Worst-Case Execution Time Analysis

Prof. Dr. Jian-Jia Chen

LS 12, TU Dortmund

29, 30 April, 2019

Most Essential Assumptions for Real-Time Systems

Upper bound on the execution times:

- Commonly, called the *Worst-Case Execution Time (WCET)*

What does Execution Time Depend on

- Input parameters
  - Algorithm parameters
  - Problem size
  - etc.
- Initial states and intermediate states of the system while executing
  - Cache configuration, replacement policies
  - Pipelines
  - Speculations
  - etc.
- Interferences from the environment
  - Scheduling
  - Interrupts
  - etc.

How to Derive the Worst-Case Execution Time (WCET)

- Most of industry's best practice
  - Measure it: determine WCET directly by running or simulating a set of inputs.
    - There is no guarantee to give an upper bound of the WCET.
    - The derived WCET could be too optimistic.
  - Exhaustive execution: by considering the set of all the possible inputs
    - In general, not possible
    - The inputs have to cover all the possible initial states and intermediate states of the system, which is also usually not possible.
- Compute it
  - In general, not possible neither, as computing (tight) WCET for a program is *uncomputable* by Turing machines.
  - Based on some structures, it is possible and the derived solution is a safe upper bound of the WCET.

Why is It Uncomputable?

Halting Problem

Given the description of a Turing machine m and its input x, the problem is to answer the question whether the machine halts on x.

Theorem

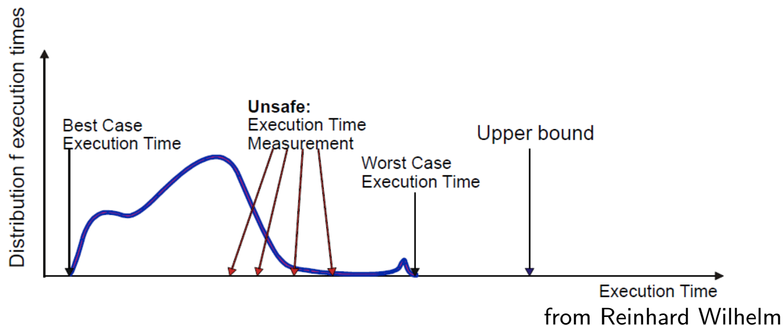
The Halting Problem is undecidable (uncomputable). In other words, one cannot use an algorithm to decide whether another algorithm m halts on a specific input.

WCET is undecidable

It is even undecidable if it terminates at all. Deriving the WCET is of course undecidable.

Please refer to the textbook of Computational Complexity by Prof. Papadimitriou.

Execution Time Distribution



from Reinhard Wilhelm

Our objectives:

- *Upper bound* of the execution time as tightly as possible.
- All control-flow paths, by considering all possible inputs.
- All paths through the architecture, resulting from the potential initial and assumed intermediate architectural states.

Timing Analysis

By considering systems, in general, with

- finite architectural configurations, finite input domains, and bounded loops and recursion,

WCET is computable.

Timing Analysis

By considering systems, in general, with

- finite architectural configurations, finite input domains, and bounded loops and recursion,

WCET is computable.

But......, the search space is too large to explore it exhaustively!

Why is It Hard for Analyzing WCET?

Execution time $e(i)$ of machine instruction i

- In the good old time:
  $e(i)$ is a constant c, which could be found in the data sheet
- Nowadays, especially for high-performance processors:
  $e(i)$ also depends on the (architectural) execution state s.

  $$\min\{e(i,s)|s \in S\} \leq e(i) \leq \max\{e(i,s)|s \in S\},$$

  where S is the set of all states.
- Using $\max\{e(i,s)|s \in S\}$ is safe for WCET, but might be not tight since some states in S might not be possible to reach by some inputs.
- Execution history, resulting in a smaller set of reachable execution states, has to be enforced to improve the tightness of the analysis.

Variability of Execution Times

x = a + b;

```
LOAD    r2, _a
LOAD    r1, _b
ADD     r3,r2,r1
```

PPC 755

Execution Time (Clock Cycles)

350
300
250
200
150
100
50
0

Best Case
Worst Case

Clock Cycles

Timing Accidents and Penalties

- Timing Accident: cause for an increase of the execution time of an instruction
- Timing Penalty: the associated increase
- Types of timing accidents
  - Cache misses
  - Pipeline stalls
  - Branch mispredictions
  - Bus collisions
  - Memory refresh of DRAM
  - TLB miss

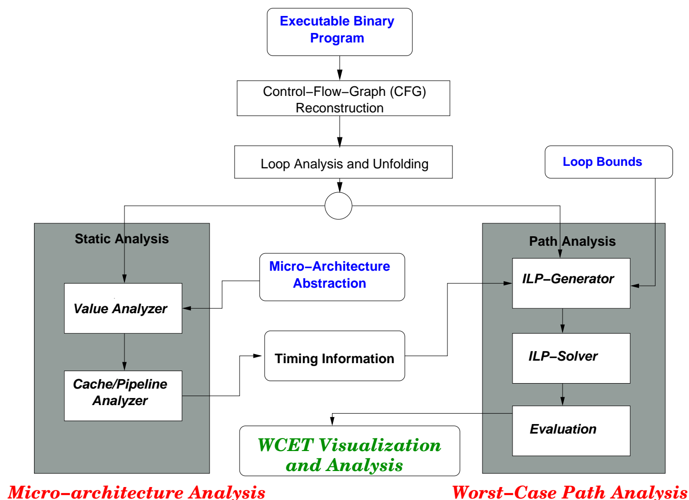
Overall Approach: Modularization

- Architecture Analysis:
  - Use Abstract Interpretation.
  - Exclude as many Timing Accidents as possible during analysis.
    - Certain timing accidents will never happen, e.g., at a certain program point, instruction fetch will never cause a cache miss.
    - The more accidents excluded, the lower (better) the upper bound.
  - Determine WCET for basic blocks, based on context information.
- Worst-Case Path Determination:
  - Map control flow graph to an Integer Linear Program (ILP).
  - Determine upper bound and associated path.

High-Level Objectives: Upper Bound of WCET

- It must be safe, i.e., not underestimate.
- It should be tight, i.e., not far away from real WCET.
- The analysis effort must be tolerable.

Overall Structure



Outline

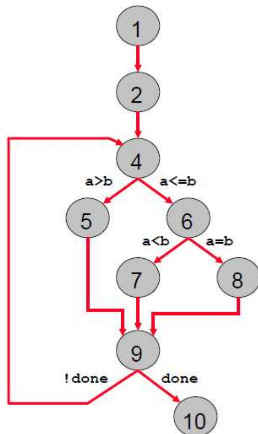
Introduction

Program Path Analysis

Static Analysis
- Value Analysis
- Cache Analysis
- Pipeline Analysis

Control Flow Graph (CFG)

```
     what_is_this {
1      read (a,b);
2      done = FALSE;
3      repeat {
4        if (a>b)
5           a = a-b;
6        elseif (b>a)
7           b = b-a;
8        else done = TRUE;
9      } until done;
10     write (a);
     }
```



Basic Blocks

Definition: *A basic block is a sequence of instructions where the control flow enters at the beginning and exits at the end, in which it is highly amenable to analysis.*

$a[0] := b[0] + c[0]$

$a[1] := b[3] + c[3]$

$a[2] := b[6] + c[6]$

$d := a[0] * a[1]$

$e := d/a[2]$

if $e < 10$ goto L

Basic Blocks

Definition: *A basic block is a sequence of instructions where the control flow enters at the beginning and exits at the end, in which it is highly amenable to analysis.*

$a[0] := b[0] + c[0]$

$a[1] := b[3] + c[3]$

$a[2] := b[6] + c[6]$

$d := a[0] * a[1]$

$e := d/a[2]$

if $e < 10$ goto L

Determining the basic blocks

- Beginning:
  - the first instruction
  - targets of un/conditional jumps
  - instructions that follow un/conditional jumps
- Ending:
  - the basic block consists of the block beginning and runs until the next block beginning (exclusive) or until the program ends

Program Path Analysis

- Problem: Which sequence of instructions is executed in the worst case (i.e., the longest execution time)?
- Input:
  - Timing information for each basic block, derived from static analysis (value/cache/pipeline analysis)
  - Loop bounds by specification
  - CFG derived from the executable binary program
- Basic Concept:
  - Transform structure of CFG into a set of (integer) linear equations
  - Solution of the Integer Linear Program (ILP) yields bound on the WCET.

Program Path Analysis: Formal Definition

Input

A CFG with N basic blocks, in which each basic block B_i has a worst-case execution time c_i, given by static analysis.

Output

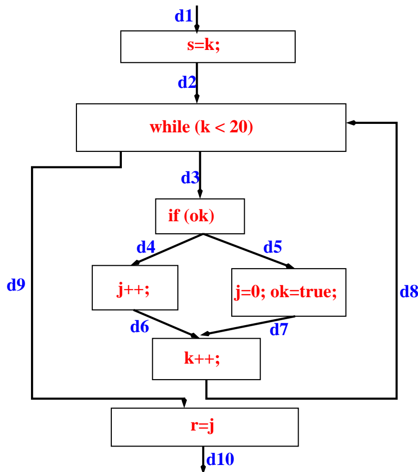
Suppose that each block B_i is executed *exactly* x_i times. What is the worst-case execution time

$$WCET = \sum_{i=1}^{N} c_i \cdot x_i,$$

such that the values of x_is satisfy the structural constraints in the CFG?

Note that additional constraints provided by the programmer (bounds for loop counters, etc.) can also be included.

Example for CFG Constraints



Flow equations: (x_i is a variable)

$d_1 = d_2 = x_1$

$d_3 + d_9 = d_2 + d_8 = x_2$

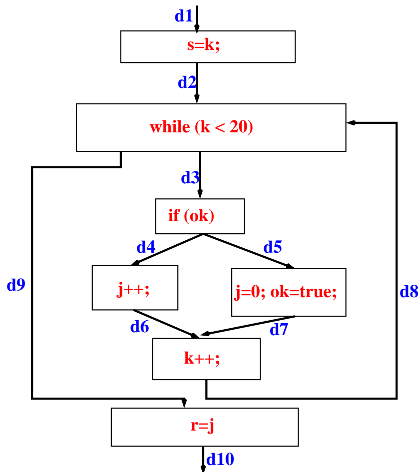
$d_4 + d_5 = d_3 = x_3$

$d_6 + d_7 = d_8 = x_4$

$d_4 = d_6 = x_5$

$d_5 = d_7 = x_6$

Example for Additional Constraints



The loop is executed for at most 20 times when k is initialized with a non-negative number:

$$x_3 \leq 20 x_1.$$

The basic block for $j = 0; ok = true;$ is executed for at most one time:

$$x_6 \leq x_1.$$

WCET: ILP Formulation

$$WCET = \max\{\sum_{i=1}^{N} c_i \cdot x_i \;|\; d_1 = 1 \text{ and } \sum_{j \in in(B_i)} d_j = \sum_{k \in out(B_i)} d_k = x_i, i = 1, \ldots, N \text{ and additional constraints}\}$$

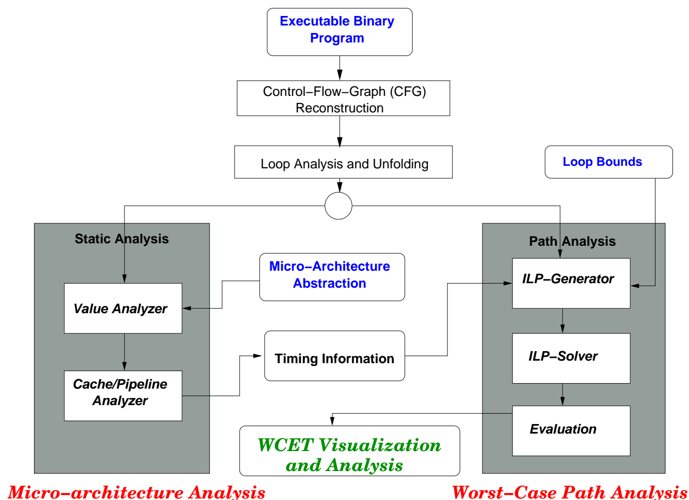
Outline

Introduction

Program Path Analysis

Static Analysis
- Value Analysis
- Cache Analysis
- Pipeline Analysis

Overall Structure



Outline

Introduction

Program Path Analysis

Static Analysis
- Value Analysis
- Cache Analysis
- Pipeline Analysis

Value Analysis: Motivation and Method

- Motivation
  - Provide access information to data-cache/pipeline analysis
  - Detect infeasible paths
  - Derive loop bounds
- Method
  - Calculate intervals at all program points
  - By considering addresses, register contents, local and global variables.

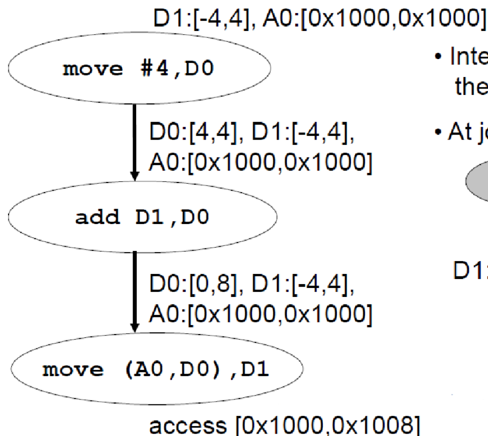
Abstract Interpretation

Perform the program's computation using value descriptions or abstract values in place of the concrete values.

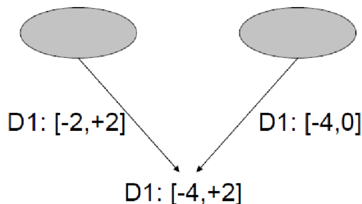
Abstract Interpretation: Manipulation

- abstract domain - related to concrete domain by abstraction and concretization functions
  - Replace an integer/double operator by using intervals
  - e.g., $L = [3, 5]$ stands for L is a value between 3 and 5
- abstract transfer functions for each statement type
  - e.g., operator $+$: $[3, 5] + [2, 6] = [5, 11]$
  - e.g., operator $-$: $[3, 5] - [2, 6] = [-3, 3]$
- a join function combining intervals from different paths
  - $[a, b]$ join $[c, d]$ becomes $[\min\{a, c\}, \max\{b, d\}]$
  - $[3, 5]$ join $[2, 4]$ becomes $[2, 5]$.

Value Analysis



- Intervals are computed along the CFG edges
- At joins, intervals are „unioned“



Outline

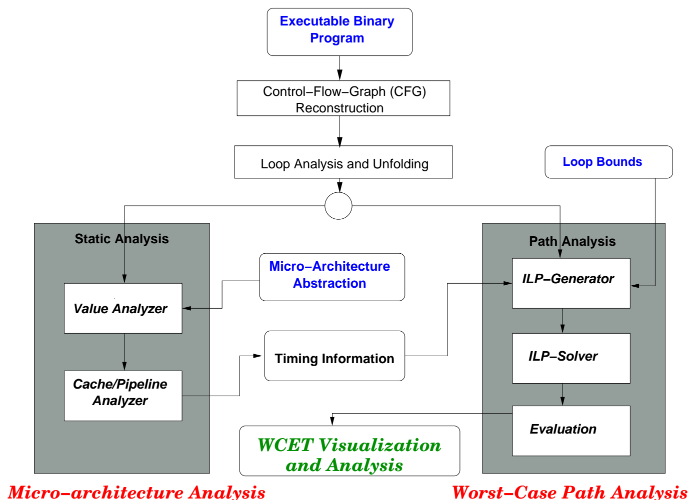
Introduction

Program Path Analysis

Static Analysis

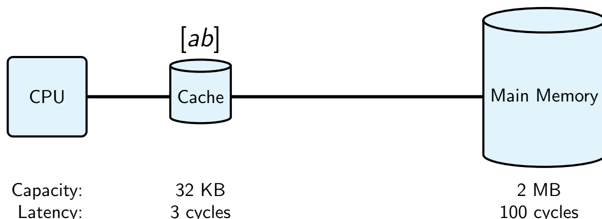
- Value Analysis
- Cache Analysis
- Pipeline Analysis

Overall Structure



Caches: Fast Memory to Deal with the Memory Wall

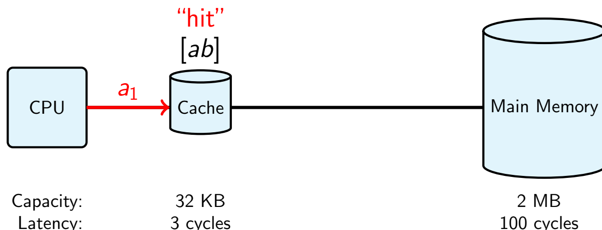
- How they work:
  - dynamically
  - managed by replacement policy



- Why they work: *principle of locality*
  - spatial
  - temporal

Caches: Fast Memory to Deal with the Memory Wall

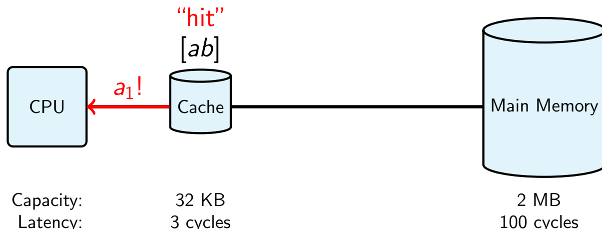
- How they work:
  - dynamically
  - managed by replacement policy



- Why they work: *principle of locality*
  - spatial
  - temporal

Caches: Fast Memory to Deal with the Memory Wall

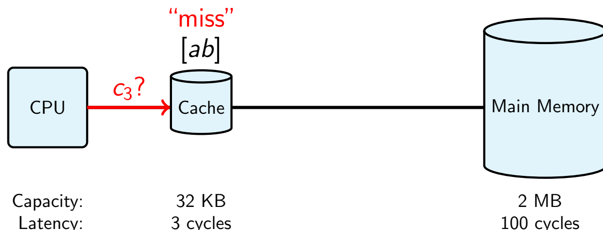
- How they work:
  - dynamically
  - managed by replacement policy



- Why they work: *principle of locality*
  - spatial
  - temporal

Caches: Fast Memory to Deal with the Memory Wall

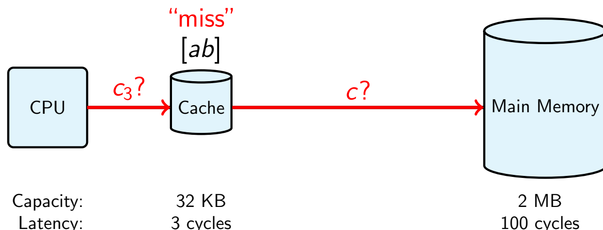
- How they work:
  - dynamically
  - managed by replacement policy



- Why they work: *principle of locality*
  - spatial
  - temporal

Caches: Fast Memory to Deal with the Memory Wall

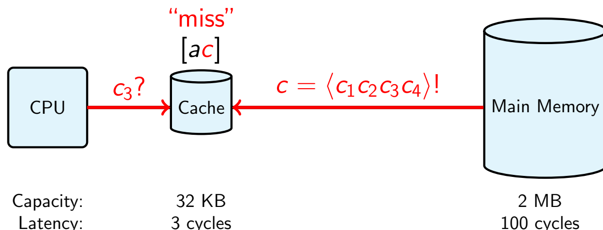
- How they work:
  - dynamically
  - managed by replacement policy



- Why they work: *principle of locality*
  - spatial
  - temporal

Caches: Fast Memory to Deal with the Memory Wall

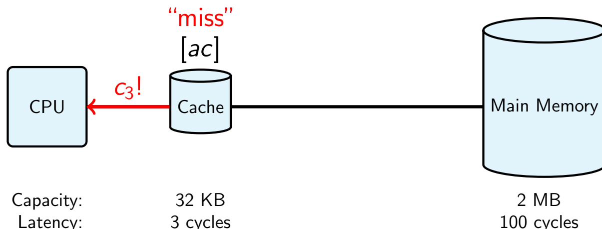
- How they work:
    - dynamically
    - managed by replacement policy



- Why they work: *principle of locality*
    - spatial
    - temporal

Caches: Fast Memory to Deal with the Memory Wall

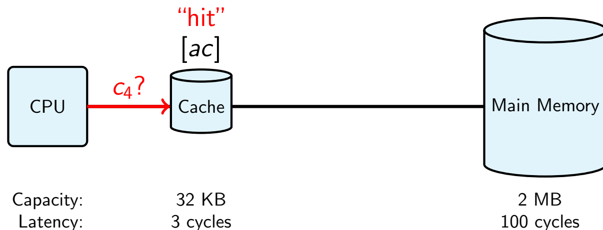
- How they work:
  - dynamically
  - managed by replacement policy



- Why they work: *principle of locality*
  - spatial
  - temporal

Caches: Fast Memory to Deal with the Memory Wall

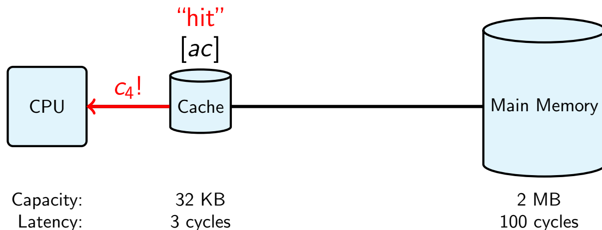
- How they work:
  - dynamically
  - managed by replacement policy



- Why they work: *principle of locality*
  - spatial
  - temporal

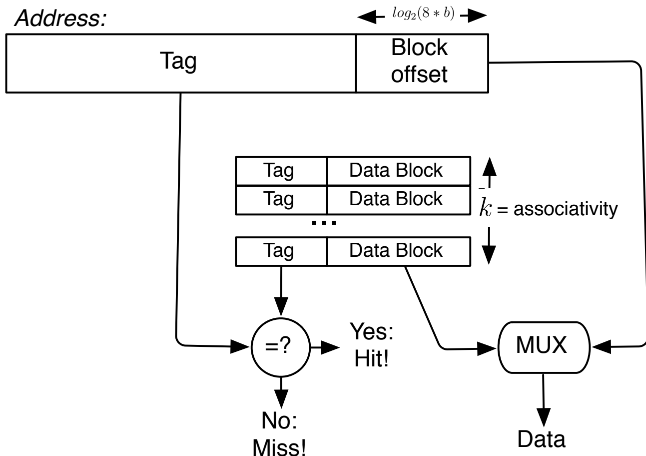
Caches: Fast Memory to Deal with the Memory Wall

- How they work:
  - dynamically
  - managed by replacement policy

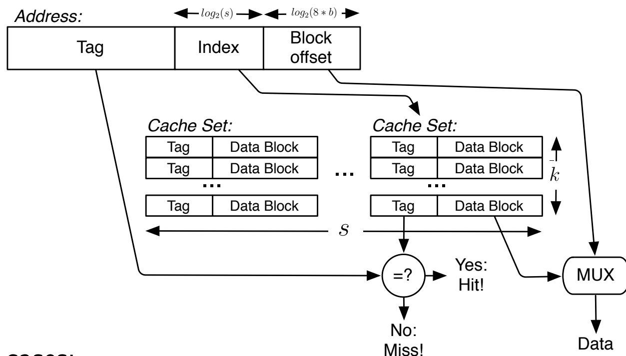


- Why they work: *principle of locality*
  - spatial
  - temporal

Fully Associative Caches



Set-Associative Caches



Special cases:

- direct-mapped cache: only one line per cache set
- fully-associative cache: only one cache set

Replacement Policies

- Least-Recently-Used (LRU) used in
  INTEL PENTIUM I and MIPS 24K/34K
- First-In First-Out (FIFO or Round-Robin) used in
  MOTOROLA POWERPC 56x, INTEL XSCALE, ARM9, ARM11
- Pseudo-LRU (PLRU) used in
  INTEL PENTIUM II-IV and POWERPC 75x
- Most Recently Used (MRU) as described in literature

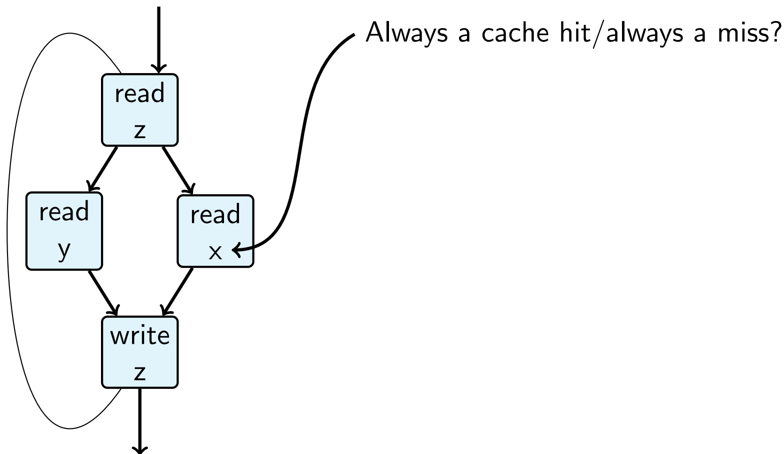
Each cache set is treated independently:
⟶ Set-associative caches are compositions of fully-associative caches.

Cache Analysis for LRU

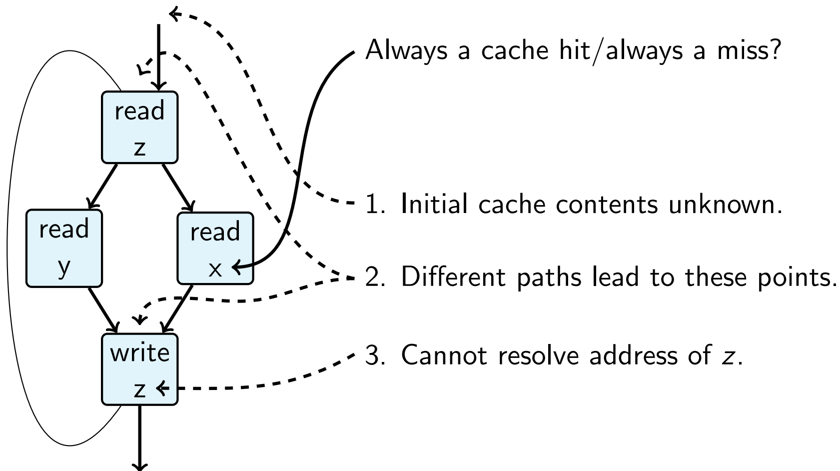
Two types of cache analyses:

1. Local guarantees: classification of individual accesses
   - Must-Analysis $\longrightarrow$ Underapproximates cache contents
   - May-Analysis $\longrightarrow$ Overapproximates cache contents
2. Global guarantees: bounds on cache hits/misses

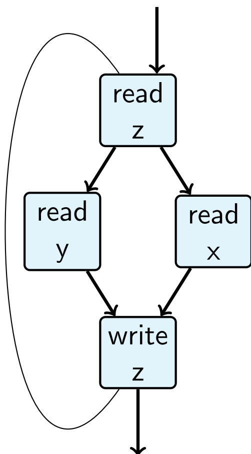
Challenges for Cache Analysis



Challenges for Cache Analysis



Using Abstract Interpretation

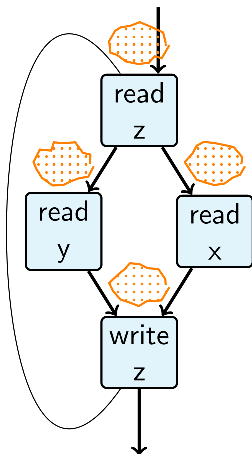


Collecting Semantics =
set of states at each program point that any execution may encounter there

Two approximations:

	Collecting Semantics	uncomputable
$\subseteq$	Cache Semantics	computable
$\subseteq$	γ(Abstract Cache Sem.)	efficiently computable

Using Abstract Interpretation

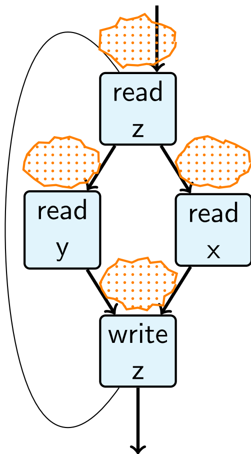


Collecting Semantics =
set of states at each program point that any execution may encounter there

Two approximations:

	Collecting Semantics	uncomputable
$\subseteq$	Cache Semantics	computable
$\subseteq$	γ(Abstract Cache Sem.)	efficiently computable

Using Abstract Interpretation

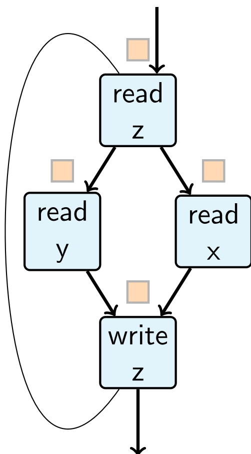


Collecting Semantics =
set of states at each program point that any execution may encounter there

Two approximations:

	Collecting Semantics	uncomputable
$\subseteq$	Cache Semantics	computable
$\subseteq$	γ(Abstract Cache Sem.)	efficiently computable

Using Abstract Interpretation

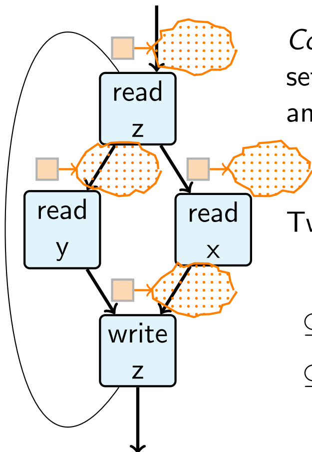


Collecting Semantics =
set of states at each program point that any execution may encounter there

Two approximations:

	Collecting Semantics	uncomputable
$\subseteq$	Cache Semantics	computable
$\subseteq$	γ(Abstract Cache Sem.)	efficiently computable

Using Abstract Interpretation

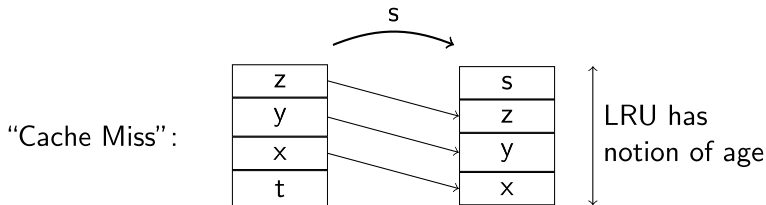


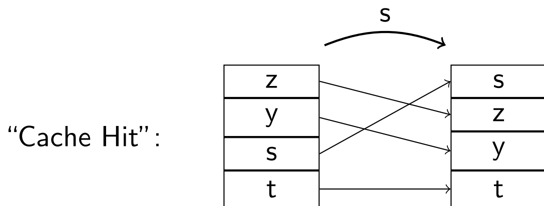
Collecting Semantics =
set of states at each program point that any execution may encounter there

Two approximations:

	Collecting Semantics	uncomputable
$\subseteq$	Cache Semantics	computable
$\subseteq$	γ(Abstract Cache Sem.)	efficiently computable

Least-Recently-Used (LRU): Concrete Behavior





LRU: Must-Analysis: Abstract Domain

- Used to predict *cache hits*.
- Maintains *upper bounds on ages* of memory blocks.
- Upper bound $\leq$ associativity $\longrightarrow$ memory block definitely cached.

Example

Abstract state:

{x}
{}
{s,t}
{}

age 0 (top), age 3 (bottom)

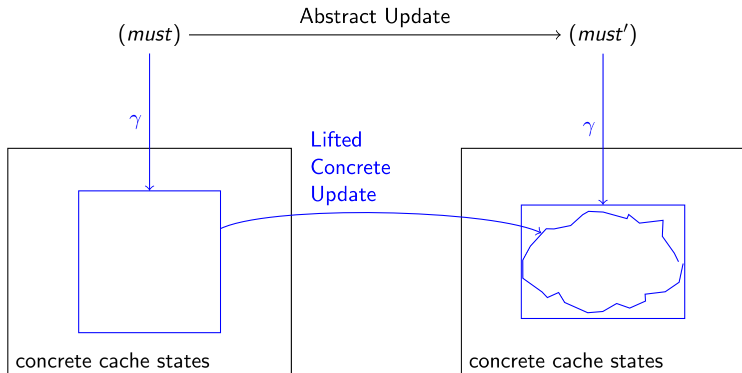
...and its interpretation:

Describes the set of all concrete cache states in which x, s, and t occur,

- x with an age of 0,
- s and t with an age not older than 2.

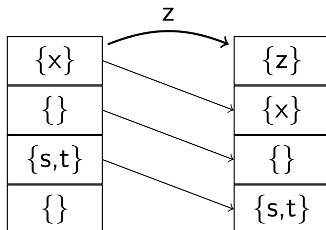
$$\gamma([\{x\},\{\},\{s,t\},\{\}]) = \{[x,s,t,a],[x,t,s,a],[x,s,t,b],\ldots\}$$

Sound Update – Local Consistency

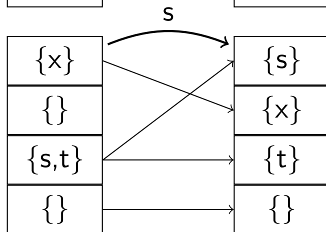


LRU: Must-Analysis: Update

"Potential Cache Miss":



"Definite Cache Hit":



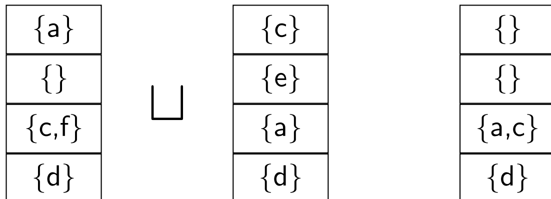
Why does t remain in the same set in Abstract Interpretation in the second case?

LRU: Must-Analysis: Join

Need to combine information where control-flow merges.

Join should be conservative:

- $\gamma(A) \subseteq \gamma(A \sqcup B)$
- $\gamma(B) \subseteq \gamma(A \sqcup B)$



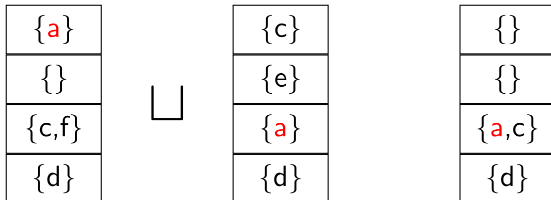
"Intersection + Maximal Age"

LRU: Must-Analysis: Join

Need to combine information where control-flow merges.

Join should be conservative:

- $\gamma(A) \subseteq \gamma(A \sqcup B)$
- $\gamma(B) \subseteq \gamma(A \sqcup B)$



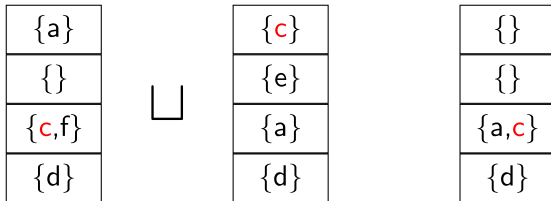
"Intersection + Maximal Age"

LRU: Must-Analysis: Join

Need to combine information where control-flow merges.

Join should be conservative:

- $\gamma(A) \subseteq \gamma(A \sqcup B)$
- $\gamma(B) \subseteq \gamma(A \sqcup B)$



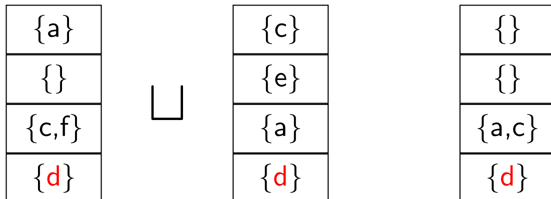
"Intersection + Maximal Age"

LRU: Must-Analysis: Join

Need to combine information where control-flow merges.

Join should be conservative:

- $\gamma(A) \subseteq \gamma(A \sqcup B)$
- $\gamma(B) \subseteq \gamma(A \sqcup B)$



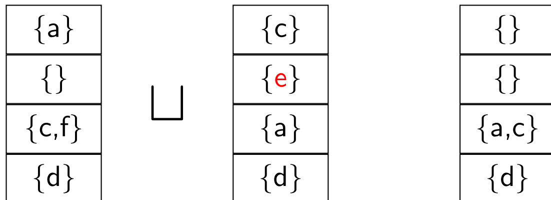
"Intersection + Maximal Age"

LRU: Must-Analysis: Join

Need to combine information where control-flow merges.

Join should be conservative:

- $\gamma(A) \subseteq \gamma(A \sqcup B)$
- $\gamma(B) \subseteq \gamma(A \sqcup B)$



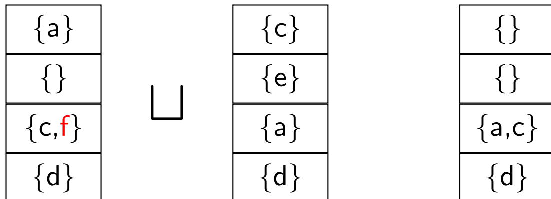
"Intersection + Maximal Age"

LRU: Must-Analysis: Join

Need to combine information where control-flow merges.

Join should be conservative:

- $\gamma(A) \subseteq \gamma(A \sqcup B)$
- $\gamma(B) \subseteq \gamma(A \sqcup B)$



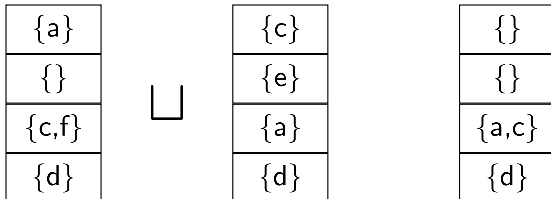
"Intersection + Maximal Age"

LRU: Must-Analysis: Join

Need to combine information where control-flow merges.

Join should be conservative:

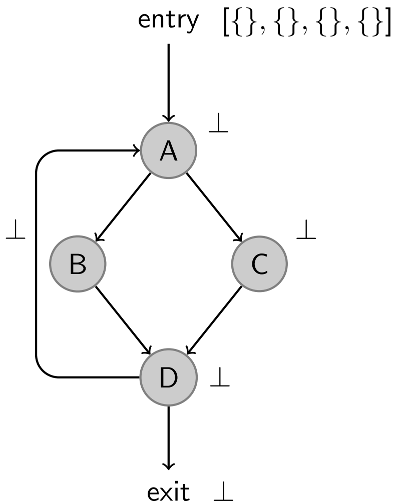
- $\gamma(A) \subseteq \gamma(A \sqcup B)$
- $\gamma(B) \subseteq \gamma(A \sqcup B)$



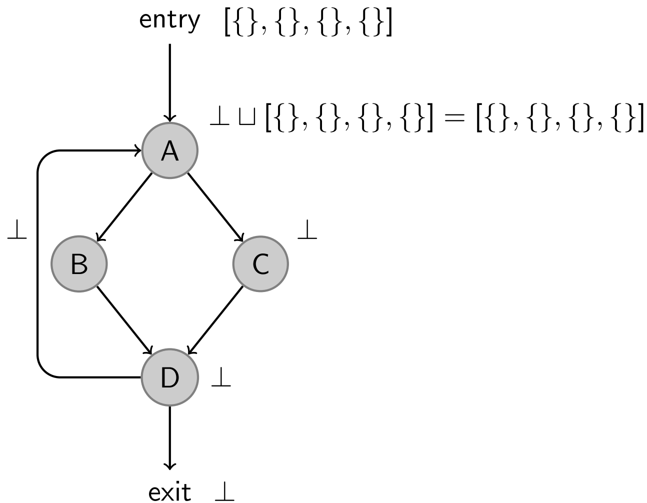
"Intersection + Maximal Age"

How many memory blocks can be in the must-cache?

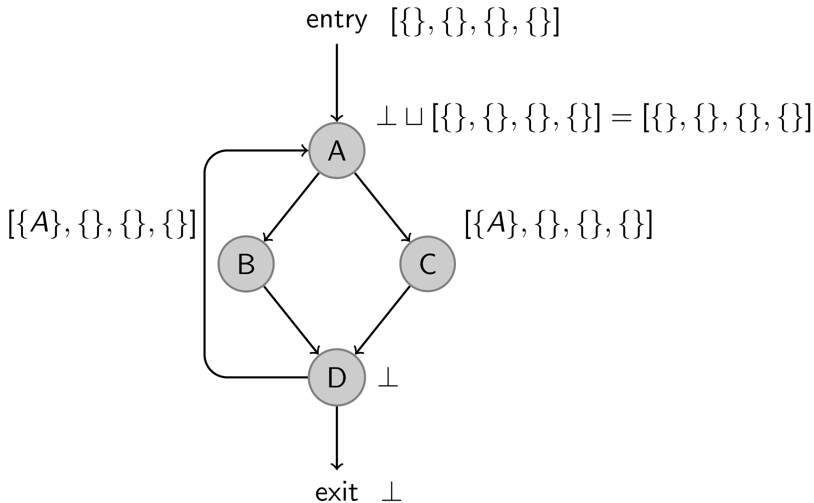
Example: Must-Analysis



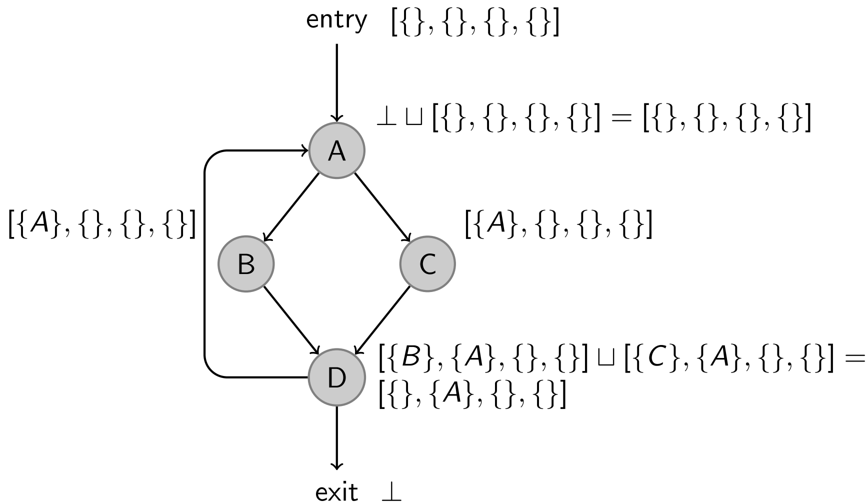
Example: Must-Analysis



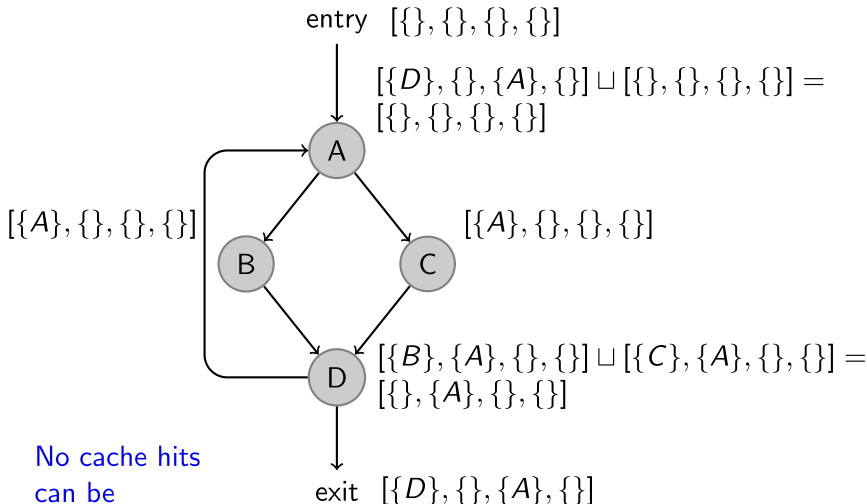
Example: Must-Analysis



Example: Must-Analysis



Example: Must-Analysis



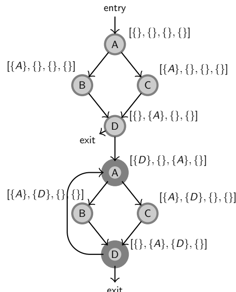
Context-Sensitive Analysis/Virtual Loop-Unrolling

- Problem:
  - The first iteration of a loop will always result in cache misses.
  - Similarly for the first execution of a function.
- Solution:
  - Virtually Unroll Loops: Distinguish the first iteration from others
  - Distinguish function calls by calling context.

Virtually unrolling the loop once:

- Accesses to A and D are provably hits after the first iteration
- Accesses to B and C can still not be classified. Within each execution of the loop, they may only miss once.

$\longrightarrow$ Persistence Analysis



LRU: May-Analysis: Abstract Domain

- Used to predict *cache misses*.
- Maintains *lower bounds on ages* of memory blocks.
- Lower bound $\geq$ associativity
  $\longrightarrow$ memory block definitely *not* cached.

Abstract state:

{x,y}	age 0
{}	
{s,t}	
{u}	age 3

and its interpretation:

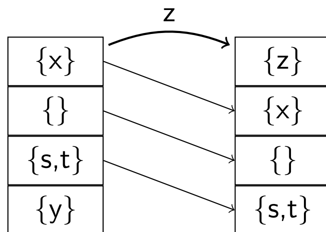
Describes a set of all concrete cache states, where no memory blocks except x, y, s, t, and u occur,

- x and y with an age of at least 0,
- s and t with an age of at least 2,
- u with an age of at least 3.

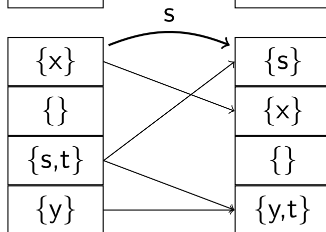
$$\gamma([\{x,y\},\{\},\{s,t\},\{u\}]) = \{[x,y,s,t],[y,x,s,t],[x,y,s,u],\ldots\}$$

LRU: May-Analysis: Update

"Definite Cache Miss":



"Potential Cache Hit":



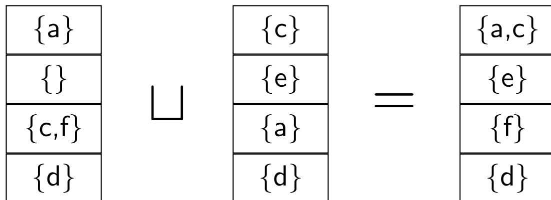
Why does t move to an older set in Abstract Interpretation in the second case?

LRU: May-Analysis: Join

Need to combine information where control-flow merges.

Join should be conservative:

- $\gamma(A) \subseteq \gamma(A \sqcup B)$
- $\gamma(B) \subseteq \gamma(A \sqcup B)$



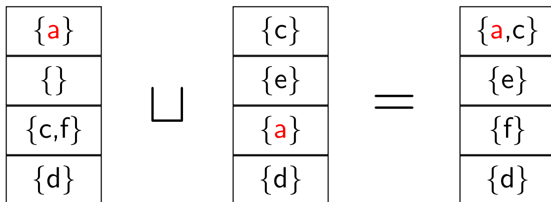
"Union + Minimal Age"

LRU: May-Analysis: Join

Need to combine information where control-flow merges.

Join should be conservative:

- $\gamma(A) \subseteq \gamma(A \sqcup B)$
- $\gamma(B) \subseteq \gamma(A \sqcup B)$



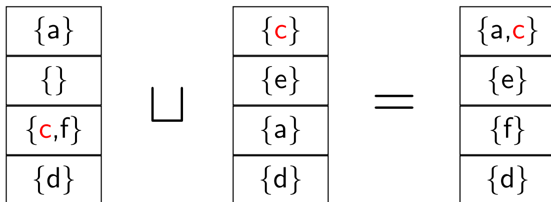
"Union + Minimal Age"

LRU: May-Analysis: Join

Need to combine information where control-flow merges.

Join should be conservative:

- $\gamma(A) \subseteq \gamma(A \sqcup B)$
- $\gamma(B) \subseteq \gamma(A \sqcup B)$



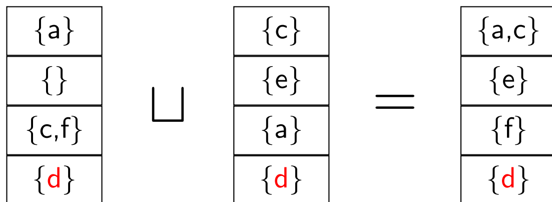
"Union + Minimal Age"

LRU: May-Analysis: Join

Need to combine information where control-flow merges.

Join should be conservative:

- $\gamma(A) \subseteq \gamma(A \sqcup B)$
- $\gamma(B) \subseteq \gamma(A \sqcup B)$



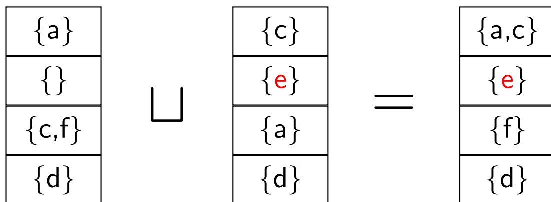
"Union + Minimal Age"

LRU: May-Analysis: Join

Need to combine information where control-flow merges.

Join should be conservative:

- $\gamma(A) \subseteq \gamma(A \sqcup B)$
- $\gamma(B) \subseteq \gamma(A \sqcup B)$



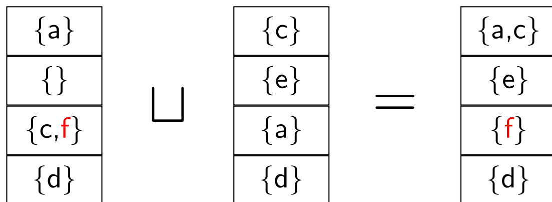
"Union + Minimal Age"

LRU: May-Analysis: Join

Need to combine information where control-flow merges.

Join should be conservative:

- $\gamma(A) \subseteq \gamma(A \sqcup B)$
- $\gamma(B) \subseteq \gamma(A \sqcup B)$



"Union + Minimal Age"

Outline

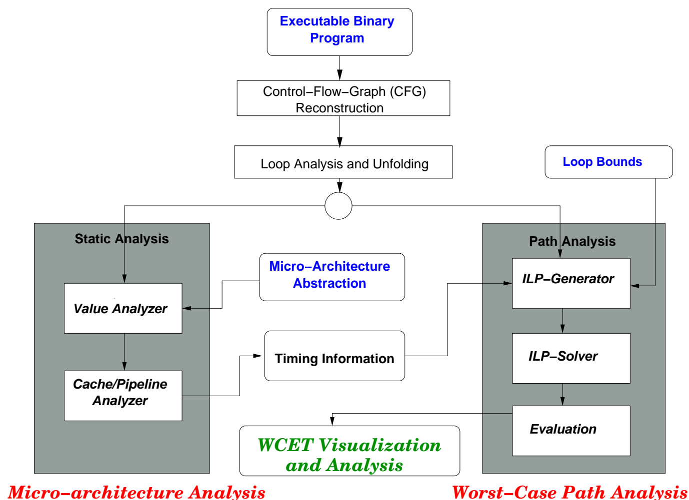
Introduction

Program Path Analysis

Static Analysis

- Value Analysis
- Cache Analysis
- Pipeline Analysis

Overall Structure



Pipelines

An instruction execution consists of several sequential phases, e.g.,

- Fetch
- Decode
- Execute
- Write Back

Inst 1	Inst 2	Inst 3	Inst 4
Fetch			
Decode	Fetch		
Execute	Decode	Fetch	
Write Back	Execute	Decode	Fetch
	Write Back	Execute	Decode
		Write Back	Execute
			Write Back

Hardware Features: Pipelines

- Instruction execution is split into several stages.
- Several instructions can be executed in parallel.
- Some pipelines can begin more than one instruction per cycle: VLIW, Superscalar.
- Some CPUs can execute instructions out-of-order.
- Practical Problems: Hazards and cache misses.
  - Data Hazards: Operands not yet available (Data Dependences)
    - By applying dependence analysis
  - Control Hazards: Conditional branch
    - By applying dependence analysis
  - Resource Hazards: Consecutive instructions use same resource
    - By applying analysis on the resource reservation tables
  - Instruction-Cache Hazards: Instruction fetch causes cache miss
    - By applying static cache analysis

CPU as a (Concrete) State Machine

- Processor (pipeline, cache, memory, inputs) viewed as a *big state machine*, performing transitions every clock cycle.
- Starting in an initial state for an instruction, transitions are performed, until a final state is reached:
  - end state: instruction has left the pipeline
  - # transitions: execution time of instruction
- function *exec* (b: basic block, s: *concrete* pipeline state) t: trace
  - interprets instruction stream of b starting in state s producing trace t
  - successor basic block is interpreted starting in initial state last(t)
  - length(t) gives number of cycles
- function *exec* (b: basic block, s: *abstract* pipeline state) t: trace
  - interprets instruction stream of b (annotated with cache information) starting in state s producing trace t
  - length(t) gives number of cycles
- Alternatives: use a set of states for an instruction instead of one starting state. This is useful when there are multiple predecessors.

Summary of WCET Analysis

- Value analysis
- Cache analysis
  - using statically computed effective addresses and loop bounds
- Pipeline analysis
  - assume cache hits where predicted,
  - assume cache misses where predicted or not excluded.
  - Only the "worst" result states of an instruction need to be considered as input states for successor instructions!

aiT-Tool