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

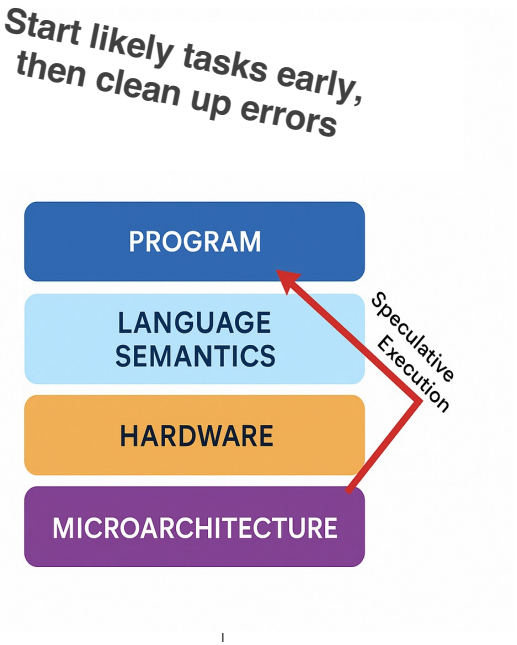
Speculative execution is a CPU-level optimization, where the processor predicts the outcome of a branch and executes instructions ahead of time to improve performance.

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

Analyses that include speculative behaviors have been developed

We will focus on secure compilation to account for speculative leakage

Outline

Memory Hierarchyand cache

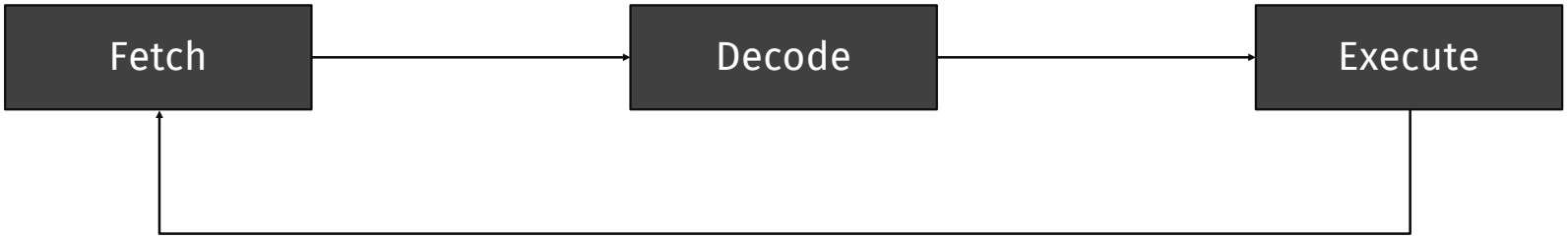
Cache Static Analysis

Speculative Execution

Security issues

Computer architecture

Do you remember how processors work?



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

No more easy gains from low-level physics!

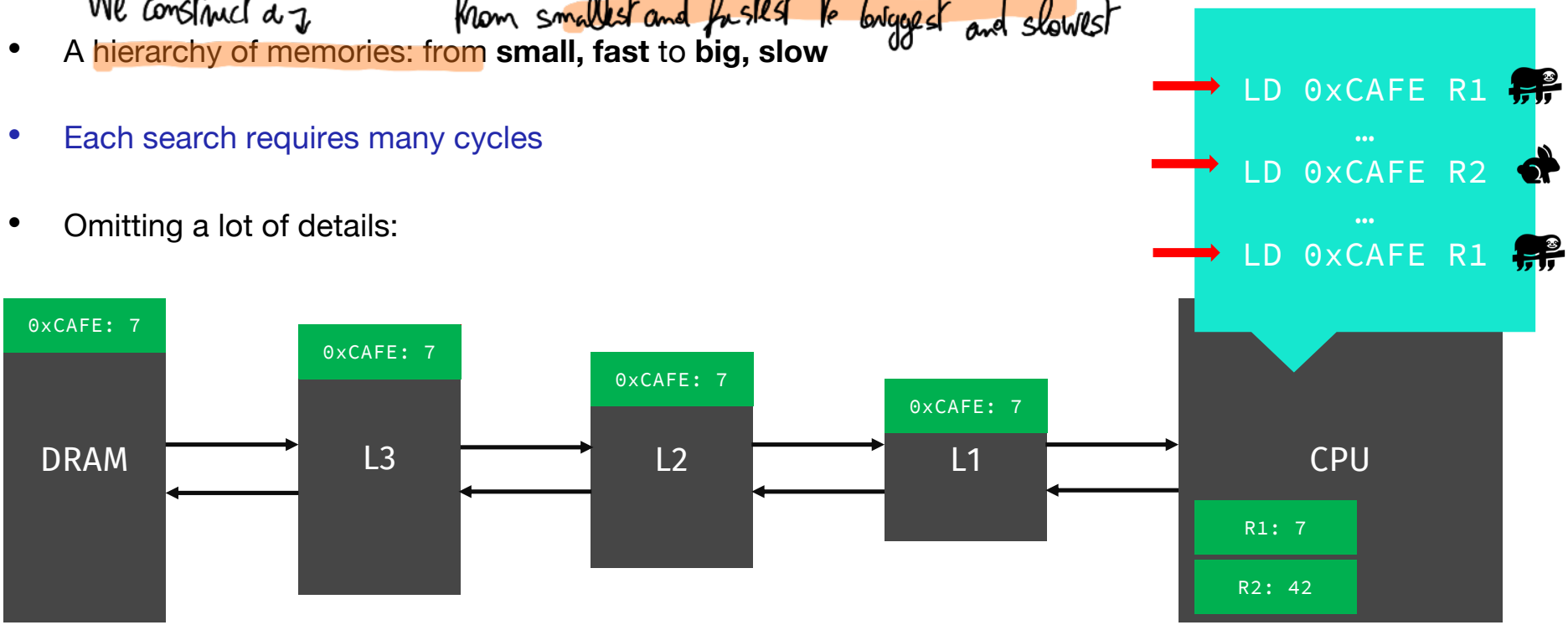
Computer architecture

- **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

We construct a ↓ from smallest and fastest to biggest and slowest

- A hierarchy of memories: from **small, fast** to **big, slow**
- 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

You follow a chain of cache queries, if L1 doesn't have it, go to L2, then L3 etc. Then propagate down the value.

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

caches were introduced to reduce bottleneck of memory accessing

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

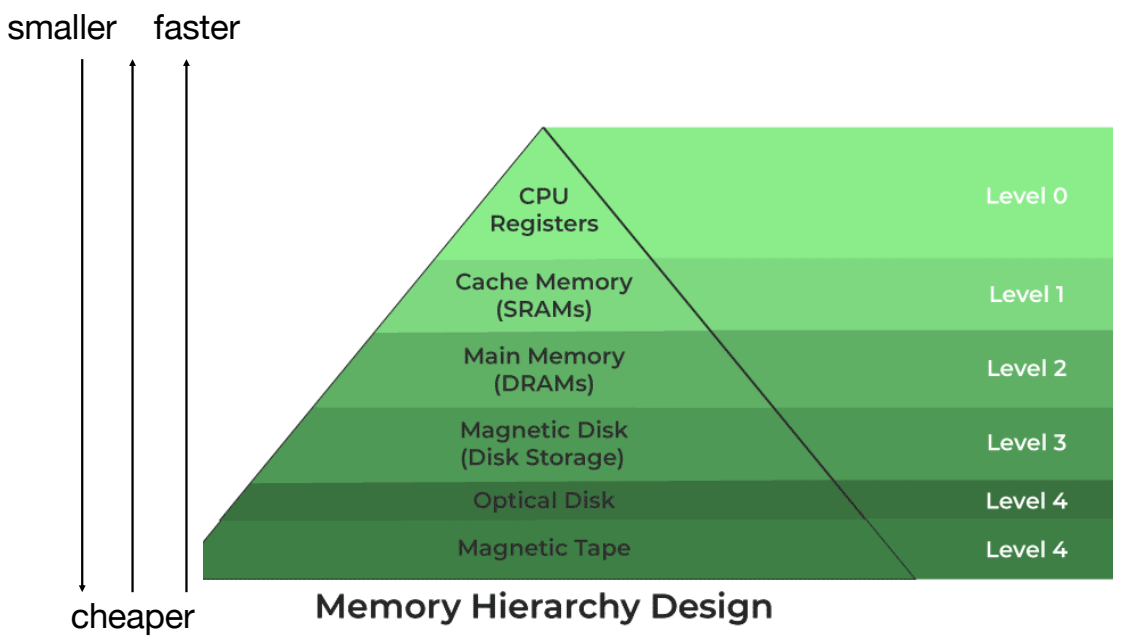
Memory hierarchy overview

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]

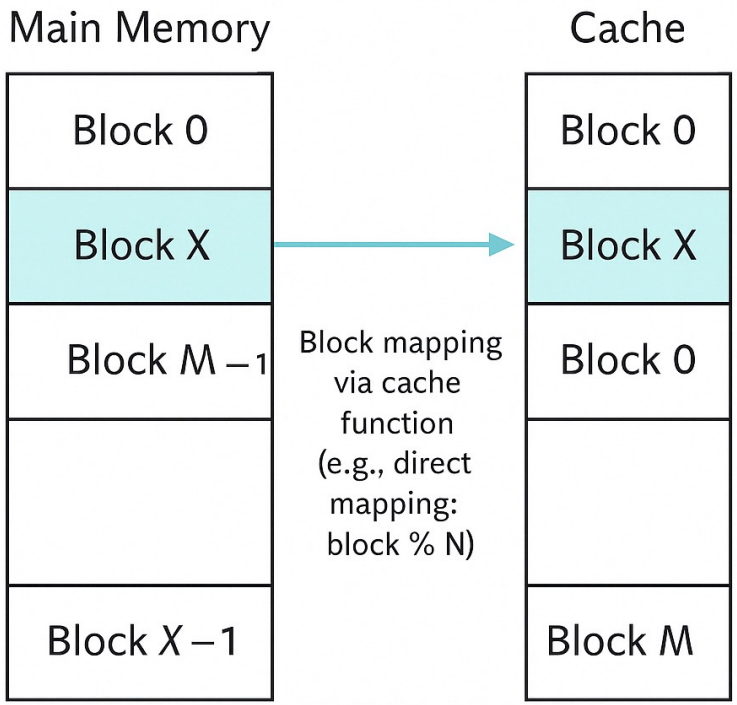
Memory blocks being read sequentially

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



You need a mapping allowing you to associate blocks from memory to cache blocks.

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:

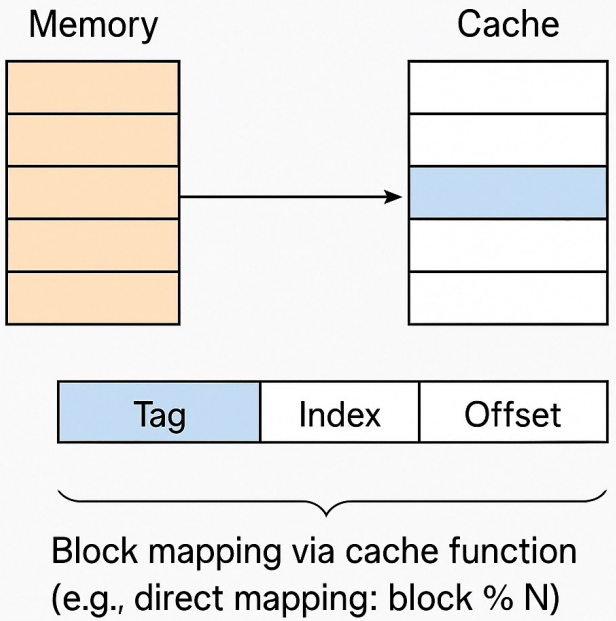
↱ we just update the cached copy and mark that cache entry as dirty to mark misalignment between memory and cache

- **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

↱ every write updates every level in the hierarchy

- **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: there must be a rule for mapping blocks in lines
- 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 (in the cache)
  - Offset: location inside block

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

Mappings are reminiscent of hash table mappings for conflict purposes

Is the tag-index-offset way to identify a memory location something that ismdkne in particular for caches? I mean, is it a way to define memory locations inside of caches?

Yes, Giovanni, the *tag-index-offset* scheme is something that's specifically used in the context of **cache memory**, not in general memory addressing.

When a cache is trying to quickly figure out whether it already has a copy of the data from main memory, it splits a memory address into three parts:

- **Offset**: Tells you where exactly the data is inside a cache block (since each block can hold multiple bytes).
- **Index**: Tells you which *cache set* or *cache line* to look into.
- **Tag**: This is the unique identifier that matches the actual memory block. It helps the cache confirm whether the data in a given line is really the one we're looking for.

So yeah, this breakdown is a cache's internal mechanism—it helps it organize and find stuff efficiently without searching the whole thing.

So if i queried a cache i would need to provide those three valuex, correct?

That's a really good instinct, Giovanni, but here's the twist: **you don't provide the tag, index, and offset separately**—you just provide the full **memory address**, and **the cache breaks it down internally** into tag, index, and offset.

So when you query a cache (say, the CPU wants to read from a specific address), the hardware does this behind the scenes:

Mapping Techniques

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

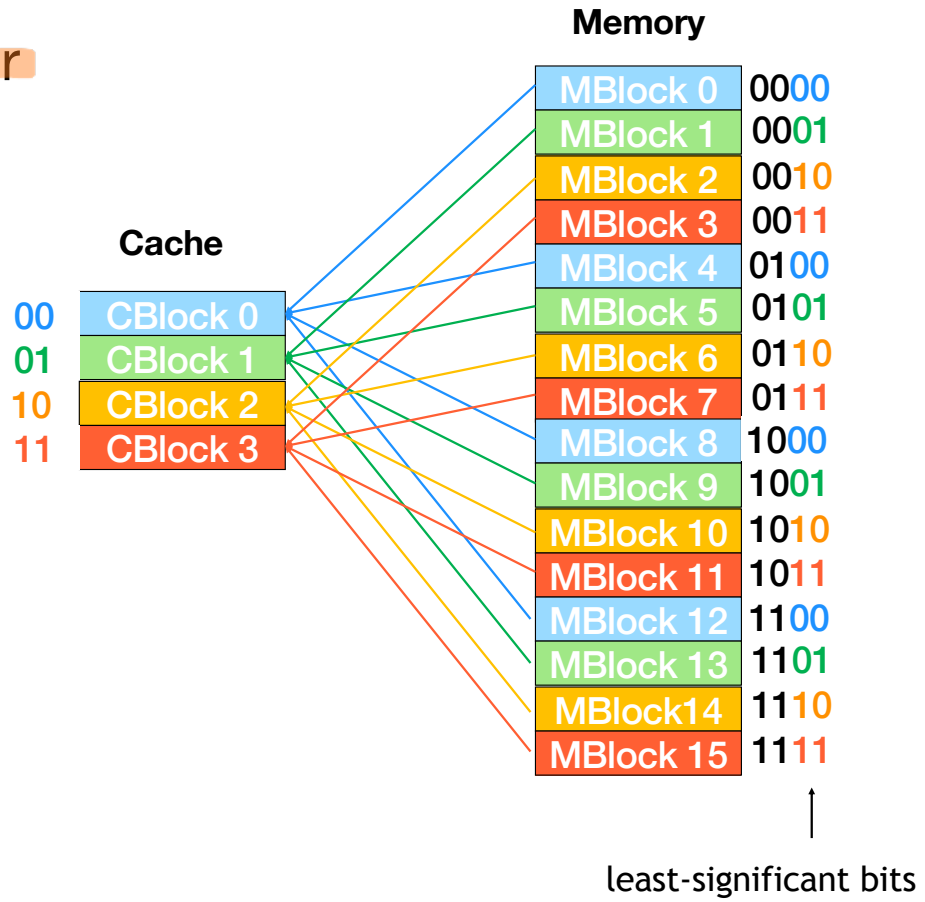
- Direct Mapping
- Fully Associative Mapping
- Set-Associative Mapping

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*

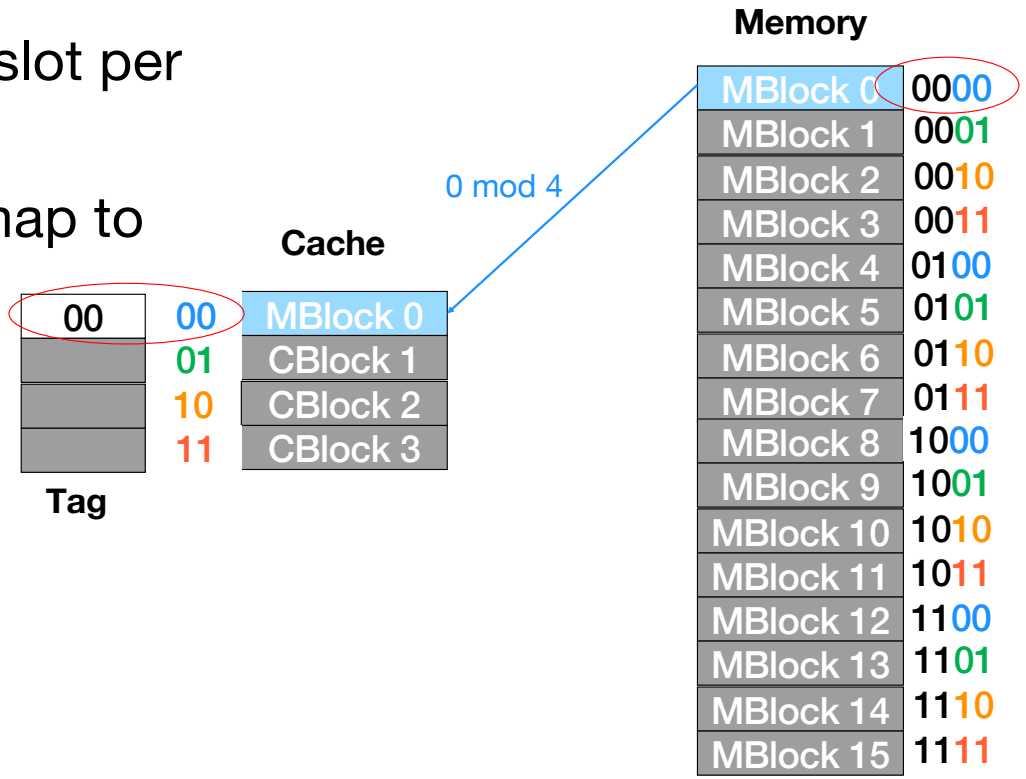


- You get tag of a block and look at line associated to that tag.

Direct Mapping

Cache resembles a hash table with one slot per bucket

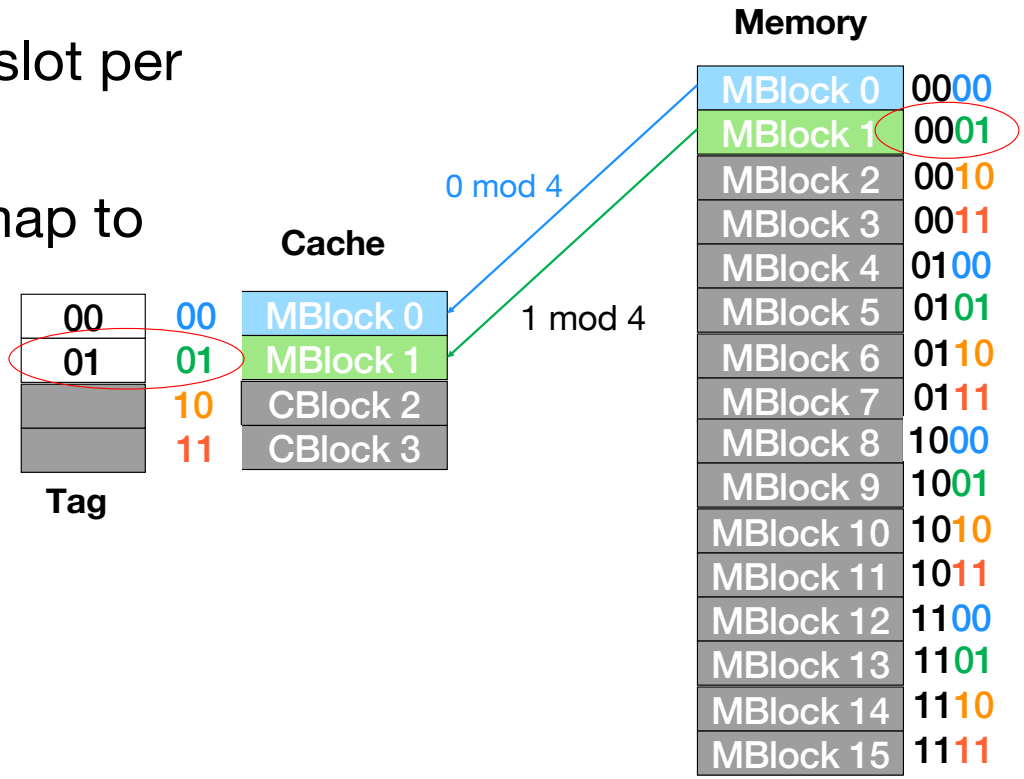
- Each main memory block will always map to the same cache block: *i mod n*



Direct Mapping

Cache resembles a hash table with one slot per bucket

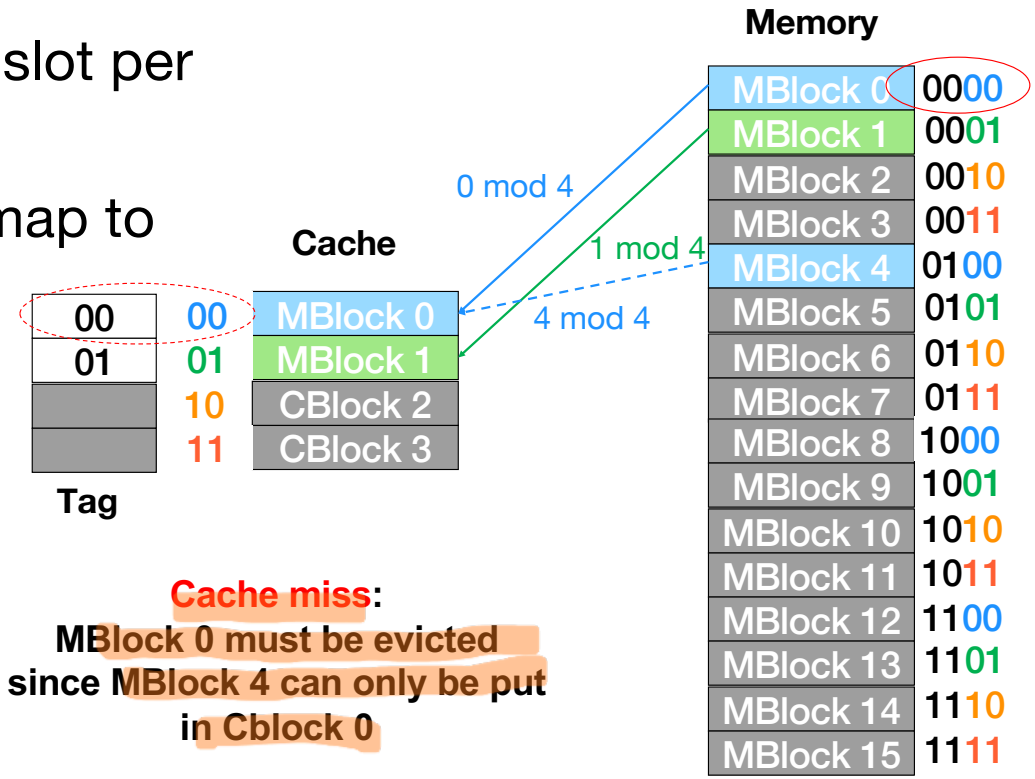
- Each main memory block will always map to the same cache block: *i mod n*



Direct Mapping

Cache resembles a hash table with one slot per bucket

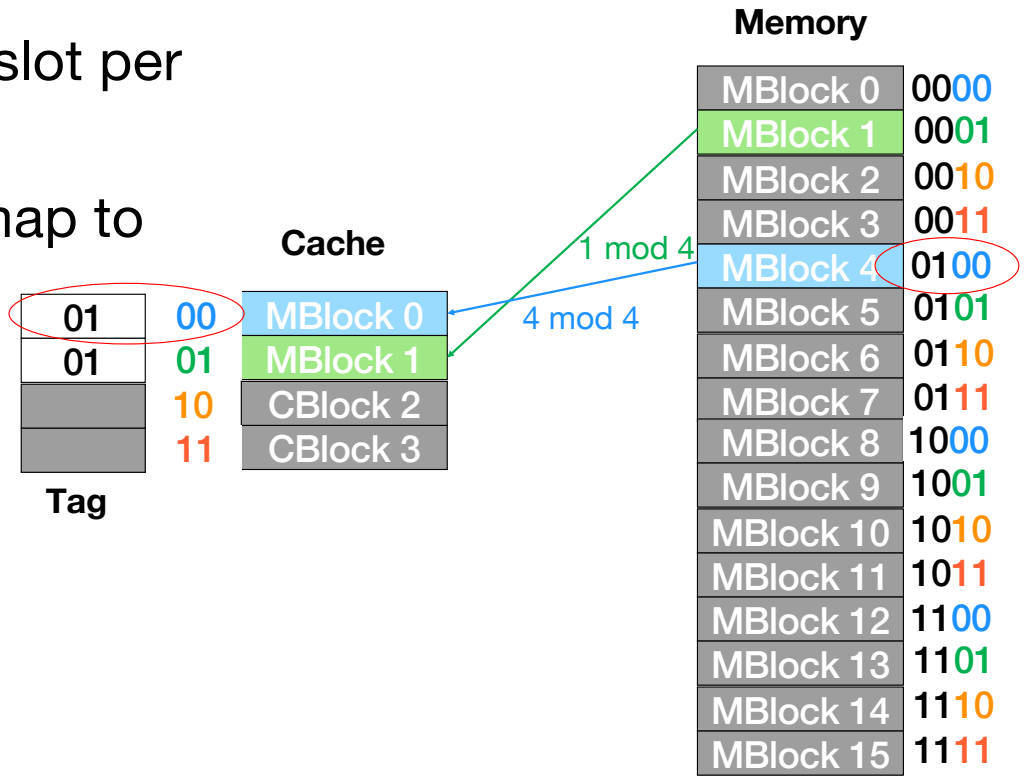
- Each main memory block will always map to the same cache block: *i mod n*
- Collisions lead to evictions



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:

block mapping is easy to be computed

- 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

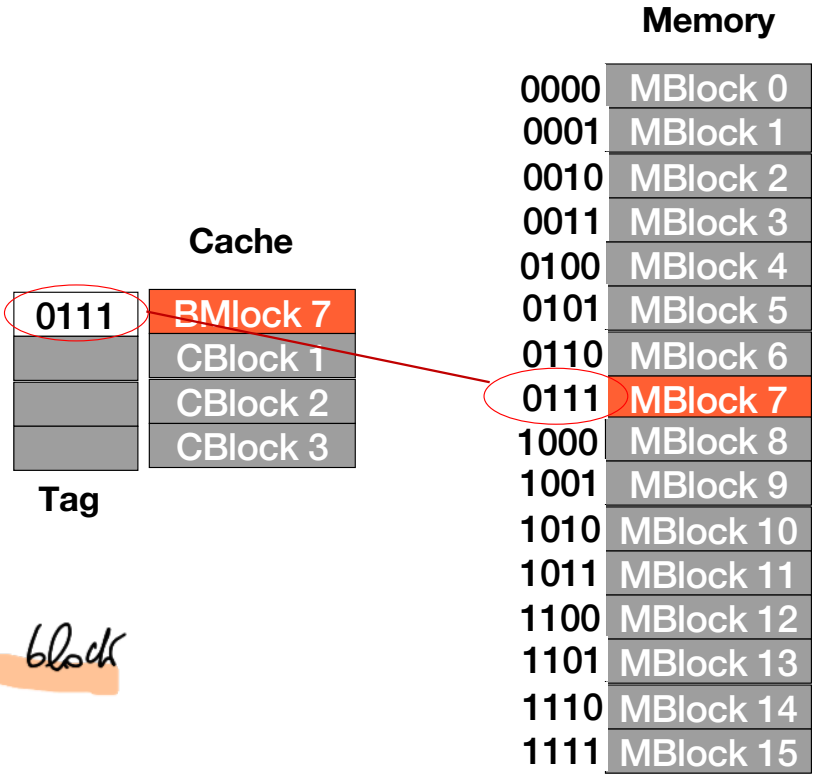
→ can have a lot of replacements

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

Fully Associative Mapping

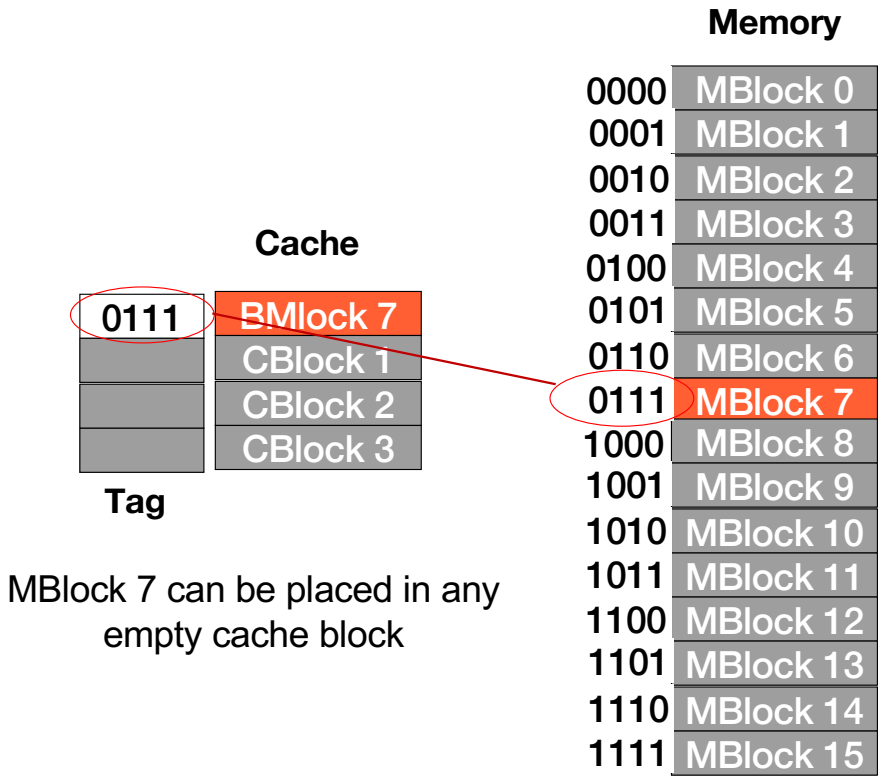
- A Main Memory block can be stored in any cache block (any unused block suits)
- Since data could be anywhere in the cache, we must check the tag of every cache block

- To check if something is there, we need to check if the associated key is in ANY 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:

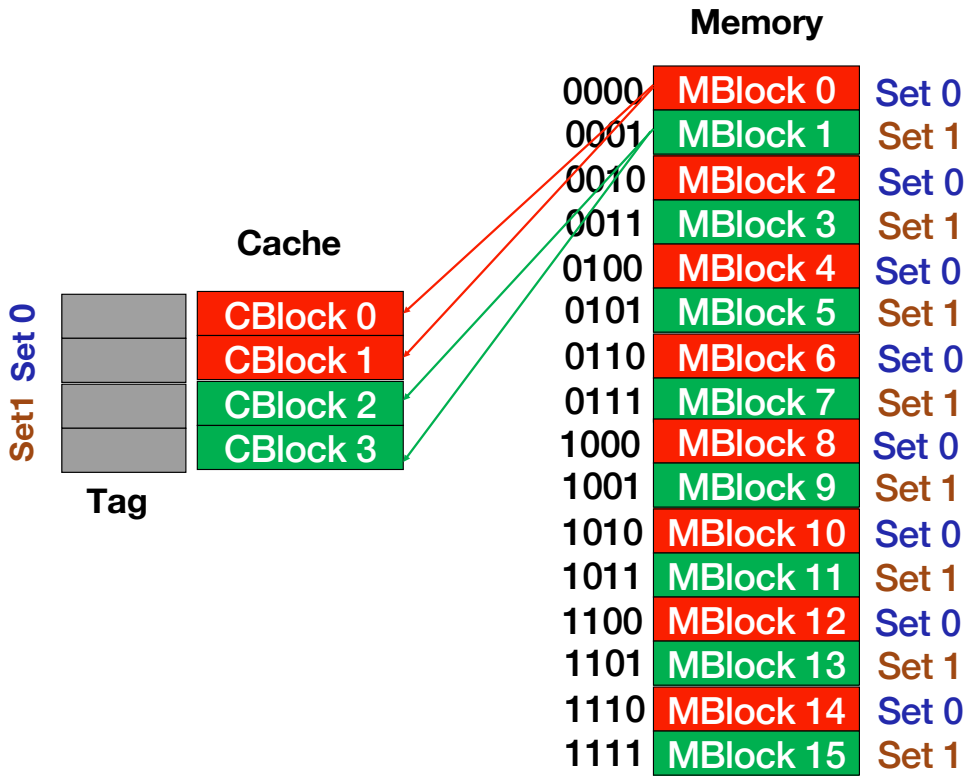
- **LRU** (Least Recently Used): discards least recently used items first, exploiting the locality principle, according to which if a block has not been used in a while, it will not be needed again anytime soon
- **LFU** (Least Frequently Used): similar to LRU, except that how many times a block was 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)

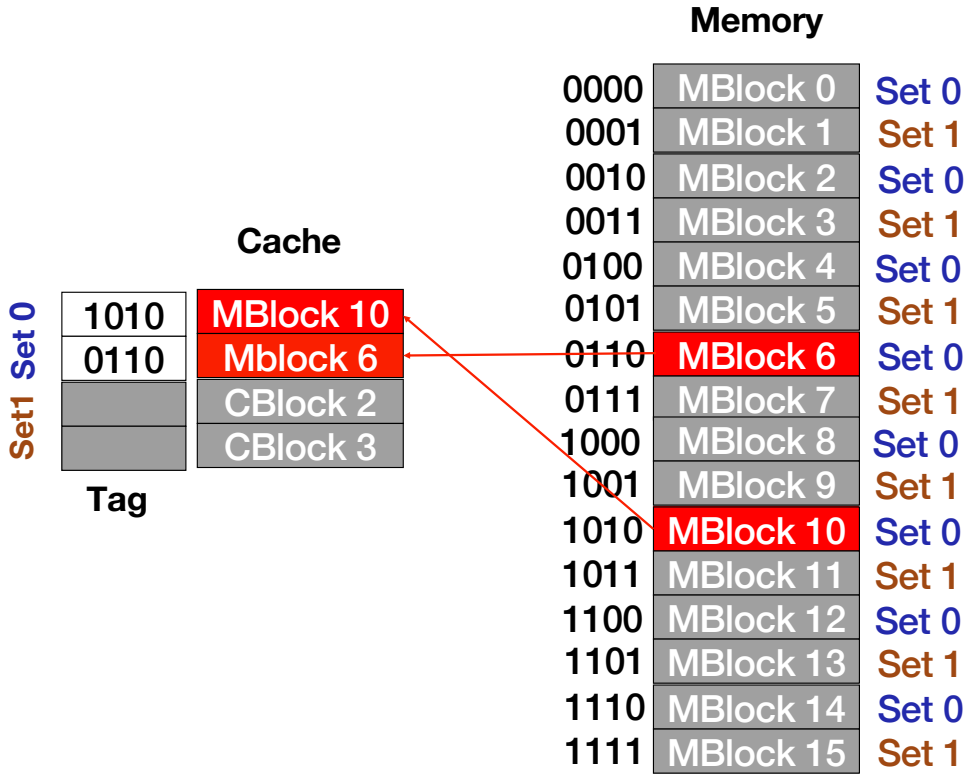


K-Set Associative Mapping

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)

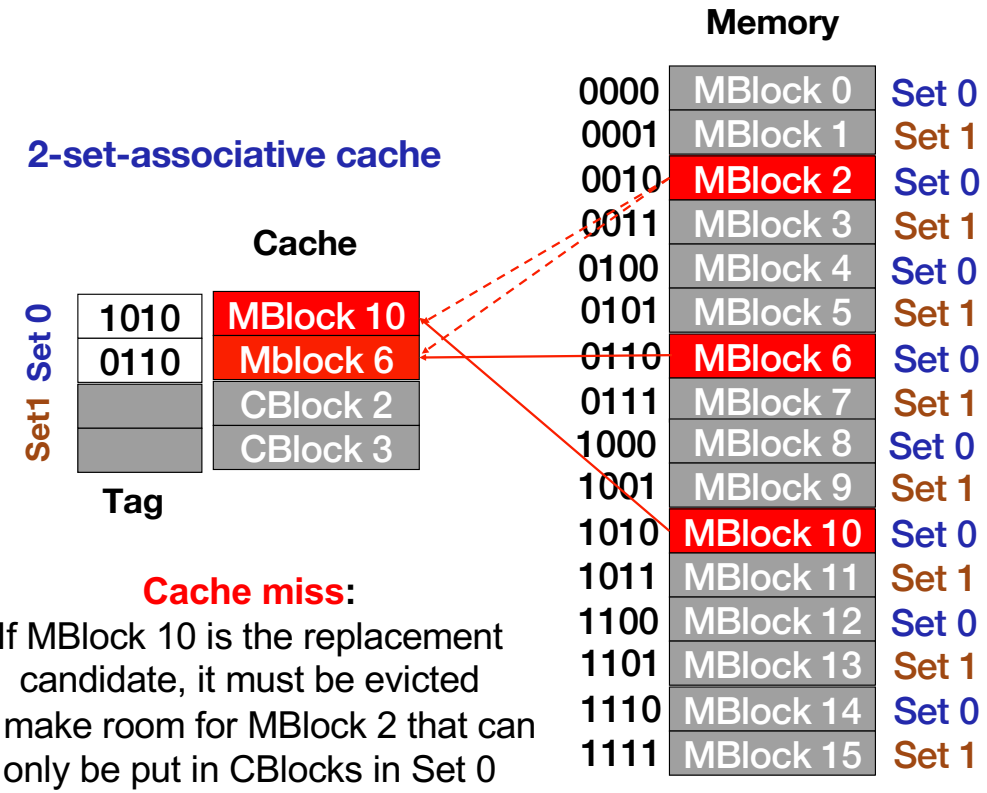


K-Set Associative Mapping

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)



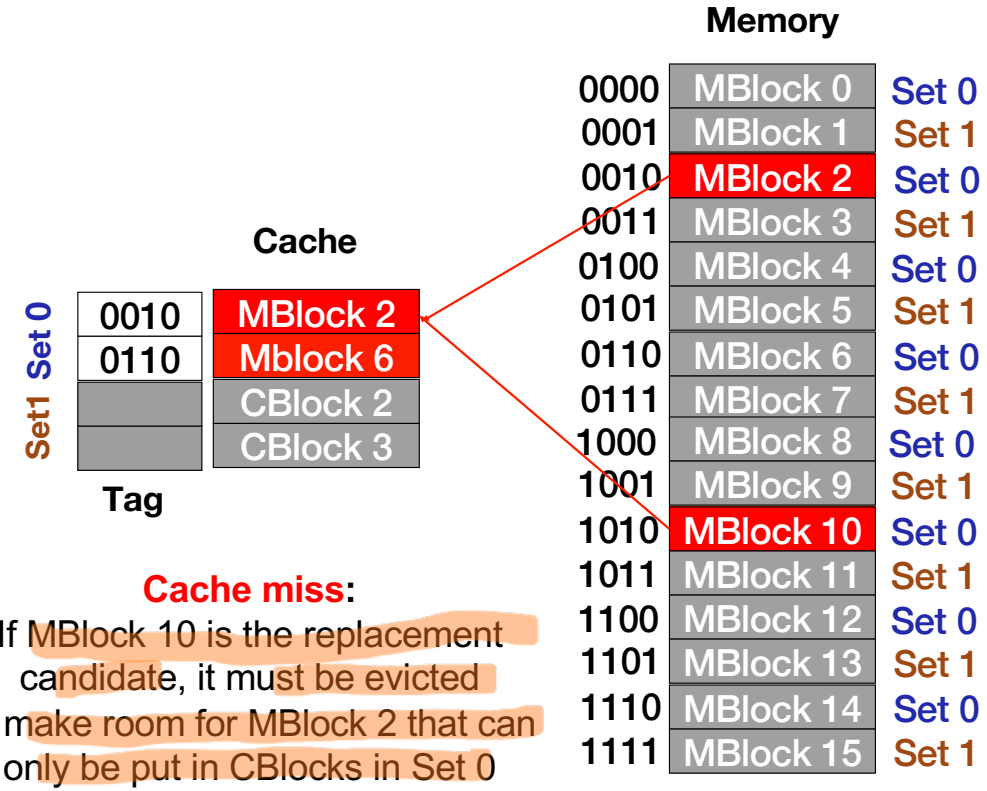
Cache miss:
If MBlock 10 is the replacement candidate, it must be evicted to make room for MBlock 2 that can only be put in CBlocks in Set 0

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

STOP HERE GO TO NEXT PPT

Outline

- Memory Hierarchyand cache
- Cache Static Analysis
- Speculative Execution
- Security issues

CACHE STATIC ANALYSIS

There are 2 possible analyses

Two types of static analysis:

↳ have a classification of cache accesses, understand if there are misses or hits.

- Local guarantees for individual accesses [classifying cache analysis]
  - **May analysis**: to overapproximate cache content — May miss
  - **Must analysis**: to underapproximate cache content — Must hit

↳ other analysis, not interested

- Global guarantees [persistence analysis]: bounds on cache misses and hits

- Focus on LRU strategy and on classifying cache analysis — So we can easily use monotone FW

CACHE STATIC ANALYSIS

The first cache analysis based on abstract interpretation (AI) was proposed by Ferdinand and Wilhelm in the 1990s

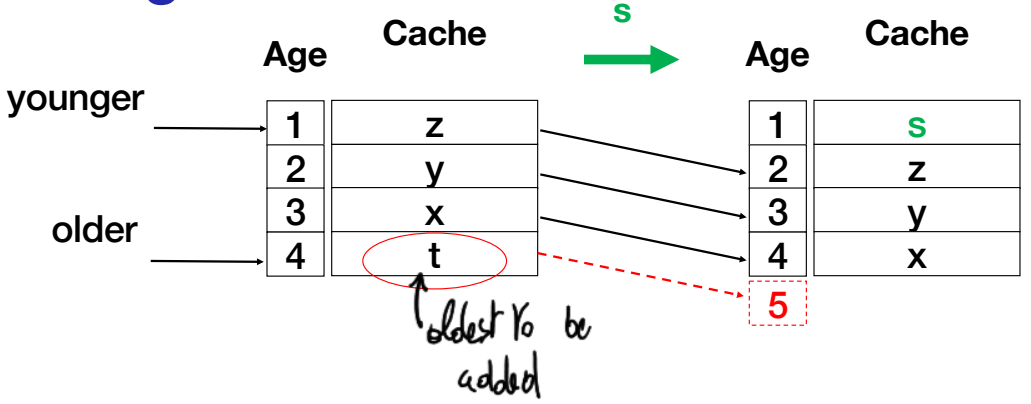
The overall approach works in two phases:

- An AI-based cache analysis computes abstract cache states at all program points as part of a fixed-point solution
- These abstract cache states are queried in order to classify memory accesses

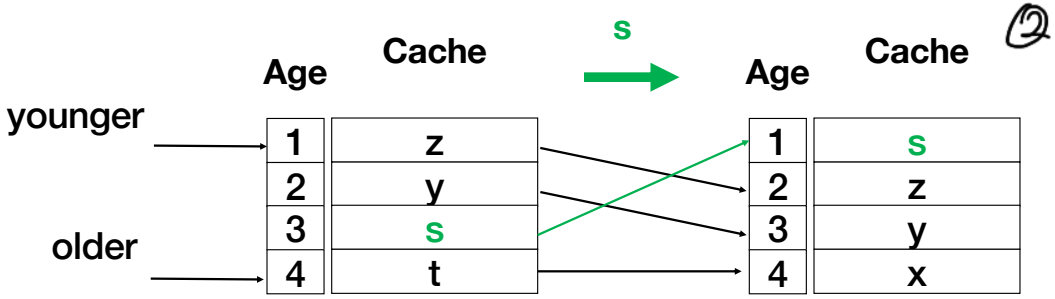
LRU concrete behaviour

LRU has a notion of age

Cache miss



Cache hit



LRU abstract behaviour

- $V = \{v_1,\ldots,v_n\}$ is the set of variables used in the program: each v_i 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
  - Age(v) = N means that v is the least recently used variable, and so the next candidate for eviction
  - Age(v) = N+1 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

- Collect at each program point the set S of all concrete cache states that are possible when program control reaches this program point
- One can represent sets of concrete cache states by abstract cache states
- **Abstract Must cache state**: information on which memory blocks will be in each of the possible concrete cache states in S.
- **Abstract May cache state**: information on which memory blocks may be in one of the concrete cache states in S

Must Analysis: abstract behaviour

Must analysis: to underapproximate cache content and to predict cache hits

It determines if a memory block is definitely in the cache, by computing abstract set states where the position (the relative age) of a memory block in the abstract set state S is an upper bound of the positions (the relative ages) of the memory block in the corresponding concrete set states

A **Must-Hit analysis** needs to compute, at each program location, an upper bound U of Age(v). If $U \leq N$,

- then v must be in the cache,
- else it is possible that v may be outside of the cache

Must Analysis

The must analysis for a program forms a complete finite join semi lattice

- The order is imposed by the set inclusion order of the underlying concrete domain (sets of concrete cache states)
- An (artificial) least element $\bot$ is used (in practice $\bot$ is used for initialization purposes for the fixpoint iteration
- In order to solve the must analysis for a program, one can construct a system of recursive equations from its control flow graph

Semilattice

In mathematics, a join-semilattice (or upper semilattice) is a partially ordered set that has a join (a least upper bound) for any nonempty finite subset.

Consider two abstract Must cache states $\hat{a}$ and $\hat{b}$ (as shown in Figure 5). Abstract cache state $\hat{a}$ represents just one concrete cache state, containing memory blocks $\{u, x, y, z\}$ with the ages 1, 2, 3, 4. Abstract cache state $\hat{b}$ represents the set of concrete cache states with memory block u having age 1, x having age at most 3, z having age at most 4, and possibly one more (unknown) block at age 2, 3, or 4. $\hat{a} \sqsubseteq \hat{b}$ means that all the concrete cache states represented by $\hat{a}$ are also represented by $\hat{b}$. In particular, this implies that all the memory blocks known to be contained in the concrete cache states described by $\hat{b}$, in the example above u, x, z, are also known to be contained in the concrete cache states described by $\hat{a}$. Furthermore, $\hat{a}$ additionally tells us that, (1) block y is guaranteed to be in the cache while $\hat{b}$ does not; (2) the age upper bound estimated for block x in $\hat{a}$ is smaller than that in $\hat{b}$. Clearly, the abstract cache state $\hat{a}$ contains better information than $\hat{b}$. As stated above, at control flow merge points cache analysis must combine the incoming information in a sound way. The operation applied to the incoming abstract cache states is the least upper bound, $\sqcup$, of the lattice. This is shown in Figure 6. As said above and made more precise later, it is some form of intersection. It determines (the best) safe information holding for all incoming paths.

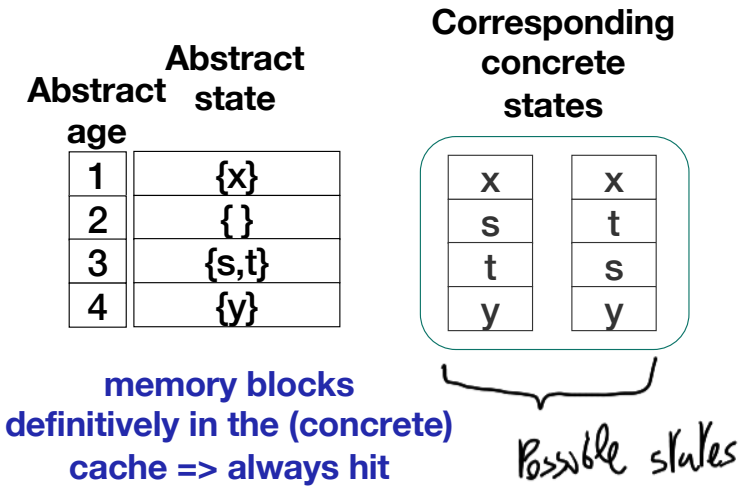
Must analysis: example

Consider a program with 4 variables x, s and t

The following abstract state describes the set of all concrete states in which x, s, t occur, with:

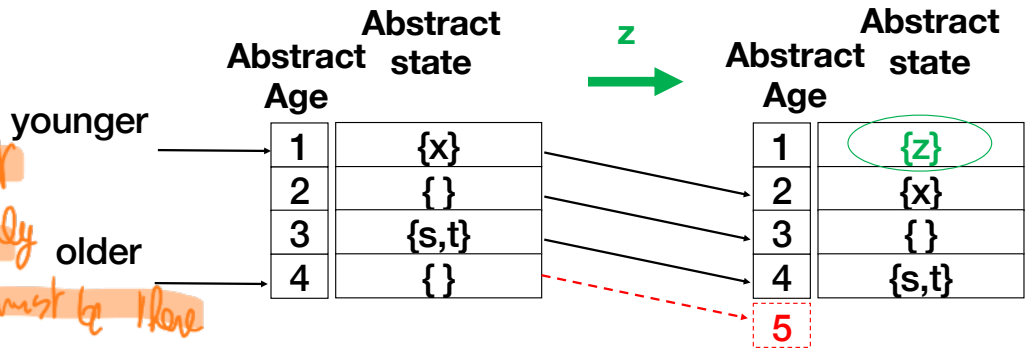
- x with age 1, and
- s and t with an age 3, i.e., not older than 3
- y with an age 4, i.e., not older than 4

Concrete age(v) ≤ Abstract age(v)

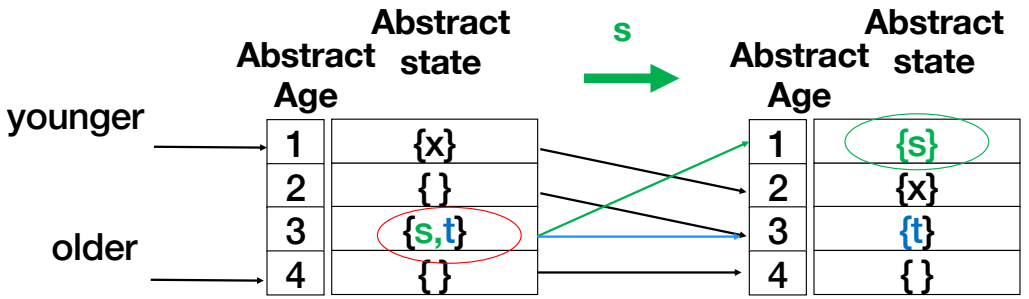


Must analysis: update on memory access

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



Potential Cache miss



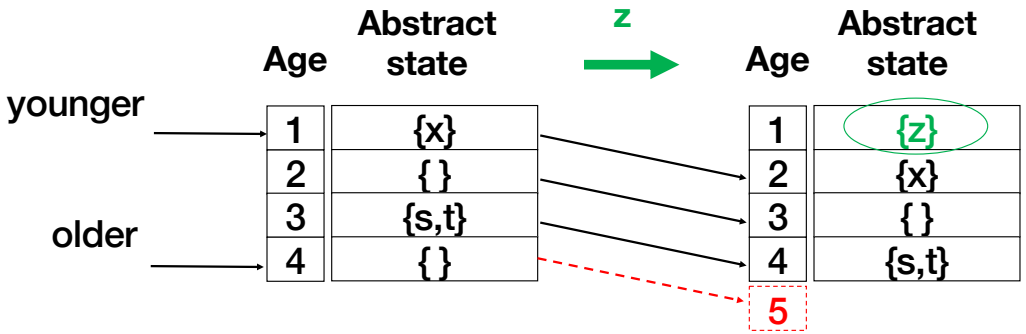
Definite Cache hit

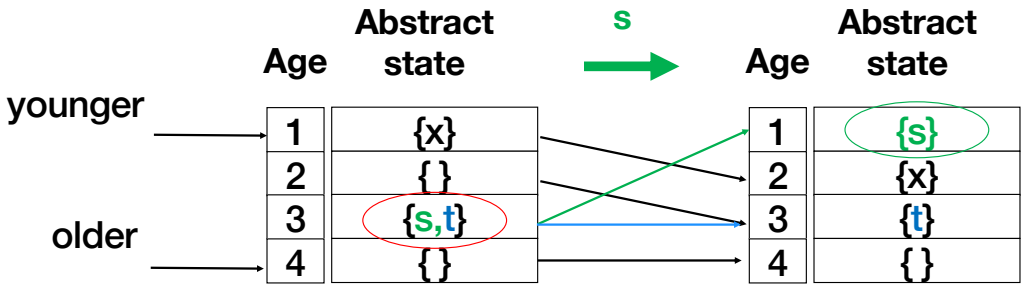
Must analysis: update update on memory access

Potential Cache miss

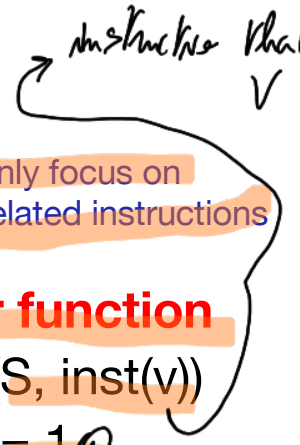
Definite Cache hit

S = ⟨Age(x),Age(s),Age(t)⟩ S′ = ⟨Age′(x),Age′(s),Age′(t)⟩





Why does t not age in the second case?



Transfer function

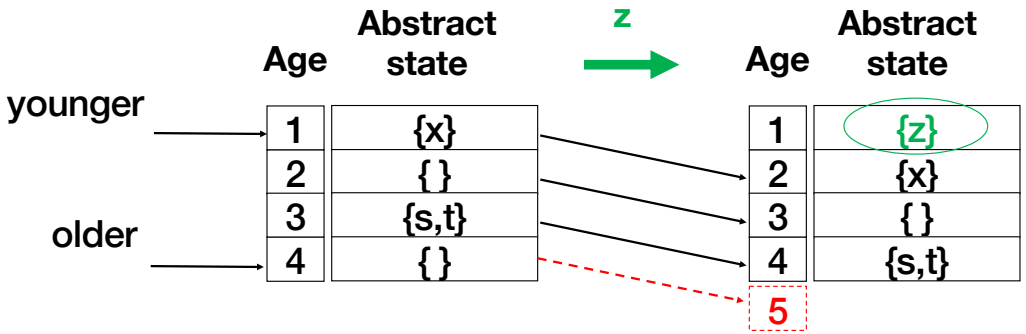
Transfer(S, inst(v))

- Age(v) = 1
- Age(u) < Age(v) ⇒ Age′(u) = Age(u) + 1
- Age′(w) = Age(w)

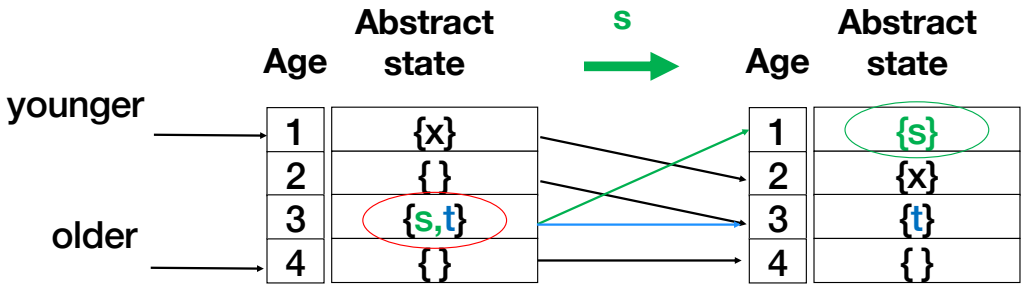
Must analysis: update update on memory access

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

Potential Cache miss



Definite Cache hit



LOADS

We only focus on memory-related instructions

Transfer function

Transfer(S, inst(v))

- Age(v) = 1
- Age(u) < Age(v) ⇒ Age′(u) = Age(u) + 1
- Age′(w) = Age(w)

Why does t not age in the second case?

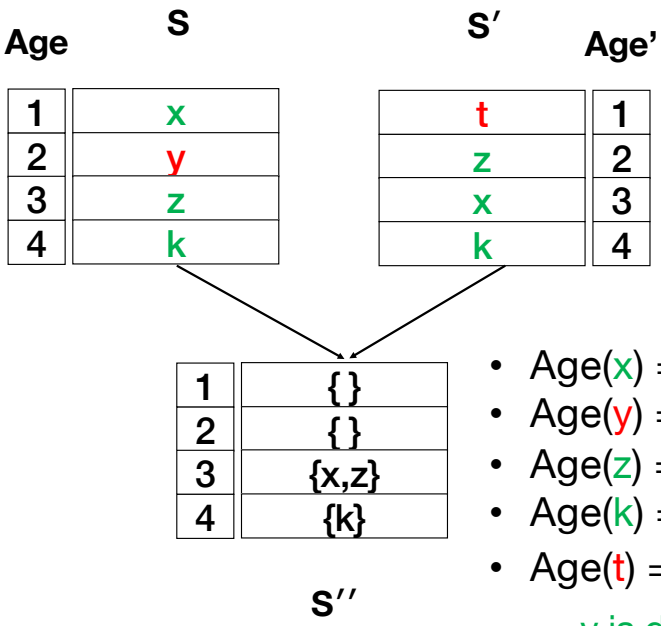
The abstract set state is an **upper bound**: the introduction of s implies a **definite** cache hit.
Hence, t has an age not older than 3

Must Analysis: join

How to combine information?

Intersection (+ maximal age)

It is a forward analysis You pick maximal age: if y is not on the cache, it is age 5.



- 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

- $S = \langle Age(v_1), \ldots, Age(v_n)\rangle$ and
- $S' = \langle Age(v'_1), \ldots, Age(v'_n)\rangle$, as follows:
- $S'' = S \sqcup S' = \langle \mathbf{max}(Age(v_1), Age(v'_1)), \ldots, \mathbf{max}(Age(v_n), Age(v'_n))\rangle$

May Analysis: abstract behaviour

May analysis: to overapproximate cache content and to predict cache misses

- It determines if a memory block is never in the cache, by computing the set of all memory blocks that may be in the cache.
- Abstract set states S, where the position (the relative age) of a memory block in the abstract set state is a lower bound on the positions (the relative ages) of the memory blocks in the corresponding concrete set states

A **May-Hit analysis** needs to compute, at each program location, a lower bound U of Age(v). If $U \geq N$,

- then v is definitely **not** cached
- else it is possible that v may be inside the cache

May analysis: example

Consider a program with 3 variables x, s and t

The following abstract state describes the set of all concrete states in which x, s, t occur, with:

- x and y with an age at least (not younger than) 1, and
- s and t with an age 3, i.e., at least 3
- u with an age 4, i.e., at least 4

Concrete age(v) ≥ Abstract age(v)

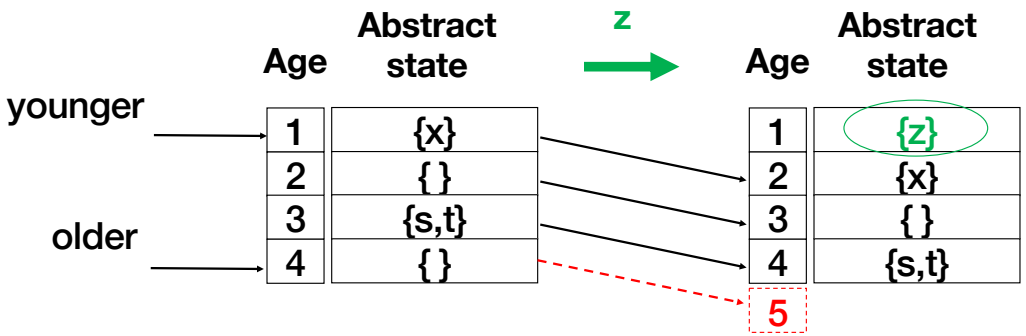
Age	Abstract state
1	{x,y}
2	{ }
3	{s,t}
4	{u}

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

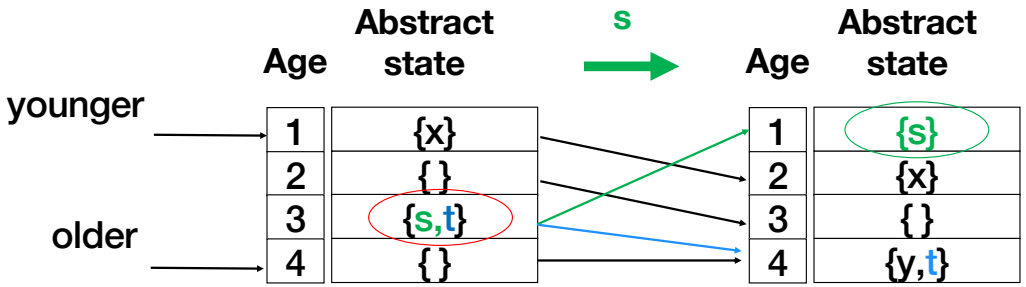
May analysis: update

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

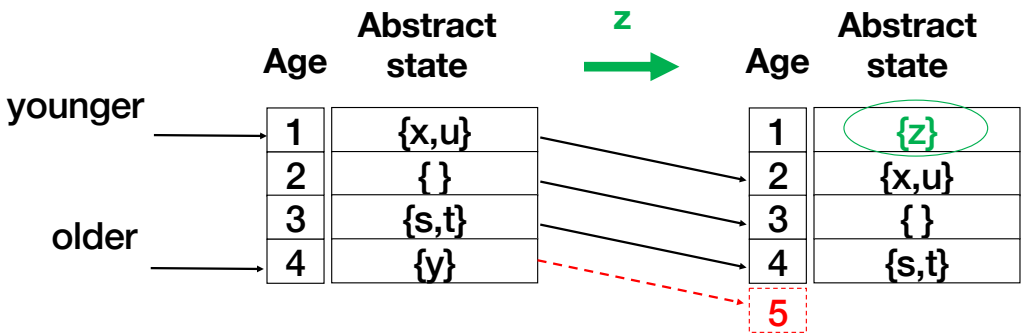
Transfer(S, inst(v))

- Age(v) = 1
- Age(u) ≤ Age(v) ⇒ Age'(u) = Age(u) + 1
- Age(u) > Age(v)
- Age'(w) = Age(w)

May analysis: update

Definite Cache miss

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



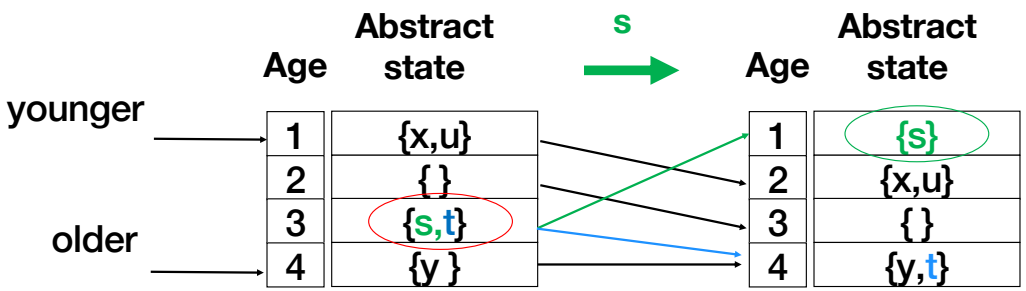
The most recently used variable (v) occupies the youngest cache line

Transfer function

Transfer(S, inst(v))

- Age(v) = 1
- Age(u) ≤ Age(v) ⇒ Age′(u) = Age(u) + 1
- Age(w) > Age(v)
- Age′(w) = Age(w)

Potential Cache hit



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.

Hence, t may have an age **at least** 4

t ages because in all concrete states it does, it is implicitly coded in how may analyses work: if t is in 3, then it can have age ≥ 3.

So:

1	{x}	CAN BE ⟹	1	x		1	x		1	-		1	-
2			2	-		2	-		2	x		2	x
3	{t, s}		3	t		3	s		3	t		3	s
4			4	s		4	t		4	s		4	t

IN ALL THOSE EXAMPLES, if s is moved to 1, t ends in 4.
FOR A MUST ANALYSIS:

1	{x}		1	x		1	x
2		⟹	2	t		2	s
3	{t, s}		3	s		3	t
4			4	-		4	-

IN ALL THOSE CASES, if s is moved, t ends in 3.

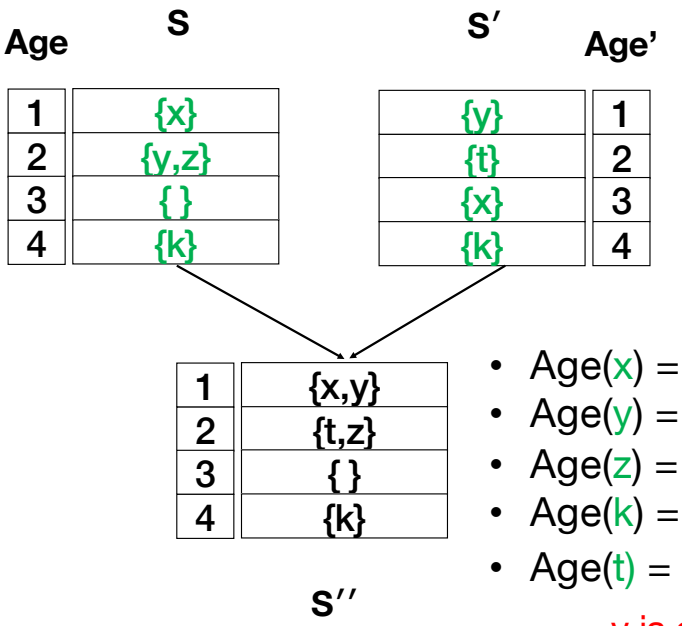
The difference is in the fact that at best, in ①, t is in 3, but s is up and will make it shift by 1. In ②, at best t is in 2 but s is up and will make it shift by 1. In ①, t can be up or same position, so it shifts; in ②, t can be down or same position so it shifts.

May Analysis: join

How to combine information?

Union (+ minimal age)

It is a forward analysis



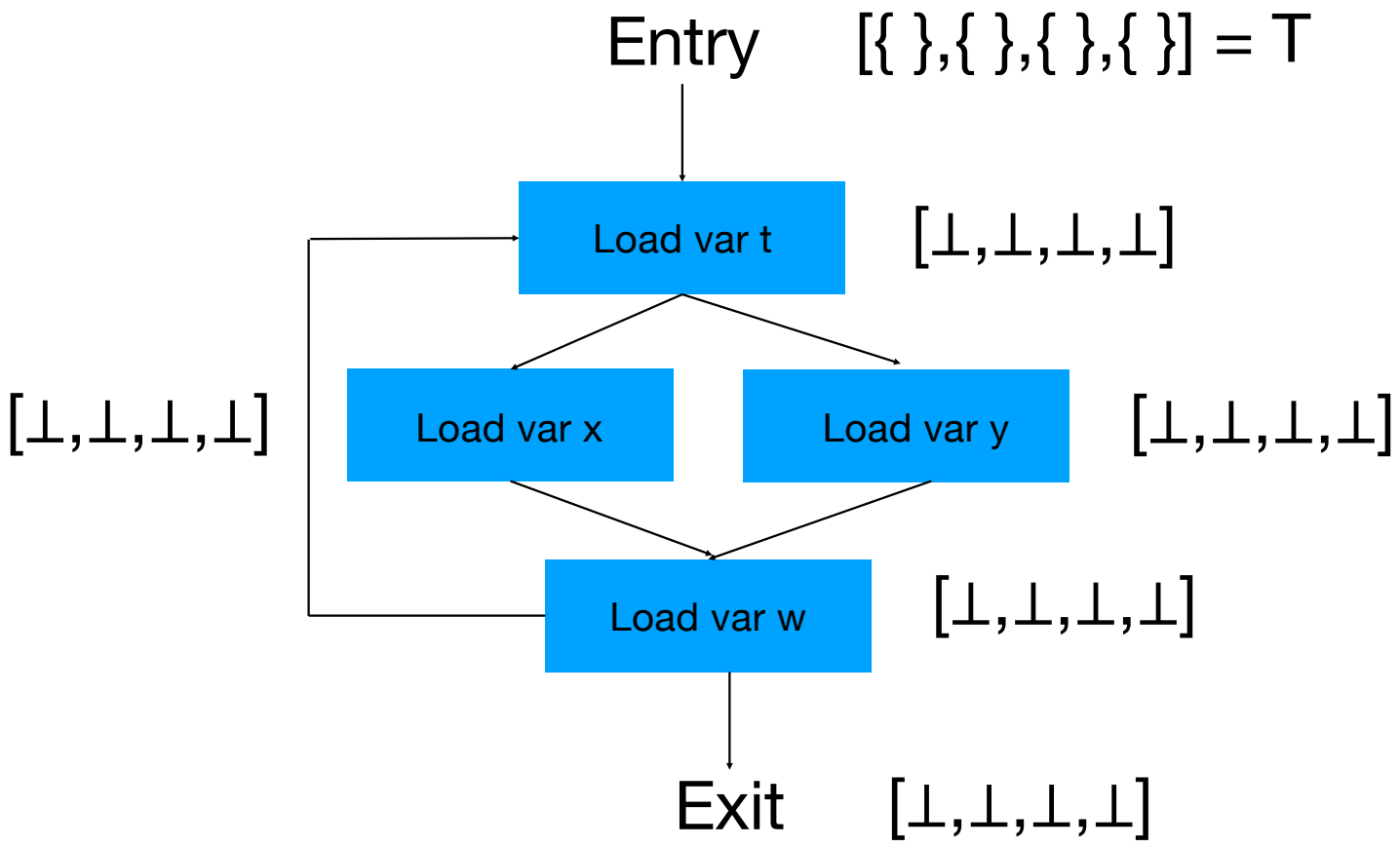
- Age(x) = 1, Age′(x) = 3 ⇒ Age′′(x) = 1
- Age(y) = 2, Age′(y) = 1 ⇒ Age′′(y) = 1
- Age(z) = 2, Age′(z) = 5 ⇒ Age′′(z) = 2
- Age(k) = 4, Age′(k) = 4 ⇒ Age′′(k) = 4
- Age(t) = 5, Age′(t) = 2 ⇒ Age′′(t) = 2

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

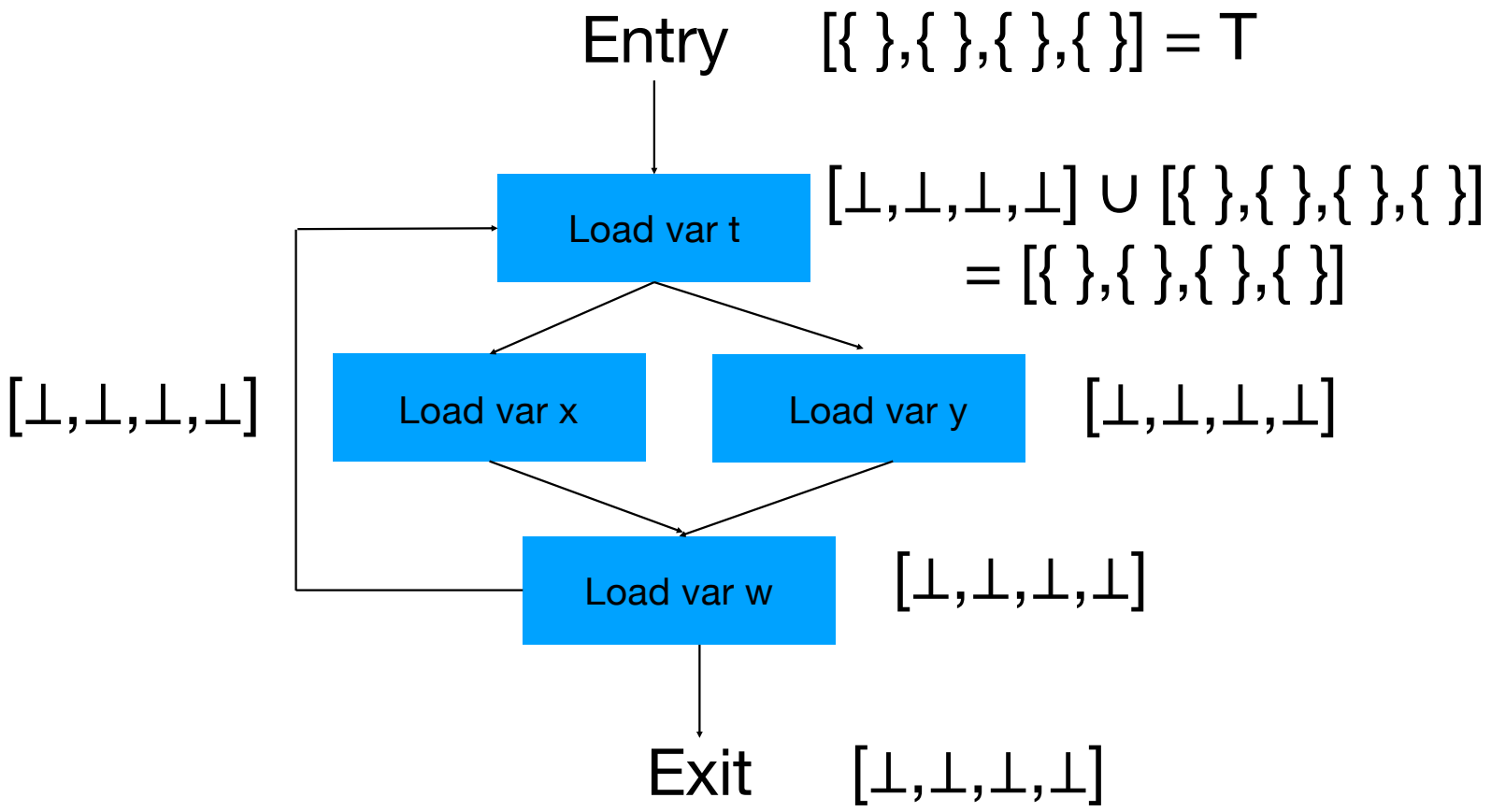
Given two states

- 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))⟩

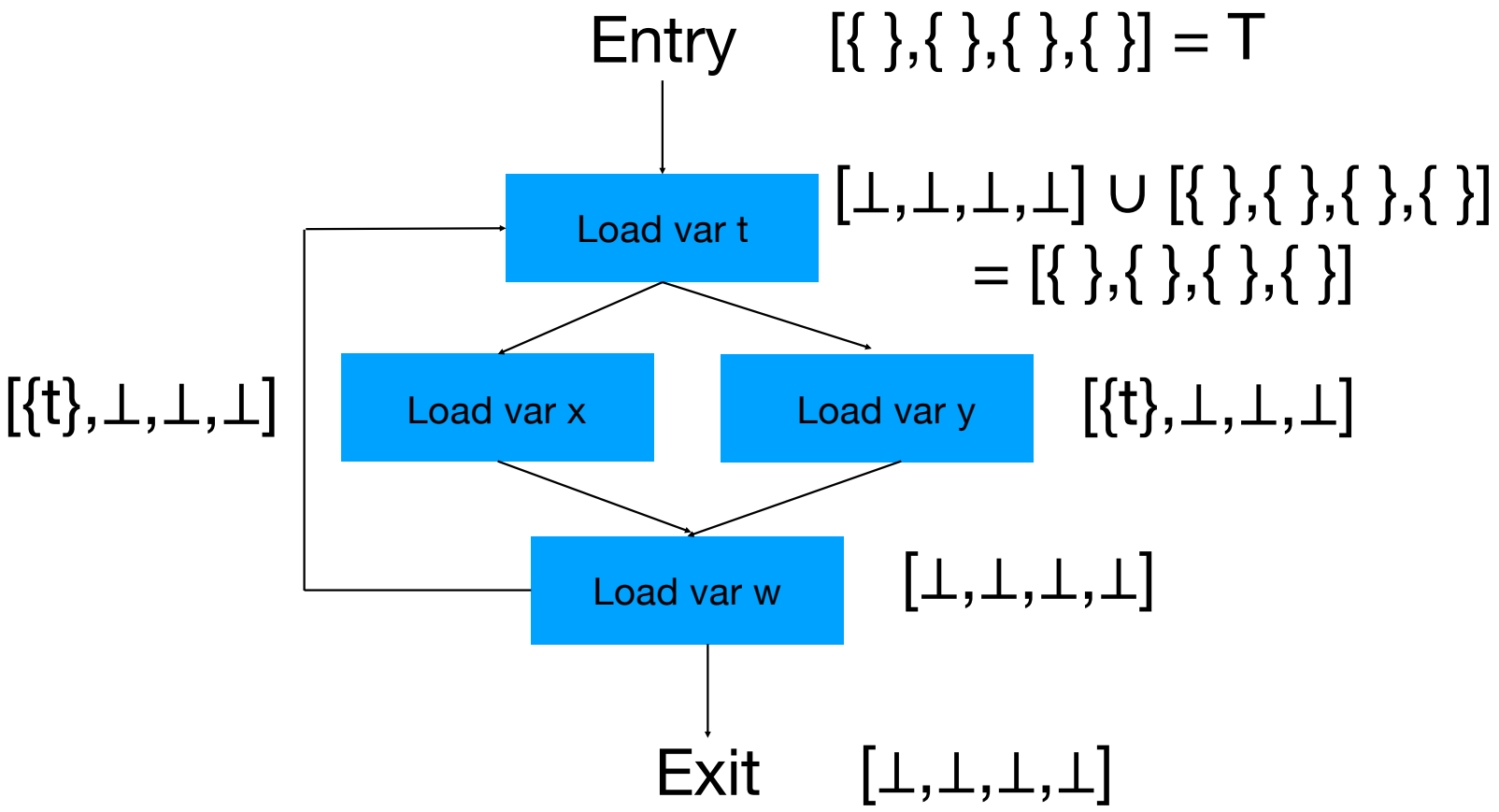
Must Analysis



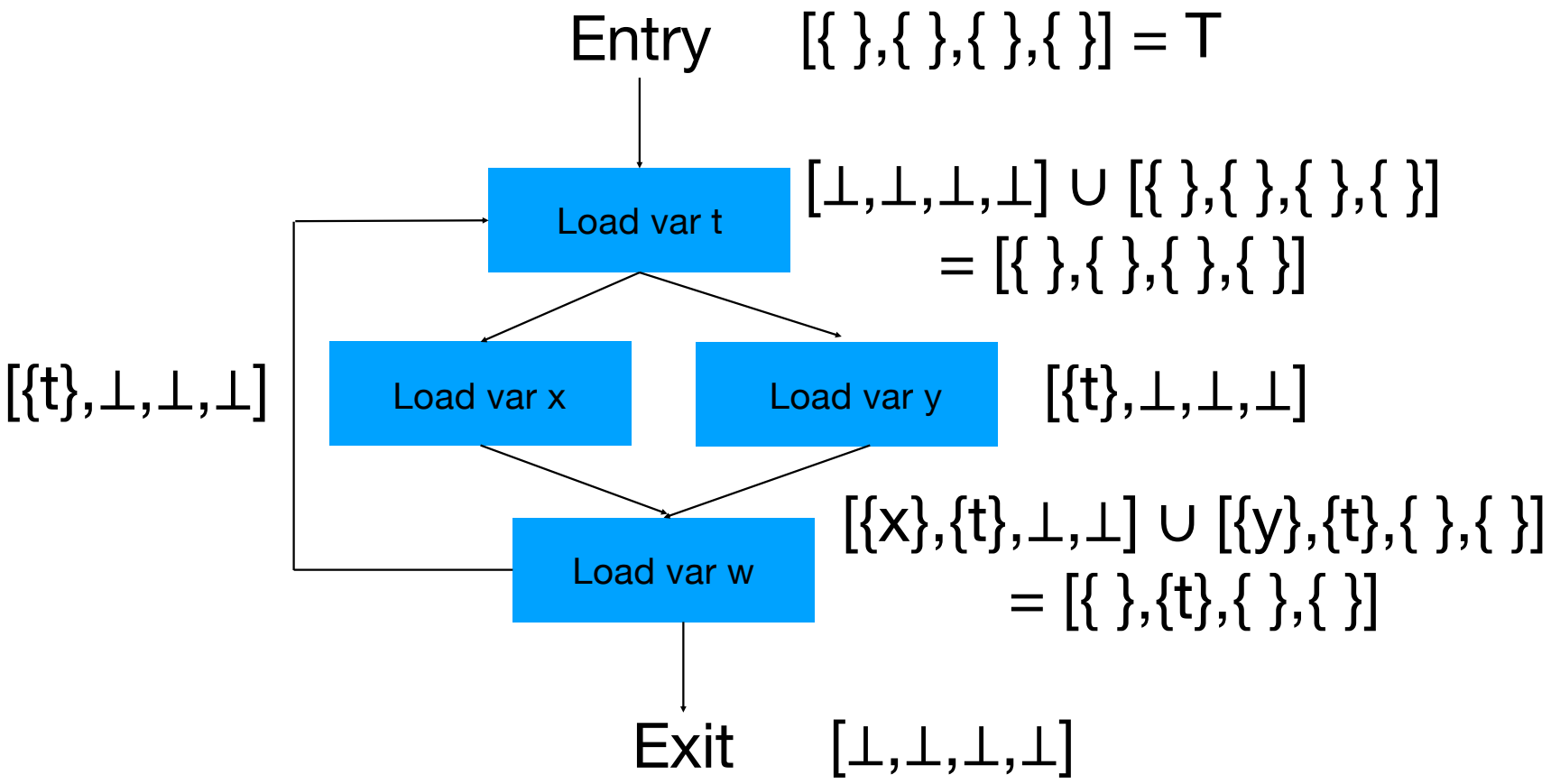
Must Analysis



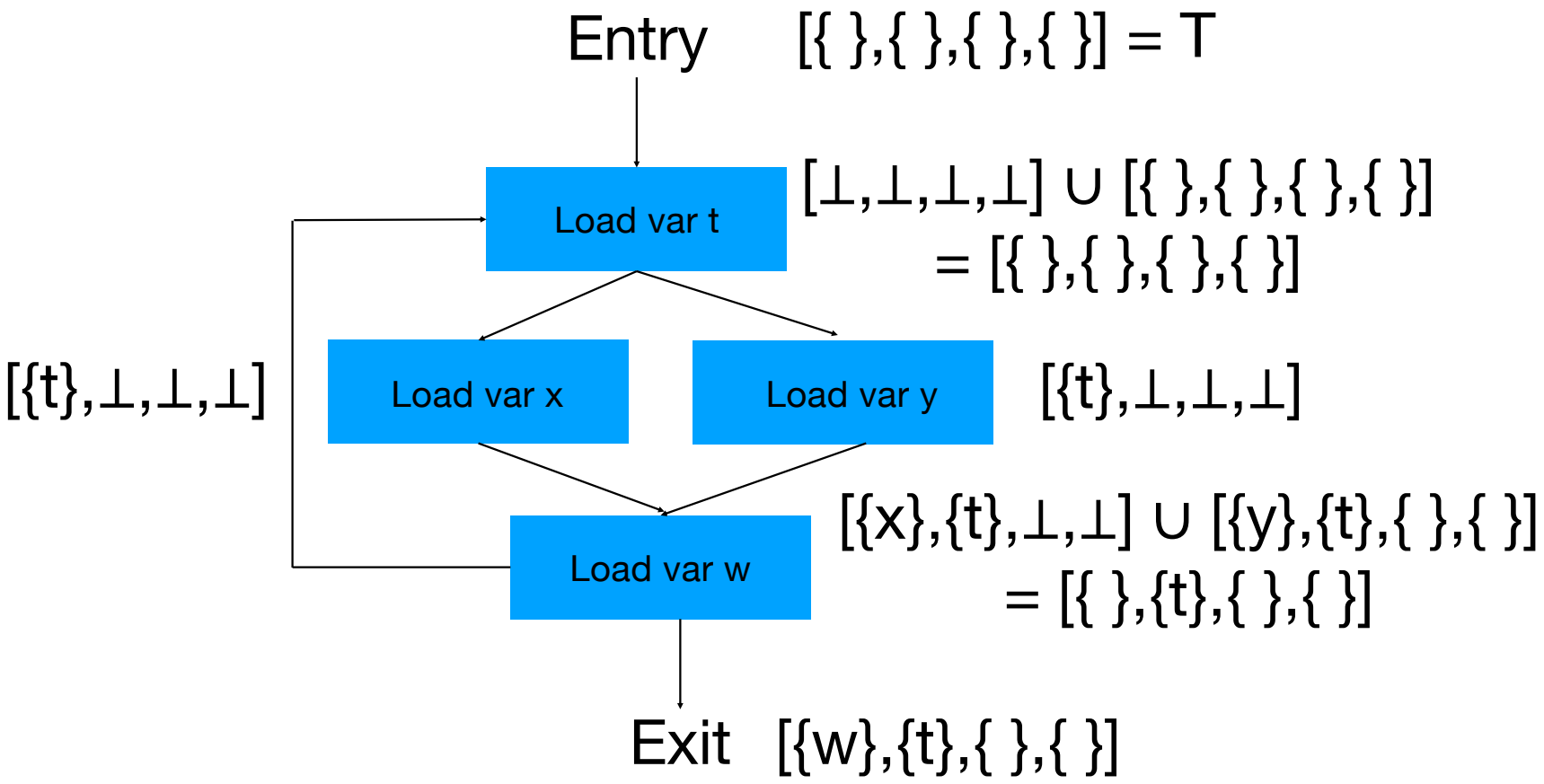
Must Analysis



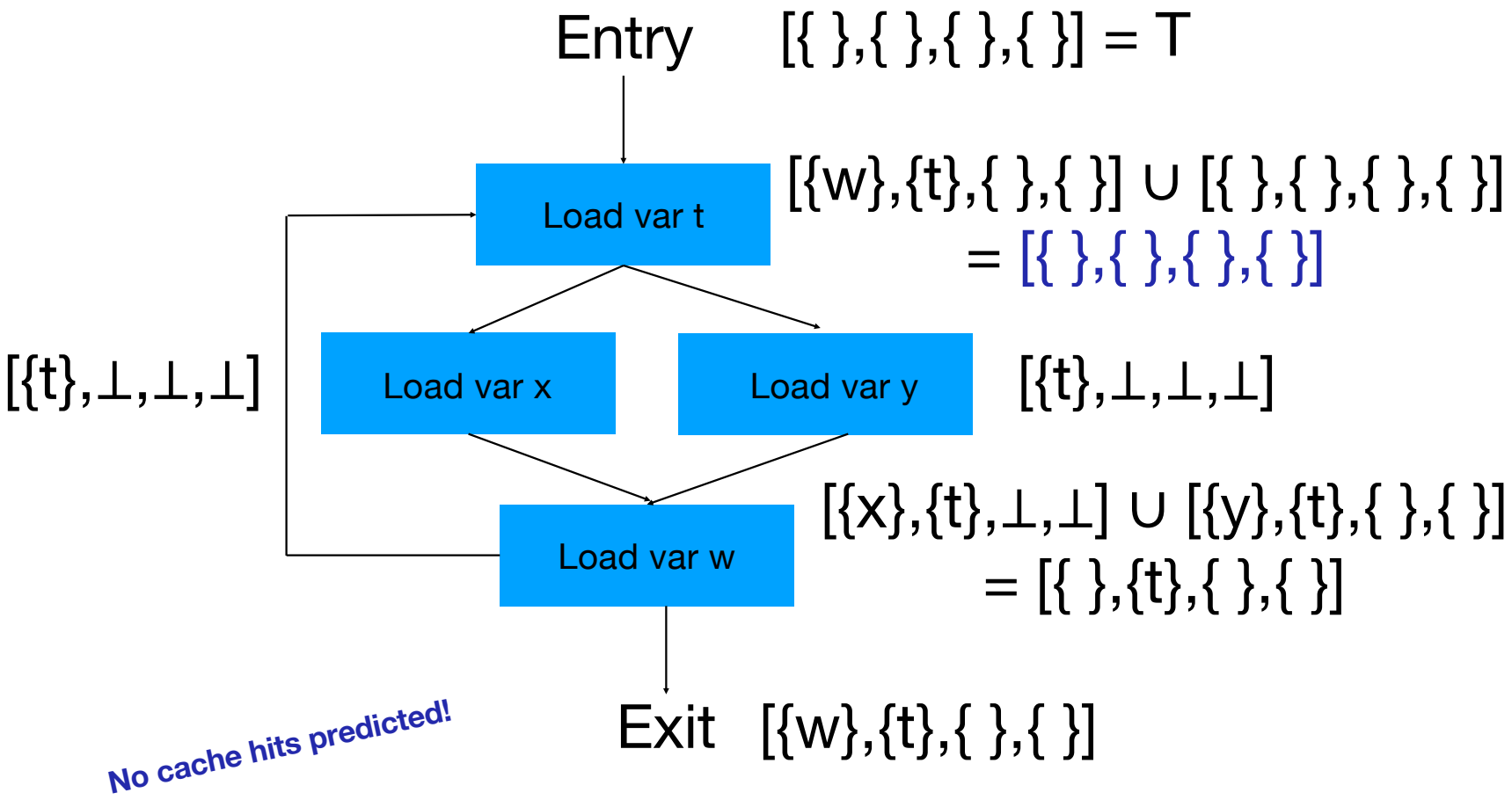
Must Analysis



Must Analysis



Must Analysis



Improving Precision by Using Contexts

Analysis can be made more precise: imagine you have a loop: on first cycle you get a miss, then not anymore.

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

Idea is unroll loops. To gain precision, otherwise 1st iteration makes a miss

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

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 not
- This information can be precisely maintained by the transfer functions due to LRU's regular cache update:
  - whether or not a block's age depends solely on its age relative 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

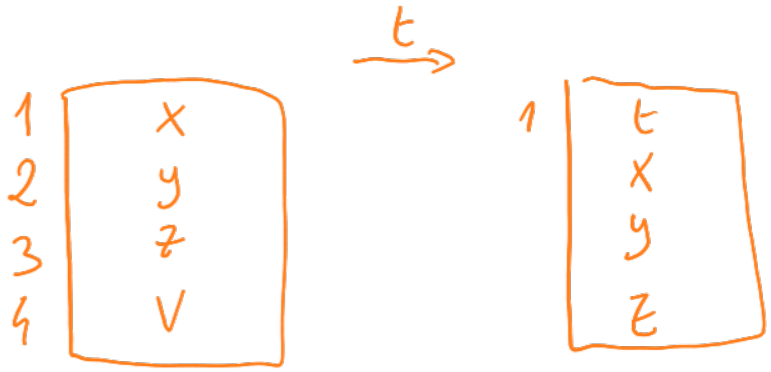
Most non-LRU replacement policies do not possess such monotone behavior

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)

$$P_1 \simeq_{ctx} P_2 \Rightarrow \forall C,\ C(P_1)\downarrow \Leftrightarrow C(P_2)\downarrow$$

$$\forall P_1, P_2:\quad P_1 \simeq_{ctx} P_2 \Leftrightarrow [P_1] \simeq_{ctx} [P_2]$$



End

1	{x, v}
2	
3	{t, s}
4	

(S, age(V))

1	w
2	x
3	y
4	z

$\xrightarrow{V}$

1	V
2	w
3	x
4	y

1	{x}
2	
3	{t, v}
4	

$\xrightarrow{t}$

1	{z}
2	{x}
3	–
4	{t, v}

1	{x}
2	
3	{t, v}
4	

$\xrightarrow{V}$

1	{v}
2	{1}
3	{r}
4	

1	$\{u, v\}$			1	$\{t\}$
2			$\xrightarrow{t}$	2	$\{u, v\}$
3	$\{t, x\}$			3	
4				4	$\{x\}$

$$\text{mst}(S, \text{age}(t))$$