# Timing Analysis and Timing Predictability

Reinhard Wilhelm
Saarland University
Saarbrücken

AbsInt
Angewandte Informatik GmbH

UNIVERSITÄT
DES
SAARLANDES

Supported by ARTIST2, ARTIST DESIGN, Predator, AVACS

# Structure of the Lectures

1. Introduction
2. Static timing analysis
   1. the problem
   2. our approach
   3. the success
   4. tool architecture
3. Cache analysis
4. Pipeline analysis
5. Value analysis

------------------------------------------------------------

1. Timing Predictability
   - caches
   - non-cache-like devices
   - future architectures
2. Conclusion

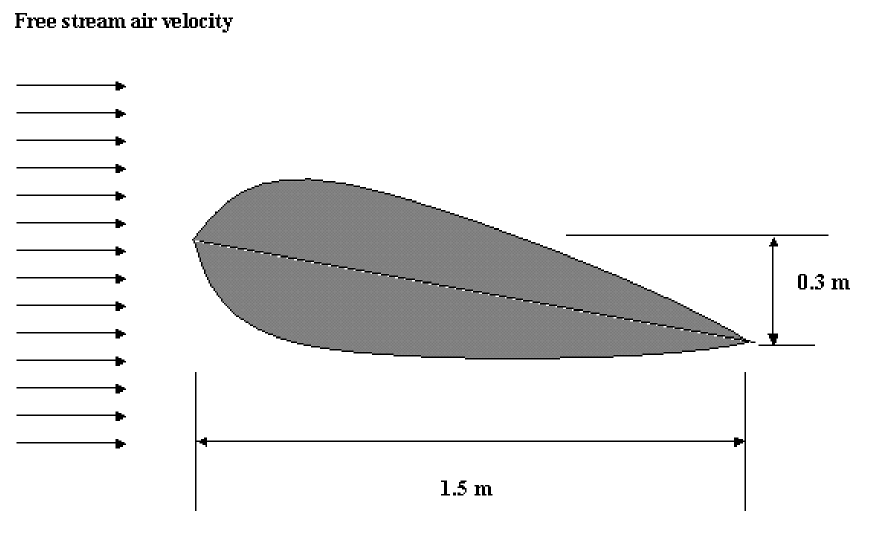
# Industrial Needs

Hard real-time systems, often in safety-critical applications abound

- Aeronautics, automotive, train industries, manufacturing control

Sideairbag in car,
Reaction in <10 mSec

Wing vibration of airplane,
sensing every 5 mSec



# Hard Real-Time Systems

- Embedded controllers are expected to finish their tasks reliably within time bounds.
- Task scheduling must be performed
- Essential: upper bound on the execution times of all tasks statically known
- Commonly called the Worst-Case Execution Time (WCET)
- Analogously, Best-Case Execution Time (BCET)

# Static Timing Analysis

Embedded controllers are expected to finish their tasks reliably within time bounds.

**The problem:**

Given

1. a software to produce some reaction,
2. a hardware platform, on which to execute the software,
3. required reaction time.

Derive: a guarantee for timeliness.

# What does Execution Time Depend on?

- the input – this has always been so and will remain so,
- the initial execution state of the platform – this is (relatively) new,
- interferences from the environment – this depends on whether the system design admits it (preemptive scheduling, interrupts).

Caused by caches, pipelines, speculation etc.

Explosion of the space of inputs **and** initial states ⇒ no exhaustive approaches feasible.

"external" interference as seen from analyzed task

# Modern Hardware Features

- Modern processors increase (average-case) performance by using:
  Caches, Pipelines, Branch Prediction, Speculation
- These features make bounds computation difficult:
  Execution times of instructions vary widely
  - Best case - everything goes smoothly: no cache miss, operands ready, needed resources free, branch correctly predicted
  - Worst case - everything goes wrong: all loads miss the cache, resources needed are occupied, operands are not ready
  - Span may be several hundred cycles

# Access Times

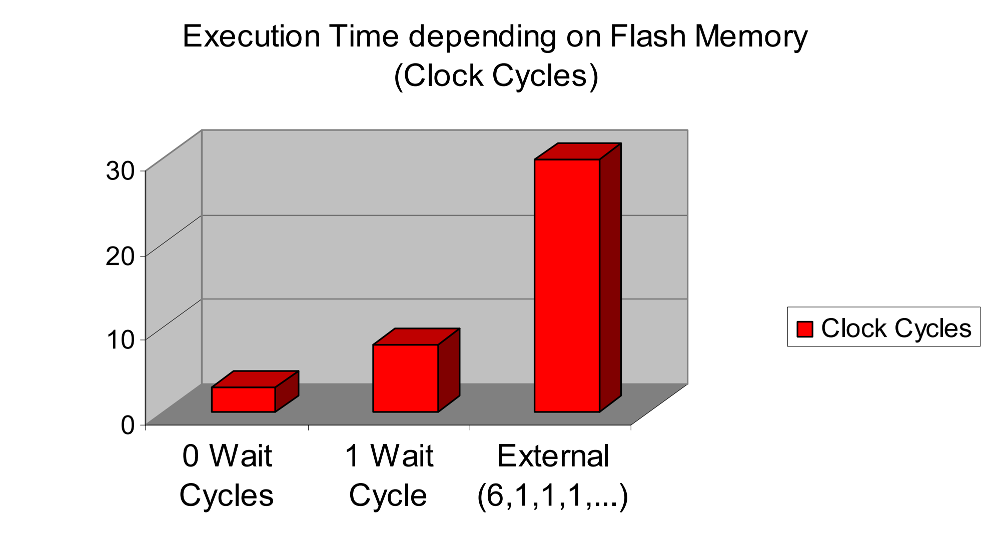
The threat:
Over-estimation by a factor of 100 ☹

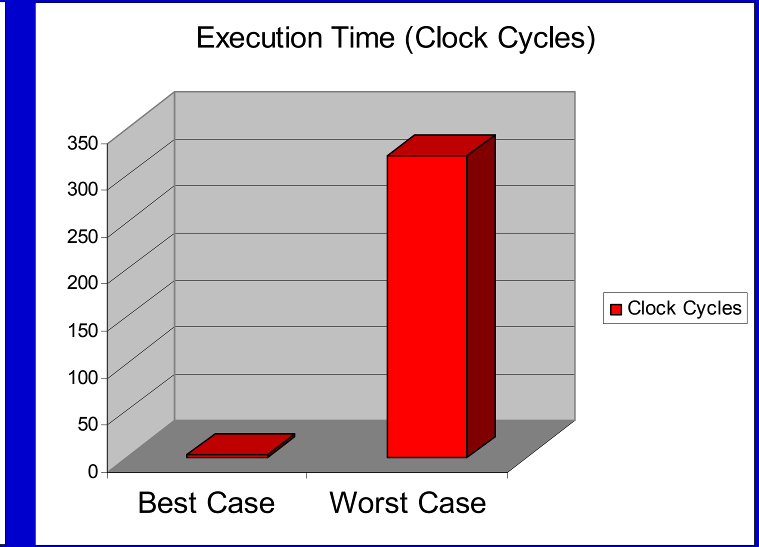
x = a + b; →

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

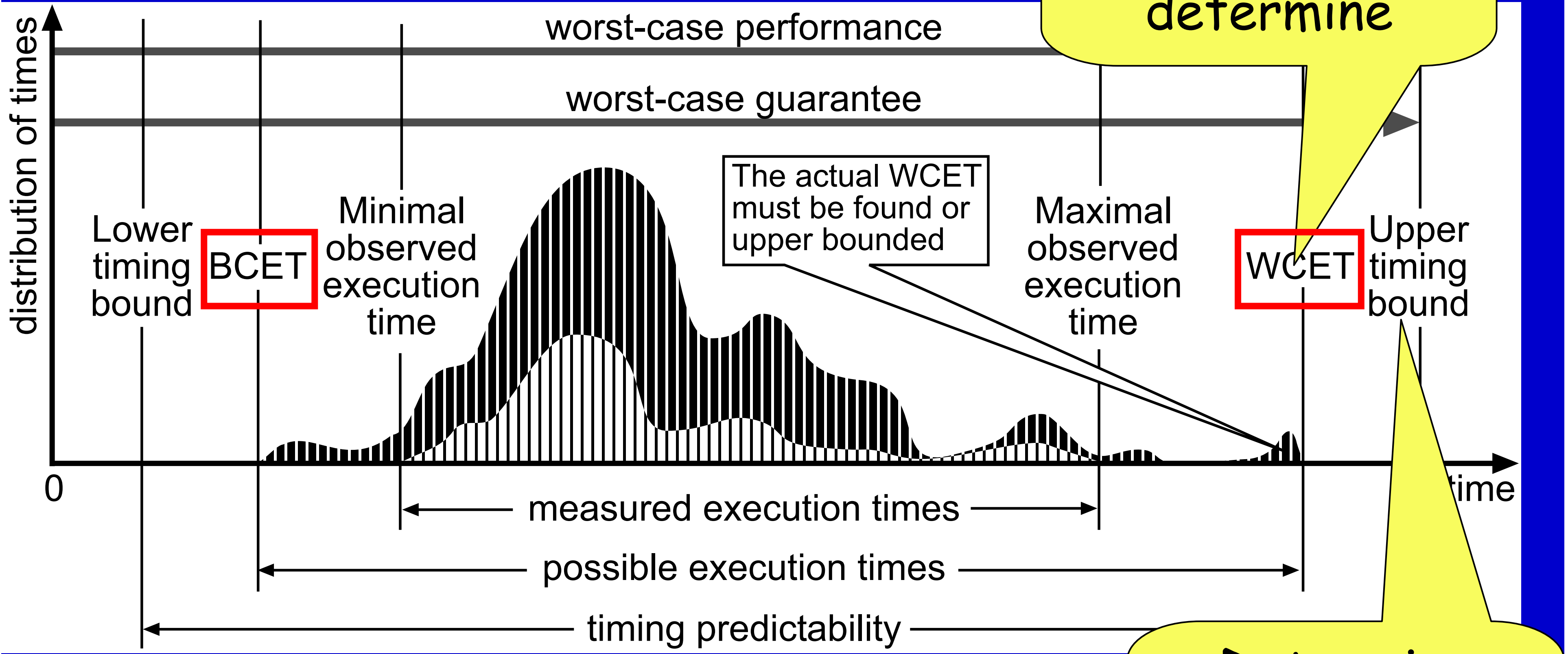
MPC 5xx

PPC 755


Execution Time depending on Flash Memory (Clock Cycles)


Execution Time (Clock Cycles)

# Notions in Timing Analysis



Hard or impossible to determine

Determine upper bounds instead

# Timing Analysis and Timing Predictability

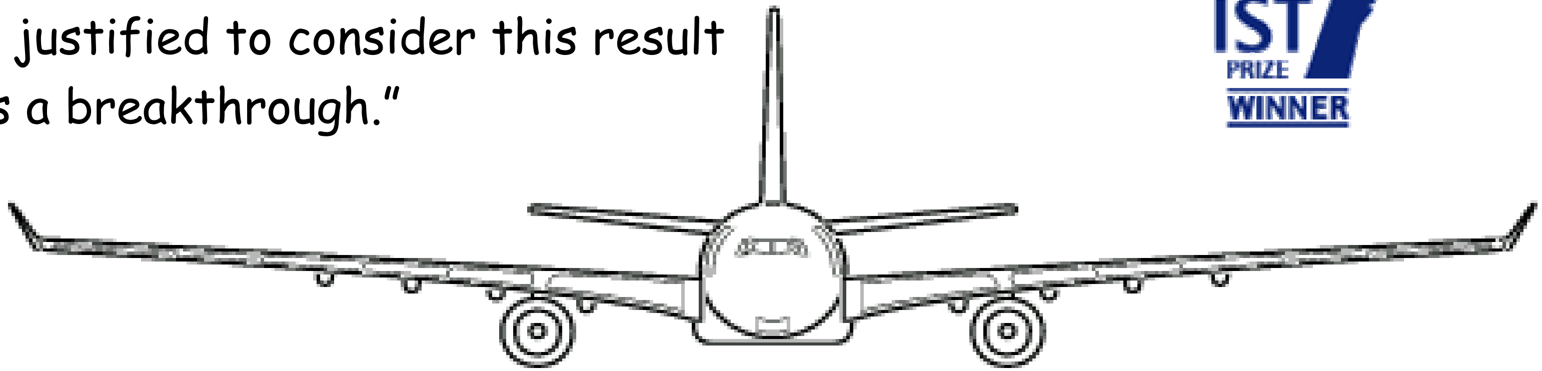
- Timing Analysis derives upper (and maybe lower) bounds
- Timing Predictability of a HW/SW system is the degree to which bounds can be determined
  - with acceptable precision,
  - with acceptable effort, and
  - with acceptable loss of (average-case) performance.
- The goal (of the Predator project) is to find a good point in this 3-dimensional space.

# Timing Analysis
# A success story for formal methods!

# aiT WCET Analyzer

IST Project DAEDALUS final review report:

"The AbsInt tool is probably the best of its kind in the world and it is justified to consider this result as a breakthrough."

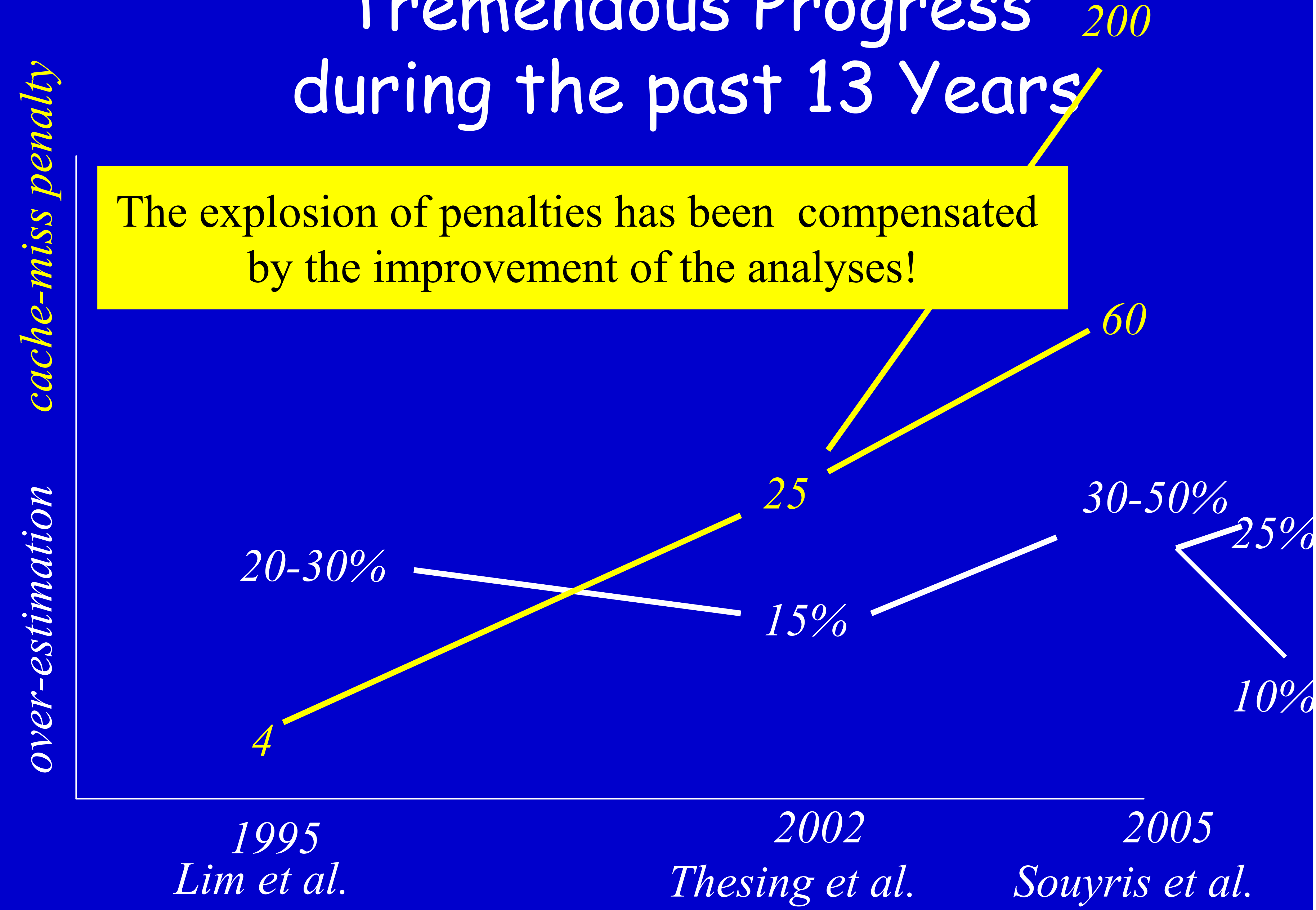


THE EUROPEAN IST PRIZE WINNER

Several time-critical subsystems of the Airbus A380 have been certified using aiT;

aiT is the only validated tool for these applications.

# Tremendous Progress during the past 13 Years



The explosion of penalties has been compensated by the improvement of the analyses!

# High-Level Requirements for Timing Analysis

- Upper bounds must be safe, i.e. not underestimated
- Upper bounds should be tight, i.e. not far away from real execution times
- Analogous for lower bounds
- Analysis effort must be tolerable

Note: all analyzed programs are terminating,
loop bounds need to be known ⇒
no decidability problem, but a complexity problem!

# Our Approach

- End-to-end measurement is not possible because of the large state space.
- We compute bounds for the execution times of instructions and basic blocks and determine a longest path in the basic-block graph of the program.
- The variability of execution times
  - may cancel out in end-to-end measurements, but that's hard to quantify,
  - exists "in pure form" on the instruction level.

# 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

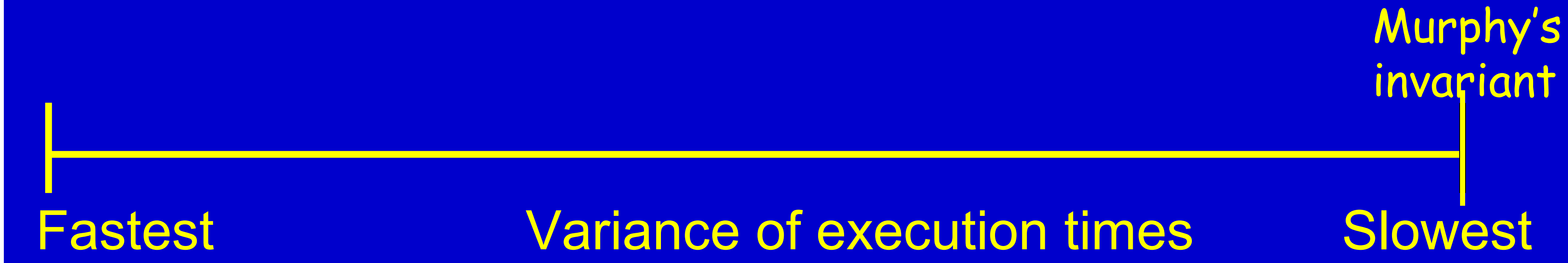
# Execution Time is History-Sensitive

Contribution of the execution of an instruction to a program's execution time

- depends on the execution state, e.g. the time for a memory access depends on the cache state
- the execution state depends on the execution history
- needed: an invariant about the set of execution states produced by all executions reaching a program point.
- We use abstract interpretation to compute these invariants.

# Deriving Run-Time Guarantees

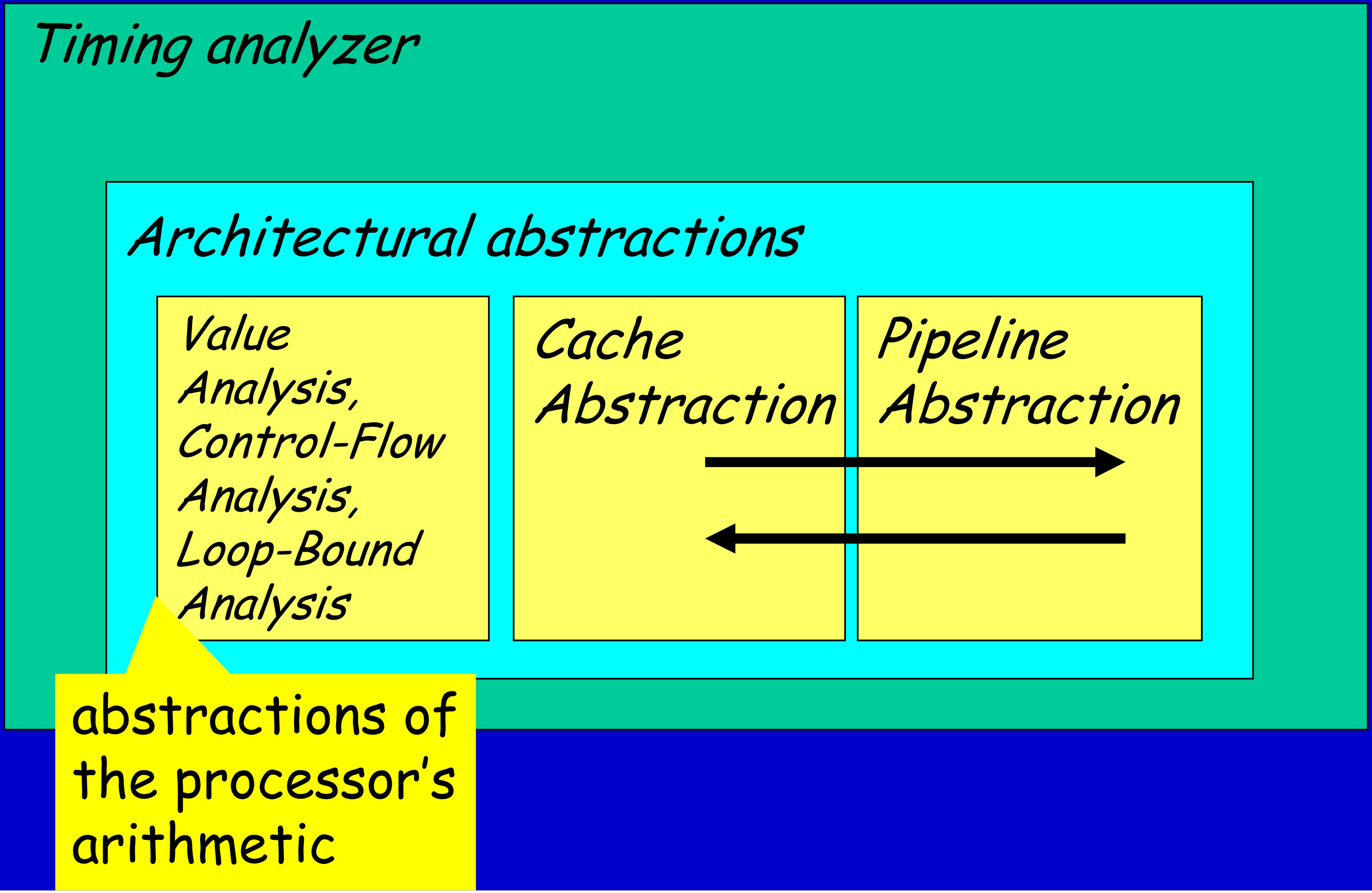
- Our method and tool, aiT, derives Safety Properties from these invariants : Certain timing accidents will never happen. Example: At program point p, instruction fetch will never cause a cache miss.
- The more accidents **excluded**, the **lower** the **upper** bound.



# Abstract Interpretation in Timing Analysis

- Abstract interpretation is always based on the semantics of the analyzed language.
- A semantics of a programming language that talks about time needs to incorporate the execution platform!
- Static timing analysis is thus based on such a semantics.

# The Architectural Abstraction inside the Timing Analyzer


Timing analyzer
Architectural abstractions
Value Analysis, Control-Flow Analysis, Loop-Bound Analysis
Cache Abstraction
Pipeline Abstraction

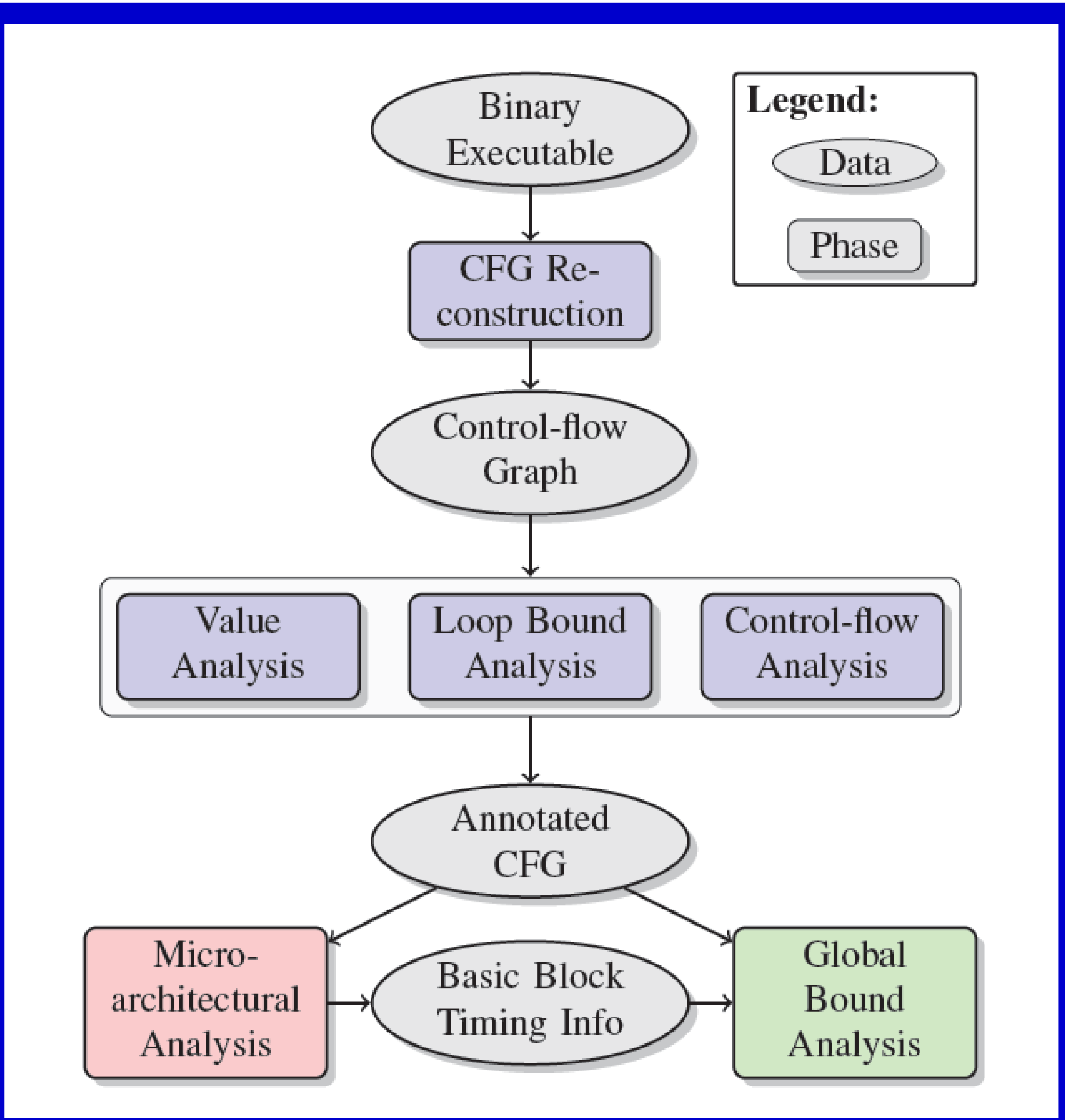
abstractions of the processor's arithmetic

# Abstract Interpretation in Timing Analysis

Determines

- invariants about the values of variables (in registers, on the stack)
  - to compute loop bounds
  - to eliminate infeasible paths
  - to determine effective memory addresses
- invariants on architectural execution state
  - Cache contents ⇒ predict hits & misses
  - Pipeline states ⇒ predict or exclude pipeline stalls

# Tool Architecture

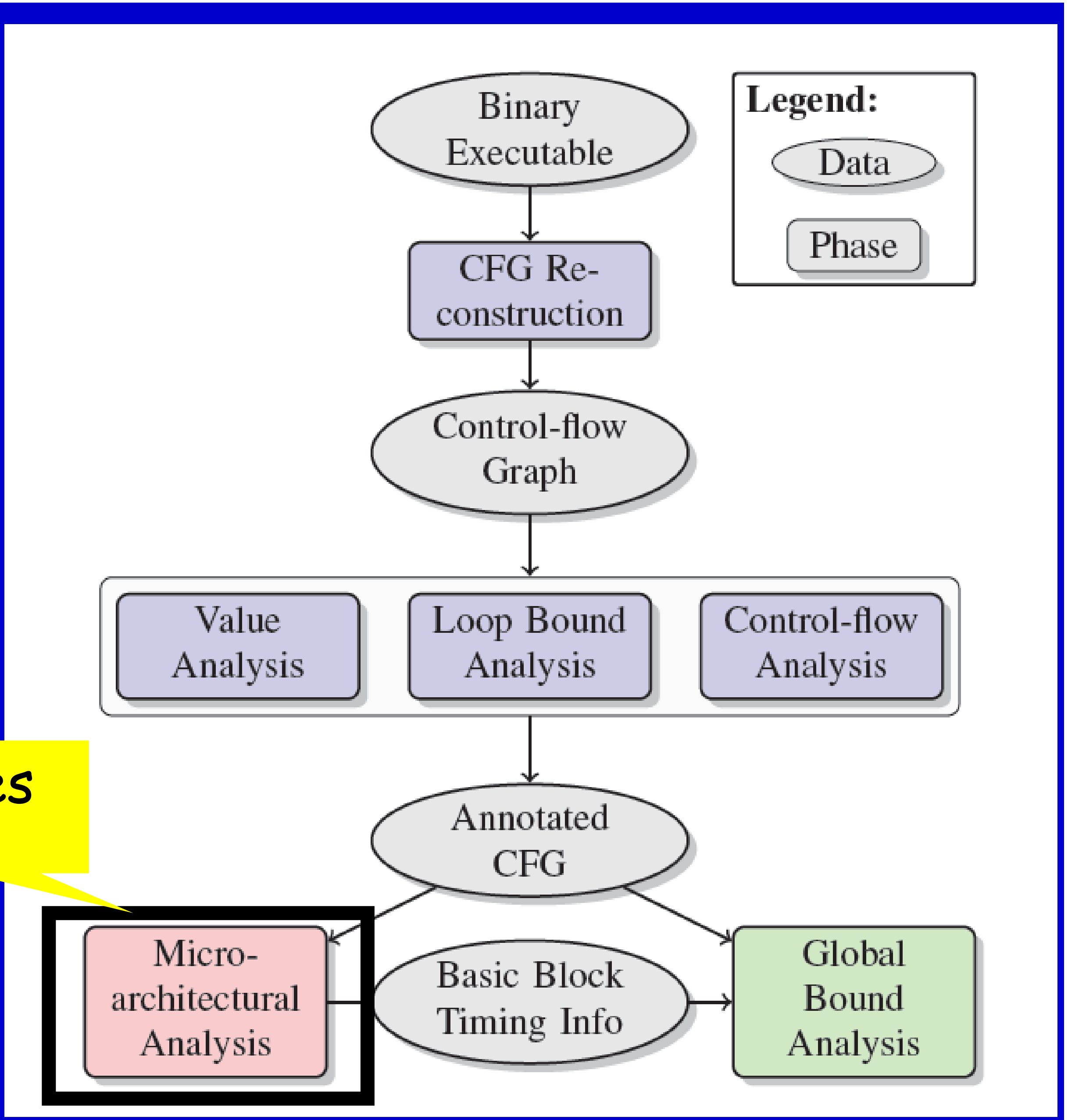


*Abstract Interpretations*

*Abstract Interpretation*

*Integer Linear Programming*

# Tool Architecture



Abstract Interpretations

Caches

Abstract Interpretation

Integer Linear Programming

# Caches:
# Small & Fast Memory on Chip

- Bridge speed gap between CPU and RAM
- Caches work well in the average case:
  - Programs access data locally (many hits)
  - Programs reuse items (instructions, data)
  - *Access patterns are distributed evenly across the cache*
- Cache performance has a strong influence on system performance!

# Caches vs. Scratchpads – an Undecided Battle

- Caches are energy hungry,
+ some cache architectures are nicely predictable.

The alternative are compiler-managed scratchpads,

+ scratchpads are economical wrt. energy,
- they need to be explicitly saved and loaded,
- they do not perform well under preemptive scheduling schemes and in interrupt-driven systems.

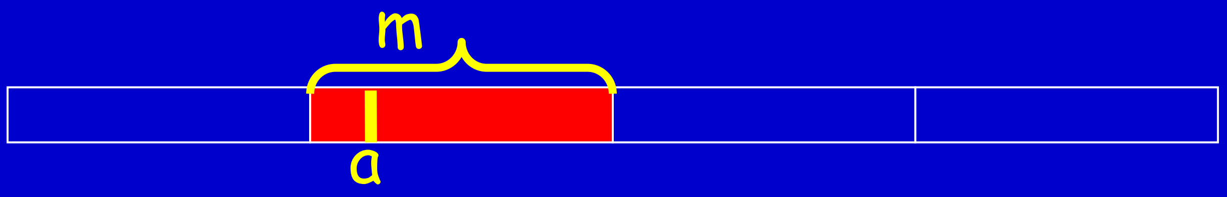
Some architects avoid caches because they don't know how to analyze the behavior.

# Caches: How they work

CPU: read/write at memory address *a*,

- sends a request for *a* to bus

Cases:



- Hit:
  - Block *m* containing *a* in the cache: request served in the next cycle
- Miss:
  - Block *m* not in the cache:
    *m* is transferred from main memory to the cache,
    *m* may replace some block in the cache,
    request for *a* is served asap while transfer still continues

# Replacement Strategies

- Several replacement strategies:

  LRU, PLRU, FIFO,...

  determine which line to replace when a memory block is to be loaded into a full cache (set)

# LRU Strategy

- Each cache set has its own replacement logic => Cache sets are independent: Everything explained in terms of one set
- LRU-Replacement Strategy:
  - Replace the block that has been Least Recently Used
  - Modeled by Ages
- Example: 4-way set associative cache

| age | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| | $m_0$ | $m_1$ | $m_2$ | $m_3$ |
| Access $m_4$ (miss) | $m_4$ | $m_0$ | $m_1$ | $m_2$ |
| Access $m_1$ (hit) | $m_1$ | $m_4$ | $m_0$ | $m_2$ |
| Access $m_5$ (miss) | $m_5$ | $m_1$ | $m_4$ | $m_0$ |

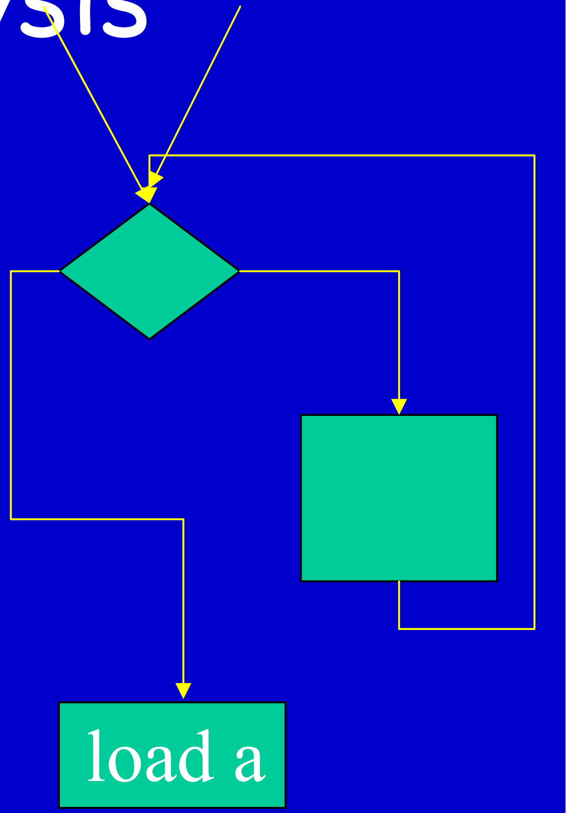
# Cache Analysis

How to statically precompute cache contents:

- Must Analysis:
  For each program point (and context), find out which blocks are in the cache → prediction of cache hits
- May Analysis:
  For each program point (and context), find out which blocks may be in the cache
  Complement says what is not in the cache → prediction of cache misses
- In the following, we consider must analysis until otherwise stated.

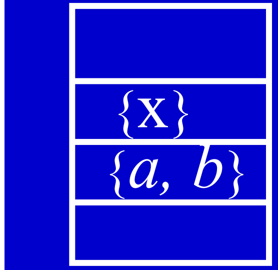
# (Must) Cache Analysis

- Consider one instruction in the program.
- There may be many paths leading to this instruction.
- How can we compute whether *a* will always be in cache independently of which path execution takes?



Question:
Is the access to a always a cache hit?

# Determine Cache-Information (abstract cache states) at each Program Point



youngest age - 0

oldest age - 3

Interpretation of this cache information: describes the set of all concrete cache states in which x, a, and b occur

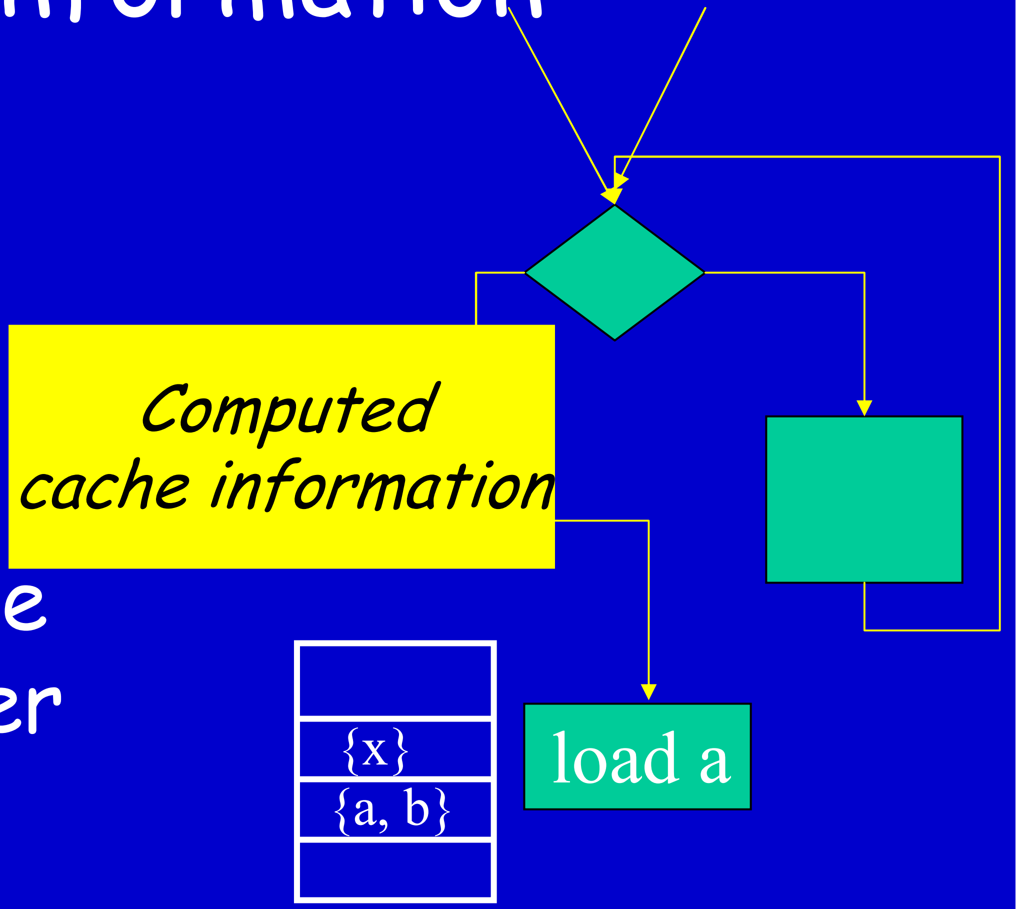
- x with an age not older than 1
- a and b with an age not older than 2,

Cache information contains

1. only memory blocks guaranteed to be in cache.
2. they are associated with their maximal age.

# Cache- Information

Cache analysis determines safe information about Cache Hits. Each predicted Cache Hit reduces the upper bound by the cache-miss penalty.



*Computed cache information*

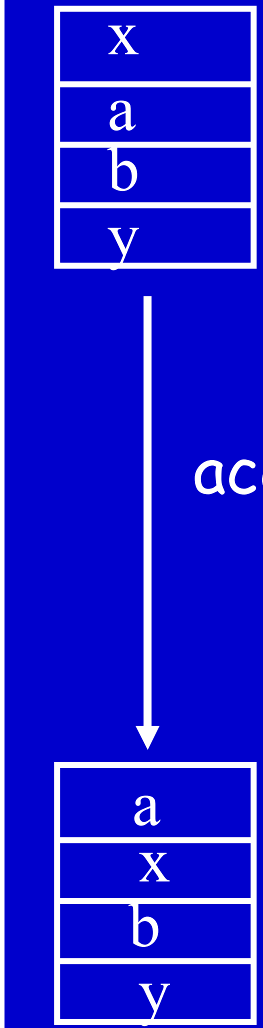
*Access to* a *is a cache hit; assume 1 cycle access time.*

# Cache Analysis – how does it work?

- How to compute for each program point an abstract cache state representing a set of memory blocks guaranteed to be in cache each time execution reaches this program point?
- Can we expect to compute the largest set?
- Trade-off between precision and efficiency – quite typical for abstract interpretation

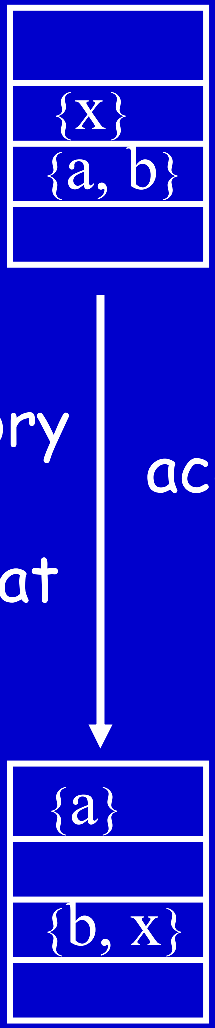
# (Must) Cache analysis of a memory access

concrete transfer function (cache)



access to a

abstract transfer function (analysis)



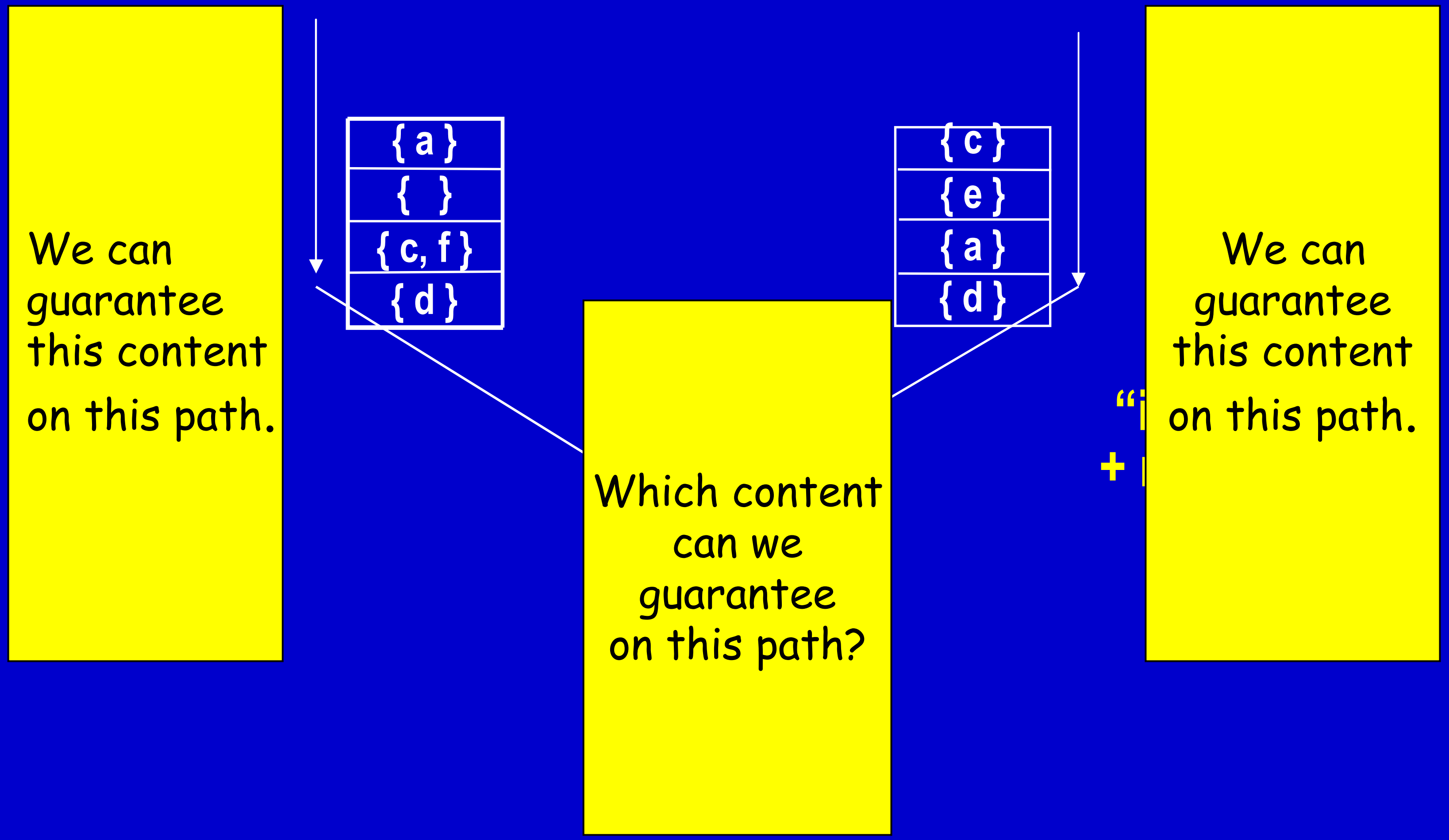
access to a

After the access to a,
a is the youngest memory block in cache,
and we must assume that x has aged.
What about b?

# Combining Cache Information

- Consider two control-flow paths to a program point:
  - for one, prediction says, set of memory blocks S1 in cache,
  - for the other, the set of memory blocks S2.
  - Cache analysis should not predict more than S1 $\cap$ S2 after the merge of paths.
  - the elements in the intersection should have their maximal age from S1 and S2.
- Suggests the following method: Compute cache information along all paths to a program point and calculate their intersection – but too many paths!
- More efficient method:
  - combine cache information on the way,
  - iterate until least fixpoint is reached.
- There is a risk of losing precision, not in case of distributive transfer functions.

# What happens when control-paths merge?



We can guarantee this content on this path.

| |
|---|
| { a } |
| { } |
| { c, f } |
| { d } |

| |
|---|
| { c } |
| { e } |
| { a } |
| { d } |

We can guarantee this content on this path.

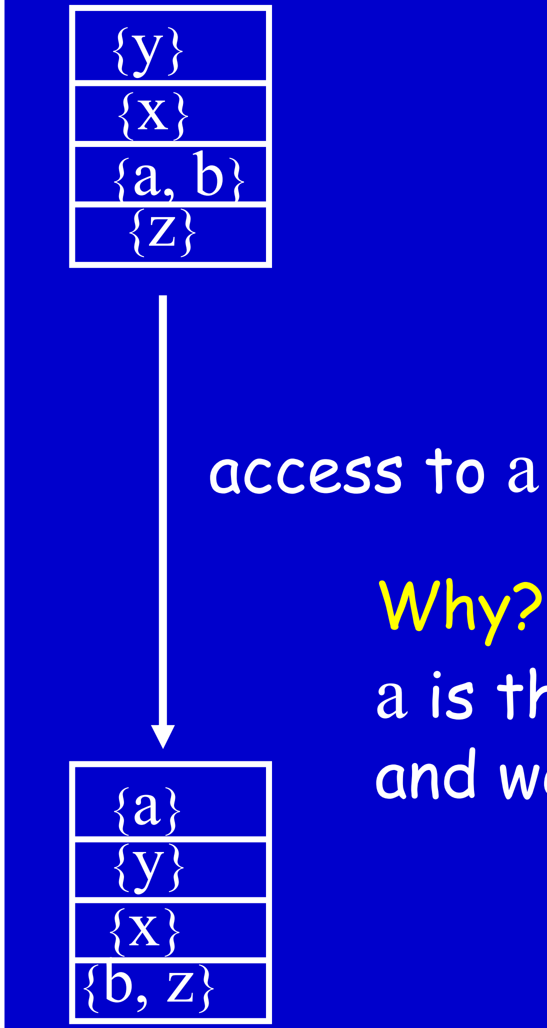
Which content can we guarantee on this path?

combine cache information at each control-flow merge point

# Must-Cache and May-Cache- Information

- The presented cache analysis is a Must Analysis. It determines safe information about cache hits.
  Each predicted cache hit reduces the upper bound.
- We can also perform a May Analysis. It determines safe information about cache misses
  Each predicted cache miss increases the lower bound.

# (May) Cache analysis of a memory access



{y}

{x}

{a, b}

{z}

access to a

{a}

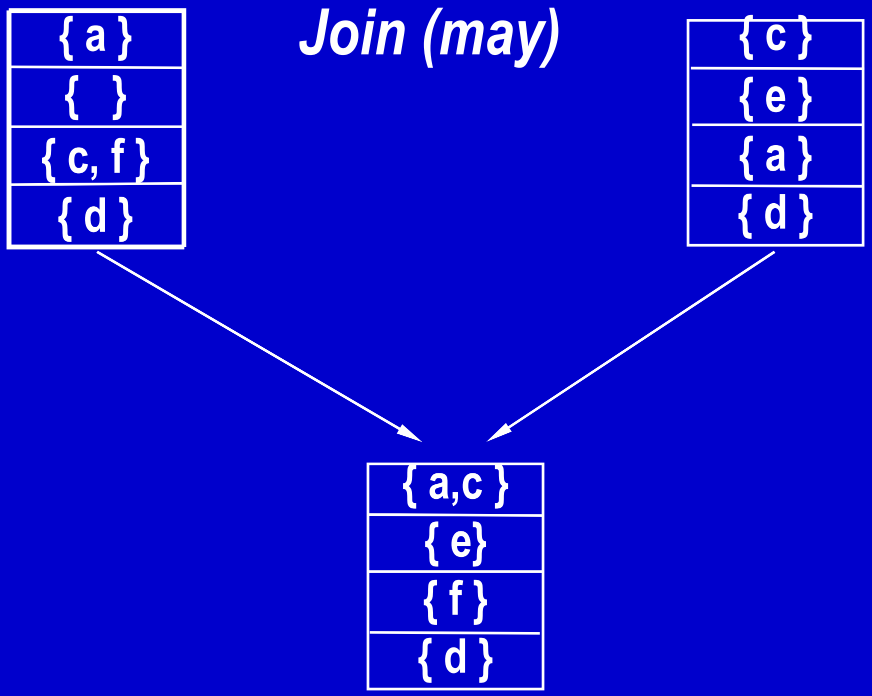
{y}

{x}

{b, z}

Why? After the access to a
a is the youngest memory block in cache,
and we must assume that x, y and b have aged.

# Cache Analysis: Join (may)



“union
+ minimal age”

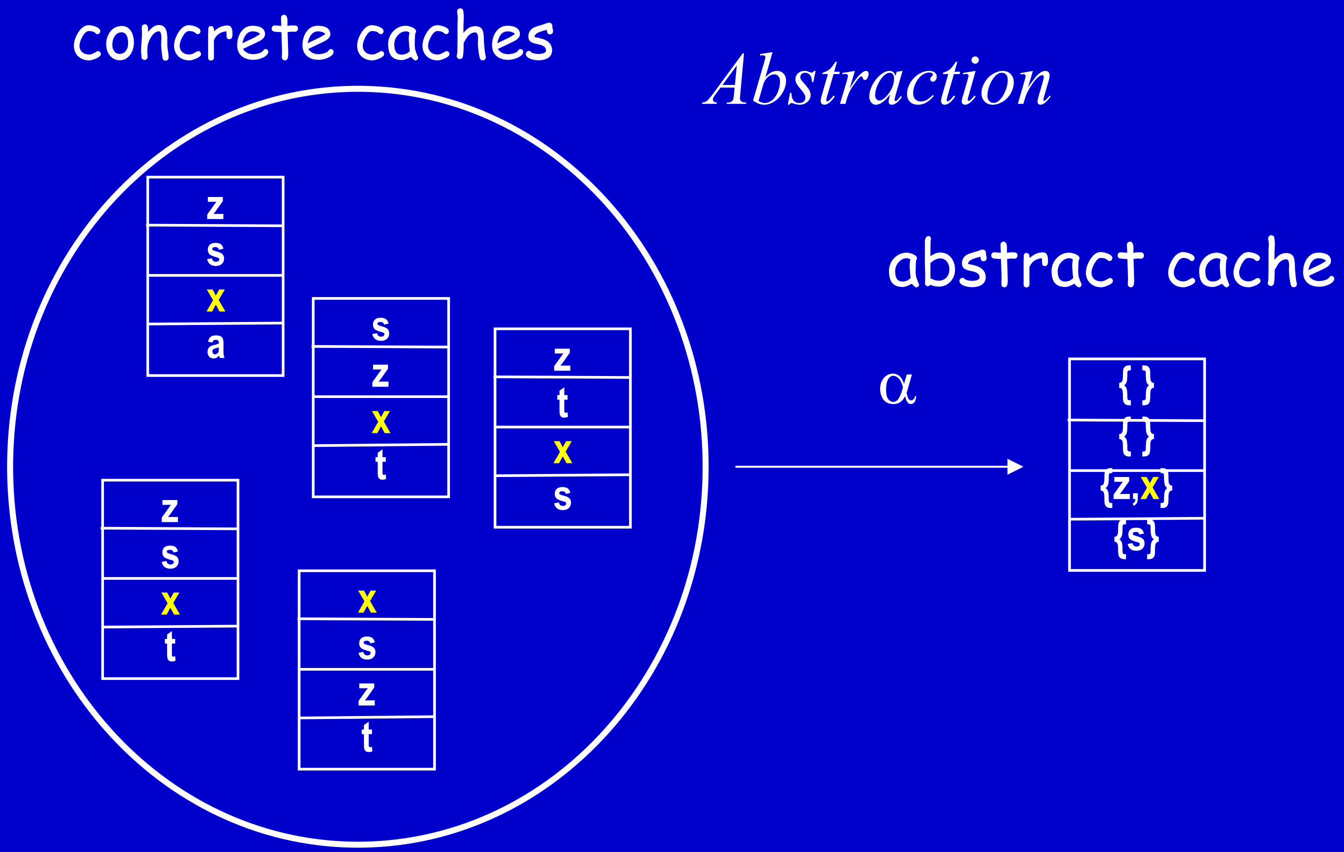
# Result of the Cache Analyses

## Categorization of memory references

| Category | Abb. | Meaning |
|---|---|---|
| always hit | **ah** | The memory reference will always result in a cache hit. |
| always miss | **am** | The memory reference will always result in a cache miss. |
| not classified | **nc** | The memory reference could neither be classified as **ah** nor **am**. |

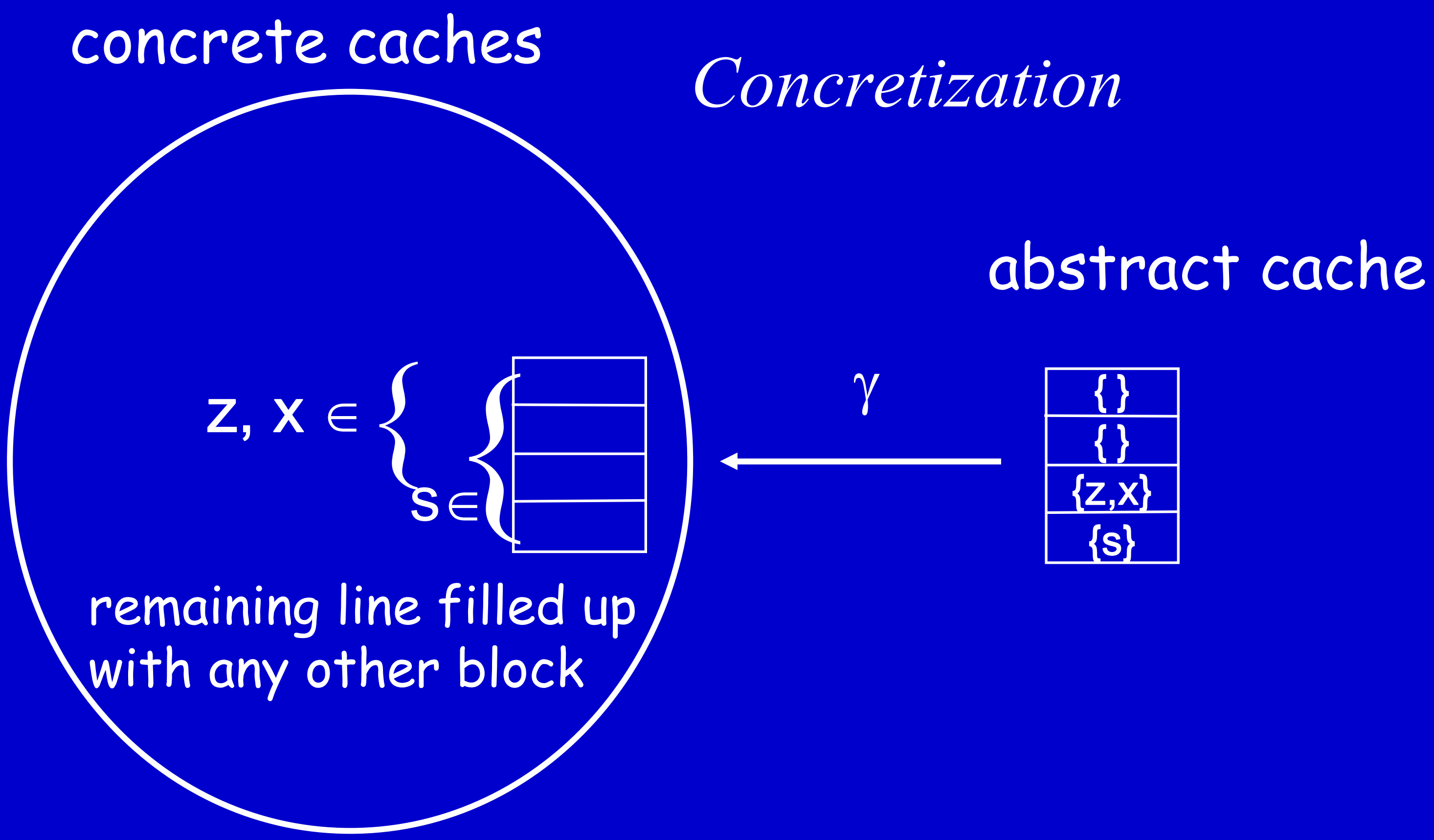
# Abstract Domain: Must Cache

Representing sets of concrete caches by their description



# Abstract Domain: Must Cache

Sets of concrete caches described by an abstract cache



concrete caches

*Concretization*

abstract cache

z, x ∈ { s ∈

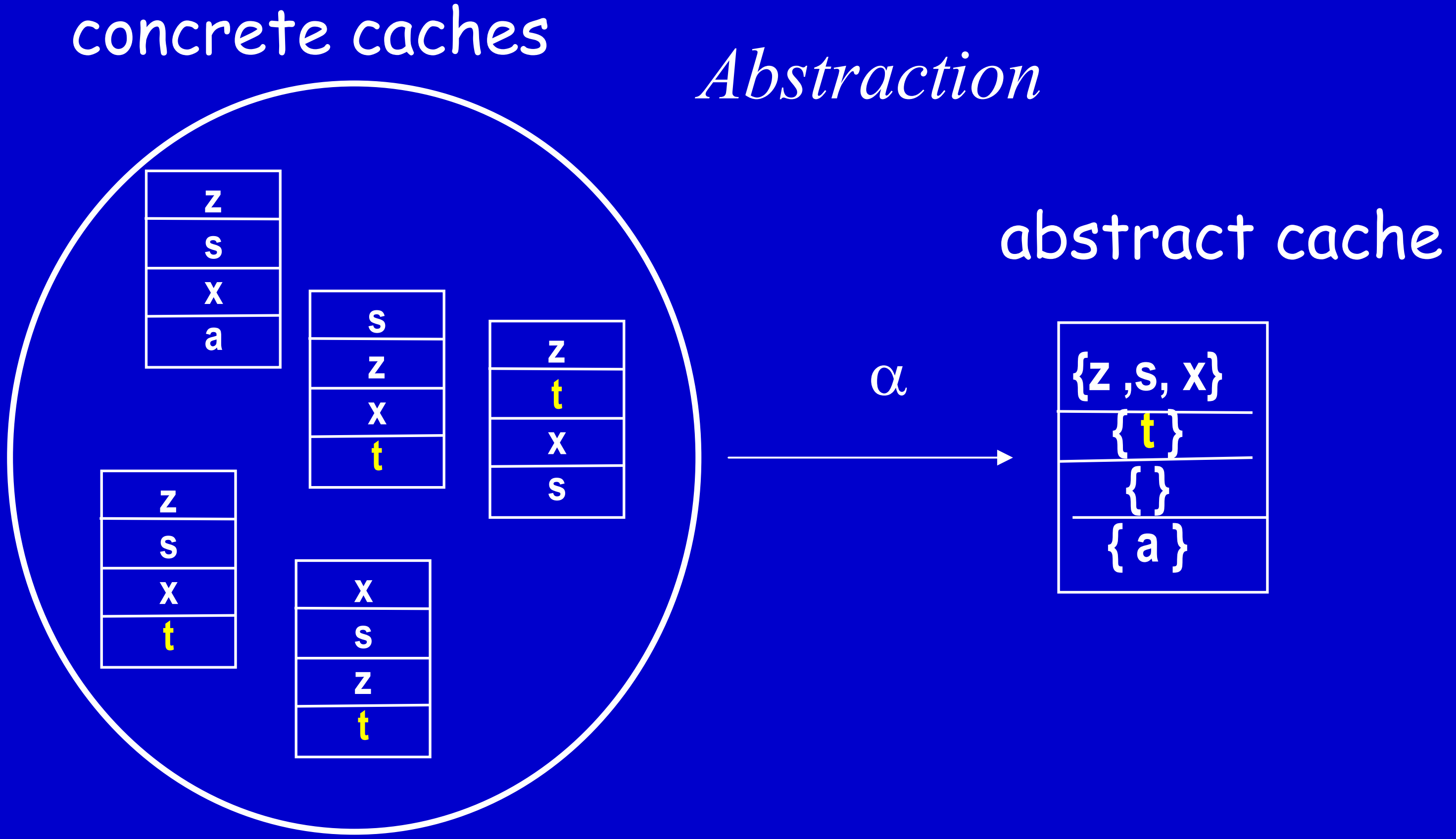
remaining line filled up with any other block

$\gamma$

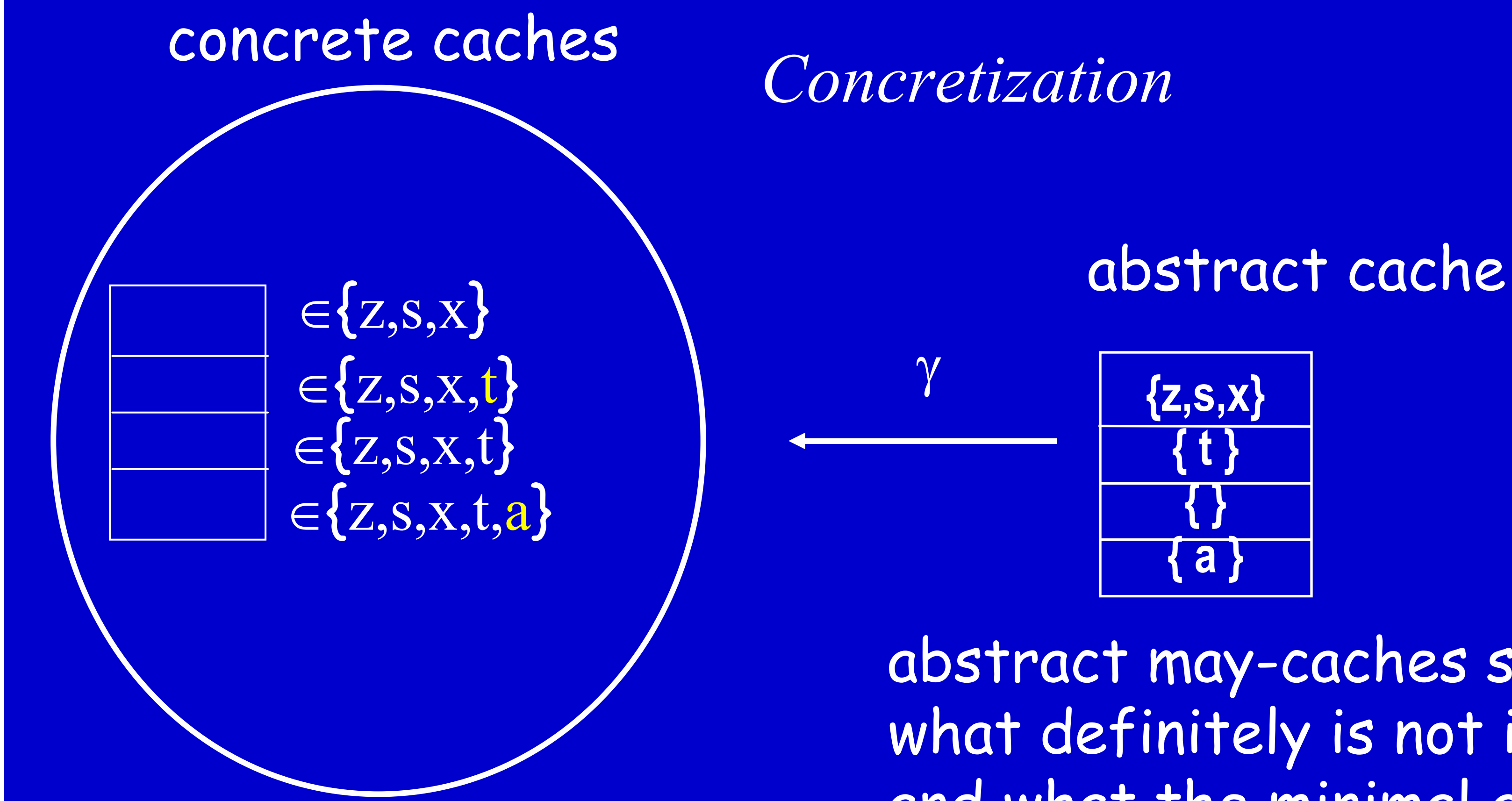
| |
|---|
| { } |
| { } |
| {z,x} |
| {s} |

over-approximation!

# Abstract Domain: May Cache



# Abstract Domain: May Cache



concrete caches

*Concretization*

$\in\{z,s,x\}$

$\in\{z,s,x,t\}$

$\in\{z,s,x,t\}$

$\in\{z,s,x,t,a\}$

$\gamma$

abstract cache

{z,s,x}

{ t }

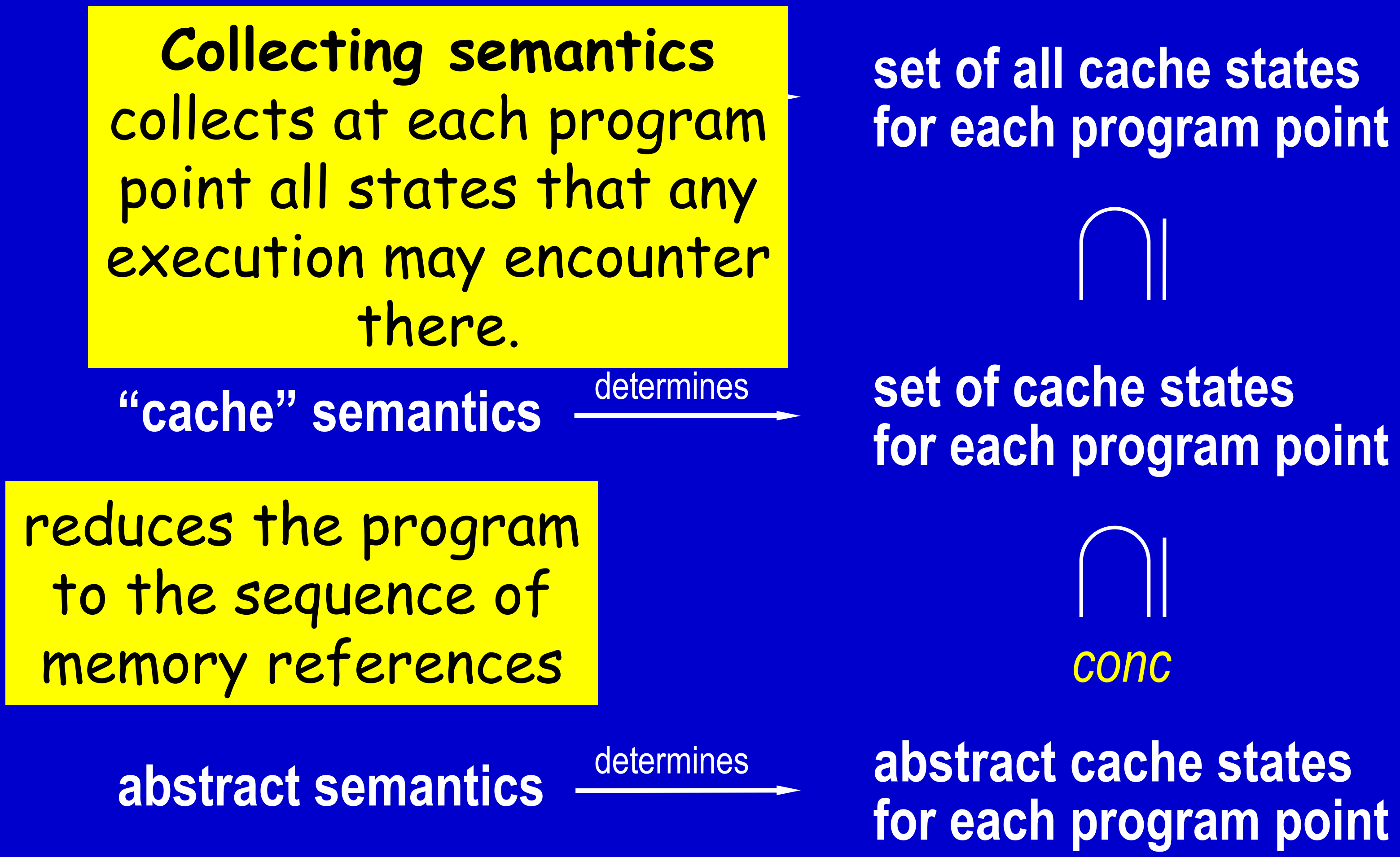
{ }

{ a }

abstract may-caches say what definitely is not in cache and what the minimal age of those is that may be in cache.

# Cache Analysis

Over-approximation of the Collecting Semantics



**Collecting semantics** collects at each program point all states that any execution may encounter there.

set of all cache states for each program point

"cache" semantics

reduces the program to the sequence of memory references

set of cache states for each program point

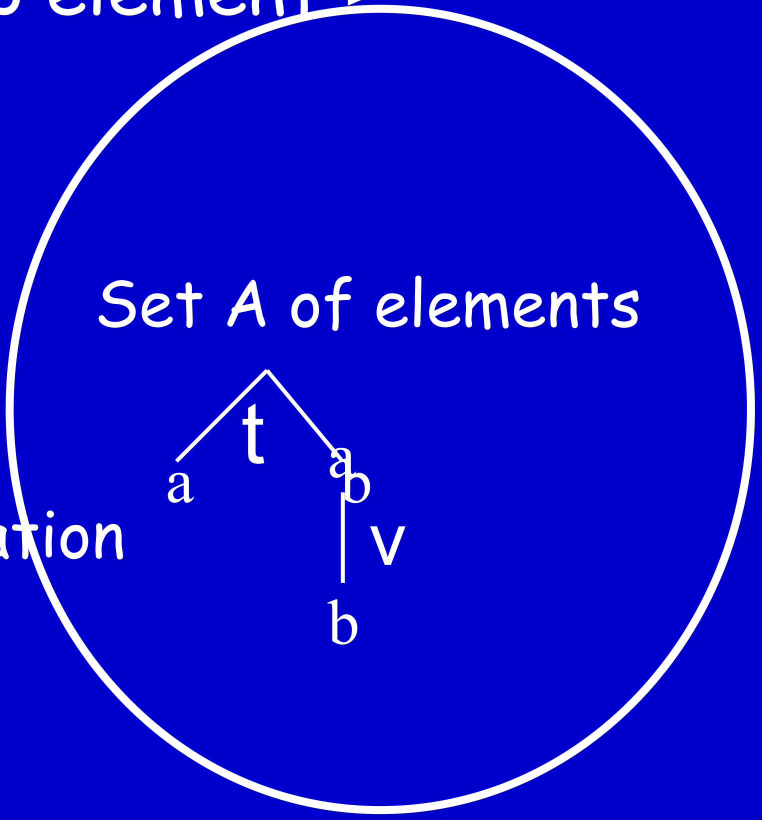
abstract semantics

abstract cache states for each program point

# Complete Lattices:
# The Mathematics of Semantic Domains

(A, v, t, u, >, ?)

Top element >



Relation between t and v:

a v b iff a t b = b

Join operator t combines information

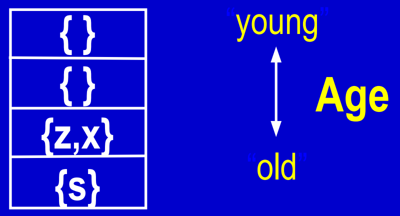
Information order v

Convention: b more precise than a

Bottom element ?

# Lattice for Must Cache

- Set A of elements
- Information order v
- Join operator t
- Top element >
- Bottom element ?



{ }<br>
{ }<br>
{z,x}<br>
{s}

young

Age

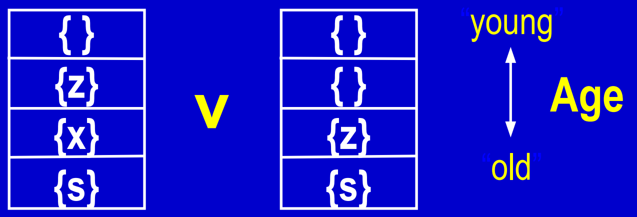
old

**Abstract cache states:**

**Upper bounds on the age of memory blocks guaranteed to be in cache**

# Lattice for Must Cache

- Set A of elements
- Information order v
- Join operator t
- Top element >
- Bottom element ?



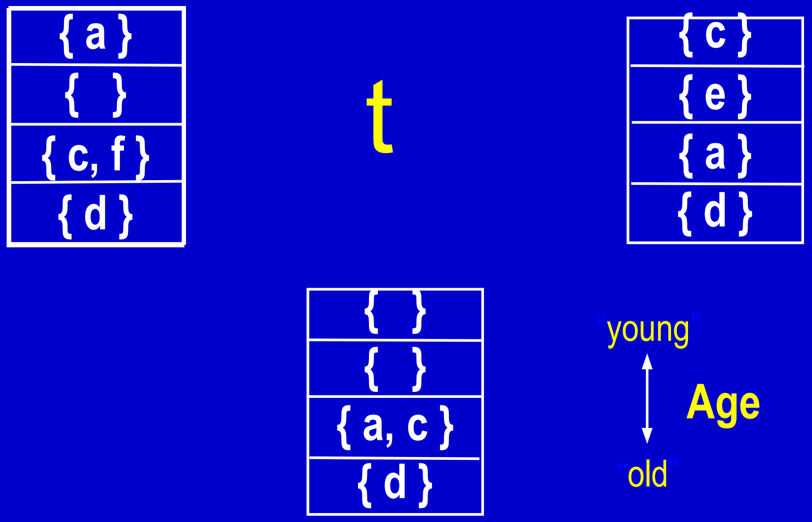
**Better precision:**

**more elements in the cache or with younger age.**

**NB. The more precise abstract cache represents less concrete cache states!**

# Lattice: Must Cache

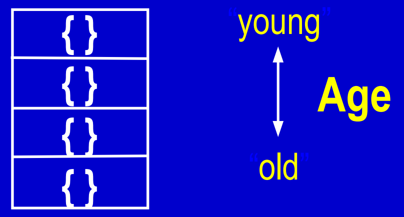
- Set A of elements
- Information order v
- Join operator t
- Top element >
- Bottom element ?



**Form the intersection and associate the elements with the maximum of their ages**

# Lattice: Must Cache

- Set A of elements
- Information order v
- Join operator t
- Top element >
- Bottom element ?



**No information:**

**All caches possible**

# Lattice: Must Cache

- Set A of elements
- Information order v
- Join operator t
- Top element >
- Bottom element ?

**Dedicated unique bottom element representing the empty set of caches**

# Galois connection – Relating Semantic Domains

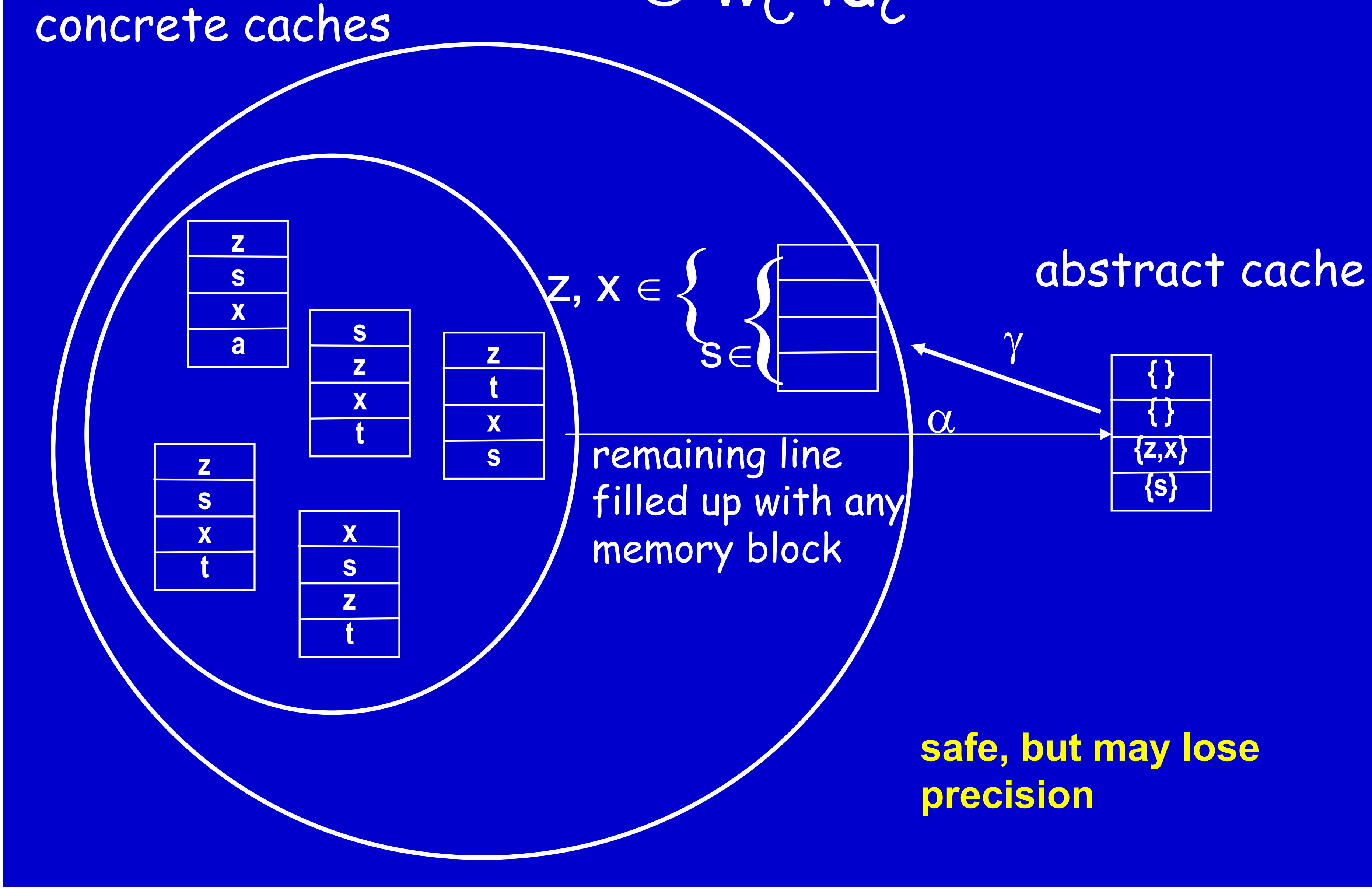
- Lattices *C*, *A*
- two monotone functions ® and °
- Abstraction: ®: *C* → *A*
- Concretization °: *A* → *C*
- (®,°) is a Galois connection if and only if

  ° • ® $\mathrm{w}_{\mathrm{C}}$ $\mathrm{id}_{C}$ and ® • ° $\mathrm{v}_{\mathrm{A}}$ $\mathrm{id}_{A}$

Switching safely between concrete and abstract domains, possibly losing precision

# Abstract Domain Must Cache

° • ® $w_C$ $id_C$



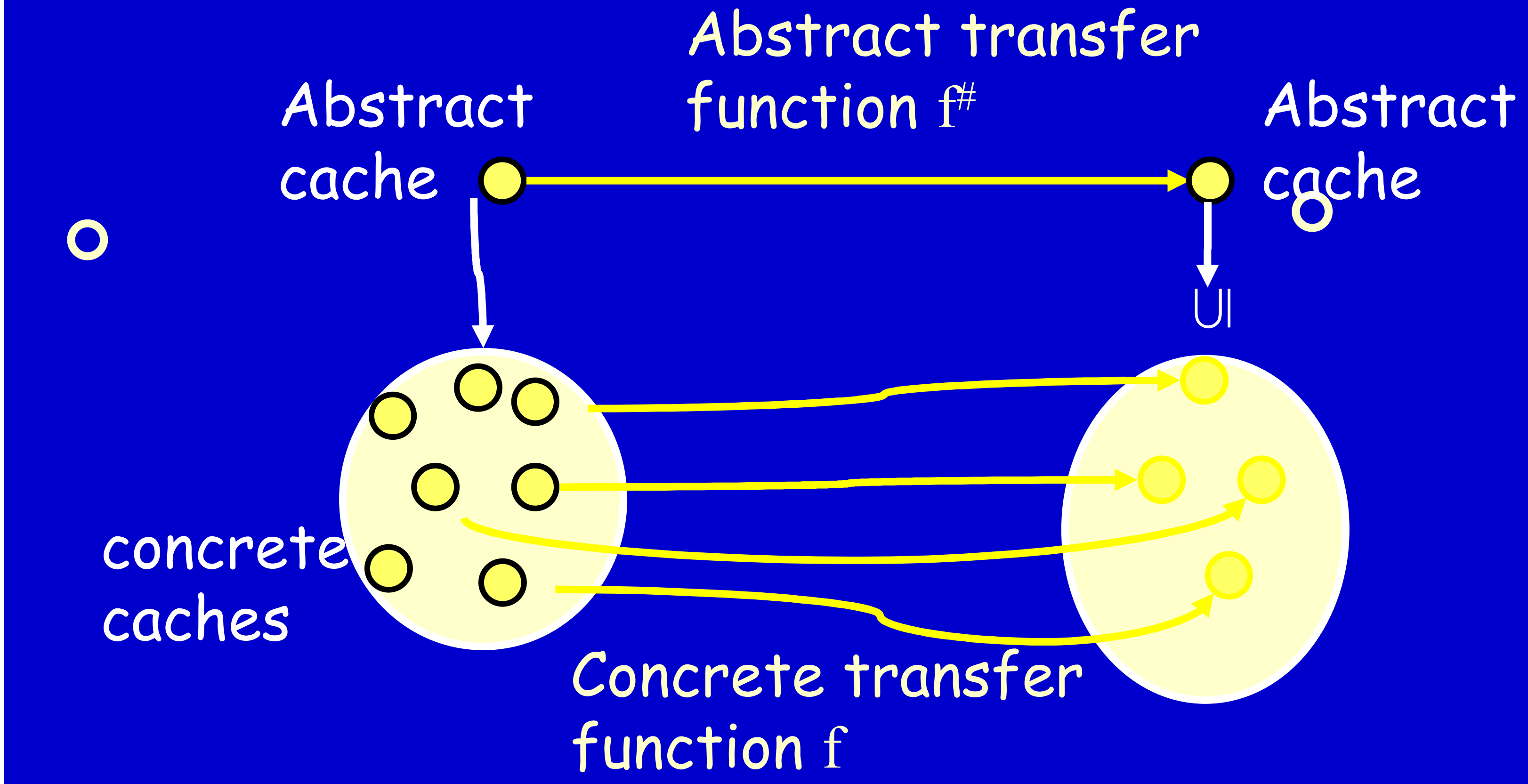
concrete caches

abstract cache

remaining line filled up with any memory block

**safe, but may lose precision**

# Correctness of the Abstract Transformer



Abstract cache

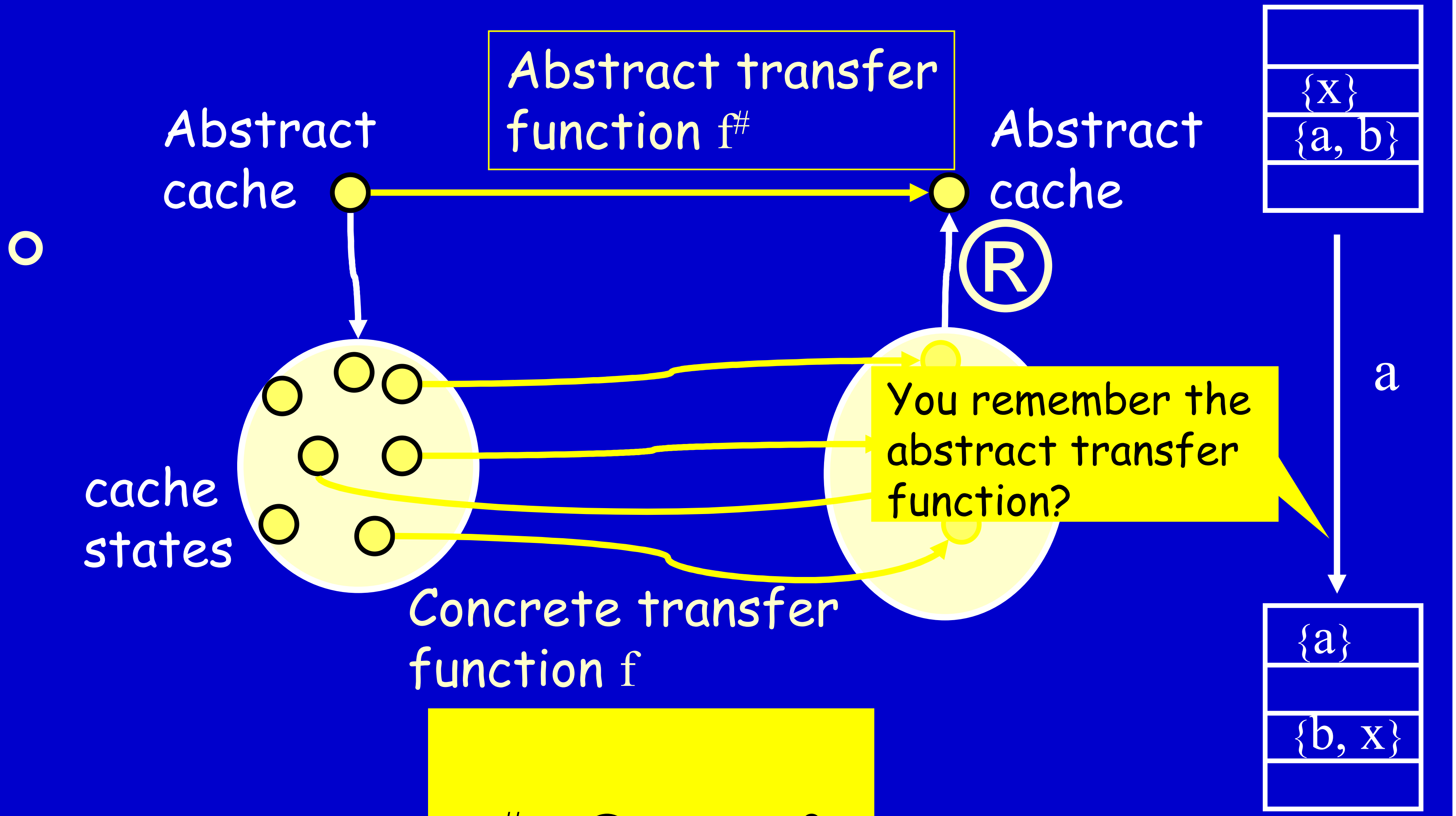
Abstract transfer function $f^{\#}$

Abstract cache

concrete caches

Concrete transfer function $f$

# Semantics II
# Cousot's Best Transformer



Abstract transfer function $f^\#$

Abstract cache

Abstract cache

®

cache states

Concrete transfer function f

You remember the abstract transfer function?

{x}
{a, b}

a

{a}
{b, x}

$f^\# = ® ± f ± °$

# Lessons Learned

- Cache analysis, an important ingredient of static timing analysis, provides for abstract domains,
- which proved to be sufficiently precise,
- have compact representation,
- have efficient transfer functions,
- which are quite natural.

# An Alternative Abstract Cache Semantics: Power set domain of cache states

- Set A of elements - sets of concrete cache states
- Information order v - set inclusion
- Join operator t - set union
- Top element > - the set of all cache states
- Bottom element ? - the empty set of caches

# Power set domain of cache states

- Potentially more precise
- Certainly not similarly efficient
- Sometimes, power-set domains are the only choice you have → pipeline analysis

# Problem Solved?

- We have shown a solution for LRU caches.
- LRU-cache analysis works smoothly
  - Favorable „structure“ of domain
  - Essential information can be summarized compactly
- LRU is the best strategy under several aspects
  - performance, predictability, sensitivity
- … and yet: LRU is not the only strategy
  - Pseudo-LRU (PowerPC 755 @ Airbus)
  - FIFO
  - worse under almost all aspects, but average-case performance!

# Abstract Interpretation – the Ingredients

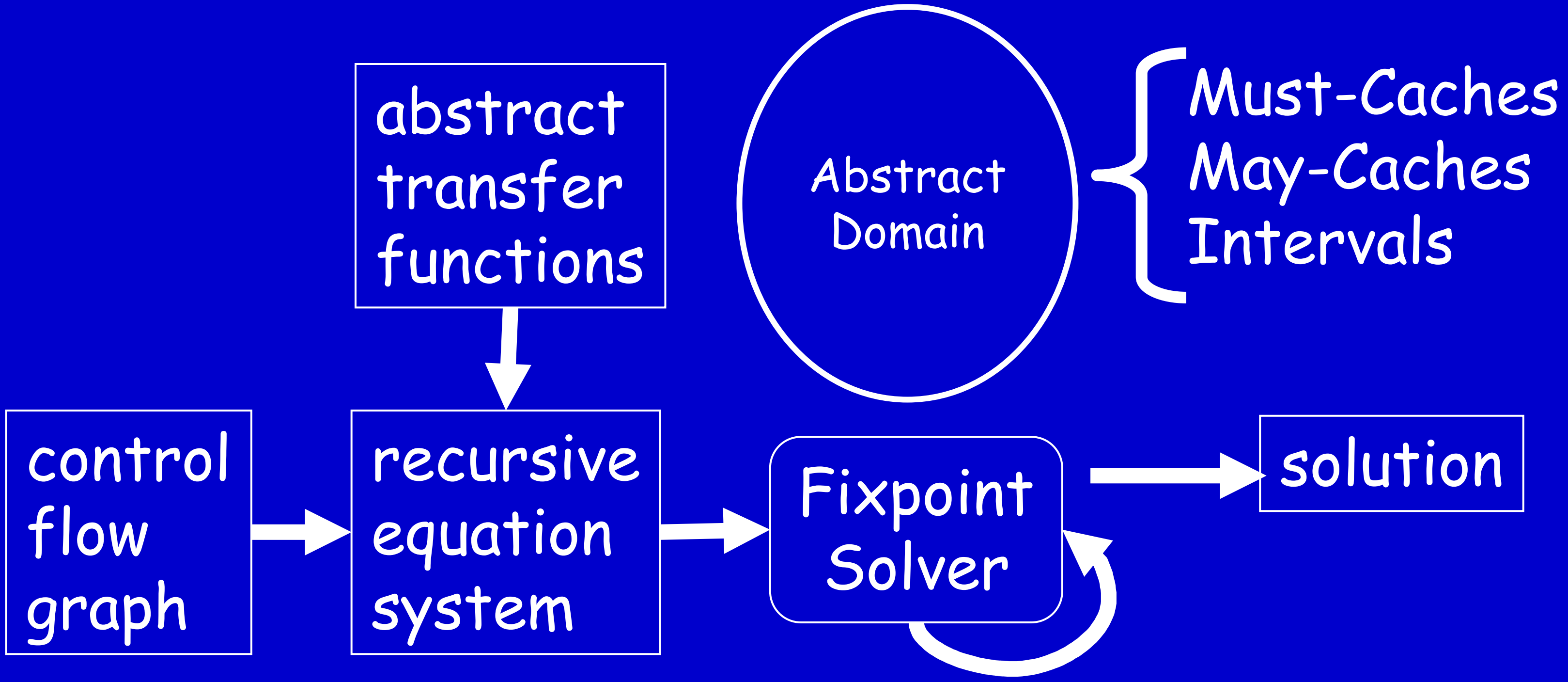
- Abstract domain –
  complete lattice (A, v, t, u, >, ?)
- (monotone) abstract transfer functions for each statement/condition/instruction
- information at program entry points

# Instantiating an Abstract Interpretation

Given control-flow graph of a program with statements/conditions/instructions at edges

- associate abstract transfer function with each edge
- associate lattice join with control-flow merge points
- induces a recursive set of equations

# Solving Static Analysis Problems



abstract transfer functions

Abstract Domain

Must-Caches
May-Caches
Intervals

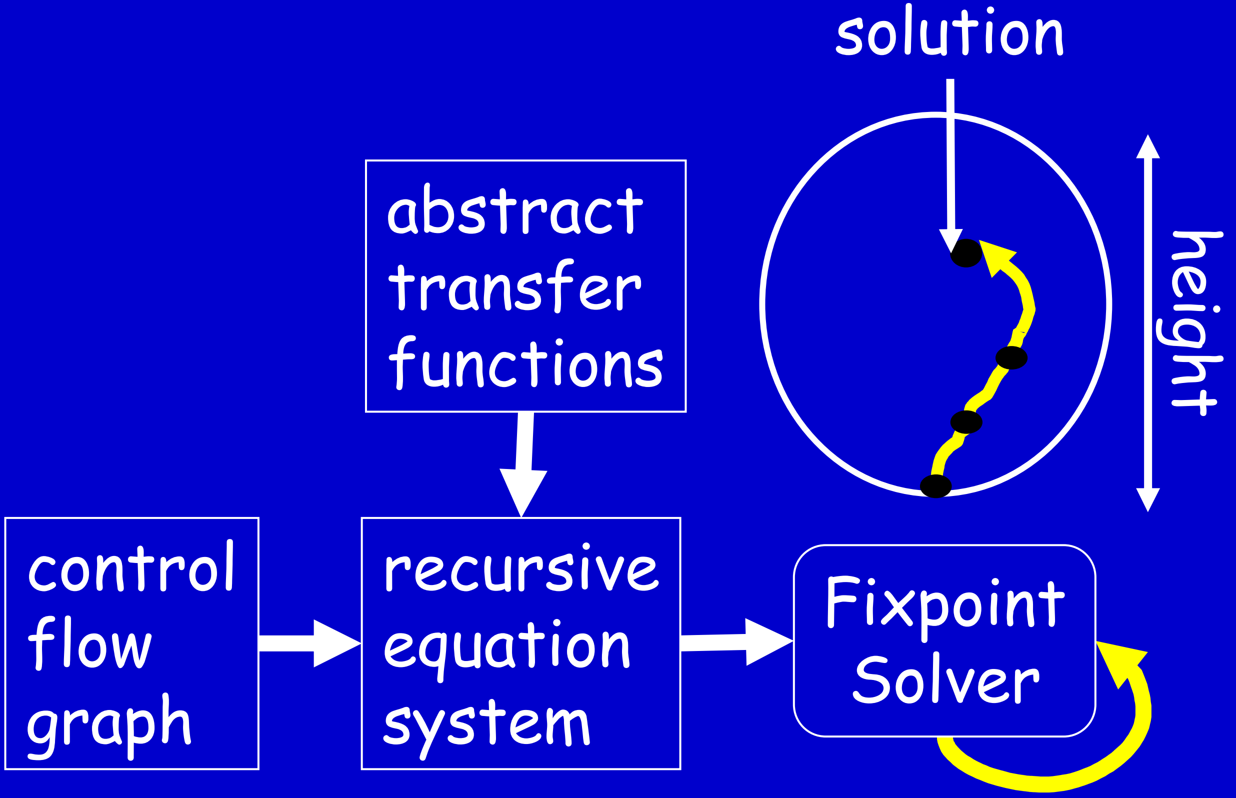
control flow graph

recursive equation system

Fixpoint Solver

solution

# Solving Static Analysis Problems by Fixpoint Iteration



abstract transfer functions

control flow graph

recursive equation system

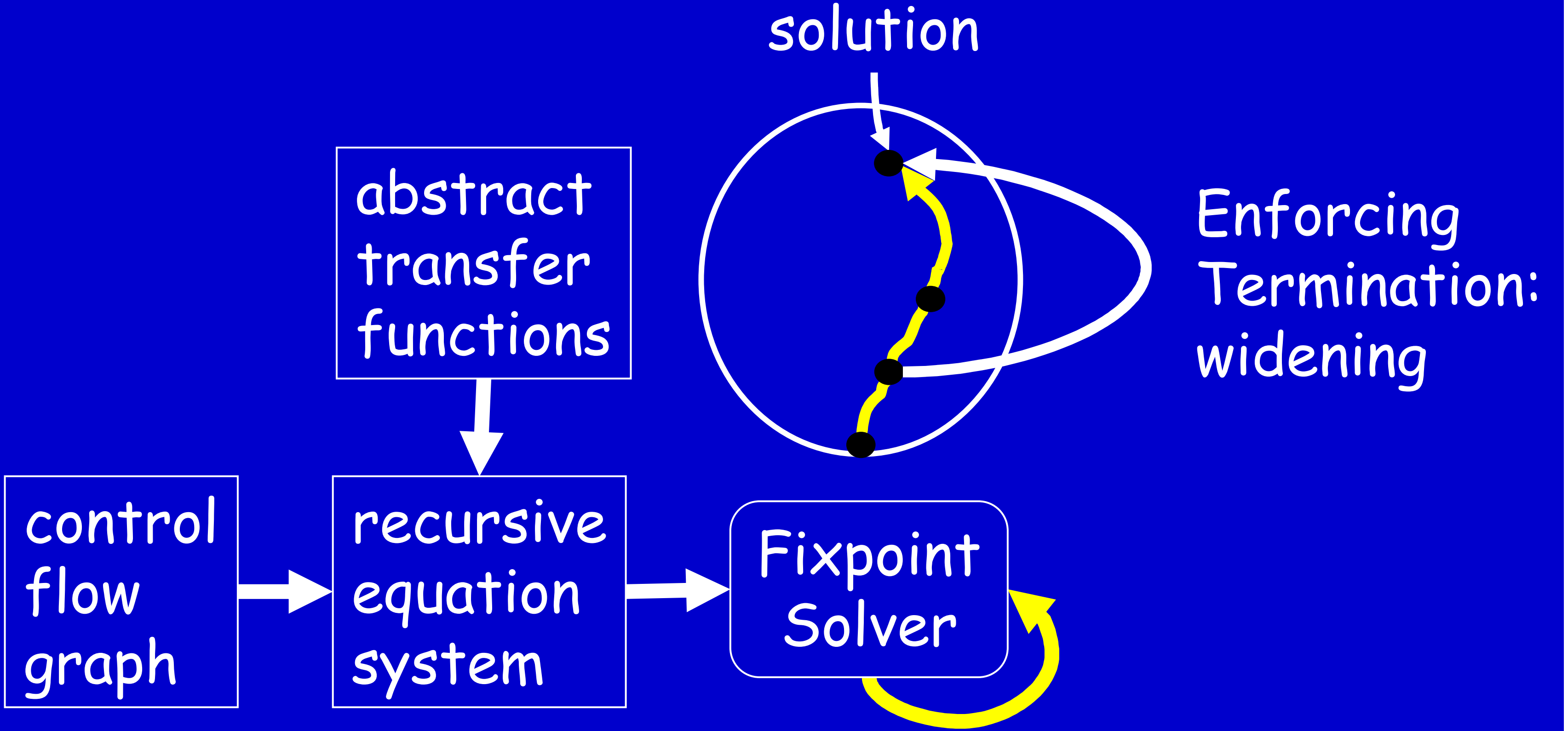
Fixpoint Solver

$X=f(X)$

Ascending Chain Condition:
+ Must-Caches
+ May-Caches
- **Intervals**

Kleene iteration:
$X^0 = ?$
$X^{i+1}=f(X^i)$

# Solving Static Analysis Problems
# Widening



solution

abstract transfer functions

Enforcing Termination: widening

control flow graph

recursive equation system

Fixpoint Solver

$X=f(X)$

Kleene iteration:

$X^0 = ?$

$X^{i+1}=f(X^i)$

# Contribution to WCET

```
while . . . do [max $n$]
  .
  .
  .
  ref to s
  .
  .
  .
od
```

***time***

$t_{miss}$

$t_{hit}$

***loop time***

$n * t_{miss}$

$n * t_{hit}$

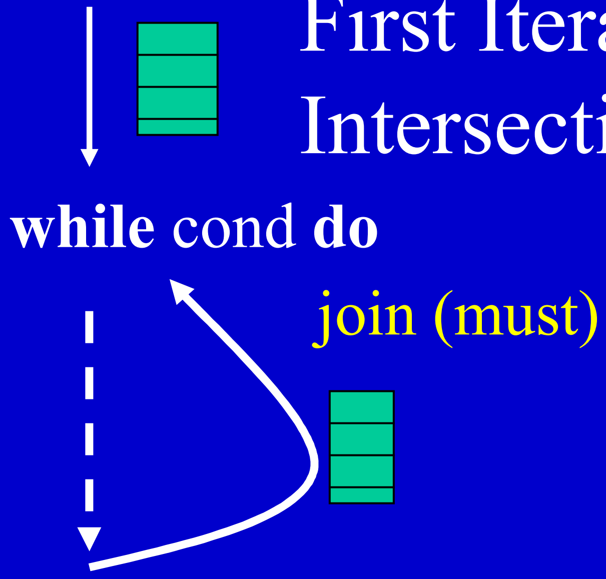
$t_{miss} + (n - 1) * t_{hit}$

$t_{hit} + (n - 1) * t_{miss}$

# Contexts

Cache contents depends on the Context, i.e. calls and loops

First Iteration loads the cache =>
Intersection loses most of the information!



**while** cond **do**

join (must)

# Distinguish basic blocks by contexts

- Transform loops into tail recursive procedures
- Treat loops and procedures in the same way
- Use interprocedural analysis techniques, VIVU
  - virtual inlining of procedures
  - virtual unrolling of loops
- Distinguish as many contexts as useful
  - 1 unrolling for caches
  - 1 unrolling for branch prediction (pipeline)

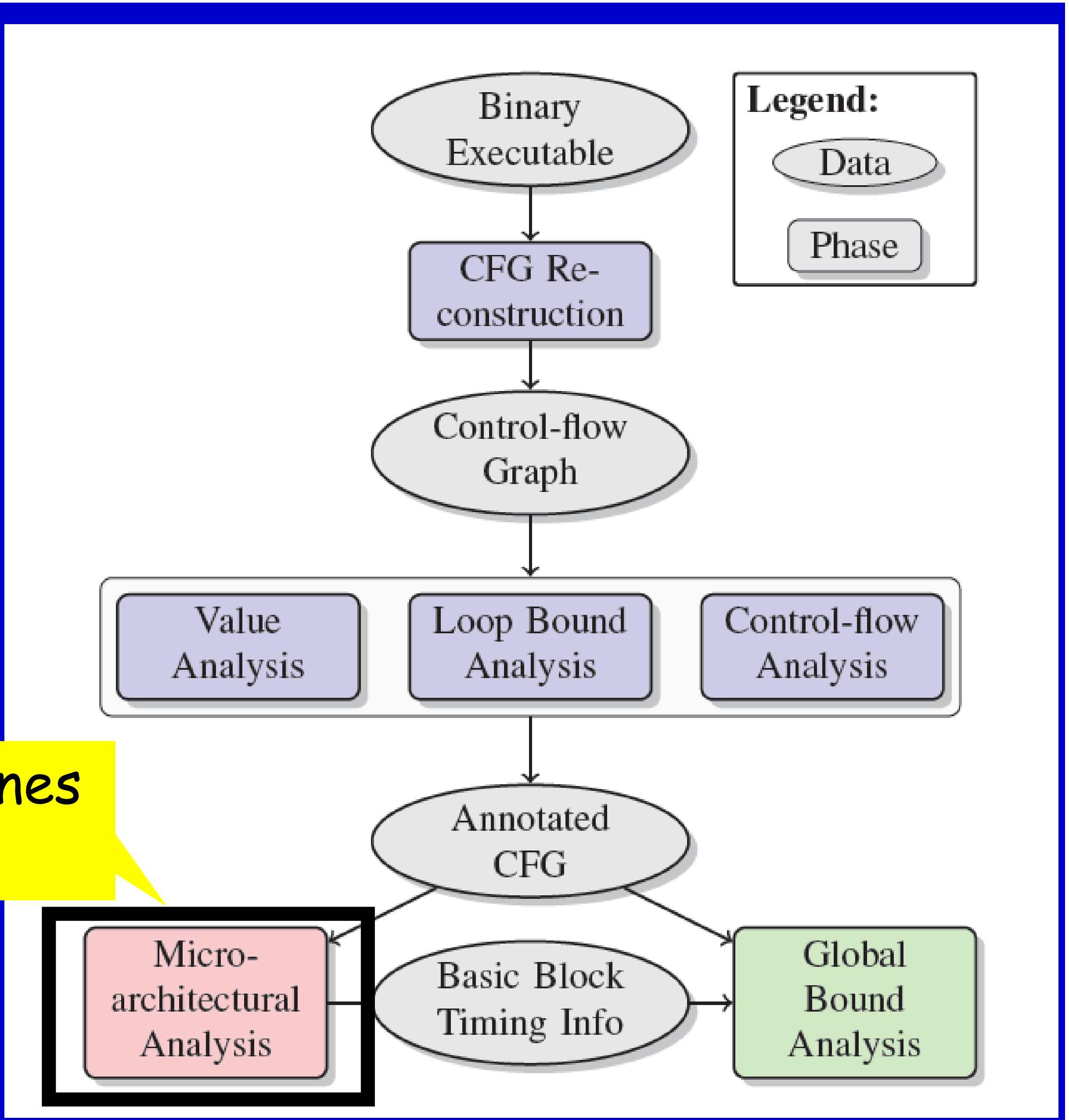
# Structure of the Lectures

1. Introduction
2. Static timing analysis
    1. the problem
    2. our approach
    3. the success
    4. tool architecture
3. Cache analysis
4. Pipeline analysis
5. Value analysis

-----------------------------------------------------------

1. Timing Predictability
    - caches
    - non-cache-like devices
    - future architectures
2. Conclusion

# Tool Architecture



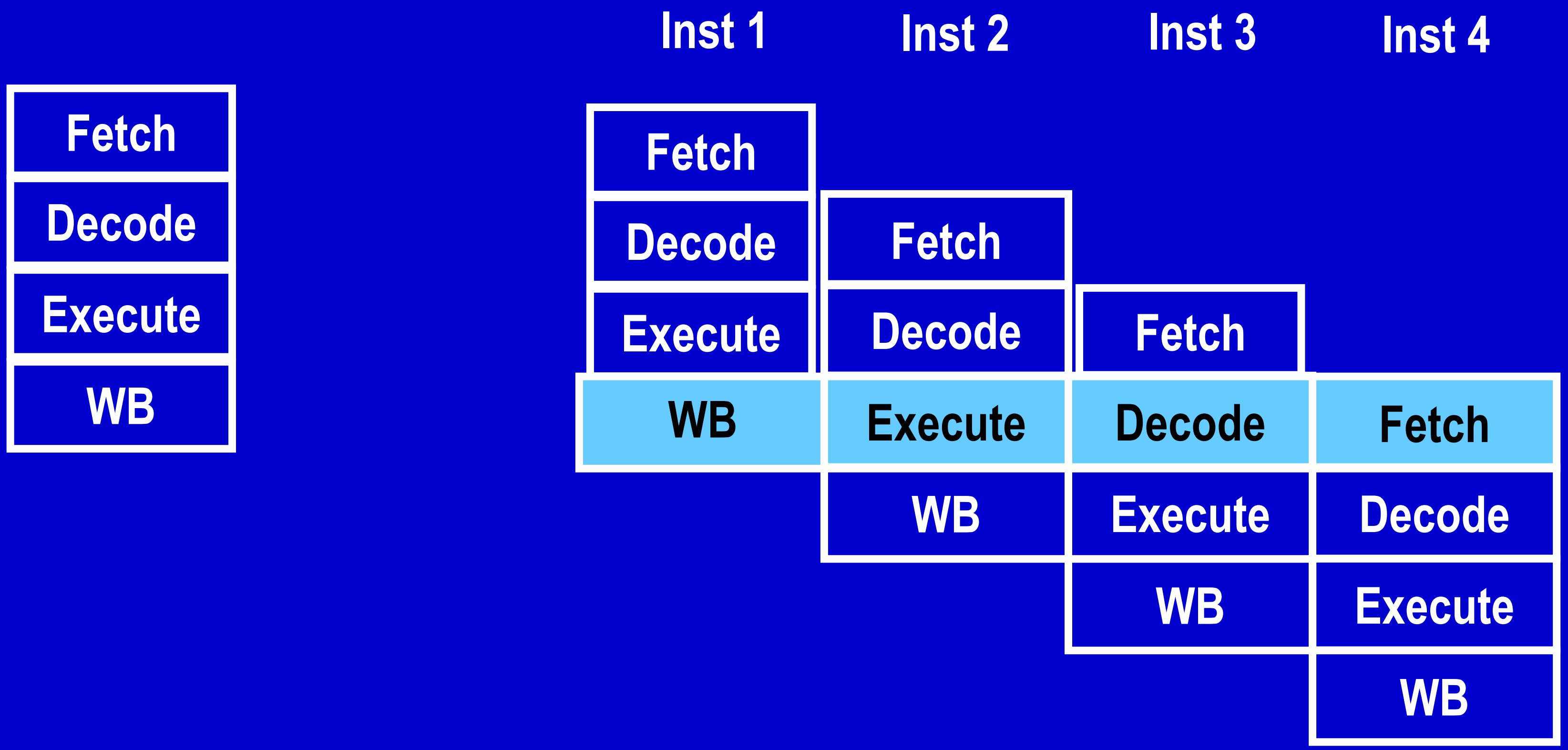
*Abstract Interpretations*

Pipelines

*Abstract Interpretation*

*Integer Linear Programming*

# Hardware Features: Pipelines



Fetch

Decode

Execute

WB

| Inst 1 | Inst 2 | Inst 3 | Inst 4 |
|---|---|---|---|
| Fetch | | | |
| Decode | Fetch | | |
| Execute | Decode | Fetch | |
| WB | Execute | Decode | Fetch |
| | WB | Execute | Decode |
| | | WB | Execute |
| | | | WB |

**Ideal Case: 1 Instruction per Cycle**

# 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

# Pipeline Hazards

Pipeline Hazards:

- Data Hazards: Operands not yet available (Data Dependences)
- Resource Hazards: Consecutive instructions use same resource
- Control Hazards: Conditional branch
- Instruction-Cache Hazards: Instruction fetch causes cache miss

# Static exclusion of hazards

Cache analysis: prediction of cache hits on instruction or operand fetch or store

**lwz r4, 20(r1)**

*Hit*

Dependence analysis: elimination of data hazards

**add r4, r5,r6**
**lwz r7, 10(r1)**
**add r8, r4, r4**

*Operand ready*

Resource reservation tables: elimination of resource hazards

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **IF** | 🟥 | 🟨 | | | | | |
| **EX** | | 🟥 | 🟨 | 🟨 | | | |
| **M** | | | 🟥 | | 🟨 | 🟨 | 🟨 |
| **F** | | | | | | | |

# 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<br>transitions are performed,<br>until a final state is reached:
  - End state: instruction has left the pipeline
  - # transitions: execution time of instruction

# A Concrete Pipeline Executing a Basic Block

**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

# An Abstract Pipeline Executing a Basic Block

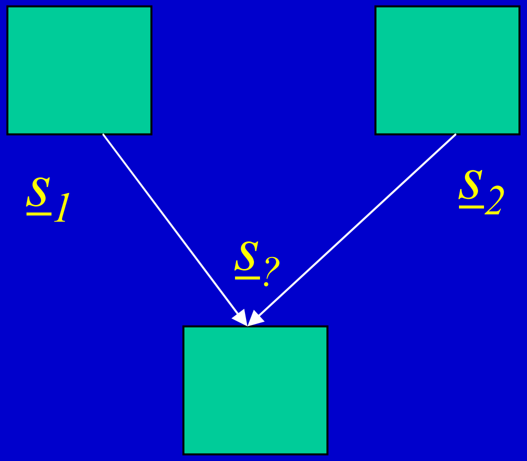
**function** *exec* (*b* : **basic block**, *s* : **abstract pipeline state**)
*t*: **trace**

interprets instruction stream of *b* (annotated with cache information) starting in state *s* producing trace *t*

*length(t)* gives number of cycles

# What is different?

- Abstract states may lack information, e.g. about cache contents.
- Traces may be longer (but never shorter).
- Starting state for successor basic block? In particular, if there are several predecessor blocks.



*Alternatives:*

*• sets of states*

*• combine by least upper bound (join), hard to find one that*

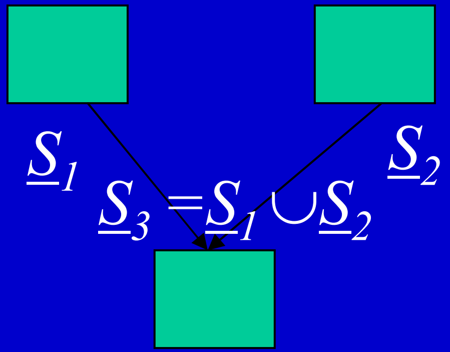
- *• preserves information and*
- *• has a compact representation.*

# Non-Locality of Local Contributions

- Interference between processor components produces **Timing Anomalies**:
  - Assuming local best case leads to higher overall execution time.
  - Assuming local worst case leads to shorter overall execution time
    Ex.: Cache miss in the context of branch prediction
- Treating components in isolation may be unsafe
- Implicit assumptions are not always correct:
  - Cache miss is not always the worst case!
  - The empty cache is not always the worst-case start!

# An Abstract Pipeline Executing a Basic Block - processor with timing anomalies -

**function analyze (*b* : basic block, *S* : analysis state) *T*: set of trace**



Analysis states = $2^{\underline{PS} \times \underline{CS}}$

PS = set of abstract pipeline states

CS = set of abstract cache states

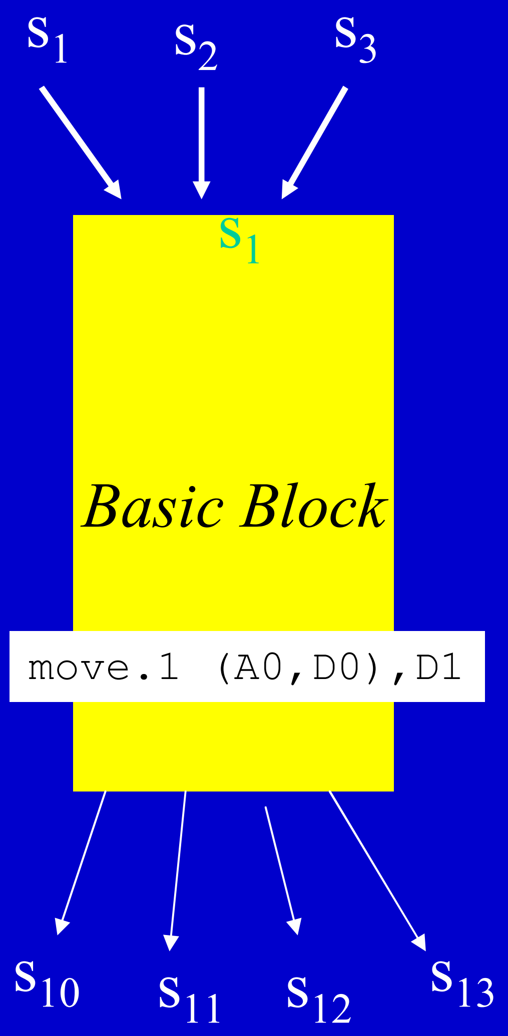
interprets instruction stream of *b* (annotated with cache information) starting in state *S* producing set of traces *T*

*max(length(T))* - upper bound for execution time

*last(T)* - set of initial states for successor block

Union for blocks with several predecessors.

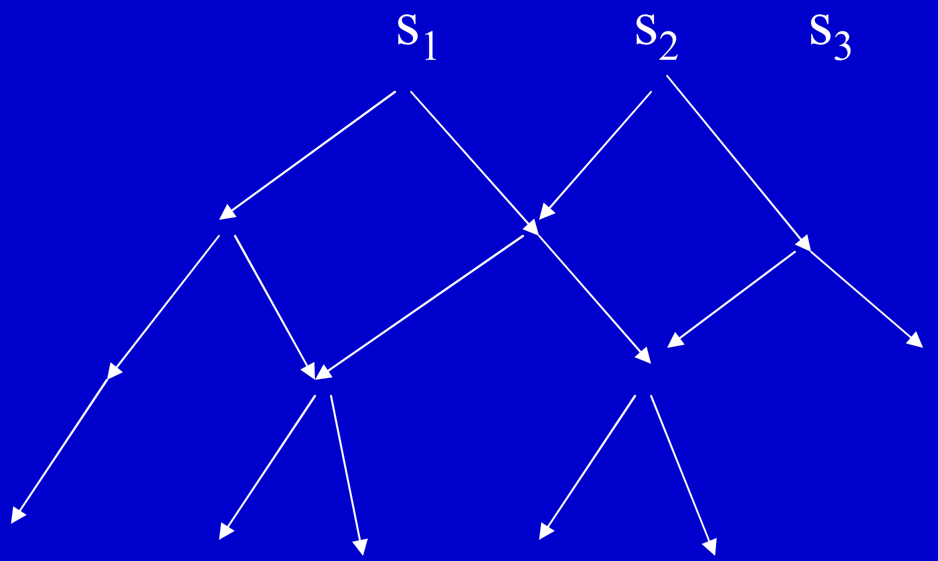
# Integrated Analysis: Overall Picture



```
move.1 (A0,D0),D1
```

Fixed point iteration over Basic Blocks (in context) $\{s_1, s_2, s_3\}$ abstract state

Cyclewise evolution of processor model for instruction



# Classification of Pipelines

- Fully timing compositional architectures:
  - no timing anomalies.
  - analysis can safely follow local worst-case paths only,
  - example: ARM7.
- Compositional architectures with constant-bounded effects:
  - exhibit timing anomalies, but no domino effects,
  - example: Infineon TriCore
- Non-compositional architectures:
  - exhibit domino effects and timing anomalies.
  - timing analysis always has to follow all paths,
  - example: PowerPC 755

# Characteristics of Pipeline Analysis

- Abstract Domain of Pipeline Analysis
  - Power set domain
    - Elements: sets of states of a state machine
  - Join: set union
- Pipeline Analysis
  - Manipulate sets of states of a state machine
  - Store sets of states to detect fixpoint
  - Forward state traversal
  - Exhaustively explore non-deterministic choices

# Abstract Pipeline Analysis vs Model Checking

- Pipeline Analysis is like state traversal in Model Checking
- Symbolic Representation: BDD
- Symbolic Pipeline Analysis:

  Topic of on-going dissertation

# Nondeterminism

- In the reduced model, one state resulted in one new state after a one-cycle transition
- Now, one state can have several successor states
  - Transitions from set of states to set of states

# Implementation

- Abstract model is implemented as a DFA
- Instructions are the nodes in the CFG
- Domain is powerset of set of abstract states
- Transfer functions at the edges in the CFG iterate cycle-wise updating each state in the current abstract value
- `max` {*# iterations for all states*} gives WCET
- From this, we can obtain WCET for basic blocks

# Why integrated analyses?

- Simple modular analysis not possible for architectures with unbounded interference between processor components
- Timing anomalies (Lundqvist/Stenström):
  - Faster execution locally assuming penalty
  - Slower execution locally removing penalty
- Domino effect: Effect only bounded in length of execution

# Timing Anomalies

Let $\Delta_{Tl}$ be an execution-time difference between two different cases for an instruction,

$\Delta_{Tg}$ the resulting difference in the overall execution time.

A Timing Anomaly occurs if either

- $\Delta_{Tl} < 0$: the instruction executes faster, and
  - $\Delta_{Tg} < \Delta_{T1}$: the overall execution is yet faster, or
  - $\Delta_{Tg} > 0$: the program runs longer than before.
- $\Delta_{Tl} > 0$: the instruction takes longer to execute, and
  - $\Delta_{Tg} > \Delta_{Tl}$: the overall execution is yet slower, or
  - $\Delta_{Tg} < 0$: the program takes less time to execute than before

# Timing Anomalies

$\Delta_{Tl} < 0$ and $\Delta_{Tg} > 0$:

Local timing merit causes global timing penalty

is critical for WCET:

using local timing-merit assumptions is unsafe

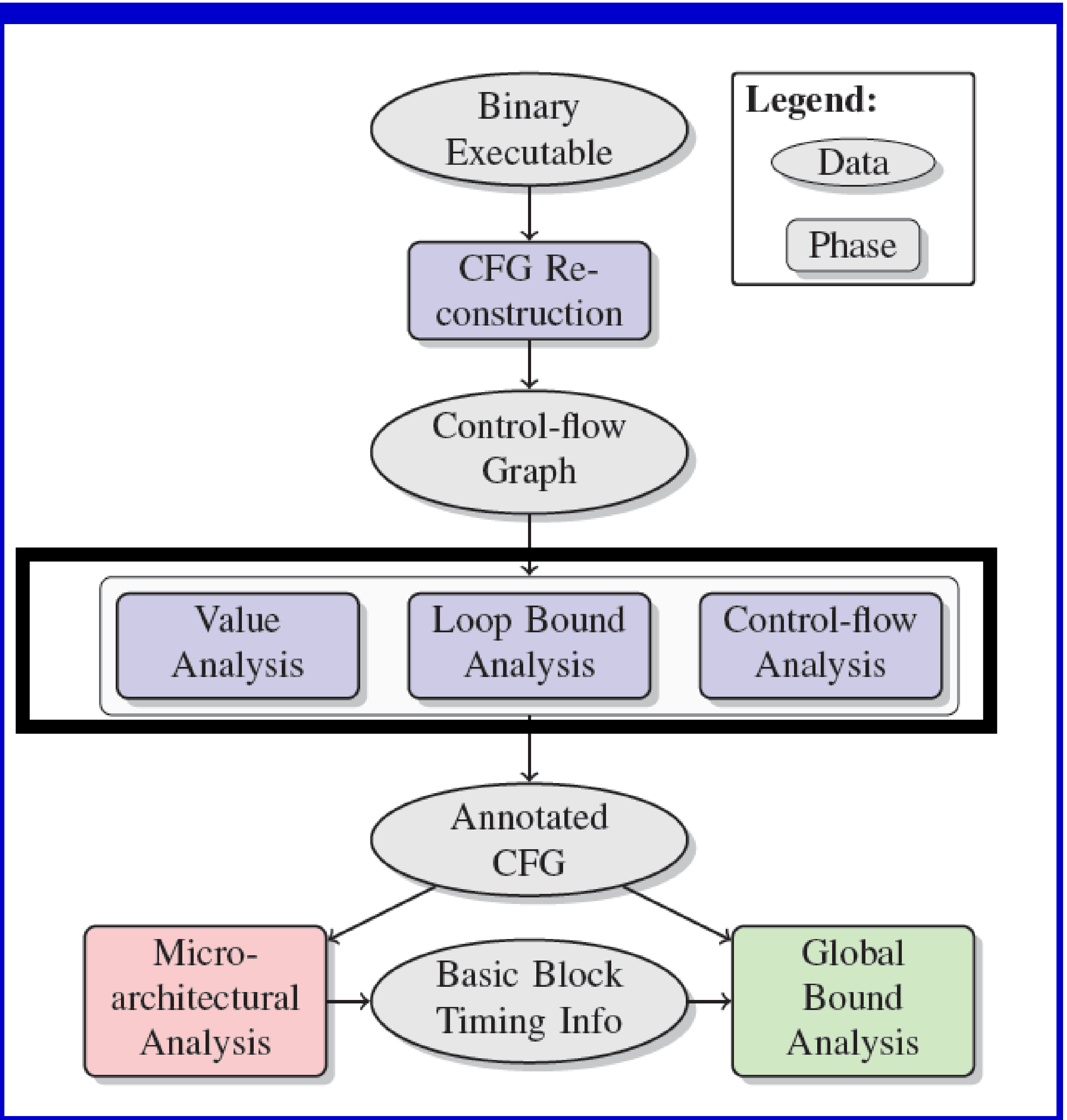
$\Delta_{Tl} > 0$ and $\Delta_{Tg} < 0$:

Local timing penalty causes global speed up

is critical for BCET:

using local timing-penalty assumptions is unsafe

# Tool Architecture



*Abstract Interpretations*

*Abstract Interpretation*

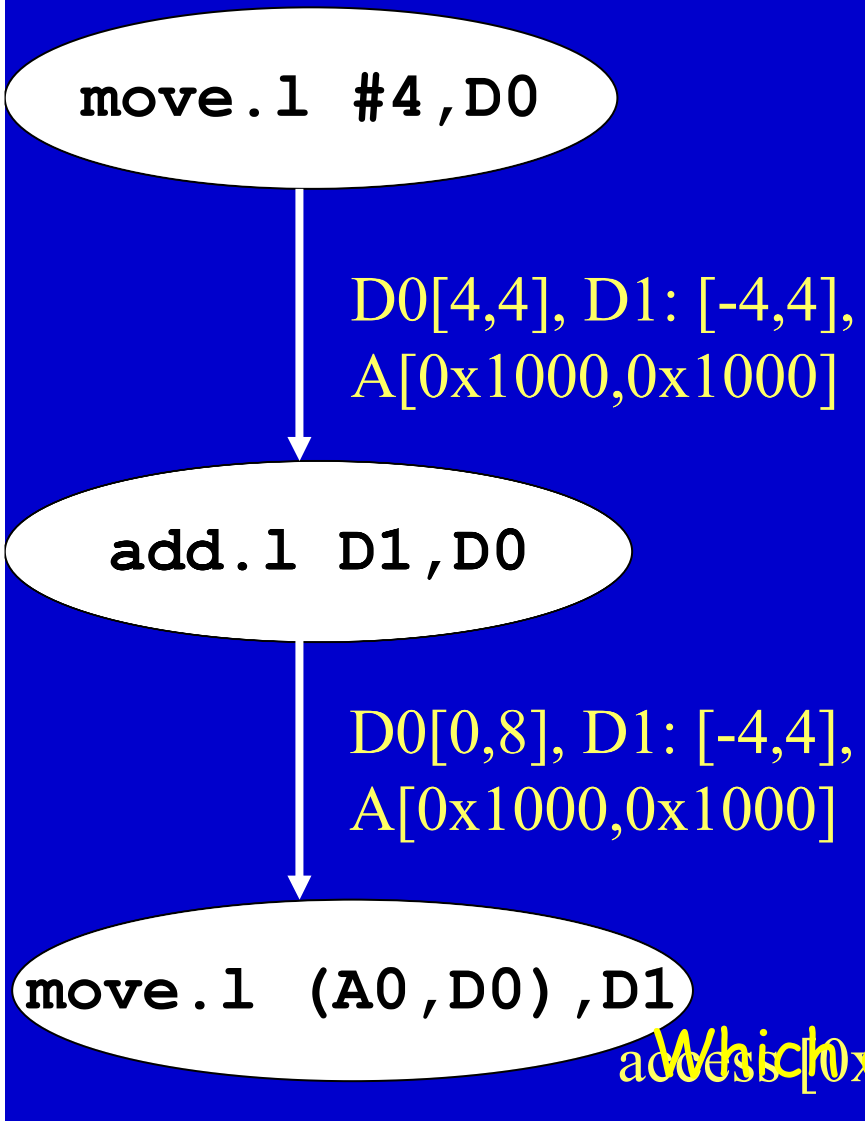
*Integer Linear Programming*

# Value Analysis

- Motivation:
  - Provide access information to data-cache/pipeline analysis
  - Detect infeasible paths
  - Derive loop bounds
- Method: calculate intervals at all program points, i.e. lower and upper bounds for the set of possible values occurring in the machine program (addresses, register contents, local and global variables) (Cousot/Cousot77)

# Value Analysis II

D1: [-4,4], A[0x1000,0x1000]



```
move.l #4,D0
```

D0[4,4], D1: [-4,4], A[0x1000,0x1000]

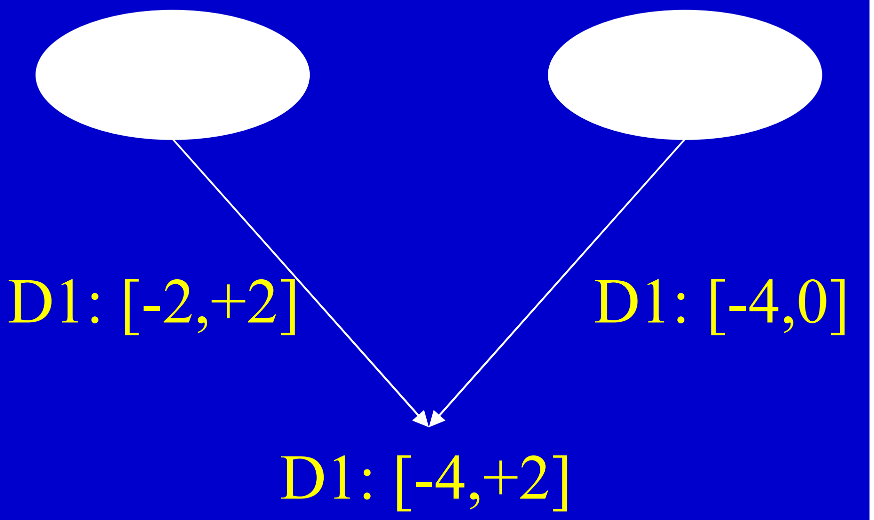
```
add.l D1,D0
```

D0[0,8], D1: [-4,4], A[0x1000,0x1000]

```
move.l (A0,D0),D1
```

- Intervals are computed along the CFG edges

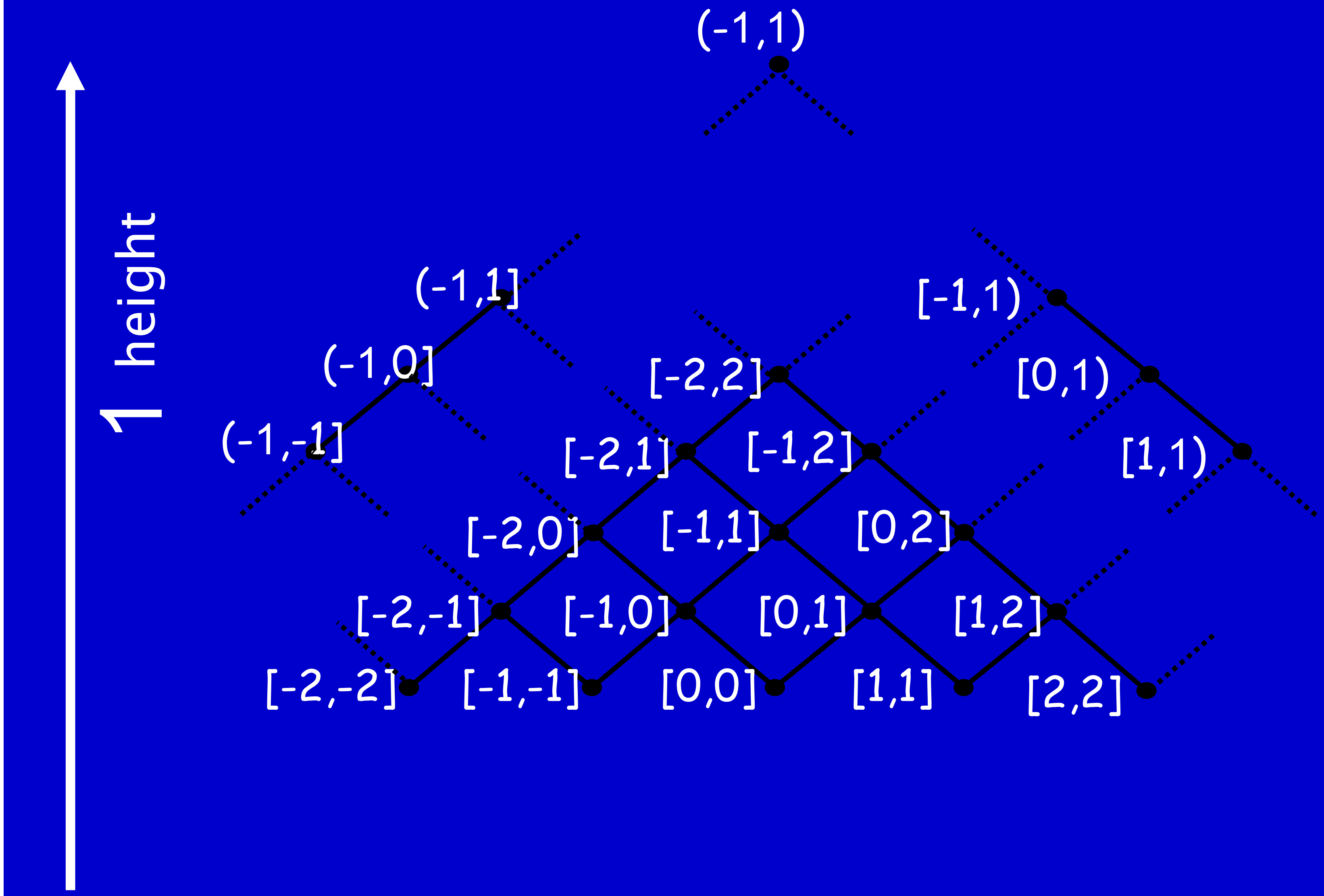
- At joins, intervals are „unioned“



access [0x1000,0x1008]

Which address is accessed here?

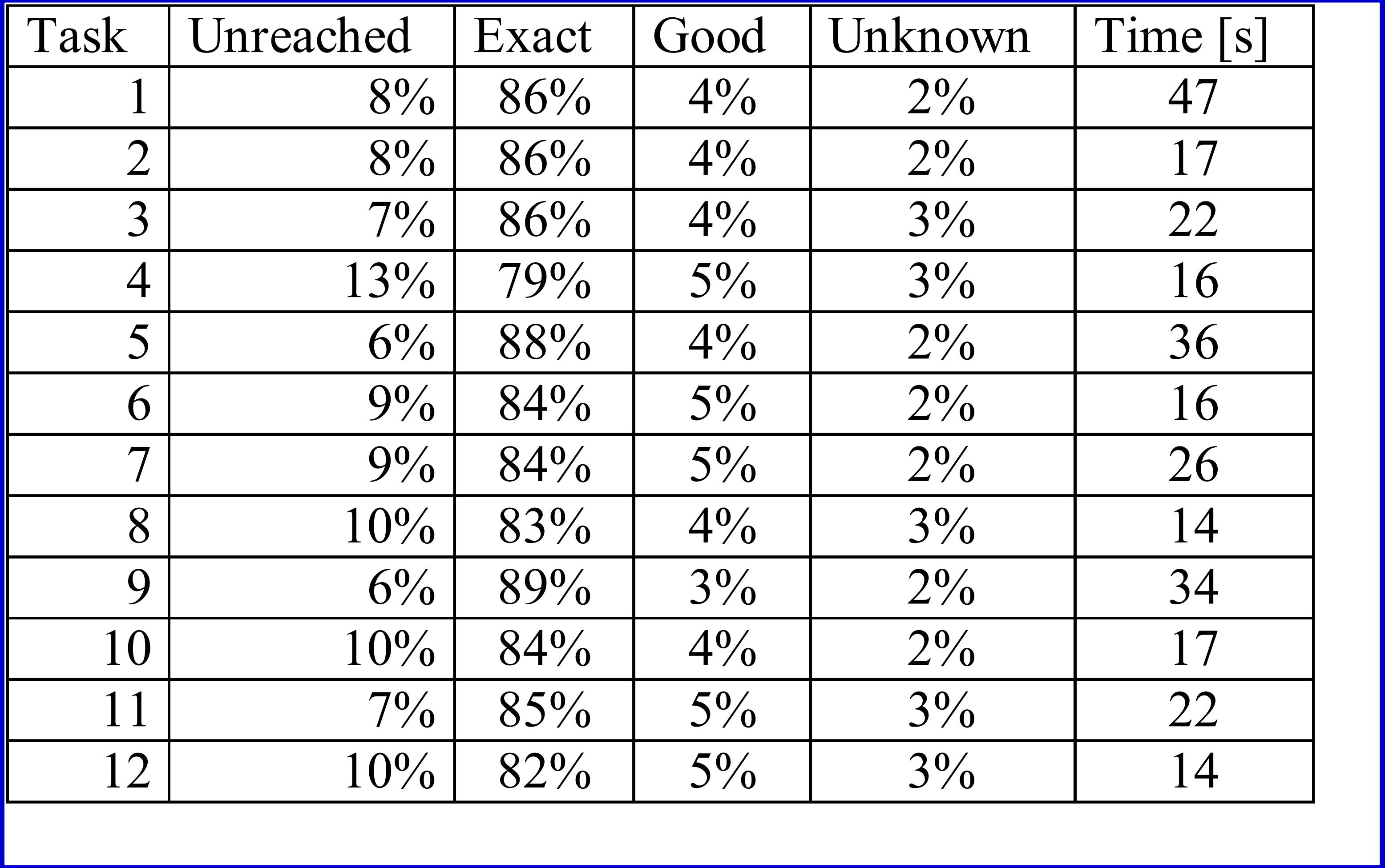
# Interval Domain



# Interval Analysis in Timing Analysis

- Data-cache analysis needs effective addresses at analysis time to know where accesses go.
- Effective addresses are approximatively precomputed by an interval analysis for the values in registers, local variables
- "Exact" intervals – singleton intervals,
- "Good" intervals – addresses fit into less than 16 cache lines.

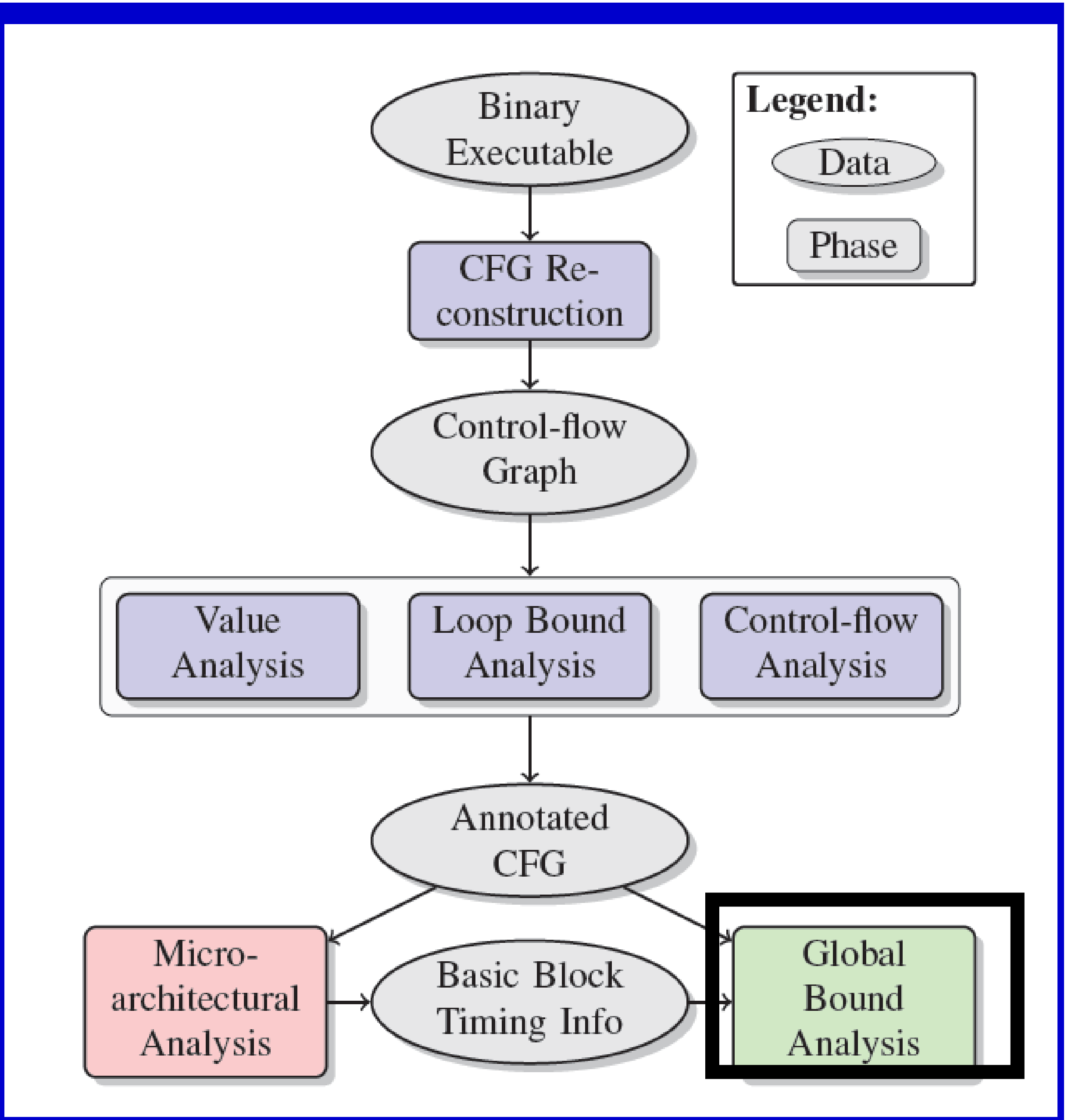
# Value Analysis (Airbus Benchmark)



| Task | Unreached | Exact | Good | Unknown | Time [s] |
|---:|---:|---:|---:|---:|---:|
| 1 | 8% | 86% | 4% | 2% | 47 |
| 2 | 8% | 86% | 4% | 2% | 17 |
| 3 | 7% | 86% | 4% | 3% | 22 |
| 4 | 13% | 79% | 5% | 3% | 16 |
| 5 | 6% | 88% | 4% | 2% | 36 |
| 6 | 9% | 84% | 5% | 2% | 16 |
| 7 | 9% | 84% | 5% | 2% | 26 |
| 8 | 10% | 83% | 4% | 3% | 14 |
| 9 | 6% | 89% | 3% | 2% | 34 |
| 10 | 10% | 84% | 4% | 2% | 17 |
| 11 | 7% | 85% | 5% | 3% | 22 |
| 12 | 10% | 82% | 5% | 3% | 14 |

1Ghz Athlon, Memory usage <= 20MB

# Tool Architecture



*Abstract Interpretations*

*Abstract Interpretation*

*Integer Linear Programming*

# Path Analysis

by Integer Linear Programming (ILP)

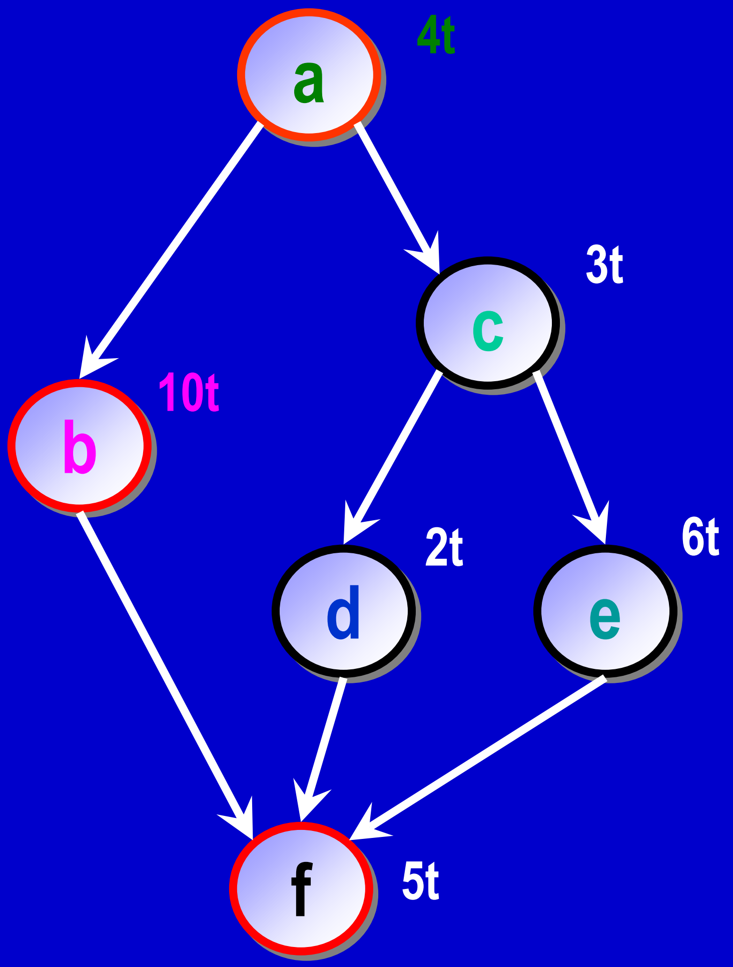
- Execution time of a program =

  $\sum_{\text{Basic Block } b}$ Execution_Time(b) x Execution_Count(b)

- ILP solver maximizes this function to determine the WCET
- Program structure described by linear constraints
  - automatically created from CFG structure
  - user provided loop/recursion bounds
  - arbitrary additional linear constraints to exclude infeasible paths

# Example (simplified constraints)

```
if a then
   b
elseif c then
   d
else
   e
endif
f
```



max: $4\,x_a + 10\,x_b + 3\,x_c + 2\,x_d + 6\,x_e + 5\,x_f$

where $x_a = x_b + x_c$

$x_c = x_d + x_e$

$x_f = x_b + x_d + x_e$

$x_a = 1$

Value of objective function: 19

|  |  |
|---|---|
| $x_a$ | 1 |
| $x_b$ | 1 |
| $x_c$ | 0 |
| $x_d$ | 0 |
| $x_e$ | 0 |
| $x_f$ | 1 |

# Structure of the Lectures

1. Introduction
2. Static timing analysis
    1. the problem
    2. our approach
    3. the success
    4. tool architecture
3. Cache analysis
4. Pipeline analysis
5. Value analysis

-----------------------------------------------------------

1. Timing Predictability
    - caches
    - non-cache-like devices
    - future architectures
2. Conclusion

# Timing Predictability

Experience has shown that the precision of results depend on system characteristics

- of the underlying hardware platform and
- of the software layers
- We will concentrate on the influence of the HW architecture on the predictability

What do we intuitively understand as Predictability?

Is it compatible with the goal of optimizing average-case performance?

What is a strategy to identify good compromises?

# Structure of the Lectures

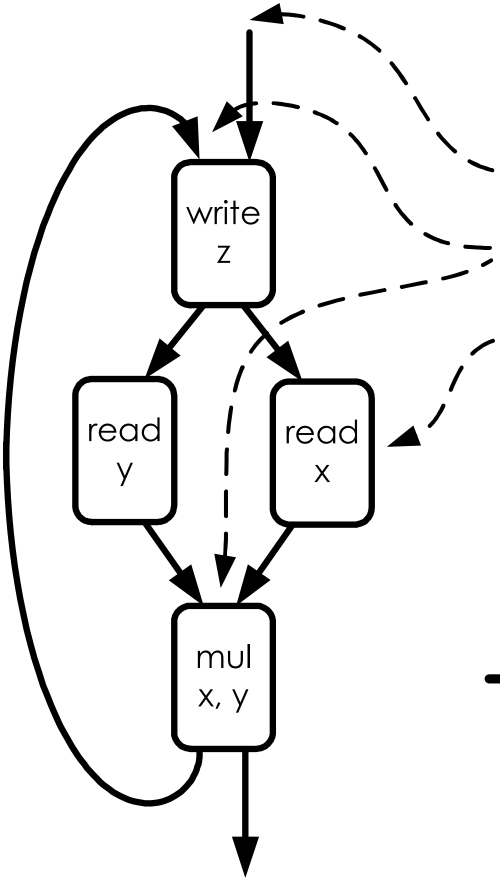
1. Introduction
2. Static timing analysis
    1. the problem
    2. our approach
    3. the success
    4. tool architecture
3. Cache analysis
4. Pipeline analysis
5. Value analysis

---

1. Timing Predictability
    - caches
    - non-cache-like devices
    - future architectures
2. Conclusion

# Predictability of Cache Replacement Policies

# Uncertainty in Cache Analysis

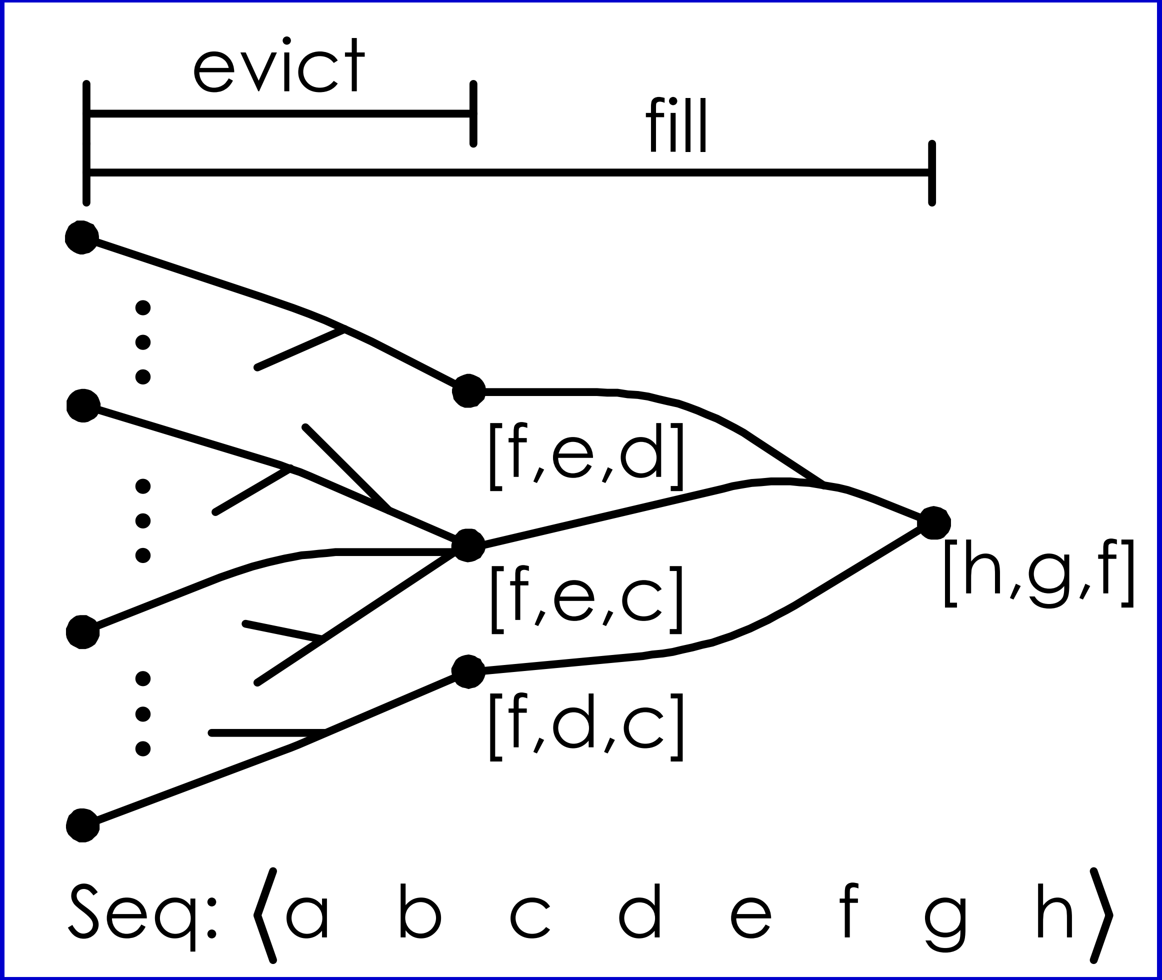


1. Initial cache contents?
2. Need to combine information
3. Cannot resolve address of x...
4. Imprecise analysis domain/ update functions

→ Need to recover information:
Predictability = Speed of Recovery

# Metrics of Predictability:

## evict & fill



Two Variants:

M = Misses Only

HM

# Meaning of evict/fill - I

- Evict: *may*-information:
  - What is definitely not in the cache?
  - Safe information about Cache Misses
- Fill: *must*-information:
  - What is definitely in the cache?
  - Safe information about Cache Hits

# Meaning of evict/fill - II

Metrics are independent of analyses:

→ evict/fill bound the precision of any static analysis!

→ Allows to analyze an analysis:

Is it as precise as it gets w.r.t. the metrics?

# Replacement Policies

- LRU – Least Recently Used

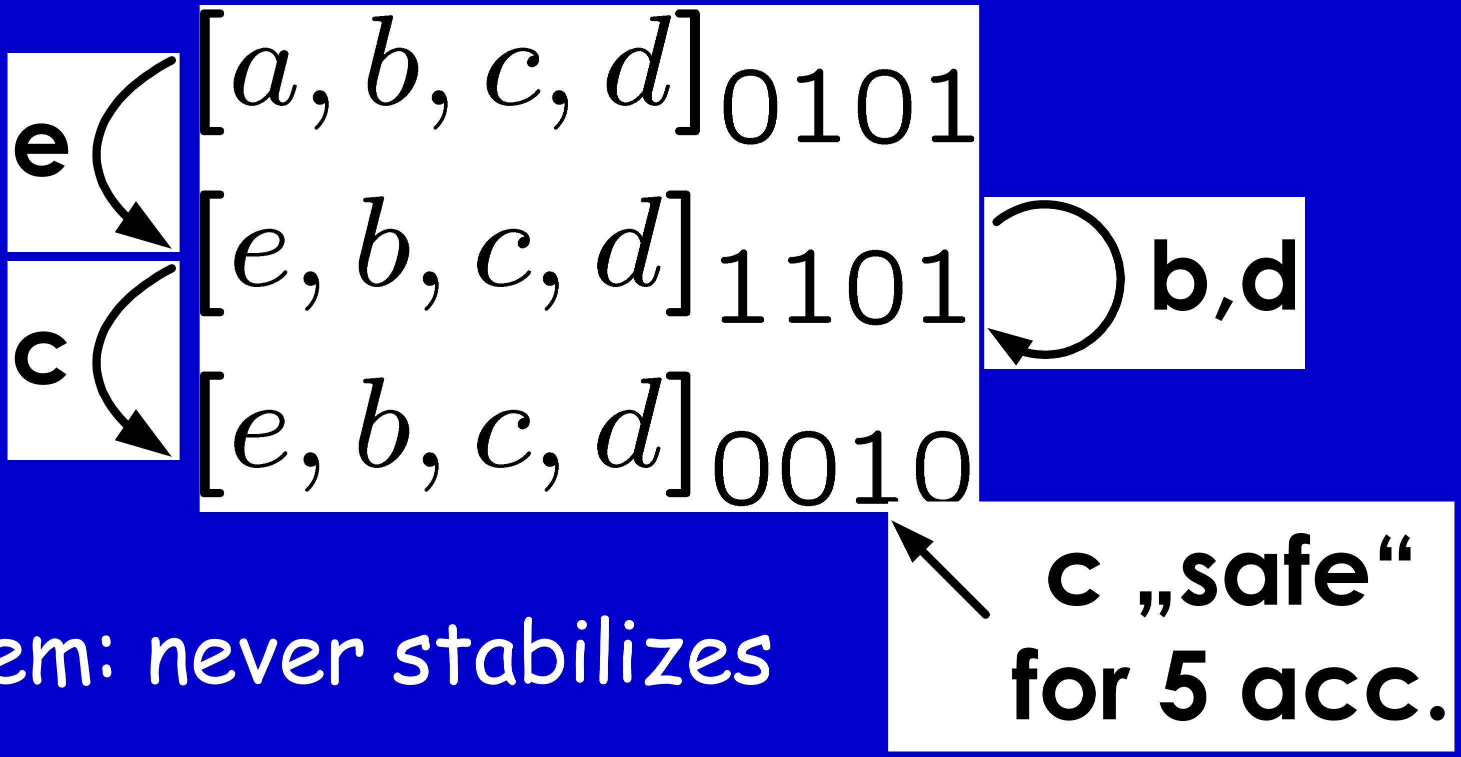
  Intel Pentium, MIPS 24K/34K
- FIFO – First-In First-Out (Round-robin)

  Intel XScale, ARM9, ARM11
- PLRU – Pseudo-LRU

  Intel Pentium II+III+IV, PowerPC 75x
- MRU – Most Recently Used

# MRU - Most Recently Used

MRU-bit records whether line was recently used



$[a, b, c, d]_{0101}$

$[e, b, c, d]_{1101}$

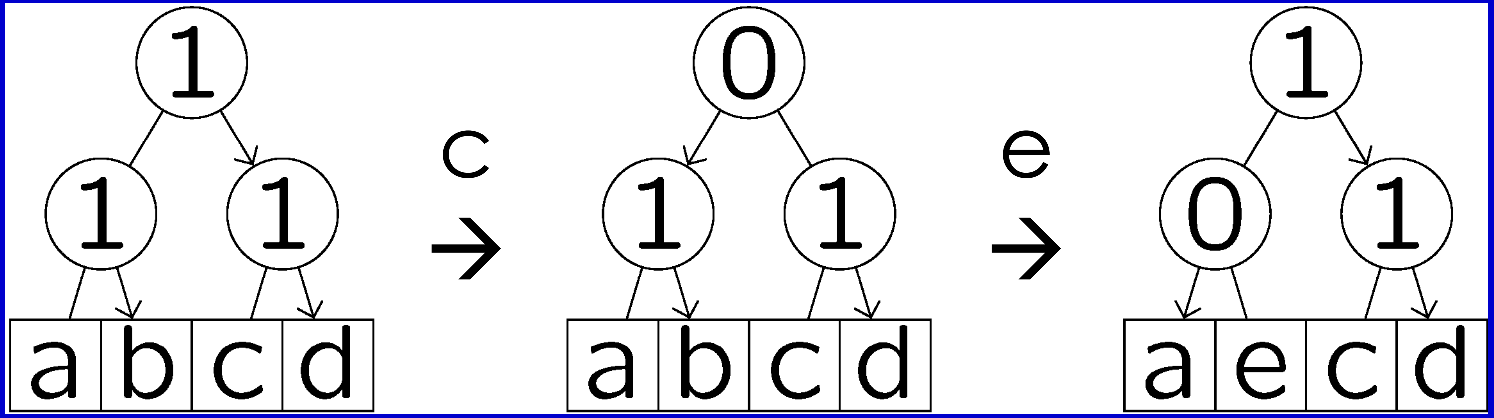
$[e, b, c, d]_{0010}$

c „safe“ for 5 acc.

Problem: never stabilizes

# Pseudo-LRU

Tree maintains order:



Problem: accesses „rejuvenate" neighborhood

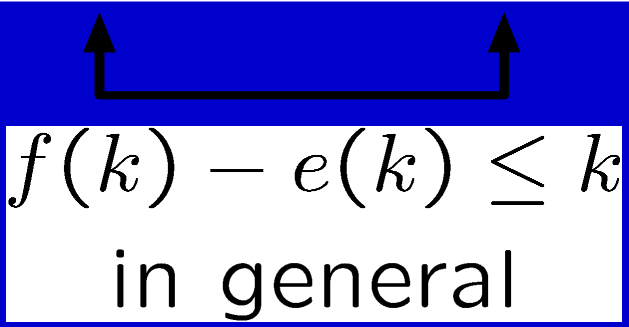
# Results: tight bounds

| Policy | $e_M(k)$ | $f_M(k)$ | $e_{HM}(k)$ | $f_{HM}(k)$ |
|---|---|---|---|---|
| LRU | $k$ | $k$ | $k$ | $k$ |
| FIFO | $k$ | $k$ | $2k-1$ | $3k-1$ |
| MRU | $2k-2$ | $\infty/2k-4^{\S}$ | $2k-2$ | $\infty/3k-4^{\S}$ |
| PLRU | $\left\{\begin{array}{c} 2k-\sqrt{2k} \\ 2k-\frac{3}{2}\sqrt{k} \end{array}\right\}$ | $2k-1$ | $\frac{k}{2}\log_2 k+1$ | $\frac{k}{2}\log_2 k+k-1$ |

| | $k=4$ | | | | $k=8$ | | | |
|---|---|---|---|---|---|---|---|---|
| Policy | $e_M$ | $f_M$ | $e_{HM}$ | $f_{HM}$ | $e_M$ | $f_M$ | $e_{HM}$ | $f_{HM}$ |
| LRU | 4 | 4 | 4 | 4 | 8 | 8 | 8 | 8 |
| FIFO | 4 | 4 | 7 | 11 | 8 | 8 | 15 | 23 |
| MRU | 6 | $\infty/4$ | 6 | $\infty/8$ | 14 | $\infty/12$ | 14 | $\infty/20$ |
| PLRU | 5 | 7 | 5 | 7 | 12 | 15 | 13 | 19 |

# Results: tight bounds

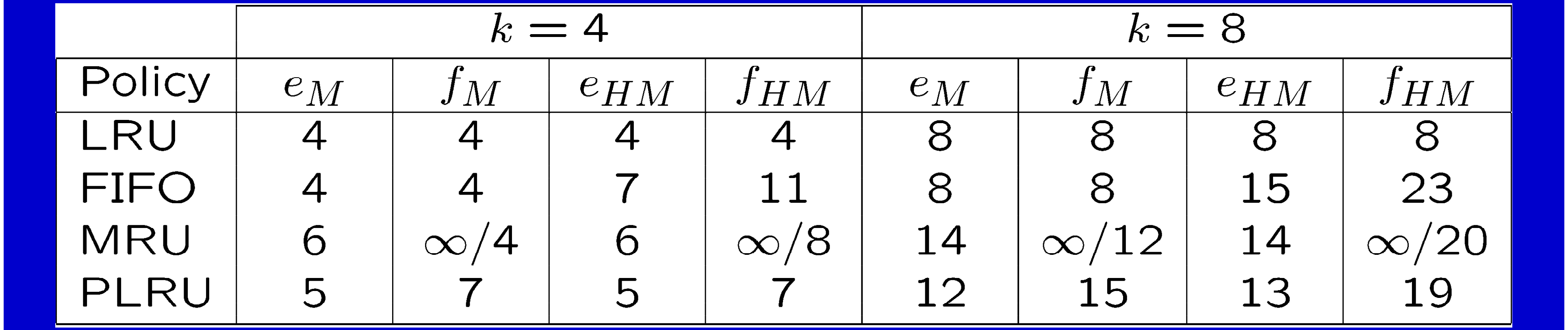
| Policy | $e_M(k)$ | $f_M(k)$ | $e_{HM}(k)$ | $f_{HM}(k)$ |
|---|---|---|---|---|
| LRU | $k$ | $k$ | $k$ | $k$ |
| FIFO | $k$ | $k$ | $2k-1$ | $3k-1$ |
| MRU | $2k-2$ | $\infty/2k-4^{\S}$ | $2k-2$ | $\infty/3k-4^{\S}$ |
| PLRU | $\left\{ \begin{array}{c} 2k-\sqrt{2k} \\ 2k-\frac{3}{2}\sqrt{k} \end{array} \right\}$ | $2k-1$ | $\frac{k}{2}\log_2 k+1$ | $\frac{k}{2}\log_2 k+k-1$ |


$$f(k)-e(k)\le k$$

in general

Generic examples prove tightness.

# Results: instances for k=4,8



| | $k=4$ | | | | $k=8$ | | | |
|---|---|---|---|---|---|---|---|---|
| Policy | $e_M$ | $f_M$ | $e_{HM}$ | $f_{HM}$ | $e_M$ | $f_M$ | $e_{HM}$ | $f_{HM}$ |
| LRU | 4 | 4 | 4 | 4 | 8 | 8 | 8 | 8 |
| FIFO | 4 | 4 | 7 | 11 | 8 | 8 | 15 | 23 |
| MRU | 6 | $\infty/4$ | 6 | $\infty/8$ | 14 | $\infty/12$ | 14 | $\infty/20$ |
| PLRU | 5 | 7 | 5 | 7 | 12 | 15 | 13 | 19 |

Question: 8-way PLRU cache, 4 instructions per line
Assume equal distribution of instructions over

256 sets:

How long a straight-line code sequence is needed to obtain precise may-information?

# Future Work I

OPT for performance

$\overset{?}{=}$ LRU for predictability

- OPT = theoretical strategy,
  optimal for performance
- LRU = used in practice,
  optimal for predictability
- Predictability of OPT?
- Other optimal policies for predictability?

# Future Work II

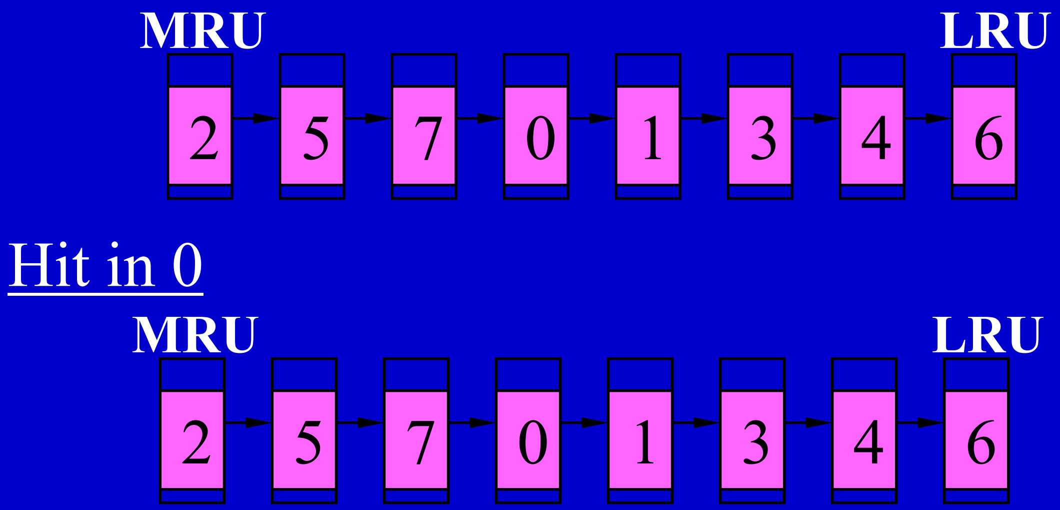
Beyond evict/fill:

- Evict/fill assume complete uncertainty
- What if there is only partial uncertainty?
- Other useful metrics?

# LRU has Optimal Predictability, so why is it Seldom Used?

- LRU is more expensive than PLRU, Random, etc.
- But it can be made fast
  - Single-cycle operation is feasible [Ackland JSSC00]
  - Pipelined update can be designed with no stalls
- Gets worse with high-associativity caches
  - Feasibility demonstrated up to 16-ways
- There is room for finding lower-cost highly-predictable schemes with good performance

# LRU algorithm

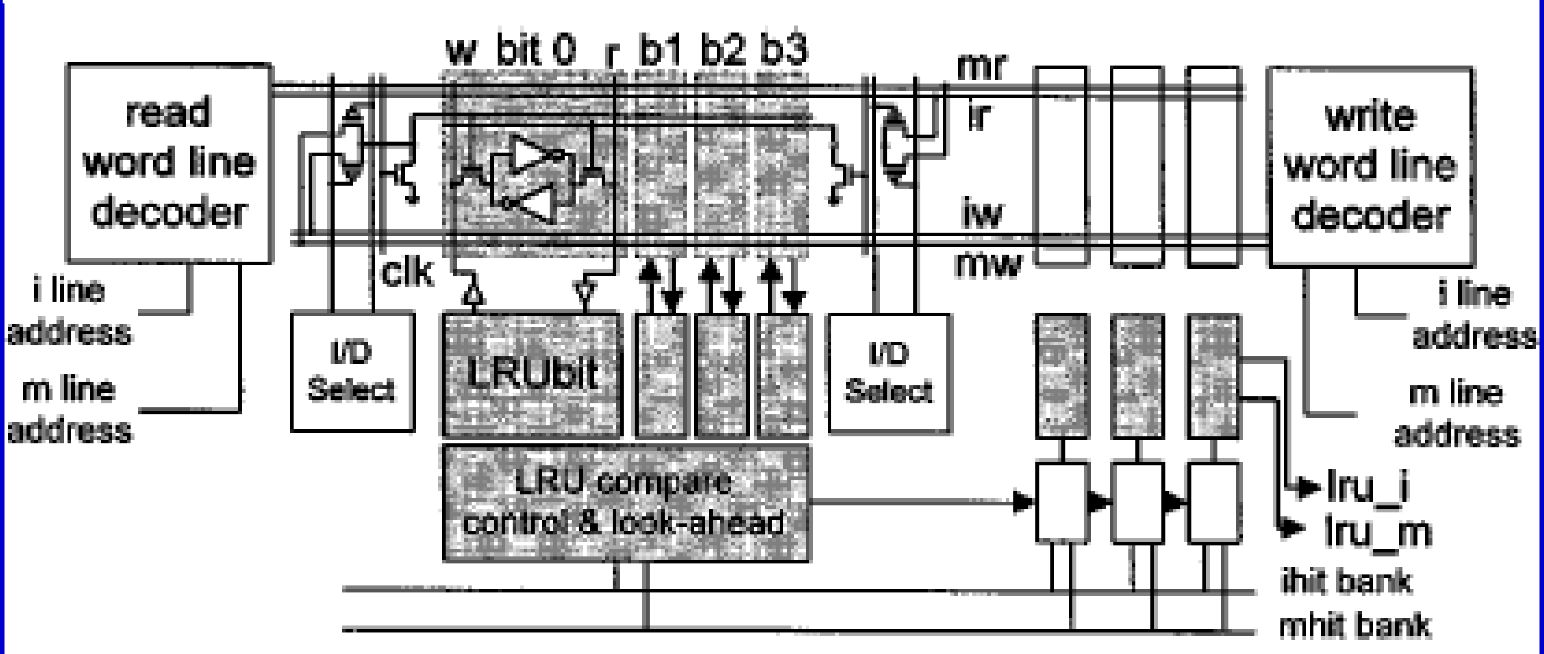


LRU stack

- Trivial, but requires an associative search-and-shift operation to locate and promote a bank to the top of the stack.
- It would be too time consuming to read the stack from the RAM, locate and shift the bank ID within the stack, and write it back to the RAM in a single cycle.

# LRU HW implementation

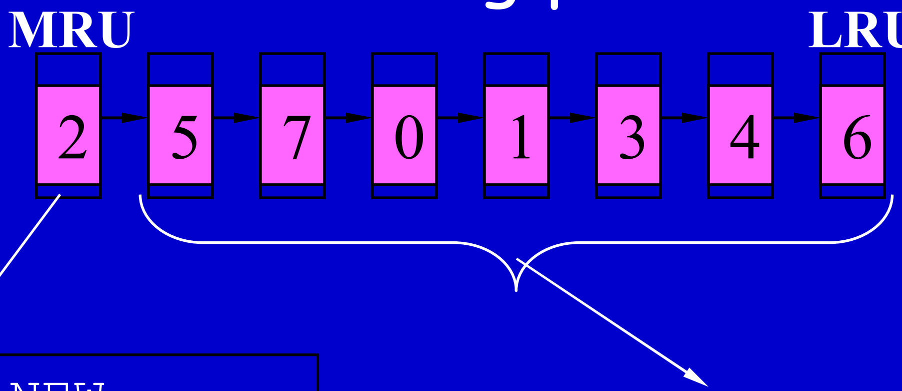
[Ackland JSSC00] LRU info is available in one cycle



- LRU-RAM produces LRU states for lines @ current ADDR
- Stores updates when state is written back: LRU is available at the same cycle when a MISS is detected

# LRU RAM Update Circuit

- Three-cycle operation
  1. LRU RAM is read
  2. LRU info is updated
  3. LRU RAM is written
- Pipelined with forwarding paths to eliminate hazards

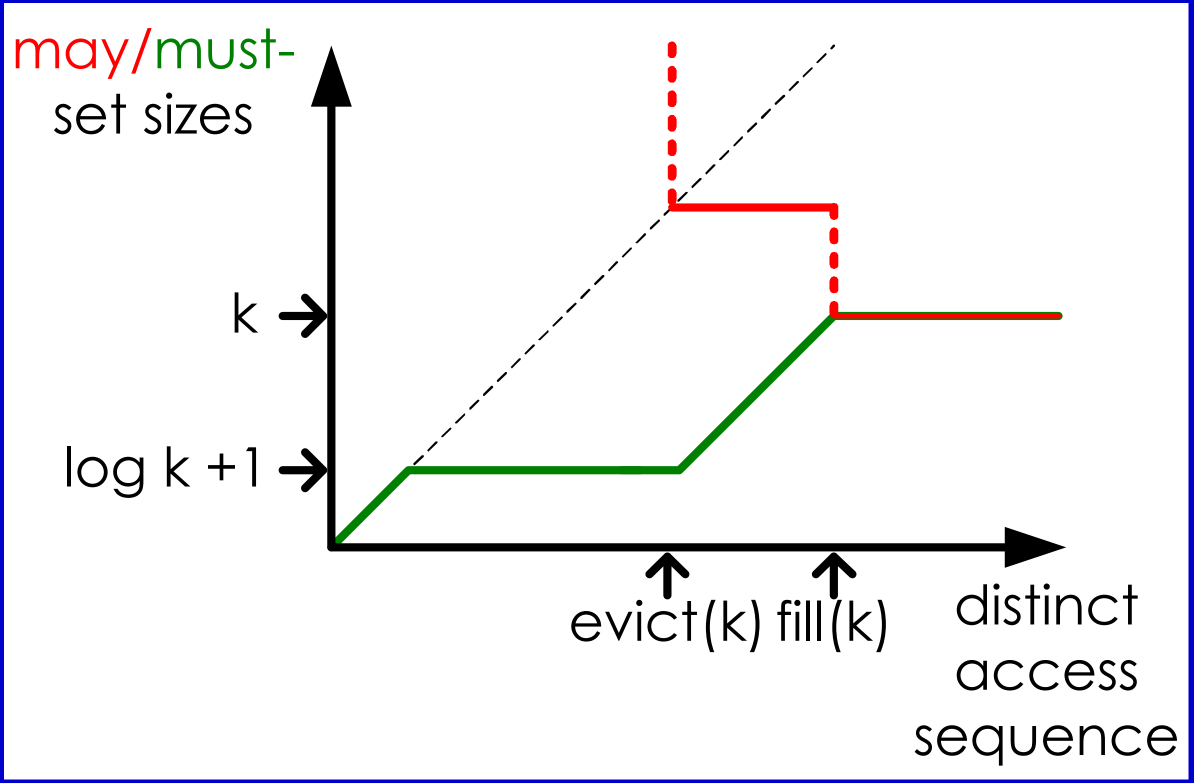


```
If  STACK[0] ≠ NEW
    STACK[0]<= NEW;
    STACK[1]<= STACK[0];
```

```
If  STACK[i] ≠ NEW
    STACK[i+1]<= STACK[i];
```

# Beyond evict/fill

Evolution of *may-* / *must-*information (PLRU):



# Structure of the Lectures

1. Introduction
2. Static timing analysis
   1. the problem
   2. our approach
   3. the success
   4. tool architecture
3. Cache analysis
4. Pipeline analysis
5. Value analysis

-----------------------------------------------------------

1. Timing Predictability
   - caches
   - non-cache-like devices
   - future architectures
2. Conclusion

# Extended the Predictability Notion

- The cache-predictability concept applies to all cache-like architecture components:
- TLBs, BTBs, other history mechanisms

# The Predictability Notion

Unpredictability

- is an inherent system property
- limits the obtainable precision of static predictions about dynamic system behavior

Digital hardware behaves deterministically (ignoring defects, thermal effects etc.)

- Transition is fully determined by current state and input
- We model hardware as a (hierarchically structured, sequentially and concurrently composed) finite state machine
- Software and inputs induce possible (hardware) component inputs

# Uncertainties About State and Input

- If initial system state and input were known, only one execution (time) were possible.
- To be safe, static analysis must take into account all possible initial states and inputs.
- Uncertainty about state implies a set of starting states and different transition paths in the architecture.
- Uncertainty about program input implies possibly different program control flow.
- Overall result: possibly different execution times

Ed wants to forbid this!

# Source and Manifestation of Unpredictability

- “Outer view” of the problem: Unpredictability manifests itself in the variance of execution time
- Shortest and longest paths through the automaton are the BCET and WCET

- “Inner view” of the problem: Where does the variance come from?
- For this, one has to look into the structure of the finite automata

# Variability of Execution Times

- is at the heart of timing unpredictability,
- is introduced at all levels of granularity
  - Memory reference
  - Instruction execution
  - Arithmetic
  - Communication
- results, in some way or other, from the interference on shared resources.

# Connection Between Automata and Uncertainty

- Uncertainty about state and input are qualitatively different:
- State uncertainty shows up at the “beginning” ≅ number of possible initial starting states the automaton may be in.
- States of automaton with high in-degree lose this initial uncertainty.

- Input uncertainty shows up while “running the automaton”.
- Nodes of automaton with high out-degree introduce uncertainty.

# State Predictability – the Outer View

Let *T(i;s)* be the execution time with component input *i* starting in hardware component state *s*.

$$\text{State predictability} := \min_{\text{Component Input } i} \; \min_{\text{State } s_1, s_2} \frac{T(i, s_1)}{T(i, s_2)}$$

The range is in [0::1], 1 means perfectly timing-predictable

The smaller the set of states, the smaller the variance and the larger the predictability.

The smaller the set of component inputs to consider, the larger the predictability.

# Input Predictability

$$\text{Input predictability} := \min_{\text{State } s} \; \min_{\text{Component Input } i_1, i_2} \frac{T(i_1, s)}{T(i_2, s)}$$

# Comparing State Predictability
# - on the basis of the variance -

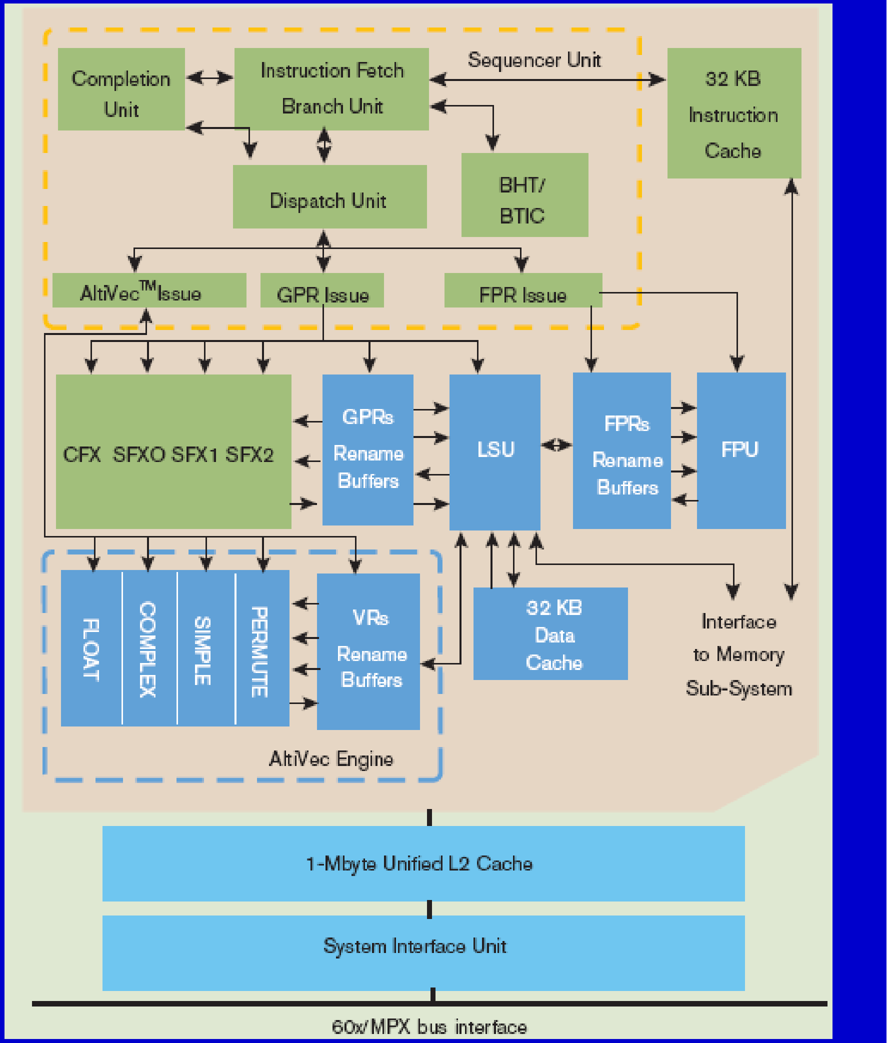
- statically scheduled processors more predictable than dynamically scheduled,
- static branch prediction more predictable than dynamic branch prediction,
- processor without cache more predictable than processor with cache,
- scheduling on several levels is most unpredictabe
- independent cache sets are more predictable than dependent cache sets
- separate I- and D-caches are more predictable than uniform caches

# Predictability – the Inner View

- We can look into the automata:
- Speed of convergence
- #reachable states
- #transitions/outdegree/indegree

# Processor Features of the MPC 7448

- Single e600 core, 600MHz-1,7GHz core clock
- 32 KB L1 data and instruction caches
- 1 MB unified L2 cache with ECC
- Up to 12 instructions in instruction queue
- Up to 16 instructions in parallel execution
- 7 stage pipeline
- 3 issue queues, GPR, FPR, AltiVec
- 11 independent execution units



# Processor Features (cont.)

- Branch Processing Unit
  - Static and dynamic branch prediction
  - Up to 3 outstanding speculative branches
  - Branch folding during fetching
- 4 Integer Units
  - 3 identical simple units (IU1s), 1 for complex operations (IU2)
- 1 Floating Point Unit with 5 stages
- 4 Vector Units
- 1 Load Store Unit with 3 stages
  - Supports hits under misses
  - 5 entry L1 load miss queue
  - 5 entry outstanding store queue
  - Data forwarding from outstanding stores to dependent loads
- Rename buffers (16 GPR/16 FPR/16 VR)
- 16 entry Completion Queue
  - Out-of-order execution but In-order completion

# Challenges and Predictability

- Speculative Execution
  - Up to 3 level of speculation due to unknown branch prediction
- Cache Prediction
  - Different pipeline paths for L1 cache hits/misses
  - Hits under misses
  - PLRU cache replacement policy for L1 caches
- Arbitration between different functional units
  - Instructions have different execution times on IU1 and IU2
- Connection to the Memory Subsystem
  - Up to 8 parallel accesses on MPX bus
- Several clock domains
  - L2 cache controller clocked with half core clock
  - Memory subsystem clocked with 100 – 200 MHz

# Architectural Complexity
# implies
# Analysis Complexity

Every hardware component whose state has an influence on the timing behavior

- must be conservatively modeled,
- contributes a multiplicative factor to the size of the search space.

History/future devices: all devices concerned with storing the past or predicting the future.

# Classification of Pipelines

- Fully timing compositional architectures:
  - no timing anomalies.
  - analysis can safely follow local worst-case paths only,
  - example: ARM7.
- Compositional architectures with constant-bounded effects:
  - exhibit timing anomalies, but no domino effects,
  - example: Infineon TriCore
- Non-compositional architectures:
  - exhibit domino effects and timing anomalies.
  - timing analysis always has to follow all paths,
  - example: PowerPC 755

# Recommendation for Pipelines

- Use compositional pipelines; often execution time is dominated by memory-access times, anyway.
- Static branch prediction only;
- One level of speculation only

# More Threats created by Computer Architects

- Out-of-order execution

*Consider all possible execution orders*

- Speculation

*ditto*

- Timing Anomalies,
  i.e., locally worst-case path does not lead to the globally worst-case path, e.g., a cache miss can contribute to a globally shorter execution if it prevents a mis-prediction.

*Considering the locally worst-case path insufficent*

# First Principles

- Reduce interference on shared resources.
- Use homogeneity in the design of history/future devices.

# Interference on Shared Resources

- can be real
  - e.g., tasks interfering on buses, memory, caches

*real non-determinism*

- can be virtual, introduced by abstraction, e.g.,
  - unknown state of branch predictor forces analysis of both transitions ⇒ interference on instruction cache
  - are responsible for timing anomalies

*artificial non-determinism*

# Design Goal:
# Reduce Interference on Shared Resources

- Integrated Modular Avionics (IMA) goes in the right direction - temporal and spatial partitioning for eliminating logical interference
- For predictability: extension towards the elimination/reduction of physical interference

# Shared Resources between Threads on Different Cores

- Strong synchronization
  ⇒ low performance
- Little synchronization
  ⇒ many potential interleavings
  ⇒ high complexity of analysis

# Recommendations for Architecture Design

Form follows function,

(Louis Sullivan)

Architecture follows application:
Exploit information about the application in the architecture design.

Design architectures to which applications can be mapped without introducing extra interferences.

# Recommendation for Application Designers

- Use knowledge about the architecture to produce an interference-free mapping.

# Separated Memories

- Characteristic of many embedded applications: little code shared between several tasks of an application ⇒ separate memories for code of threads running on different cores

# Shared Data

- Often:
    - reading data when task is started,
    - writing data when task terminate
- deterministic scheme for access to shared data memory<br>required cache performance determines
    - partition of L2-caches
    - bus schedule
- Crossbar instead of shared bus

# Conclusion

- Feasibility, efficiency, and precision of timing analysis strongly depend on the execution platform.
- Several principles were proposed to support timing analysis.

# Relevant Publications (from my group)

- *C. Ferdinand et al.: Cache Behavior Prediction by Abstract Interpretation. Science of Computer Programming 35(2): 163-189 (1999)*
- *C. Ferdinand et al.: Reliable and Precise WCET Determination of a Real-Life Processor*, EMSOFT 2001
- *M. Langenbach et al.: Pipeline Modeling for Timing Analysis*, SAS 2002
- *R. Heckmann et al.: The Influence of Processor Architecture on the Design and the Results of WCET Tools*, IEEE Proc. on Real-Time Systems, July 2003
- *St. Thesing et al.: An Abstract Interpretation-based Timing Validation of Hard Real-Time Avionics Software*, IPDS 2003
- *R. Wilhelm: AI + ILP is good for WCET, MC is not, nor ILP alone*, VMCAI 2004
- *L. Thiele, R. Wilhelm: Design for Timing Predictability, 25th Anniversary edition of the Kluwer Journal* Real-Time Systems, *Dec. 2004*
- *J. Reineke et al.: Predictability of Cache Replacement Policies, Real-Time Systems, 2007*
- *R. Wilhelm: Determination of Execution-Time Bounds, CRC Handbook on Embedded Systems, 2005*
- *R. Wilhelm et al. : The worst-case execution-time problem—overview of methods and survey of tools, ACM Transactions on Embedded Computing Systems (TECS), Volume 7 , Issue 3 (April 2008)*
- *R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, C. Ferdinand: Desiderata for Future Architectures in Time-critical Embedded Systems, submitted to TCAD*