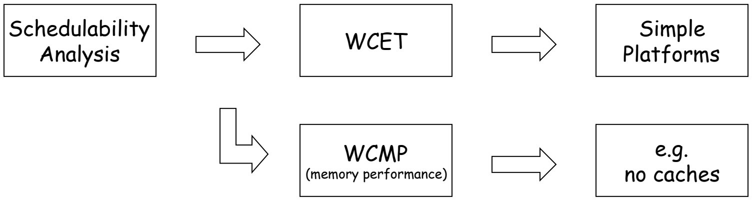
Introduction to Cache Analysis for Real-Time Systems

[C. Ferdinand and R. Wilhelm, “Efficient and Precise Cache Behavior Prediction for Real-Time Systems”, Real-Time Systems, 17, 131-181, (1999)]



- Ignoring cache leads to significant resource under-utilization.
- Q: How to appropriately account for cache?

Worst-Case Execution Time (WCET)

- WCET analysis must be **safe**
- WCET analysis must be **tight**
- WCET depends on
  - execution path (in program)
  - cache behavior (depends on execution history)
  - pipelining (depends on very recent execution history)

Problems with Cache Memories

Two fundamental problems with cache analysis:

1. Large differences in cache behavior (and execution time!) result from minor changes
    - in program code
    - in input data

2. Inter-task cache interference

WCET analysis for single tasks remains prerequisite for multi-task analysis!

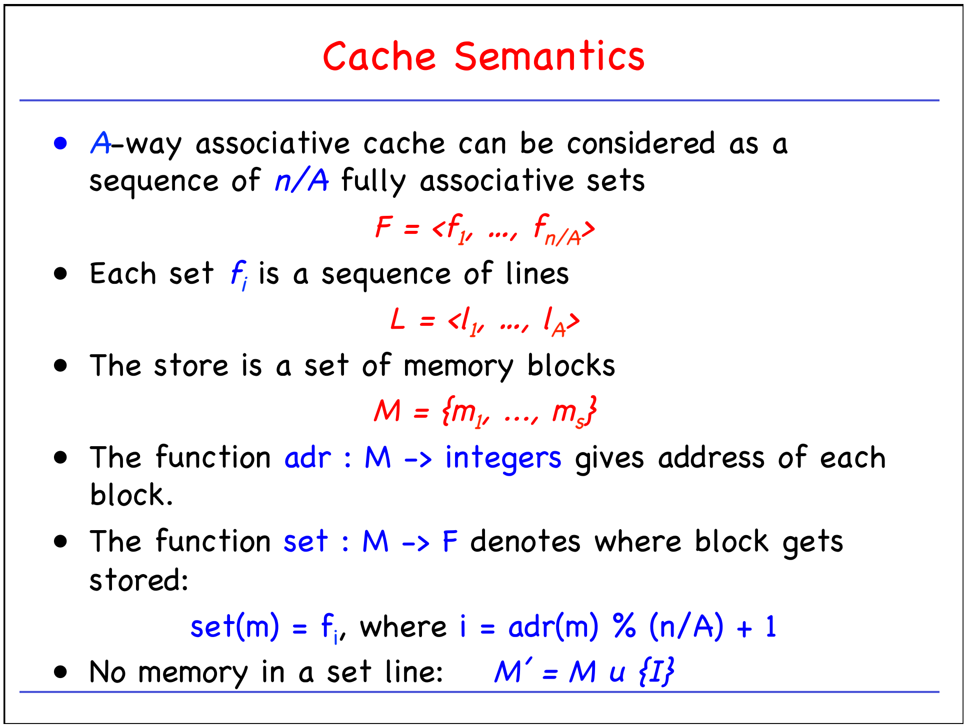
Cache Memories

Major parameters of caches:

1. **Capacity**: how many bytes in the cache?

2. **Line size** (block size): number of contiguous bytes that are transferred from memory on cache miss.
   Cache contains $n = capacity / line\_size$ blocks

3. **Associativity**: In how many cache locations can a particular block reside?
   $A = 1$ => "direct mapped"
   $A = n$ => "fully associative"



Cache Semantics

- A-way associative cache can be considered as a sequence of n/A fully associative sets

$$F = <f_1, \ldots, f_{n/A}>$$

- Each set f_i is a sequence of lines

$$L = <l_1, \ldots, l_A>$$

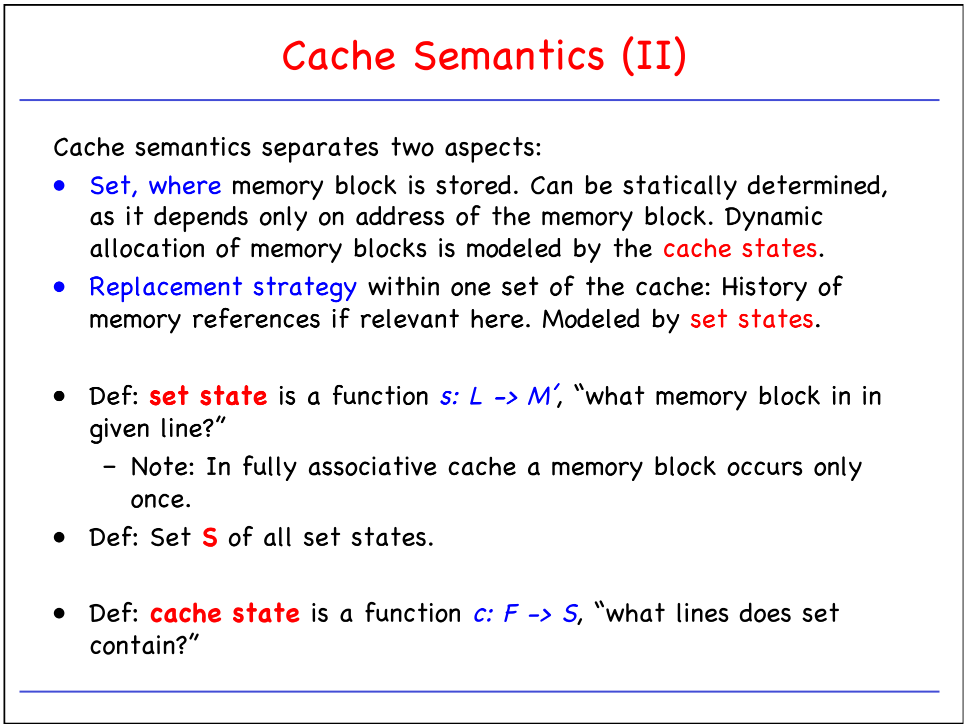
- The store is a set of memory blocks

$$M = \{m_1, \ldots, m_s\}$$

- The function adr : M -> integers gives address of each block.
- The function set : M -> F denotes where block gets stored:

$$\text{set}(m) = f_i, \text{ where } i = \text{adr}(m) \ \% \ (n/A) + 1$$

- No memory in a set line: $M' = M u \{I\}$



Cache Semantics (II)

Cache semantics separates two aspects:

- Set, where memory block is stored. Can be statically determined, as it depends only on address of the memory block. Dynamic allocation of memory blocks is modeled by the cache states.
- Replacement strategy within one set of the cache: History of memory references if relevant here. Modeled by set states.

- Def: **set state** is a function $s: L \to M'$, "what memory block in in given line?"
  - Note: In fully associative cache a memory block occurs only once.
- Def: Set **S** of all set states.

- Def: **cache state** is a function $c: F \to S$, "what lines does set contain?"

LRU Replacement Policy

- The side effects of referencing memory on the set/cache is represented by an *update* function. We note that
  - behavior of sets is independent of each other
  - order of blocks within set indicates relative age
- We number cache lines according to relative age of their memory block: $s(l_x) = m, m != I$ describes the relative age of block m according to LRU, not its physical position in the set.
- Def: **set update function** $U_S: S \times M \rightarrow S$ describes new set state for given set state and referenced memory block.
- Def: **cache update function** $U_C: C \times M \rightarrow C$ describes new cache state for a given cache state and a referenced memory block.

LRU Replacement Policy (II)

Definition 5 (set update). A set update function $\mathcal{U}_S\colon S \times M \rightarrow S$ describes the new set state for a given set state and a referenced memory block.

Definition 6 (cache update). A cache update function $\mathcal{U}_C\colon C \times M \rightarrow C$ describes the new cache state for a given cache state and a referenced memory block.

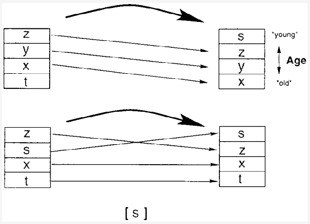


Figure 3. Update of a concrete fully associative set.

The most recently referenced memory block is put in the first position l_1 of the set. If the referenced memory block m is in the set already, then all memory blocks in the set that have been more recently used than m are shifted by one position to the next set line, i.e., they increase their relative age by one. If the memory block m is not yet in the set, then all memory blocks in the set are shifted and, if the set is full, the 'oldest', i.e., least recently used memory block is removed from the set.

Updates of fully associative sets with LRU replacement strategy are modeled in the following way (see Figure 3):

$$\mathcal{U}_S(s, m) = \begin{cases} [l_1 \mapsto m, & \\ \ l_i \mapsto s(l_{i-1}) \mid i = 2 \ldots h, & \\ \ l_i \mapsto s(l_i) \mid i = h+1 \ldots A]; & \text{if } \exists l_h\colon s(l_h) = m \\ & \\ [l_1 \mapsto m, & \\ \ l_i \mapsto s(l_{i-1}) \text{ for } i = 2 \ldots A]; & \text{otherwise} \end{cases}$$

Updates of A-way set associative caches are modeled in the following way:

$$\mathcal{U}_C(c, m) = c[set(m) \mapsto \mathcal{U}_S(c(set(m)), m)]$$

Control Flow Representation

- Program represented as **Control Flow Graph** (CFG):
  - Nodes are Basic Blocks.
  - **Basic block** is "a sequence of instructions in which control flow enters at the beginning and leaves at the end without halt or possibility of branching except at the end."
  - For each basic block the sequence of memory references is known.
- We can map control flow nodes to sequences of memory blocks (at least for instruction caches) and represent this as function

  $$L: V \rightarrow M^*$$

- We can extend U_C to sequences of memory references:

  $$U_C(c, \langle m_1, \ldots, m_y \rangle) = U_C(\ldots U_C(c, m_1) \ldots, m_y)$$

- Extend UC to path $\langle k_1, \ldots, k_p \rangle$ in control flow graph:

  $$U_C(c, \langle k_1, \ldots, k_p \rangle) = U_C(c, L(k_1), \ldots, L(k_p))$$

Must Analysis vs. May Analysis

- **Must Analysis** determines set of memory blocks definitely in the cache whenever control reaches a given program point.
- **May Analysis** determines all memory blocks that may be in the cache at a given program point.
- May analysis is used to guarantee **absence** of a memory block in the cache.
- Analysis for basic blocks and paths of basic blocks is simple.
- What about when paths merge?!

Abstract Cache States and Join Function

Definition 7 (abstract set state). An *abstract set state* $\hat{s}: L \to 2^M$ maps set lines to sets of memory blocks, where:

$$\forall l_a, l_b \in L: \forall m \in M: m \in (\hat{s}(l_a) \cap \hat{s}(l_b)) \Rightarrow l_a = l_b \tag{4}$$

The absence of any memory block is represented by the empty set $\{\}$. $\hat{S}$ denotes the set of all abstract set states.

Definition 8 (abstract cache state). An *abstract cache state* $\hat{c}: F \to \hat{S}$ maps sets to abstract set states, where:

$$\forall f_y \in F: \forall l_x \in L: \forall m \in M: m \in \hat{c}(f_y)(l_x) \Rightarrow set(m) = f_y \tag{5}$$

$\hat{C}$ denotes the set of all abstract cache states.

On control flow nodes with at least two predecessors, join functions are used to combine the abstract cache states.

Definition 9 (join function). A *join function* $\hat{\mathcal{J}}: \hat{C} \times \hat{C} \mapsto \hat{C}$ combines two abstract cache states.

MUST Analysis

- An abstract cache state *c*ˆ describes a set of concrete cache states *c*, and an abstract set state *s*ˆ describes a set of concrete set states *s*.
- To determine if a memory block is definitely in the cache we use abstract set states where the position (the relative *age*) of a memory block in the abstract set state *s*ˆ is an upper bound of the positions (the relative *ages*) of the memory block in the concrete set states that *s*ˆ represents.
- *ma* ∈ *s*ˆ(*lx*) means that the memory block *ma* is in the cache. The position (relative age) of a memory block *ma* in a set can only be changed by references to memory blocks *mb* with *set*(*ma*) = *set*(*mb*), i.e., by memory references that go into the same set. The position is not changed by references to memory blocks *mb* ∈ *s*ˆ(*ly*) where *y* ≤ *x*, i.e., memory blocks that are already in the cache and are "younger" or the same age as *ma*.
- *ma* will stay in the cache at least for the next *A* − *x* references that go to the same set and are not yet in the cache or are *older* than *ma*.

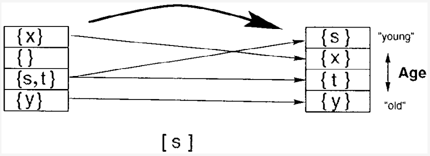


Figure 4. Update of an abstract fully associative set for the Must analysis.

$$\hat{\mathcal{U}}^{\cap}_{\hat{S}}(\hat{s},m) = \begin{cases} [l_1 \mapsto \{m\}, & \\ \;l_i \mapsto \hat{s}(l_{i-1}) \mid i = 2\dots h-1, & \\ \;l_h \mapsto \hat{s}(l_{h-1}) \cup (\hat{s}(l_h) - \{m\}), & \\ \;l_i \mapsto \hat{s}(l_i) \mid i = h+1\dots A]; & \text{if } \exists l_h\colon m \in \hat{s}(l_h) \\ & \\ [l_1 \mapsto \{m\}, & \\ \;l_i \mapsto \hat{s}(l_{i-1}) \mid i = 2\dots A]; & \text{otherwise} \end{cases}$$

$\hat{\mathcal{J}}^{\cap}_{\hat{S}}(\hat{s}_1, \hat{s}_2) = \hat{s}$, where:

$$\hat{s}(l_x) = \{m \mid \exists l_a, l_b \text{ with } m \in \hat{s}_1(l_a), m \in \hat{s}_2(l_b) \text{ and } x = max(a,b)\}$$