



Critical Embedded Real-Time Systems

Systèmes Temps Réel Embarqués Critiques

STREC - WCET - Cache Analysis

Florian Brandner

Télécom ParisTech

Outline

Sub-Module Outline

Worst-Case Execution Time Analysis:

1. Introduction
2. **Static cache analysis (single task)**
 - Recap: cache organization
 - Cache analysis overview
 - Hit & miss classification
 - Persistences

Cache Organization

Cache Principles

What is a cache?

- A relatively small and fast memory
- Connected to a larger and slower cache/memory
- Stores data (or instructions) *currently* used
 - Implemented as a kind of dictionary
 - **Cache hit:**
Data requested by the processor is in the cache
⇒ Immediate response
 - **Cache miss:**
Data requested by the processor is not in the cache
⇒ Data is fetched from larger cache/memory
⇒ Delayed response

Cache Misses

Sources of misses can be grouped in three categories:

- **Compulsory misses:**

Occur when new data is accessed that was *never* referenced before

- **Capacity misses:**

Occur due to the limited size of the cache, regardless of the cache's internal design

(i.e., the amount of data accessed is larger than the cache)

- **Conflict misses:**

Are due to the internal organization of the cache

(i.e., could theoretically be avoided by an ideal cache design)

Cache Design

A cache can be seen as a kind of dictionary with k entries:

- Each entry is associated with the following information
 - Valid flag:
Flag indicating whether the data is valid
 - Tag:
The address of the data held by the entry
 - Data:
The data held by the entry
- Entries are stored in a memory
- Cache accesses to address a :
 1. Check whether an entry's tag matches a
 2. Check whether that entry is valid
 3. Yes? \implies hit
 4. No? \implies miss

Set-Associative Cache

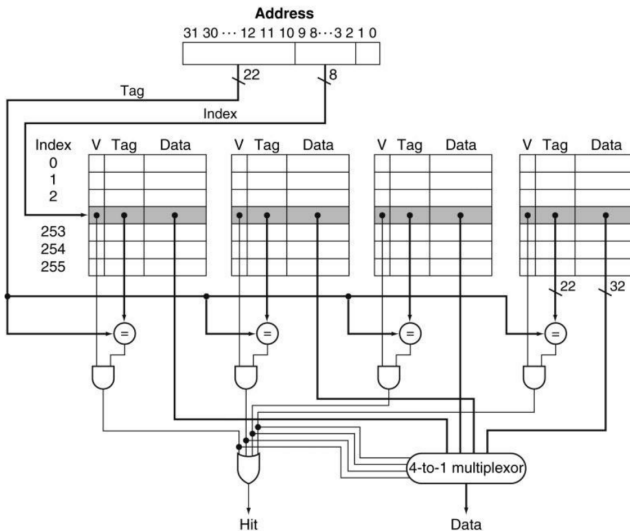
Organize cache in **lines** to reduce conflicts:

- The cache contains k entries
- Each line contains a **set** of s entries
- Each entry holds a block of b bytes
- Address a maps to a line:

$$(a \div b) \bmod (k \div s)$$

- Cache look-up:
 - Read line $(a \div b) \bmod (k \div s)$ from cache
 - Compare the tags of the line's s entries with a
 - Select the matching entry (if one exists)
 - Check the entry's valid flag

Set-Associative Cache (2)



Example: A 4-way set-associative cache.

Replacement Policy

Which entry should be used on a cache miss?¹

- Several policies are possible
- **First-In, First-Out:**
 - Simple to implement
 - Replace the entry that was loaded first
 - aka: Round-Robin
- **Least-Recently Used (LRU):**
 - Widely used strategy in practice (rather expensive through)
 - Replace block that was not used the longest
 - Preserve blocks that have recently been used (cf. temporal locality)
- ...

¹Especially once all valid flags are set.

Least-Recently Used

Implemented as an age counter for each entry:

- Counters are updated on each access to an address a
- Counters are in the range $[0, 1, \dots, s - 1]$
- Hit:
 1. If the age of a 's entry is 0: done
 2. Otherwise: set the age of that entry to 0
 3. Increment the age of the line's other entries by 1
- Miss:
 1. Fetch data from backing store
 2. Select entry with age $s - 1$
 3. Set the counter of that entry to 0
 4. Set the valid flag, the tag, and the data accordingly
 5. Increment the age of the line's other entries by 1

Example: LRU Replacement

Cache state when performing memory accesses:

- Assume the following set-associative cache:
 - Block size: $b = 2^0 = 1$ byte
 - Entries: $k = 2^3 = 8$
 - Associativity: $s = 2^1 = 2$
 - Address Width: 5 bits
- The cache is initially empty (i.e., all valid flags are 0)
- Accessed addresses:
22, 26, 22, 26, 16, 3, 16, 18, 26

Example: LRU Replacement (1)

Index	Valid	Age	Set 0		Valid	Age	Set 1	
			Tag	Data			Tag	Data
0	0	0	—	—	0	0	—	—
1	0	0	—	—	0	0	—	—
2	0	0	—	—	0	0	—	—
3	0	0	—	—	0	0	—	—

Initially: Cache is entirely empty.

Example: LRU Replacement (2)

Index	Valid	Age	Set 0		Valid	Age	Set 1	
			Tag	Data			Tag	Data
0	0	0	—	—	0	0	—	—
1	0	0	—	—	0	0	—	—
2	1	0	101_2	$M(10110_2)$	0	0	—	—
3	0	0	—	—	0	0	—	—

Miss: Compulsory miss for address 22.

Example: LRU Replacement (3)

Index	Valid	Age	Set 0		Valid	Age	Set 1	
			Tag	Data			Tag	Data
0	0	0	—	—	0	0	—	—
1	0	0	—	—	0	0	—	—
2	1	1	101_2	$M(10110_2)$	1	0	110_2	$M(11010_2)$
3	0	0	—	—	0	0	—	—

Miss: Compulsory miss for address 26.
(same line, but no conflict)

Example: LRU Replacement (4)

Index	Set 0				Set 1			
	Valid	Age	Tag	Data	Valid	Age	Tag	Data
0	1	0	100_2	$M(10000_2)$	0	0	—	—
1	0	0	—	—	0	0	—	—
2	1	1	101_2	$M(10110_2)$	1	0	110_2	$M(11010_2)$
3	0	0	—	—	0	0	—	—

Hits: Cache hits for addresses 22 and 26.
(intermittently switch age)

Miss: Compulsory miss for address 16.

Example: LRU Replacement (5)

Index	Set 0				Set 1			
	Valid	Age	Tag	Data	Valid	Age	Tag	Data
0	1	0	100 ₂	$M(10000_2)$	0	0	—	—
1	0	0	—	—	0	0	—	—
2	1	1	101 ₂	$M(10110_2)$	1	0	110 ₂	$M(11010_2)$
3	1	0	000 ₂	$M(00011_2)$	0	0	—	—

Miss: Compulsory miss for address 3.

Example: LRU Replacement (6)

Index	Valid	Age	Set 0		Valid	Age	Set 1	
			Tag	Data			Tag	Data
0	1	0	100_2	$M(10000_2)$	0	0	—	—
1	0	0	—	—	0	0	—	—
2	1	0	100_2	$M(10010_2)$	1	1	110_2	$M(11010_2)$
3	1	0	000_2	$M(00011_2)$	0	0	—	—

Hit: Cache hit for address 16.

Miss: Compulsory miss for address 18.
(conflict with addresses 22)

Example: LRU Replacement (7)

Index	Valid	Age	Set 0		Valid	Age	Set 1	
			Tag	Data			Tag	Data
0	1	0	100_2	$M(10000_2)$	0	0	—	—
1	0	0	—	—	0	0	—	—
2	1	1	100_2	$M(10010_2)$	1	0	110_2	$M(11010_2)$
3	1	0	000_2	$M(00011_2)$	0	0	—	—

Hit: Cache hit for address 26.

Write Policy (Hit)

Determines how memory stores are handled:

- Two basic options for a **write hit**
- **Write-through:**
 - Write data into the cache and to backing store
 - Long delay (waiting for slow higher-level caches)
- **Write-back:**
 - Write data only to the cache
 - Data is incoherent between cache and backing store
 - Backing store updated once data is evicted from cache
 - Implementation:
Add an additional *dirty* bit to each cache entry
- What happens on a **write miss**?

Write Policy (Miss)

Should data be loaded to the cache on a write miss?

- **Write-allocate:**

- First load cache block from backing store
- Then use same strategy as for write hits

- **Write-no-allocate:**

- Does not load from backing store
- Write immediately to backing store

- Both can be combined with write-through/-back, but usually
 - Write-through is combined with write-no-allocate
 - Write-back is combined with write-allocate

This Course

From now on we will assume:

- Separate data and instruction caches
- LRU replacement policy
- Write-through with write-no-allocate

Cache Analysis

Cache Analysis

Compute the time required for cache misses:


- Analyze cache states before each memory access
 - Is the accessed data in the cache?
 - How often do cache hits occur?
 - How often do the expensive cache misses occur?
- Problems:
 - Access addresses need to be known **precisely**
 - Behavior of accesses in loops **changes** over time

Example: Cache Analysis Join

Combining two cache states (addresses)*

0x100	0x200
0x103	0x105

0x100	0x200
0x103	0x107



0x100	0x200
0x103	??

* Cache configuration

2-way set-associative, 1 word blocks, 2 cache lines, LRU replacement

Example: Cache Analysis

Initial cache state (addresses)*

0x100	0x200
0x103	??

* Cache configuration
2-way set-associative, 1 word blocks, 2 cache lines, LRU replacement

Example: Cache Analysis

Initial cache state (addresses)*

0x100	0x200
0x103	??

lw [0x100]

0x100	0x200
0x103	??

Classified as hit

*Cache configuration

2-way set-associative, 1 word blocks, 2 cache lines, LRU replacement

Example: Cache Analysis

Initial cache state (addresses)*

0x100	0x200
0x103	??

lw [0x100]

0x100	0x200
0x103	??

Classified as hit

lw [0x300]

0x300	0x100
0x103	??

Classified as miss

*Cache configuration

2-way set-associative, 1 word blocks, 2 cache lines, LRU replacement

Example: Cache Analysis

Initial cache state (addresses)*

0x100	0x200
0x103	??

lw [0x100]	0x100	0x200
	0x103	??

Classified as hit

lw [0x300]	0x300	0x100
	0x103	??

Classified as miss

lw [0x105]	0x100	0x200
	0x105	??

Classification unclear

*Cache configuration

2-way set-associative, 1 word blocks, 2 cache lines, LRU replacement

Example: Cache Analysis

Initial cache state (addresses)*

0x100	0x200
0x103	??

lw [0x100]	<table><tr><td>0x100</td><td>0x200</td></tr><tr><td>0x103</td><td>??</td></tr></table>	0x100	0x200	0x103	??	Classified as hit
0x100	0x200					
0x103	??					
lw [0x300]	<table><tr><td>0x300</td><td>0x100</td></tr><tr><td>0x103</td><td>??</td></tr></table>	0x300	0x100	0x103	??	Classified as miss
0x300	0x100					
0x103	??					
lw [0x105]	<table><tr><td>0x100</td><td>0x200</td></tr><tr><td>0x105</td><td>??</td></tr></table>	0x100	0x200	0x105	??	Classification unclear
0x100	0x200					
0x105	??					
lw [??]	<table><tr><td>??</td><td>??</td></tr><tr><td>??</td><td>??</td></tr></table>	??	??	??	??	Classification unclear
??	??					
??	??					

*Cache configuration
2-way set-associative, 1 word blocks, 2 cache lines, LRU replacement

Cache Hit & Miss Classification

Basic Idea

For each memory/cache access:²

1. Determine the set of memory blocks potentially accessed
 - For instance: range analysis (last lecture)
2. Determine the age of each memory block
 - Topic of today's lecture
3. Use the minimum/maximum age to classify hits/misses
 - Topic of today's lecture

²Recall: assume a set-associative cache with LRU

Memory Blocks

Abstraction used during the analysis to track the cache state:

- Address range in memory corresponding to a cache block
- Aligned with the cache block size
- Matches the size of a cache block (b from above)
- Notations:
 - $mb_l(i)$ denotes the set of memory blocks of cache line l , potentially accessed by instruction i
 - This set might be empty
(e.g., instructions not accessing memory, such as `addi` on MIPS)

Age

Associate each memory block with an age counter:

- Counter range: $[0, 1, \dots, s]$ (s entries per set)
- Difference with *real* cache:
 - Track age of all memory blocks not just those in the cache
 - Memory blocks that are not in the cache have age s
(compare with counter range of actual cache)
- Notations:
 - $age(u)$ denotes the age of memory block u .

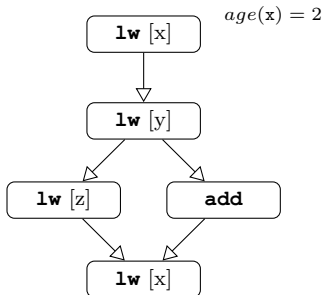
Access Classification

Classify each memory/cache access into a category:

- Always Hit (AH):
The age of all potentially accessed memory blocks must be smaller than s .
- Always Miss (AM):
The age of any potentially accessed memory blocks may never be smaller than s .
- Not classified (NC):
None of the above applies.

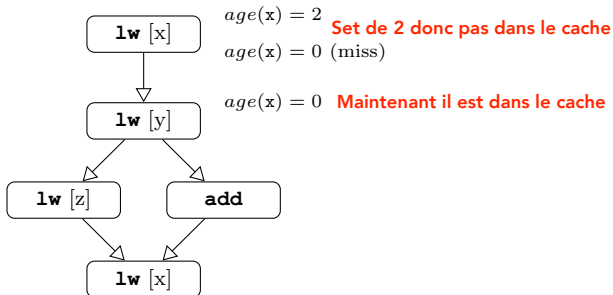
Example: Age-Based Cache Analysis

Computing the age of cache block x :



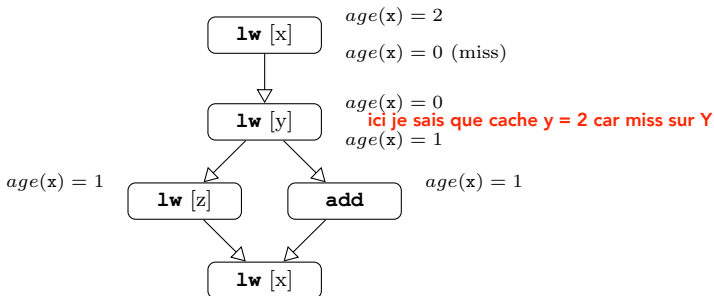
Example: Age-Based Cache Analysis

Computing the age of cache block x :



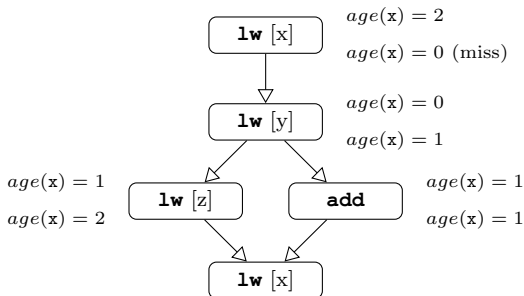
Example: Age-Based Cache Analysis

Computing the age of cache block x :



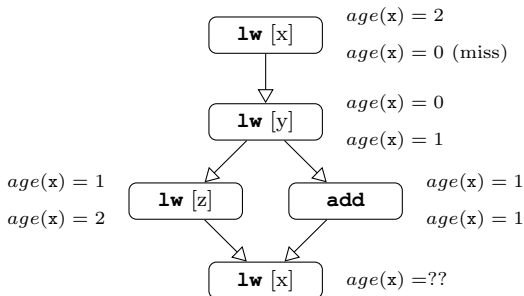
Example: Age-Based Cache Analysis

Computing the age of cache block x :



Example: Age-Based Cache Analysis

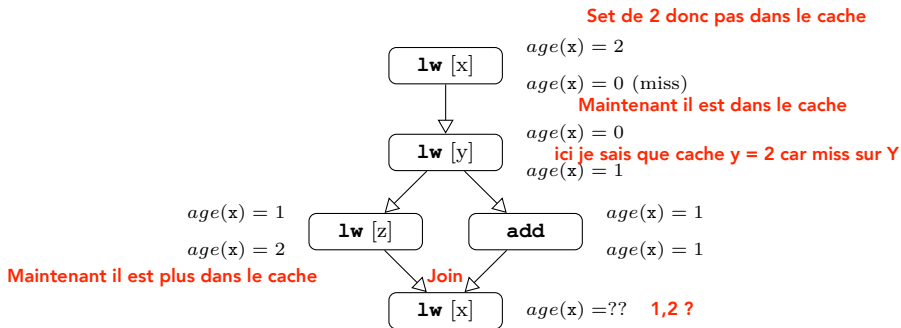
Computing the age of cache block x :



Groupe Exercise Age-Based Cache Analysis

What is the classification of the last access to x ?

- What can be said about the age of the memory block?
- Hint: recall the words may and must in the category definitions



Analysis Problems

The above classification gives rise to two analysis problems

- **Must analysis:** (pessimist)
Compute maximum age of memory blocks, i.e., ages appearing in any real execution **must** be equal or smaller to the computed age.
- **May analysis:** (optimist)
Compute minimum age of memory blocks, i.e., there **may** exist a real execution with an age as low as the computed age.

Must Analysis

Data-flow analysis computes maximum age of memory blocks:

- Domain:
 - $CS = MB_l \times \{0, 1, \dots, s\}$
 - MB_l denotes the set of memory blocks of a cache line l
 - s denotes the number of cache sets
- Notations:
 - $age(c, u)$ gives the age of memory block u for cache state c
 - Only memory blocks *in* the cache will be shown (i.e., only those with an age smaller than s)

Must Analysis: Join Operator (\sqcup_{MUST})

Select the maximum age for each memory block from cache states $c_1, c_2 \in CS$:

$$c_1 \sqcup_{MUST} c_2 = \{(u, a) \mid \exists (u, a_1) \in c_1, (u, a_2) \in c_2 : a = \max(a_1, a_2)\}$$

Must Analysis: Transfer Function (1)

Lets consider a single memory block for now:

- Assume a state $c \in CS$ and a memory block $u \in MB_I$
- The cache state after a memory load is then given by:

$$update_{MUST}(c, u) = \{(v, a) | v \in MB_I: a = must_age(c, u, v)\}$$

$$must_age(c, u, v) = \begin{cases} 0 & , \text{ if } u = v \\ age(c, v) & , \text{ if } age(c, v) \geq age(c, u) \\ age(c, v) + 1 & , \text{ if } age(c, v) < age(c, u) \end{cases}$$

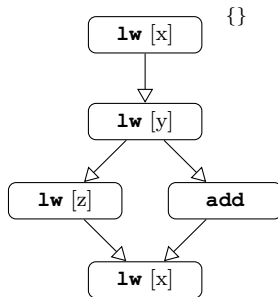
- Memory stores do not impact the cache state (write-through, write-no-allocate)

Must Analysis: Transfer Function (2)

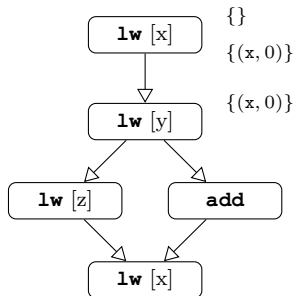
The transfer function for cache state c and instruction i then is:

$$t_{MUST}(c, i) = \begin{cases} c & , \text{ if } mb_I(i) = \emptyset \\ update_{MUST}(c, u) & , \text{ if } mb_I(i) = \{u\} \\ \text{error} & , \text{ otherwise (not yet handled)} \end{cases}$$

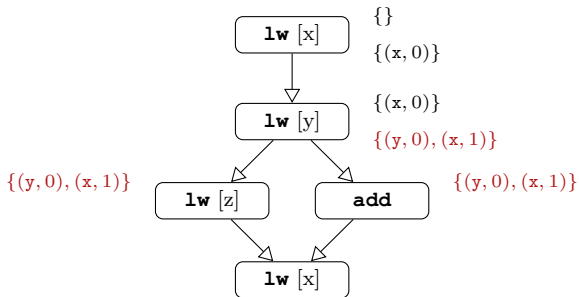
Example: Must Cache Analysis



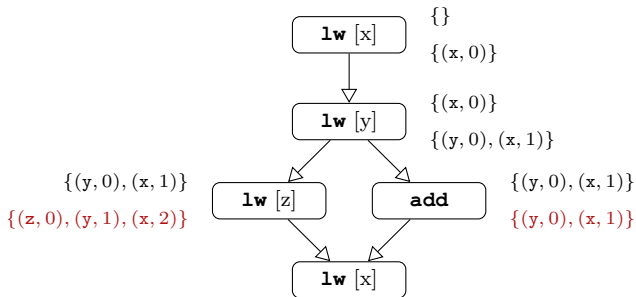
Example: Must Cache Analysis



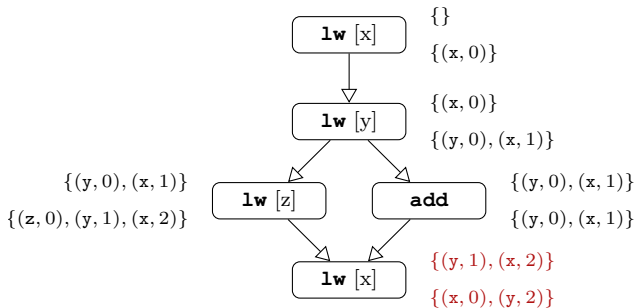
Example: Must Cache Analysis



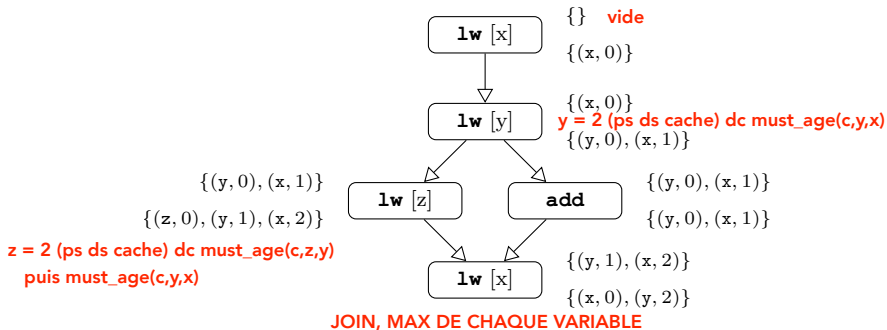
Example: Must Cache Analysis



Example: Must Cache Analysis



Example: Must Cache Analysis

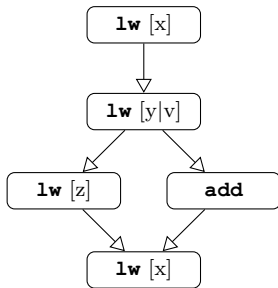


On ne peut pas placer y comme hit car trop grand.

Group Exercise: Must Cache Analysis

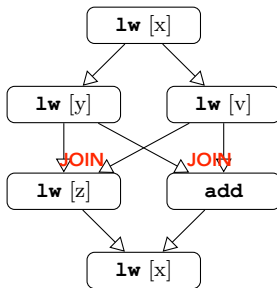
What if the accessed memory blocks are not precisely known?

- Assume that the second load might either access y or v (but of course never both)
- Is information regarding other memory blocks *lost*?



Representing Uncertain Accesses

- Can be seen as a form of *control-flow* decision
- Simply handle both cases separately
- Then apply the join operator
- Example:



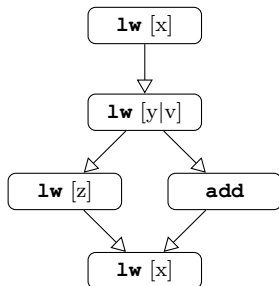
Must Analysis: Transfer Function (3)

The transfer function for cache state c and instruction i then is:

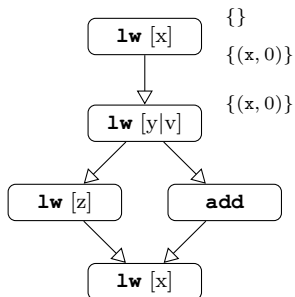
$$t_{MUST}(c, i) = \begin{cases} c & , \text{ if } mb_I(i) = \emptyset \\ \sqcup_{u \in mb_I(i)}^{MUST} update_{MUST}(c, u) & , \text{ otherwise} \end{cases}$$

ON FAIT DES JOIN A TT VA

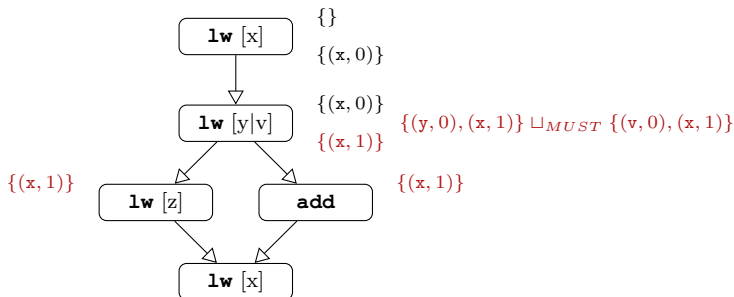
Example: Uncertain Accesses



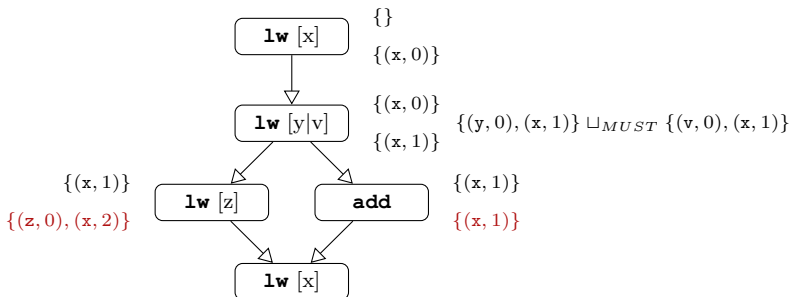
Example: Uncertain Accesses



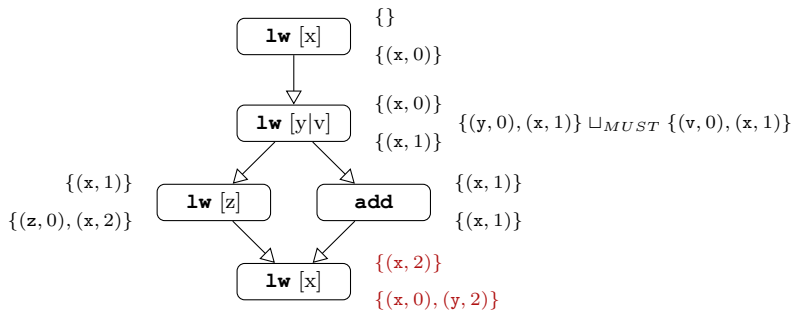
Example: Uncertain Accesses



Example: Uncertain Accesses

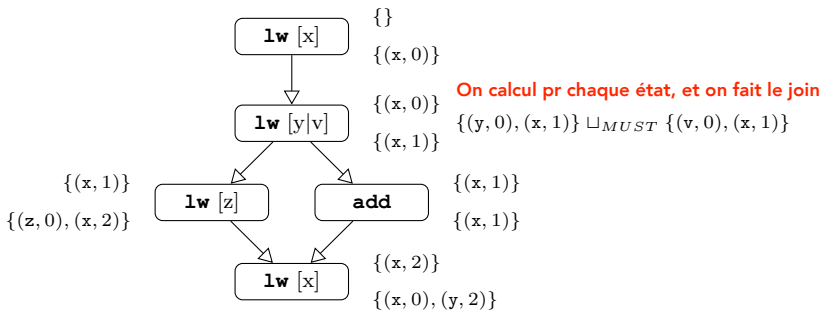


Example: Uncertain Accesses



Texte

Example: Uncertain Accesses



Potentiellement ici, y ou v n'est pas dans le cache => On ne sait pas

May Analysis

Similar data-flow analysis as the Must analysis before:

- Domain: (same as for Must)
 - $CS = MB_l \times \{0, 1, \dots, s\}$
 - MB_l denotes the set of memory blocks of cache line l
 - s denotes the number of cache sets

- Join Operator:

For any $c_1, c_2 \in CS$ the join operator is given by:

$$c_1 \sqcup_{MAY} c_2 = \{(u, a) \mid \exists (u, a_1) \in c_1, (u, a_2) \in c_2 : a = \min(a_1, a_2)\}$$

May Analysis: Transfer Function

Again, first consider a single memory block:

- Assume a state $c \in CS$ and a memory block $u \in MB_I$
- The cache state after a memory load is then given by:

$$update_{MAY}(c, u) = \{(v, a) \mid v \in MB_I : a = may_age(c, u, v)\}$$

$$may_age(c, u, v) = \begin{cases} 0 & , \text{ if } u = v \\ age(c, v) & , \text{ if } age(c, v) > age(c, u) \\ age(c, v) + 1 & , \text{ if } age(c, v) \leq age(c, u) \wedge \\ & age(c, v) < s \\ s & , \text{ otherwise} \end{cases}$$

- Memory stores do not impact the cache state
(as before, write-through, write-no-allocate)
- The actual transfer function is similar to the Must analysis

Group Exercise: May versus Must Analysis

The age functions of the May and Must analyses are similar:

- Try to explain the differences
- Hint: Recall that the Must analysis provides a maximum and the May analysis a minimum age!

$$must_age(c, u, v) = \begin{cases} 0 & , \text{ if } u = v \\ age(c, v) & , \text{ if } age(c, v) \geq age(c, u) \\ age(c, v) + 1 & , \text{ if } age(c, v) < age(c, u) \end{cases}$$

$$may_age(c, u, v) = \begin{cases} 0 & , \text{ if } u = v \\ age(c, v) & , \text{ if } age(c, v) > age(c, u) \\ age(c, v) + 1 & , \text{ if } age(c, v) \leq age(c, u) \wedge \\ & age(c, v) < s \\ s & , \text{ otherwise} \end{cases}$$

May versus Must Analysis

Must analysis:

- $age(c, v)$ represents the **maximum age**, i.e., the actual age might be **smaller**.
- Due to $age(c, v) \geq age(c, u)$, the access to u cannot increase the age of v .

$$age(cs, v) = 2: \begin{array}{|c|c|c|c|} \hline & & & \\ \hline \end{array} \geq age(cs, u) = 1: \begin{array}{|c|c|c|c|} \hline & & & \\ \hline \end{array}$$

$$\text{Case 1: } age(v) = 2: \begin{array}{|c|c|c|c|} \hline & & v & \\ \hline \end{array} \quad age(u) = 1: \begin{array}{|c|c|c|c|} \hline & u & & \\ \hline \end{array} \implies age(v) = 2: \begin{array}{|c|c|c|c|} \hline u & & v & \\ \hline \end{array}$$

$$\text{Case 2: } age(v) = 0: \begin{array}{|c|c|c|c|} \hline v & & & \\ \hline \end{array} \quad age(u) = 1: \begin{array}{|c|c|c|c|} \hline & u & & \\ \hline \end{array} \implies age(v) = 2: \begin{array}{|c|c|c|c|} \hline u & v & & \\ \hline \end{array}$$

- Similar argument when $age(c, v) < age(c, u)$.

May versus Must Analysis (2)

May analysis:

- $age(c, v)$ represents the **minimum age**, i.e., the actual age might be **larger**.
- Due to $age(c, v) > age(c, u)$ the access to u thus cannot increase the age of v

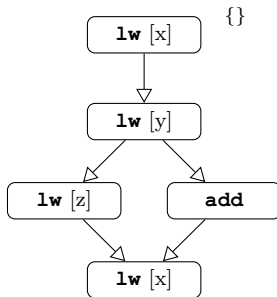
$$age(cs, v) = 2: \begin{array}{|c|c|c|c|} \hline & & & \\ \hline \end{array} > age(cs, u) = 1: \begin{array}{|c|c|c|c|} \hline & & & \\ \hline \end{array}$$

$$\text{Case 1: } age(v) = 2: \begin{array}{|c|c|c|c|} \hline & & v & \\ \hline \end{array} \quad age(u) = 1: \begin{array}{|c|c|c|c|} \hline & u & & \\ \hline \end{array} \implies age(v) = 2: \begin{array}{|c|c|c|c|} \hline u & & v & \\ \hline \end{array}$$

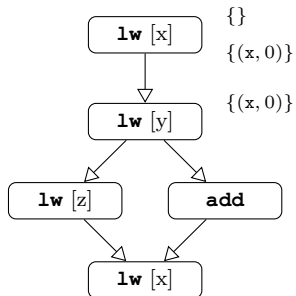
$$\text{Case 2: } age(v) = 2: \begin{array}{|c|c|c|c|} \hline & & v & \\ \hline \end{array} \quad age(u) = 3: \begin{array}{|c|c|c|c|} \hline & & & u \\ \hline \end{array} \implies age(v) = 3: \begin{array}{|c|c|c|c|} \hline u & & & v \\ \hline \end{array}$$

- Similar argument when $age(c, v) \leq age(c, u)$
(attention $age(c, v)$ might become too large here)

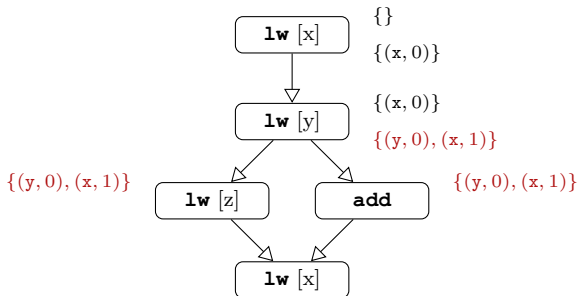
Example: May Analysis



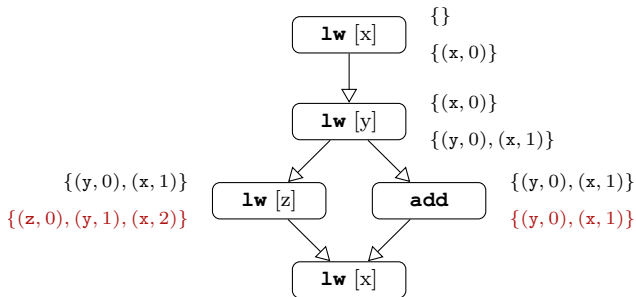
Example: May Analysis



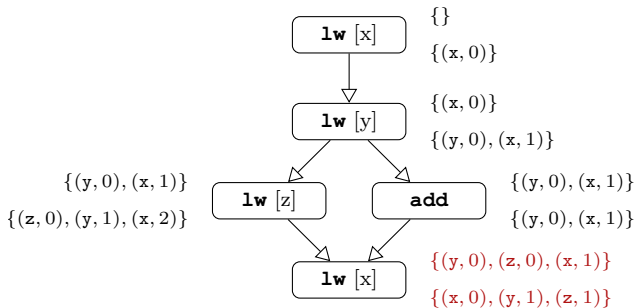
Example: May Analysis



Example: May Analysis



Example: May Analysis



Final Classification

Classification derived from May/Must analyses:

- Always Hit (AH):
Must analysis age of memory blocks in $mb_l(i)$ has to be lower than s .
- Always Miss (AM):
May analysis age of memory blocks in $mb_l(i)$ has to be equal to s .
- Not classified (NC):
None of the above applies.

Integration with IPET

Access classifications are easy to integrate into IPET

- Always Hit (AH):
Usually does not require additional costs.
- Always Miss (AM):
Add miss costs to the weight of the instruction's basic block
- Not classified (NC):
Often considered as expensive as a miss.³

³This is only safe on architectures without timing anomalies (out of scope of this course – see SE201 at Télécom ParisTech to get an idea)

Persistence

Accesses in Loops

Behavior of realistic programs:

- Often repeatedly access the same data in loops
- Observation:
 - First iteration: cache miss (compulsory miss)
 - Other iterations: often hits in cache
 - The executed code itself exhibits typically this behavior (instruction cache)
- Problem:

This is cannot be handled by simple hit/miss classification.

Persistence

Introduce the notion of *persistence*:

- Data that remains in the cache once loaded
- Typically with regard to a scope
(e.g., a loop, a function, . . .)
- New classification:
 - **First Miss** / First Hit
 - Accounting for one miss each time the scope is entered

Persistence Analysis (Idea)

Determine persistent memory accesses within a scope:

- Various possible approaches
 - Combine loop peeling with Must analysis
 - Bound set of conflicting accesses
 - ...
- Typically focus on loops
 - Here in particular loop nests

Summary

- Caches hide long memory access latencies
 - Considerably improve average-case execution time
 - Need to be considered during WCET analysis
- Cache design
 - Organized in sets of fixed-sized cache blocks
 - Replacement policy (Least Recently Used)
 - Write strategy (Write-through, no-allocation)
- Cache analysis
 - Classify memory accesses (Always Hit/Miss, Not Classified)
 - Must analysis: cache blocks that *must* be in the cache
 - May analysis: cache blocks that *may* be in the cache