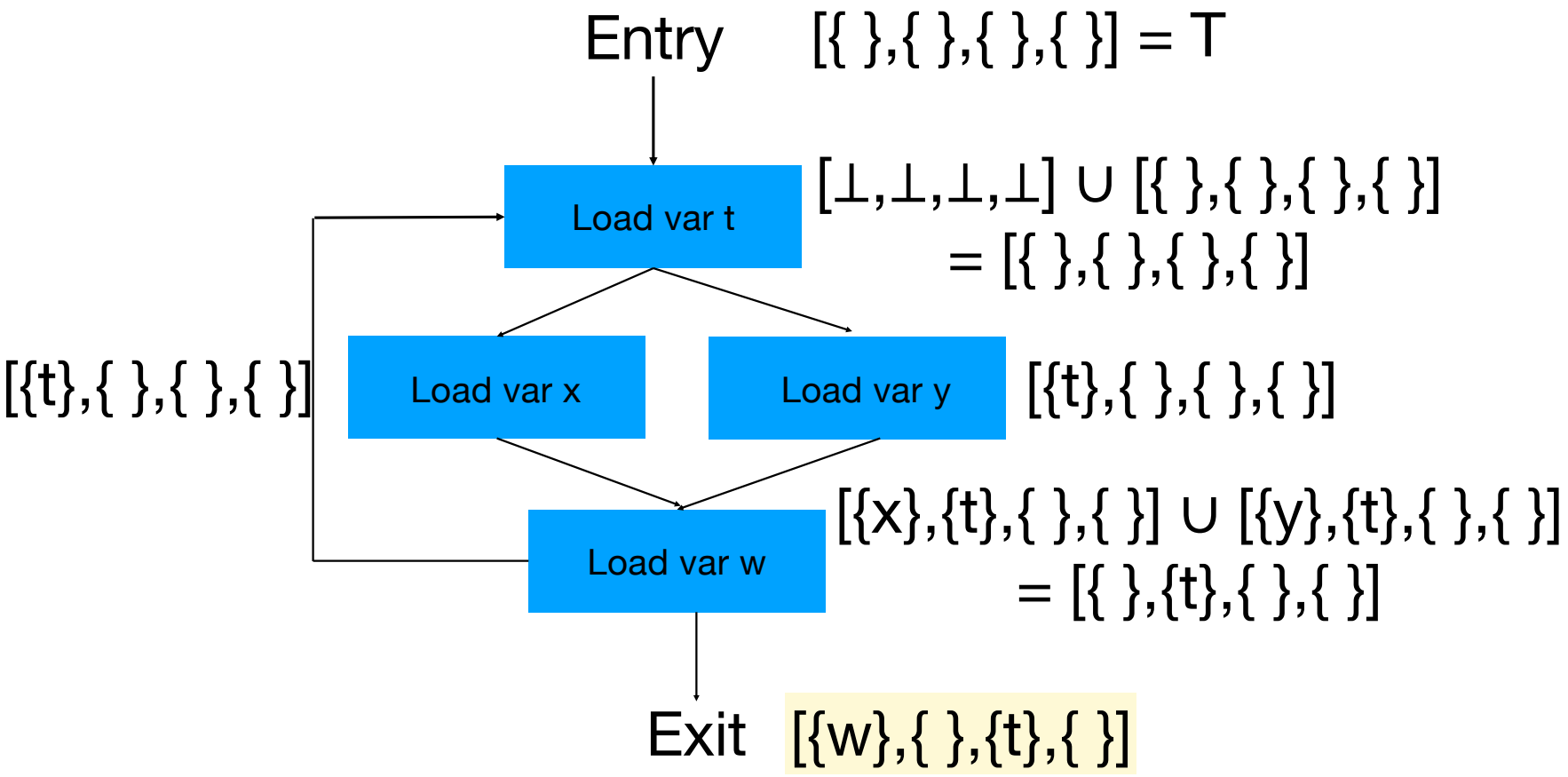
Back to Must Analysis: example



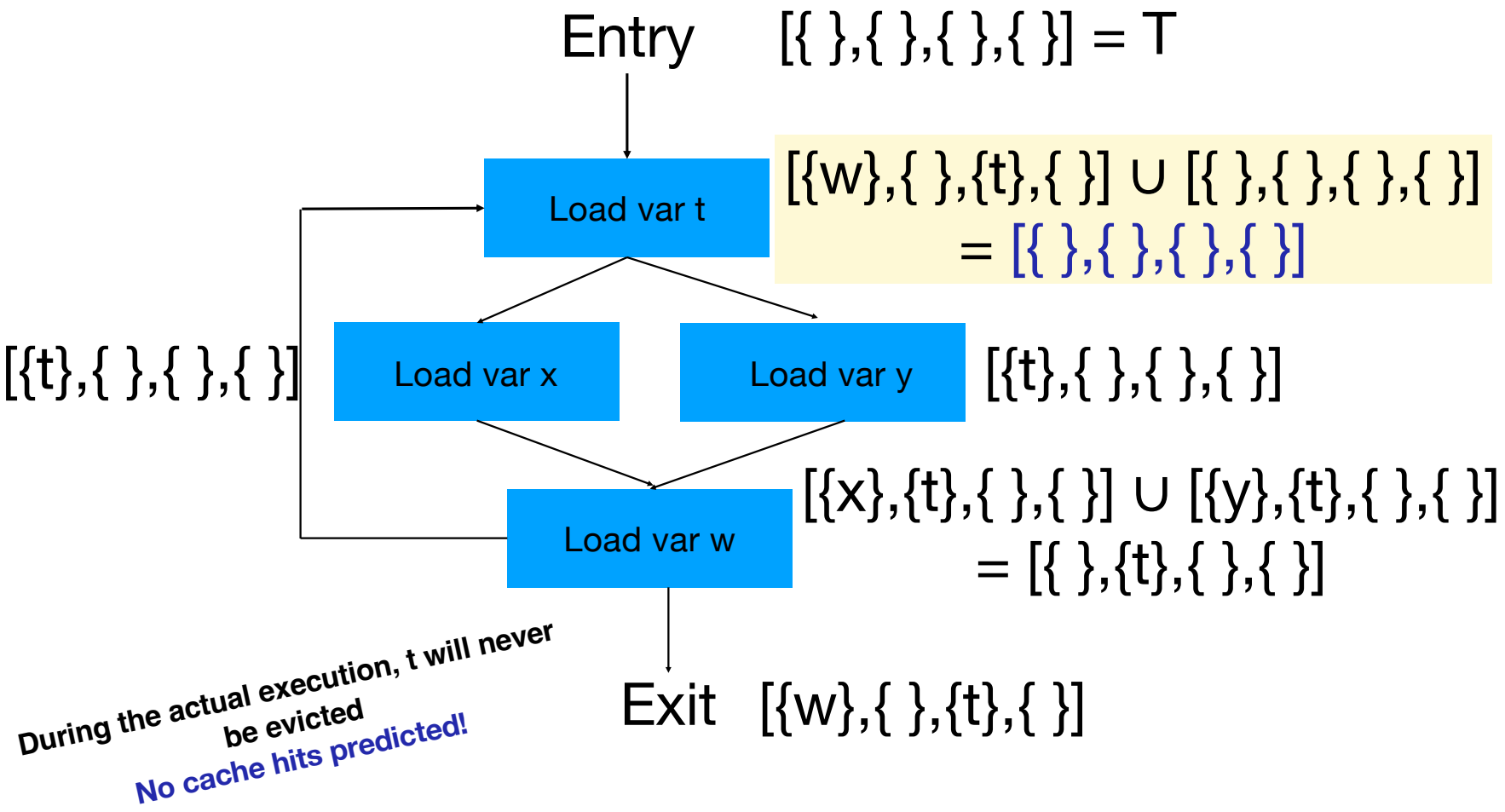
Back to Must Analysis: example



Back to Must Analysis: example



Back to Must Analysis: example



Improving Precision by Using Contexts

Only relying on classifying analyses, such as Must and May analyses, may still largely over-estimate the memory-accesses

Problem

- The first iteration of a loop will always result in cache misses
- Similarly for the first execution of a function

Improving Precision by Using Contexts

Only relying on classifying analyses, such as Must and May analyses, may still largely overestimate the memory-accesses

Problem

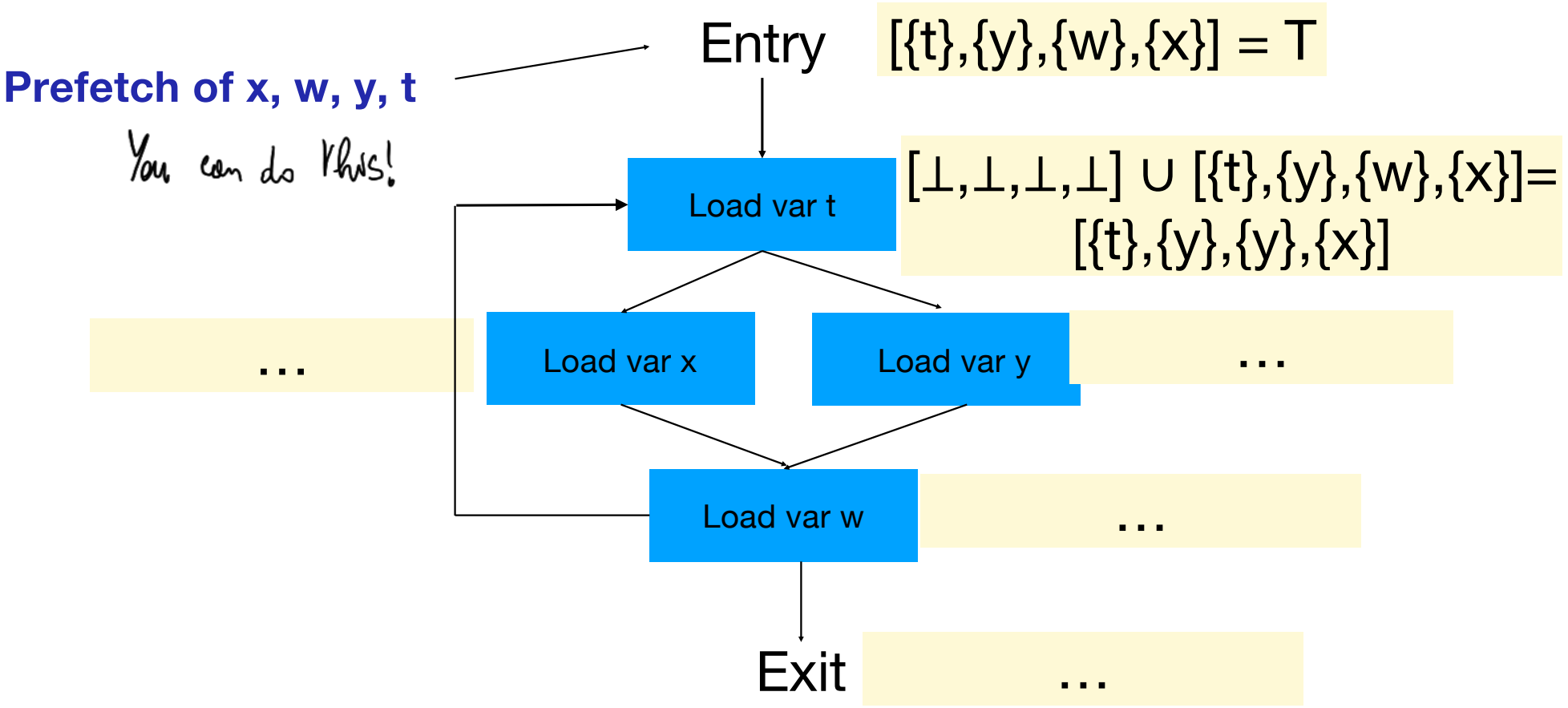
- The first iteration of a loop will always result in cache misses
- Similarly for the first execution of a function

Solution

Context-sensitivity is needed to enhance precision

- Virtual unrolling of loops: distinguish the first iteration from the others (e.g., in our example, accesses to t and w are probably hits after the first iteration)
- Virtual Inlining: distinguishing function calls by calling contexts

Back to Must Analysis: example



Possible optimizations

A possible optimization is to introduce

- a new classification, besides always hit and always miss:"**defined unknown**", and

- a further cache analysis that safely establishes that some blocks are "**defined unknown**" under LRU replacement

Analysis for Definitely Unknown

A memory access is "**definitely unknown**"

- if there is a concrete execution in which the access misses and another in which it hits.
- The aim of this new analysis is to prove the existence of such executions to classify an access as "**definitely unknown**"

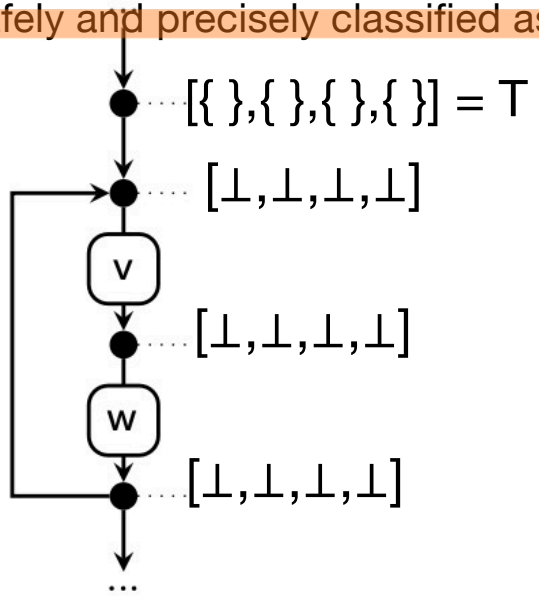
- Note the differently from classical may/must analysis and most other abstract interpretations, which compute properties that hold **for all executions**, here the question is to prove that **there exist** two executions with suitable properties

Example

- Standard **may** and **must** cache analyses classify the memory accesses to v and w as **unknown**, because the first iteration causes misses and subsequent ones result in hits—yielding no definitive outcome
- **EH** and **EM** analyses provide complementary insights:
  - **EH** shows that hits are possible
  - **EM** that misses are possible

[Exists hit and Exists miss analyses. We can provide a classification]

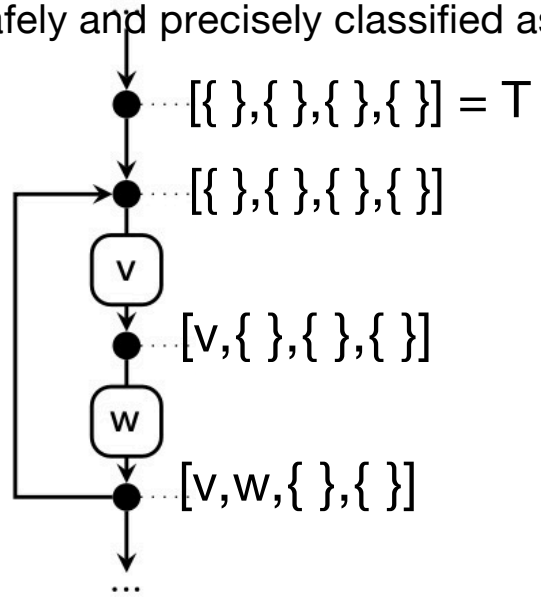
- Combining these, the accesses to **v** and **w** are safely and precisely classified as **definitely unknown**
  - they both miss in the first loop iteration, while
  - they hit in all subsequent iterations



Example

- Standard **may** and **must** cache analyses classify the memory accesses to v and w as **unknown**, because the first iteration causes misses and subsequent ones result in hits—yielding no definitive outcome
- **EH** and **EM** analyses provide complementary insights:
  - **EH** shows that hits are possible
  - **EM** that misses are possible

- Combining these, the accesses to **v** and **w** are safely and precisely classified as **definitely unknown**
  - they both miss in the first loop iteration, while
  - they hit in all subsequent iterations



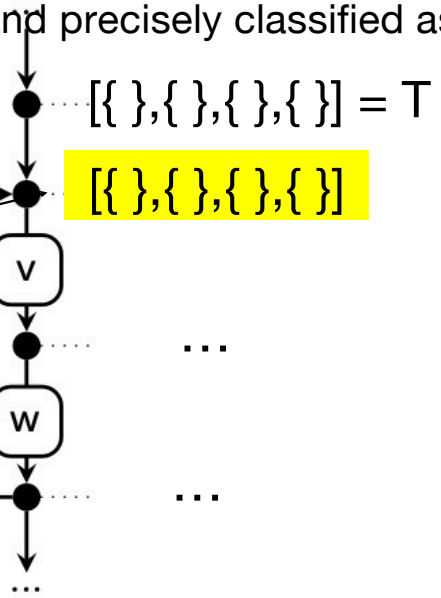
Example

- Standard **may** and **must** cache analyses classify the memory accesses to v and w as **unknown**, because the first iteration causes misses and subsequent ones result in hits—yielding no definitive outcome
- **EH** and **EM** analyses provide complementary insights:
  - **EH** shows that hits are possible
  - **EM** that misses are possible

- Combining these, the accesses to **v** and **w** are safely and precisely classified as **definitely unknown**
  - they both miss in the first loop iteration, while
  - they hit in all subsequent iterations

$[v,w,\{\},\{\}] \sqcup [\{\},\{\},\{\},\{\}] =$
$[\{\},\{\},\{\},\{\}]$

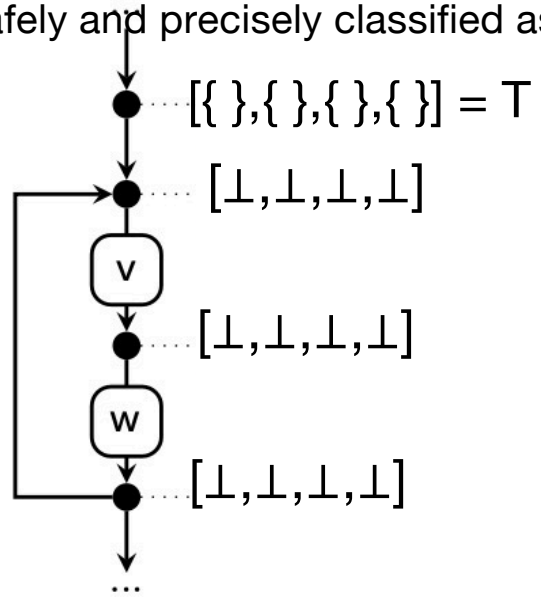
here I would lose info



Example

- Standard **may** and **must** cache analyses classify the memory accesses to v and w as **unknown**, because the first iteration causes misses and subsequent ones result in hits—yielding no definitive outcome
- **EH** and **EM** analyses provide complementary insights:
  - **EH** shows that hits are possible
  - **EM** that misses are possible

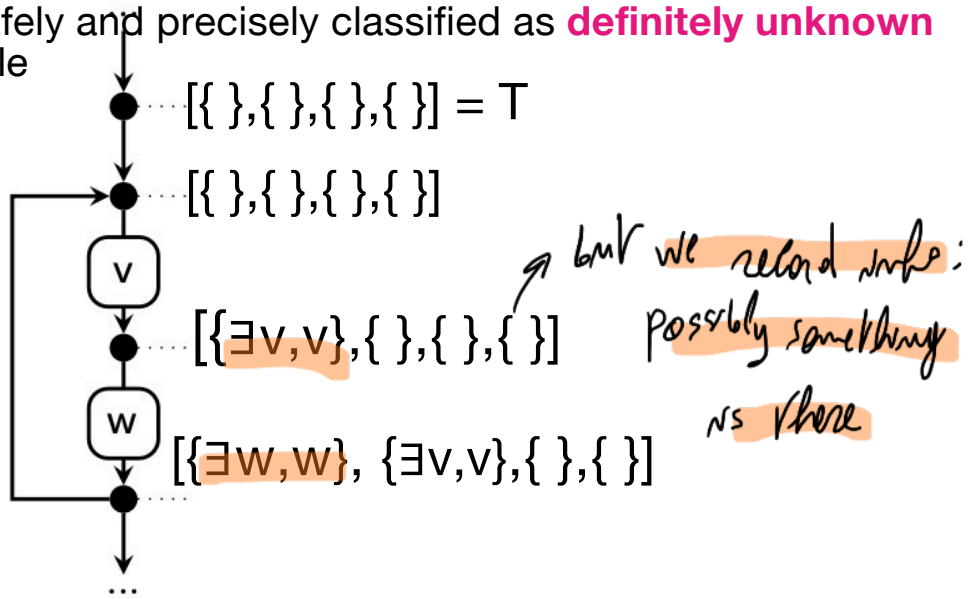
- Combining these, the accesses to **v** and **w** are safely and precisely classified as **definitely unknown**
  - they both miss in the first loop iteration, while
  - they hit in all subsequent iterations



Example

- Standard **may** and **must** cache analyses classify the memory accesses to v and w as **unknown**, because the first iteration causes misses and subsequent ones result in hits—yielding no definitive outcome
- **EH** and **EM** analyses provide complementary insights:
  - **EH** shows that hits are possible
  - **EM** that misses are possible

- Combining these, the accesses to **v** and **w** are safely and precisely classified as **definitely unknown**
  - **they both miss in the first loop iteration**, while
  - they hit in all subsequent iterations

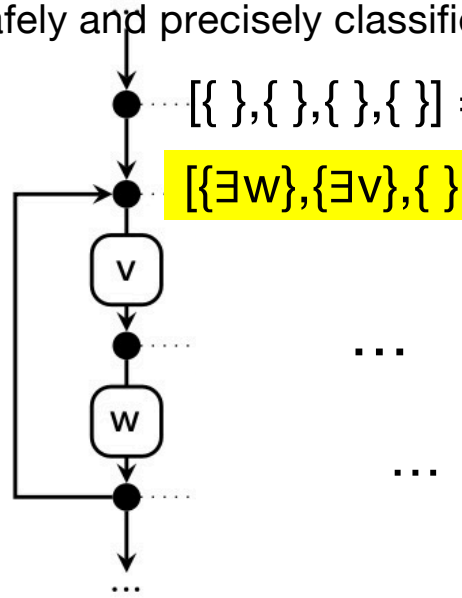


Example

- Standard **may** and **must** cache analyses classify the memory accesses to v and w as **unknown**, because the first iteration causes misses and subsequent ones result in hits—yielding no definitive outcome
- **EH** and **EM** analyses provide complementary insights:
  - **EH** shows that hits are possible
  - **EM** that misses are possible

- Combining these, the accesses to **v** and **w** are safely and precisely classified as **definitely unknown**
  - they both miss in the first loop iteration, while
  - **they hit in all subsequent iterations**

$$[\{\exists w, w\}, \{\exists v, v\}, \{\,\}, \{\,\}] \sqcup [\{\,\}, \{\,\}, \{\,\}, \{\,\}] = [\{\exists w\}, \{\exists v\}, \{\,\}, \{\,\}]$$



Analysis for Definitely Unknown: EH intuition

An access to a block **a** results in a **hit**

- if **a** has been accessed recently, i.e., **a**'s age is low

- We would need to determine the minimal age that **a** may have in a reachable cache state immediately prior to the access in question

- The access can be a **hit** if and only if this minimal age is lower than the cache's dimension: **minAge(a) ≤ N**

- Instead of computing exact minimal ages, an **Exists Hit (EH) analysis** to compute **safe upper bounds on minimal ages**

Analysis for Definitely Unknown: EM intuition

An access to a block **a** results in a **miss**

- if **a** has not been accessed recently, i.e., **a**'s age is high

- We would need to determine the maximal age that **a** may have in a reachable cache state immediately prior to the access in question

- The access can be a **miss** if and only if this maximal age is higher than the cache's dimension: **minAge(a) ≥ N**

- Instead of computing exact maximal ages, an **Exists Miss (EM) analysis** to compute **safe lower bounds on maximal ages**

Analysis for Definitely Unknown

- EH analysis to determine that an access results in a hit in at least some of all possible executions: if the minimum age of the block prior to the access is guaranteed to be less than the cache's dimension

- Similarly, EM analysis can be used to determine that an access results in a miss in at least some of the possible executions: if this maximal age of the block prior to the access is higher than the cache's dimension

Summary

Cache analysis for LRU efficiently represents sets of cache states by bounding the age of memory blocks from **above** and from **below**

- The block age bounds capture precisely the information required to classify blocks as **cached** or **uncached**

- This information can be precisely maintained by the transfer functions due to LRU's regular cache update:
  - the new block's age only depends on how its previous age relates to the accessed block's age
  - upper and lower bounds can be precisely updated due to the monotonicity of the operation, regardless of its previous age, and whether it was cached or not, the accessed block is always assigned the youngest age

Most non-LRU replacement policies do not possess such monotone behavior

Bibliograhy

- Mingsong Lv, Nan Guan, Jan Reineke, Reinhard Wilhelm, Wang Yi:
  A Survey on Static Cache Analysis for Real-Time Systems.
  Leibniz Trans. Embed. Syst. 3(1): 05:1-05:48 (2016)

End