Static Timing Analysis of Hard Real-Time Systems

Reinhard Wilhelm
Saarbrücken

AbsInt
Angewandte Informatik GmbH

UNIVERSITÄT
DES
SAARLANDES

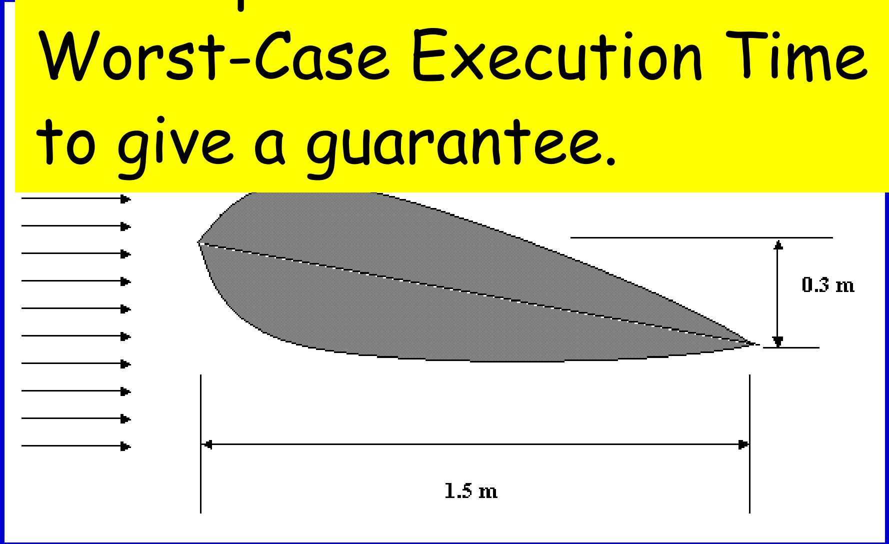
Hard Real-Time Systems

- Hard real-time systems, often in safety-critical applications abound
  - Aeronautics, automotive, train industries, manufacturing control

Sideairbag in car,
Reaction in <10 mSec

Embedded controllers must finish their tasks within given time bounds. Developers would like to know the Worst-Case Execution Time (WCET) to give a guarantee.

Wing vibration of airplane,
sensing every 5 mSec



Crankshaft-synchronous
tasks in cars < 45 uSec

Structure of the Lecture

- The Problem
- Sketch of a solution and report of success
- Tool architecture
- Analyzing cache behavior
- An excursion to Predictability
- Analyzing pipeline behavior
- Ongoing and future work

Static Timing Analysis
- the Problem -

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.

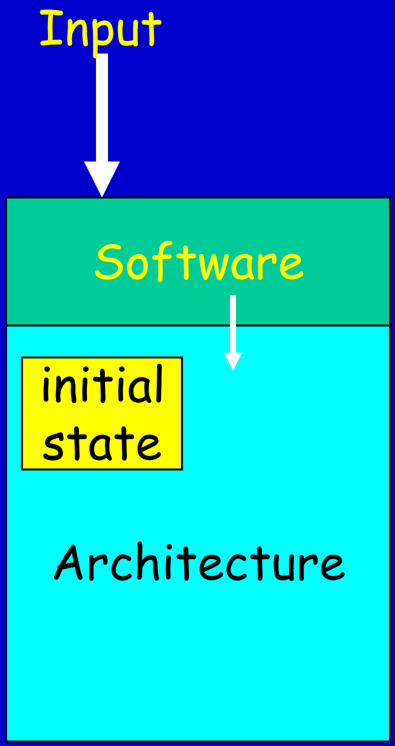
What does Execution Time Depend on?

- 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 ⇒ all exhaustive approaches infeasible

"external" interference as seen from analyzed task



Modern Hardware Features

- Modern processors increase 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

(Concrete) Instruction Execution

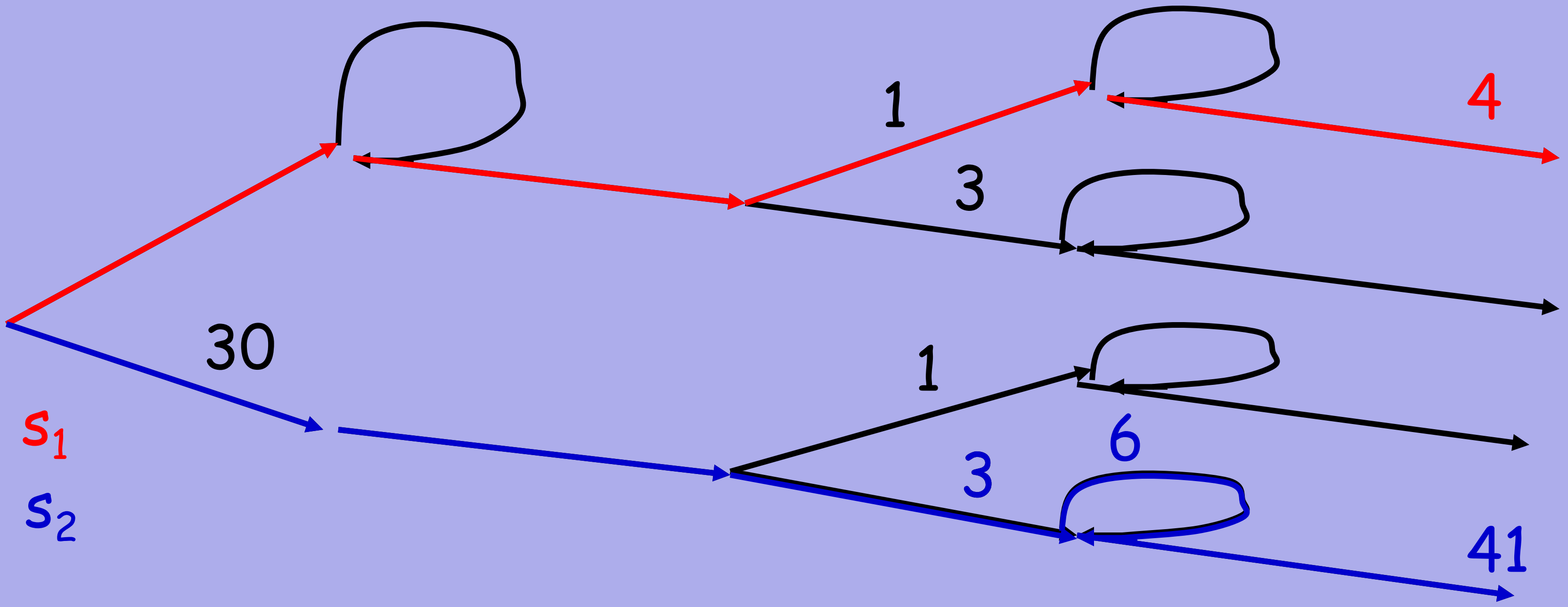
`mul`

Fetch I-Cache miss?

Issue Unit occupied?

Execute Multicycle?

Retire Pending instructions?



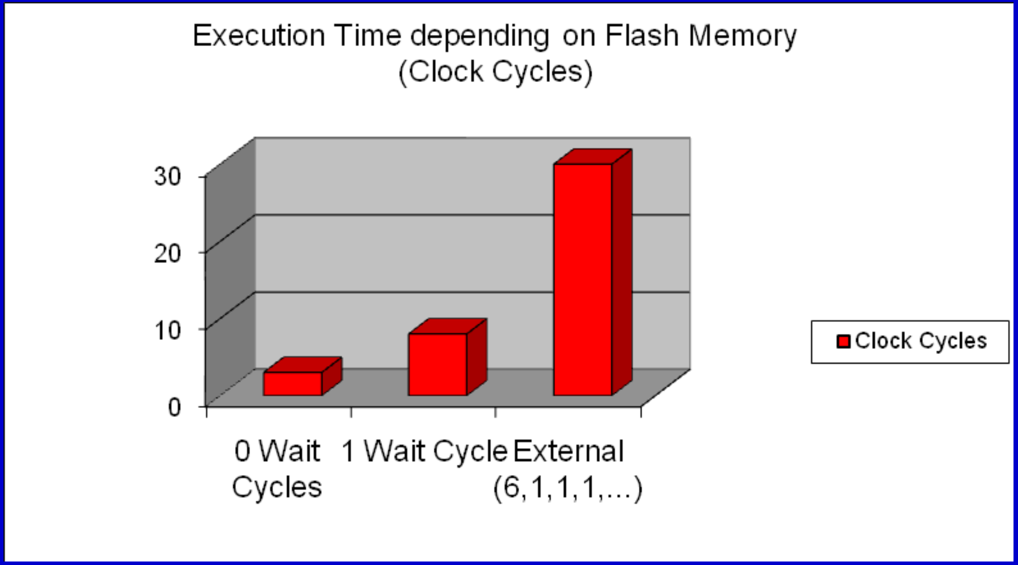
Access Times

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

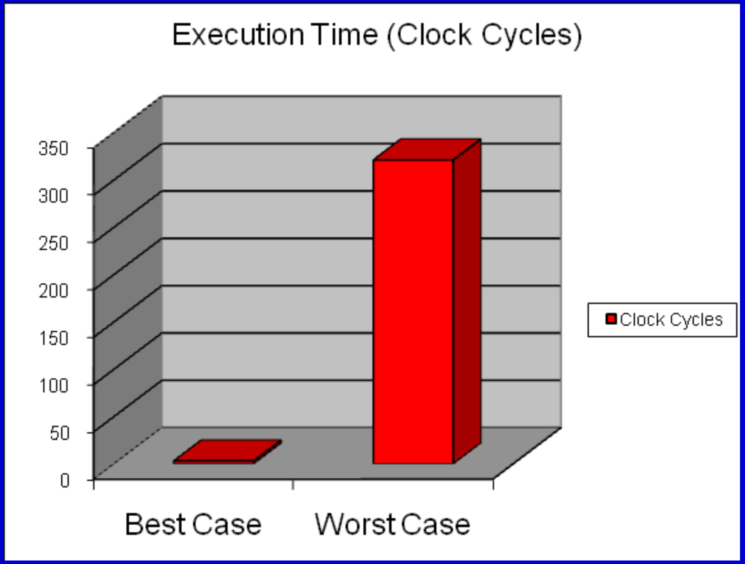
x = a + b; →

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

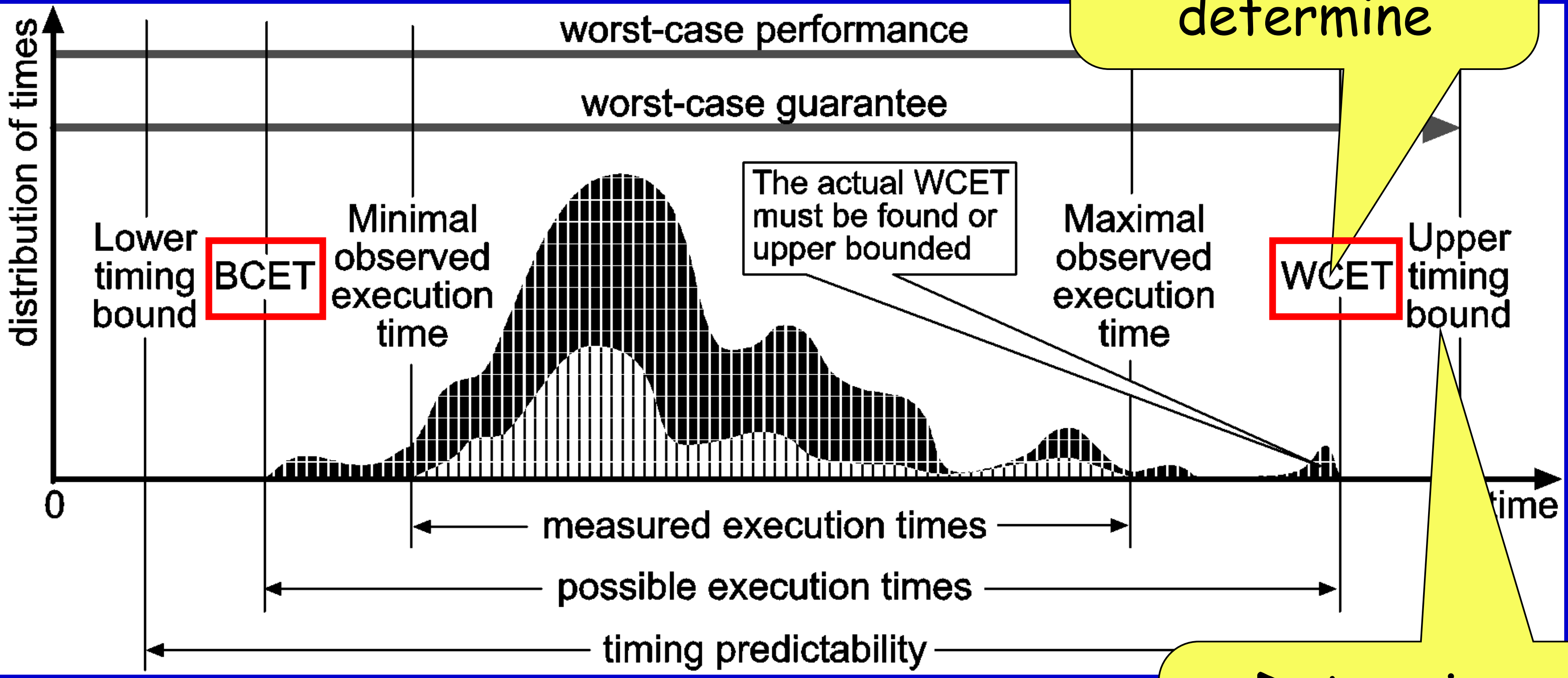
MPC 5xx


Execution Time depending on Flash Memory (Clock Cycles)

PPC 755


Execution Time (Clock Cycles)

Notions in Timing Analysis



Hard or impossible to determine

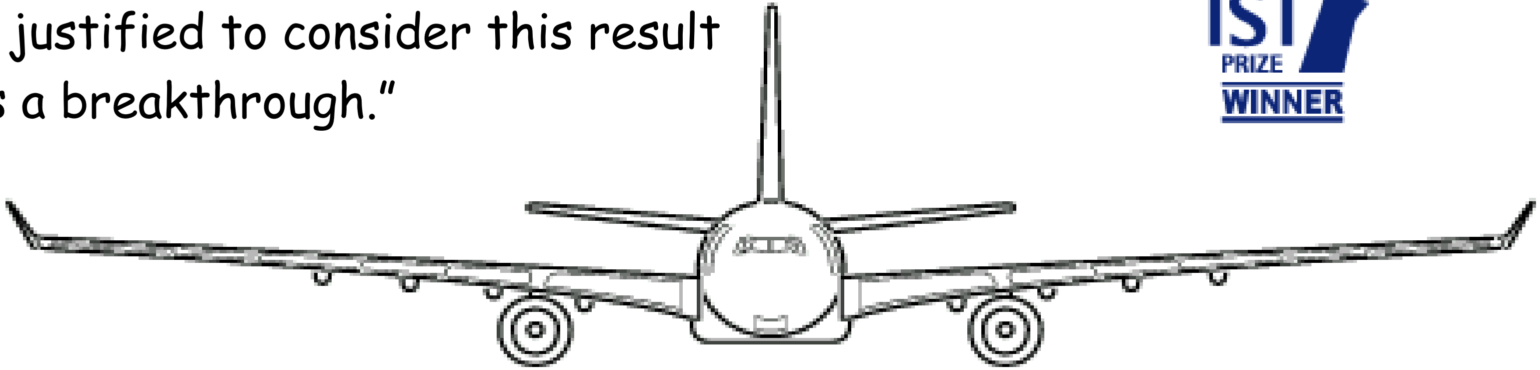
Determine upper bounds instead

aiT WCET Analyzer

IST Project DAEDALUS final review report:

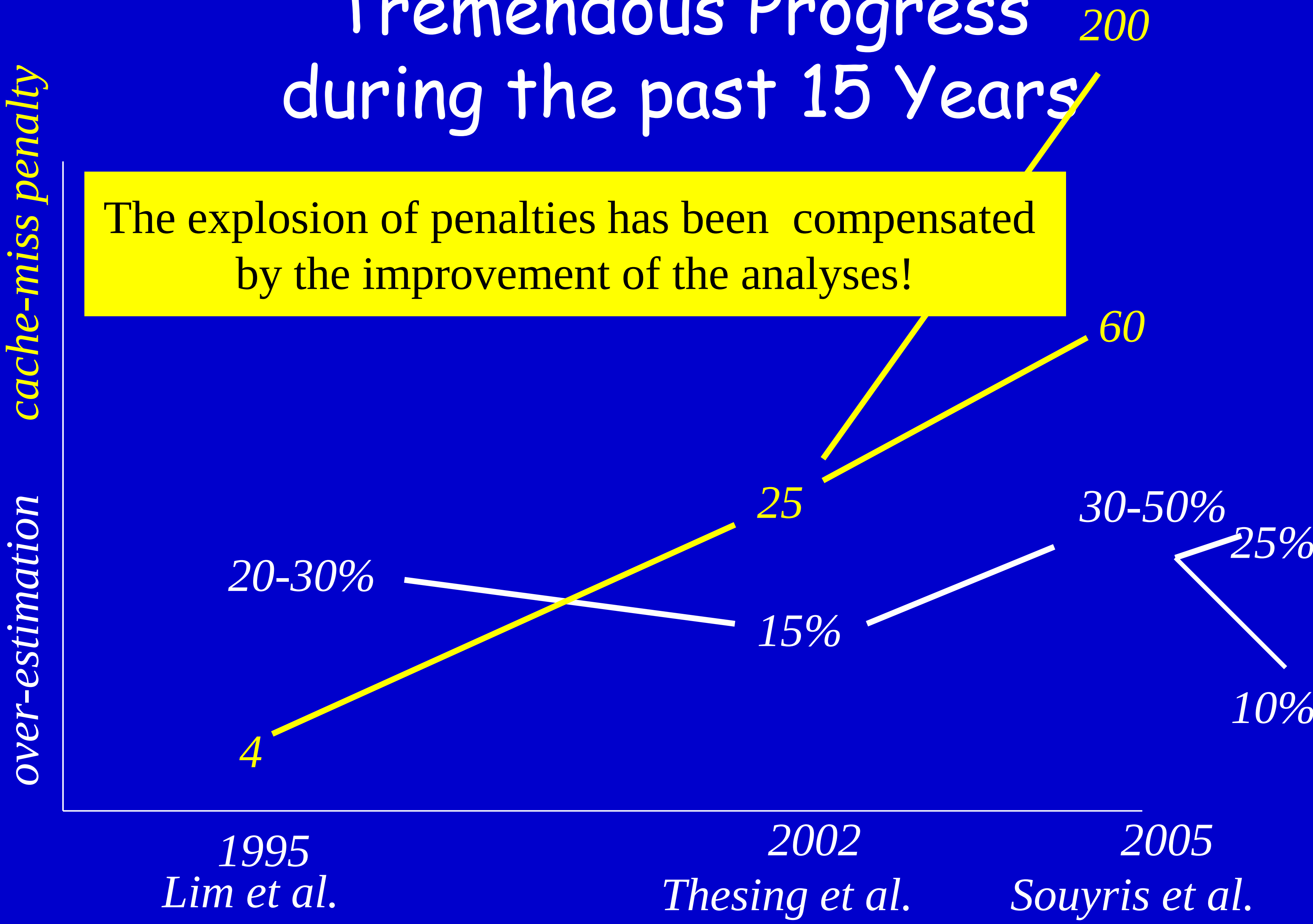
"The AbsInt tool is probably the best of its kind in the world and it is justified to consider this result as a breakthrough."

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 15 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!

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

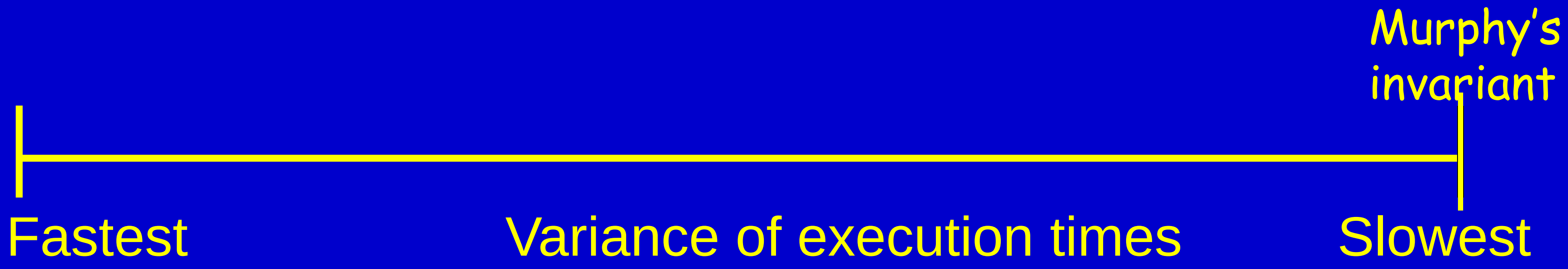
History-Sensitivity of Instruction Execution-Time

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 results from the execution history.
- Needed: an invariant about the set of execution states produced by all executions reaching a program point.

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.



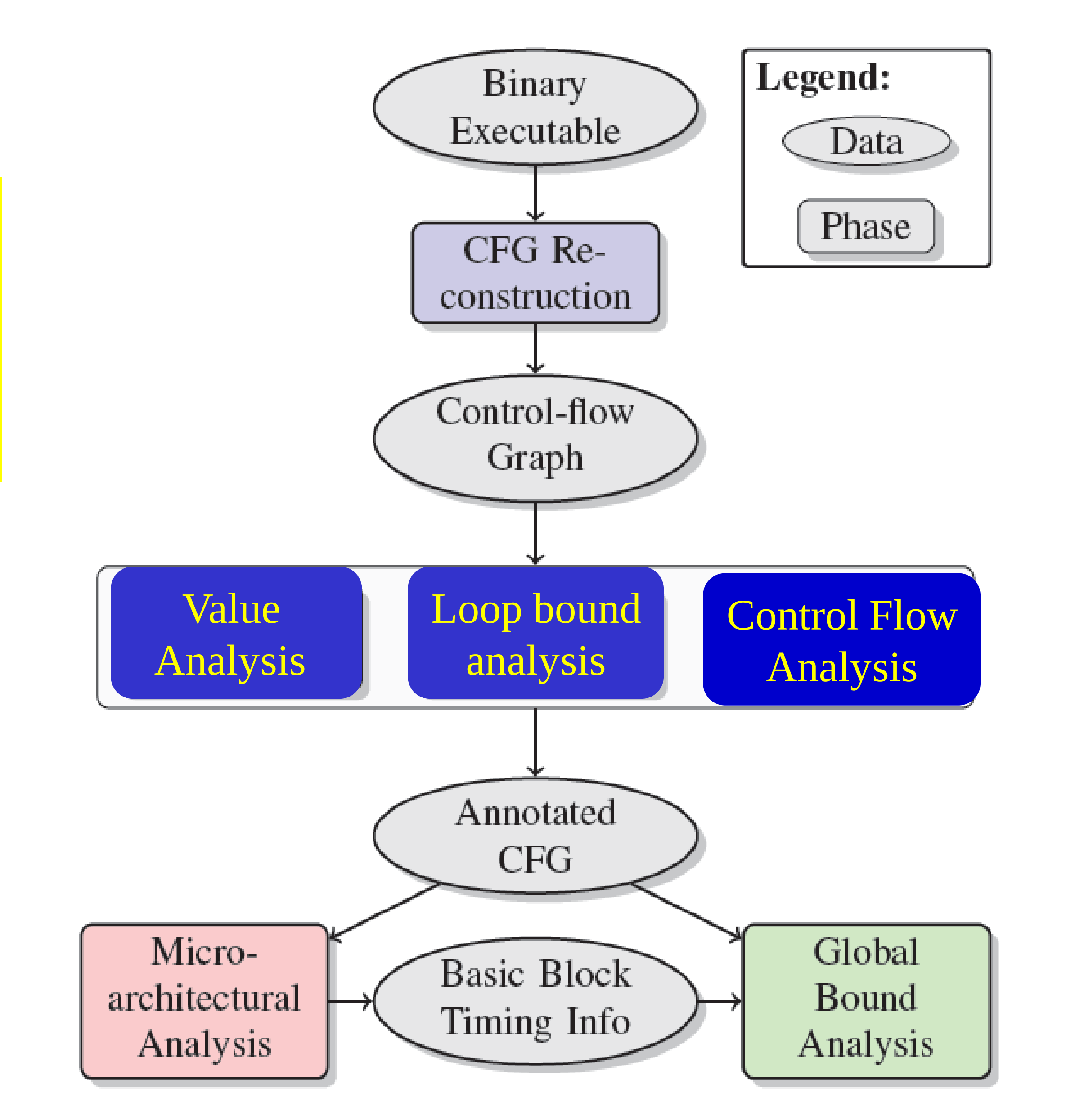
Tool Architecture

Determines enclosing intervals for the set of ters and , used for ddresses.

Determines

Determines infeasible paths

Abstract Interpretations

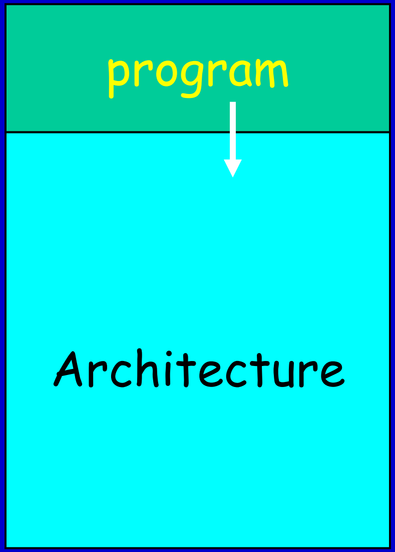


Abstract Interpretation

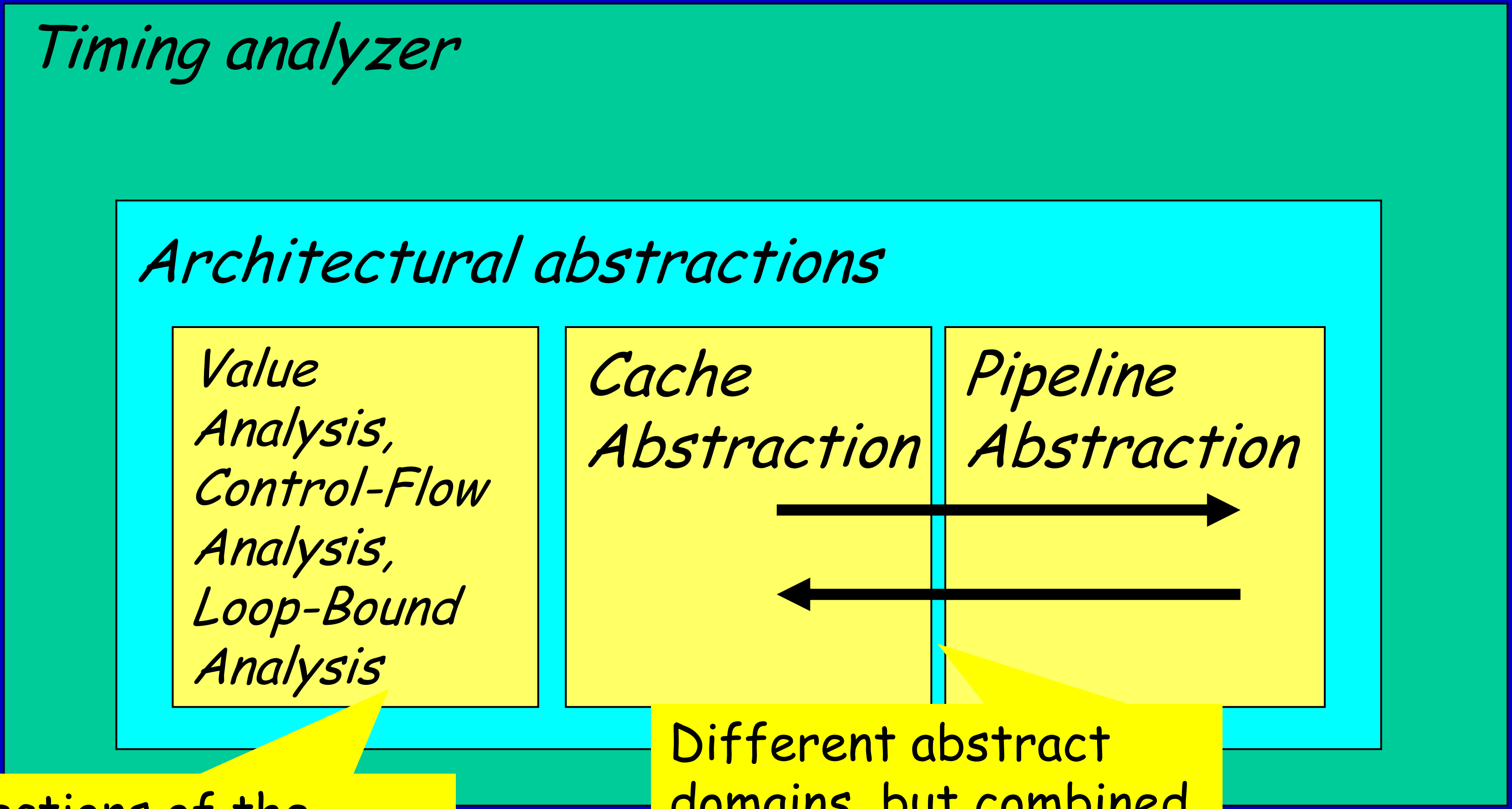
Integer Linear Programming

Abstract Interpretation in Timing Analysis

- Abstract interpretation is always based on the semantics of the analyzed language.
- A semantics of a programming language that refers to 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

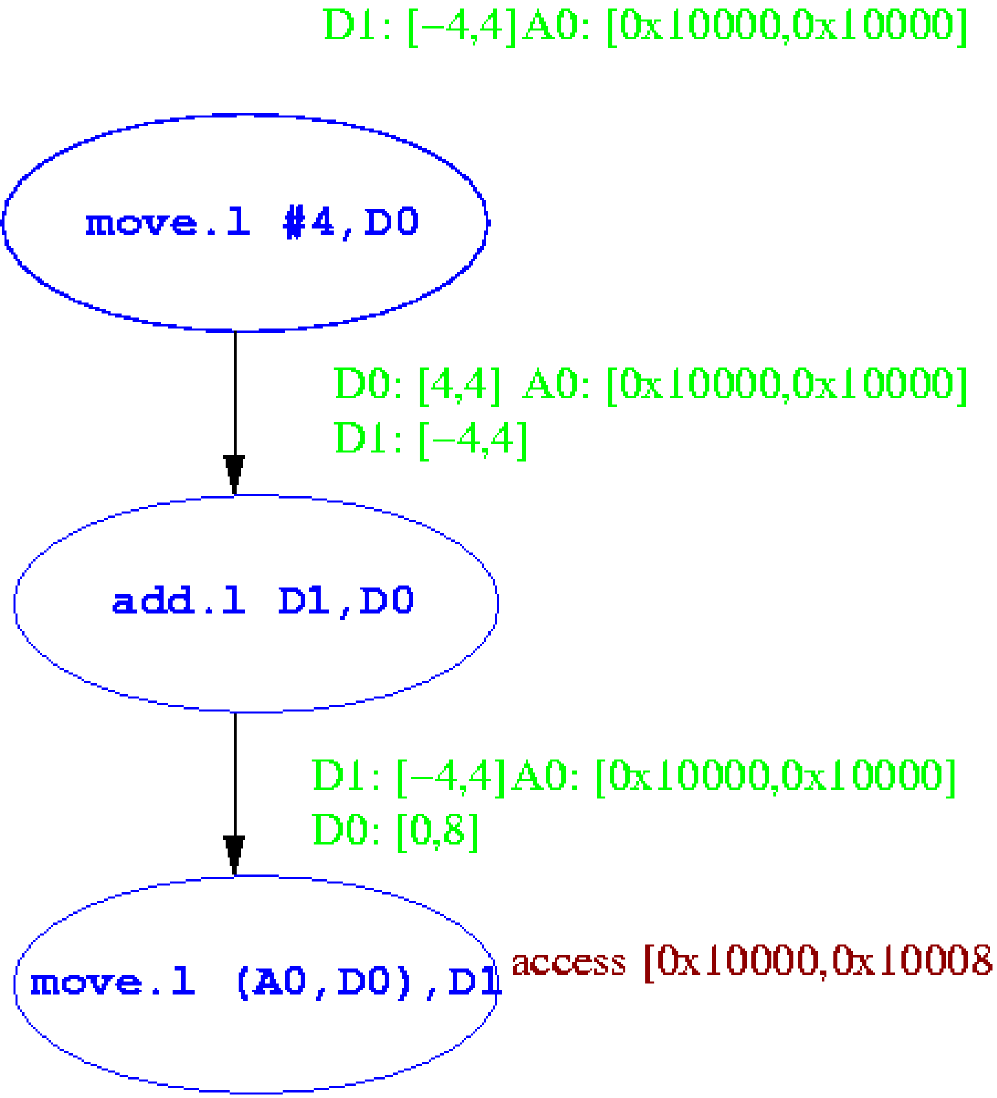
abstractions of the processor's arithmetic, separate analyses

Different abstract domains, but combined analyses due to cyclic dependences

Value Analysis

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

Value Analysis II



D1: [−4,4] A0: [0x10000,0x10000]

move.l #4,D0

D0: [4,4] A0: [0x10000,0x10000]
D1: [−4,4]

add.l D1,D0

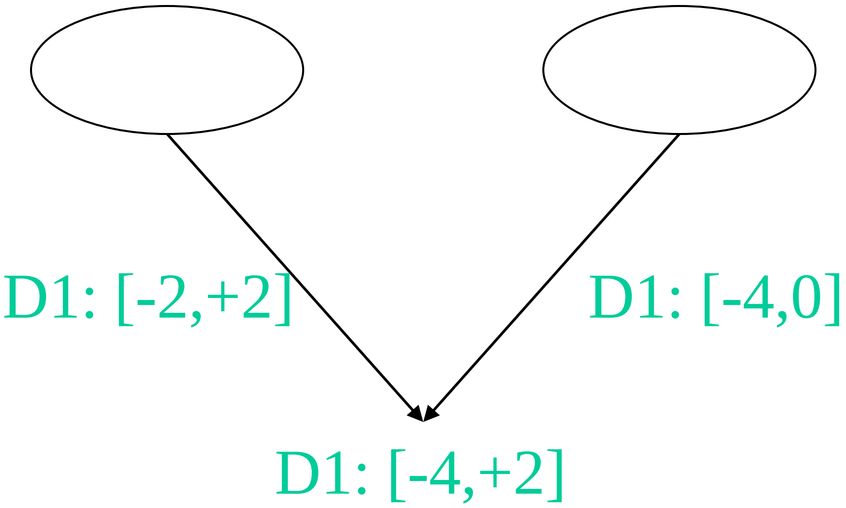
D1: [−4,4] A0: [0x10000,0x10000]
D0: [0,8]

move.l (A0,D0),D1

access [0x10000,0x10008

- Intervals are computed along the CFG edges

- At joins, intervals are „unioned“



D1: [-2,+2]

D1: [-4,0]

D1: [-4,+2]

Reducing Complex Domains

Components with domains of states $C_1, C_2, \ldots, C_k$

Analysis has to track domain $C_1 \times C_2 \times \ldots \times C_k$

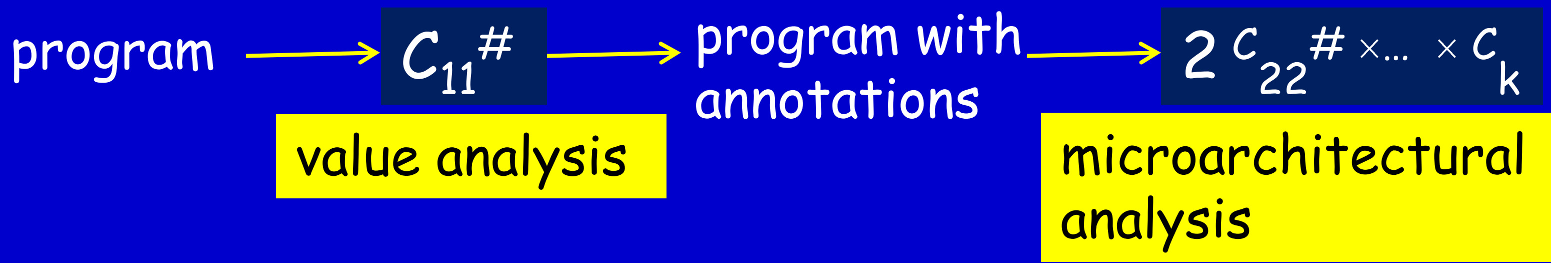
Start with the powerset domain $2^{C_1 \times C_2 \times \ldots \times C_k}$

Find an abstract domain $C_1^\#$ transform into $C_1^\# \times 2^{C_2 \times \ldots \times C_k}$

This has worked for caches and cache-like devices.

Find abstractions $C_{11}^\#$ and $C_{12}^\#$ factor out $C_{11}^\#$ and transform rest into $2^{C_{22}^\# \times \ldots \times C_k}$

This has worked for the arithmetic of the pipeline.



program → $C_{11}^\#$ → program with annotations → $2^{C_{22}^\# \times \ldots \times C_k}$

value analysis

microarchitectural analysis

Complexity Issues

Independent-attribute analysis

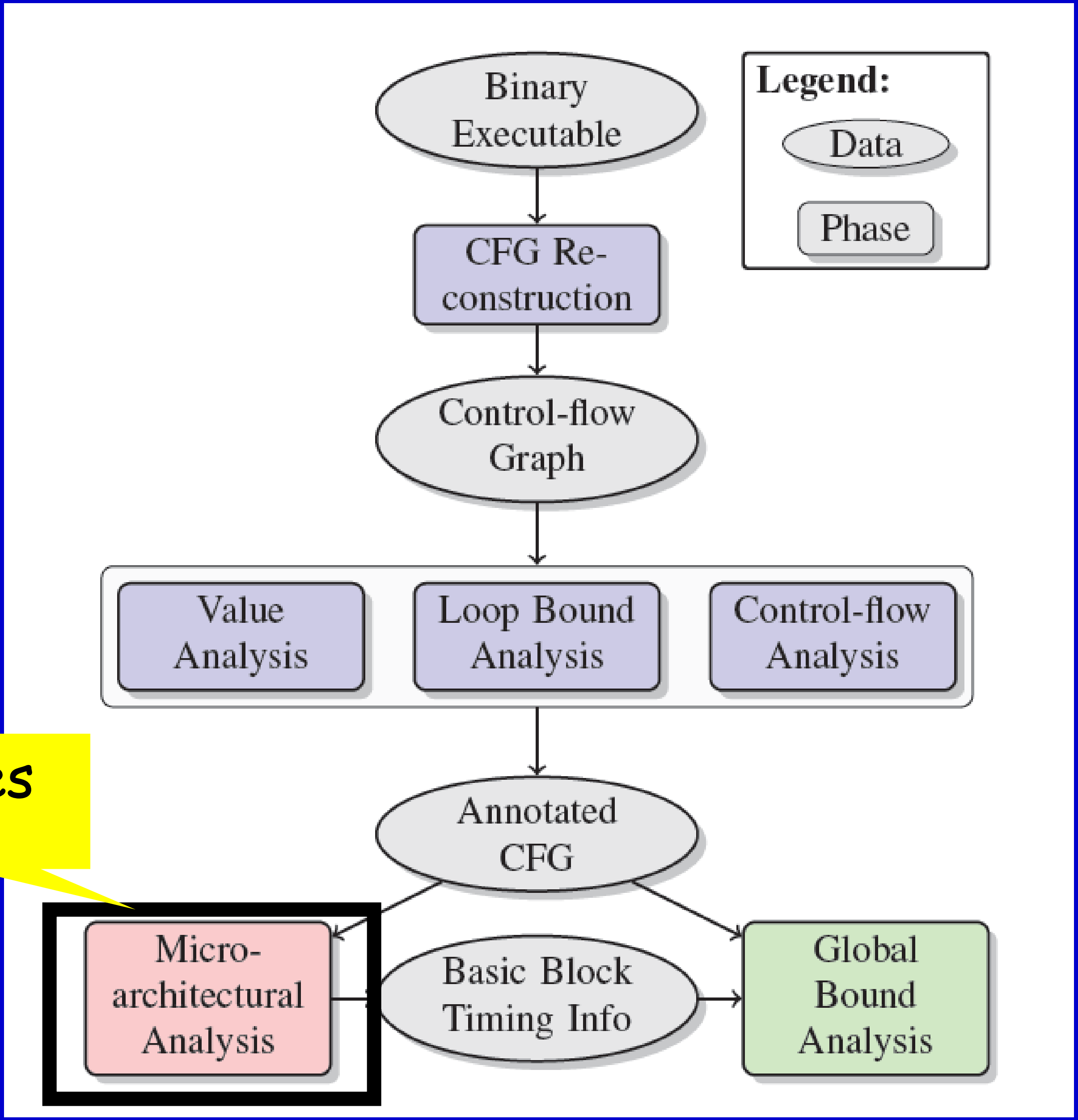
- Feasible for domains with no dependences or tolerable loss in precision
- Examples: value analysis, cache analysis
- Efficient!

Relational analysis

- Necessary for mutually dependent domains
- Examples: pipeline analysis
- Highly complex

Other parameters:
Structure of the underlying domain, e.g. height of lattice;
Determines speed of convergence of fixed-point iteration.

Tool Architecture



Abstract Interpretations

Caches

Abstract Interpretation

Integer Linear Programming

Caches: Small & Fast Memory on Chip

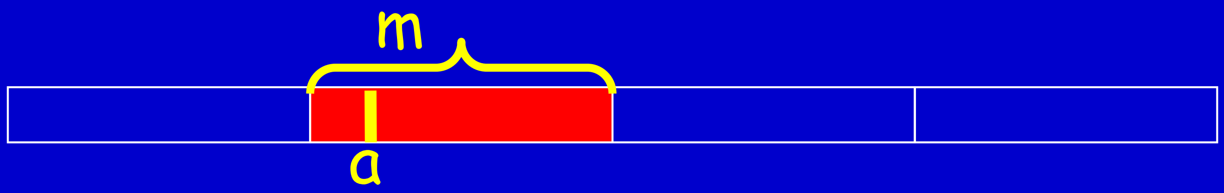
- Bridge speed gap between CPU and RAM
- Caches work well in the average case:
  - Programs access data locally (many hits)
  - Programs reuse items (instructions, data)
  - Access patterns are distributed evenly across the cache
- Cache performance has a strong influence on system performance!
- The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

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 strategy: LRU, PLRU, FIFO,...determine which line to replace in a full cache (set)

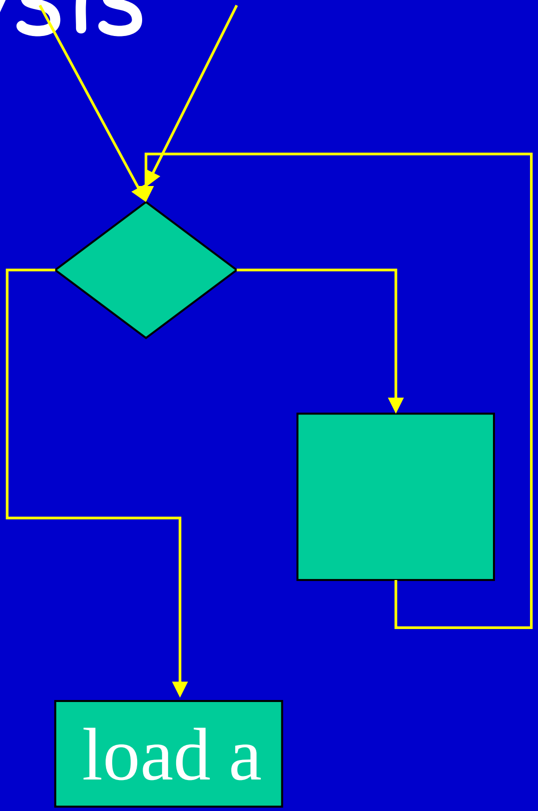
Cache Analysis

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.

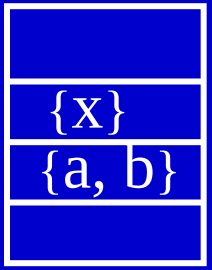
(Must) Cache Analysis

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



{x}

{a, b}

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

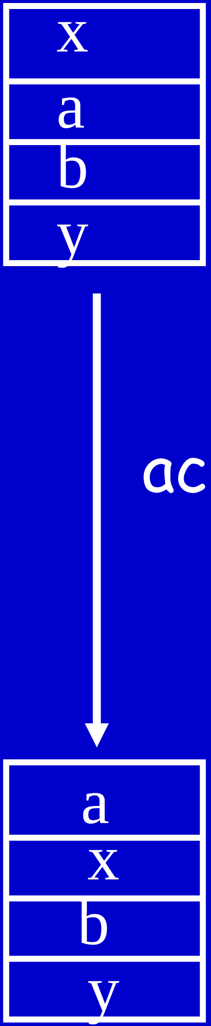
1. only memory blocks guaranteed to be in cache.
2. they are associated with their maximal age.

Cache 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 with LRU replacement

concrete transfer function (cache)

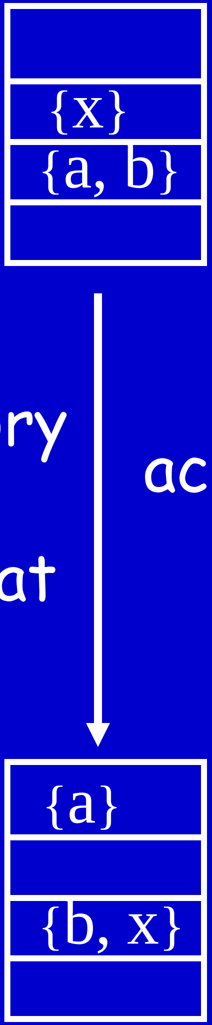


x
a
b
y

access to a

a
x
b
y

abstract transfer function (analysis)



{x}
{a, b}

access to a

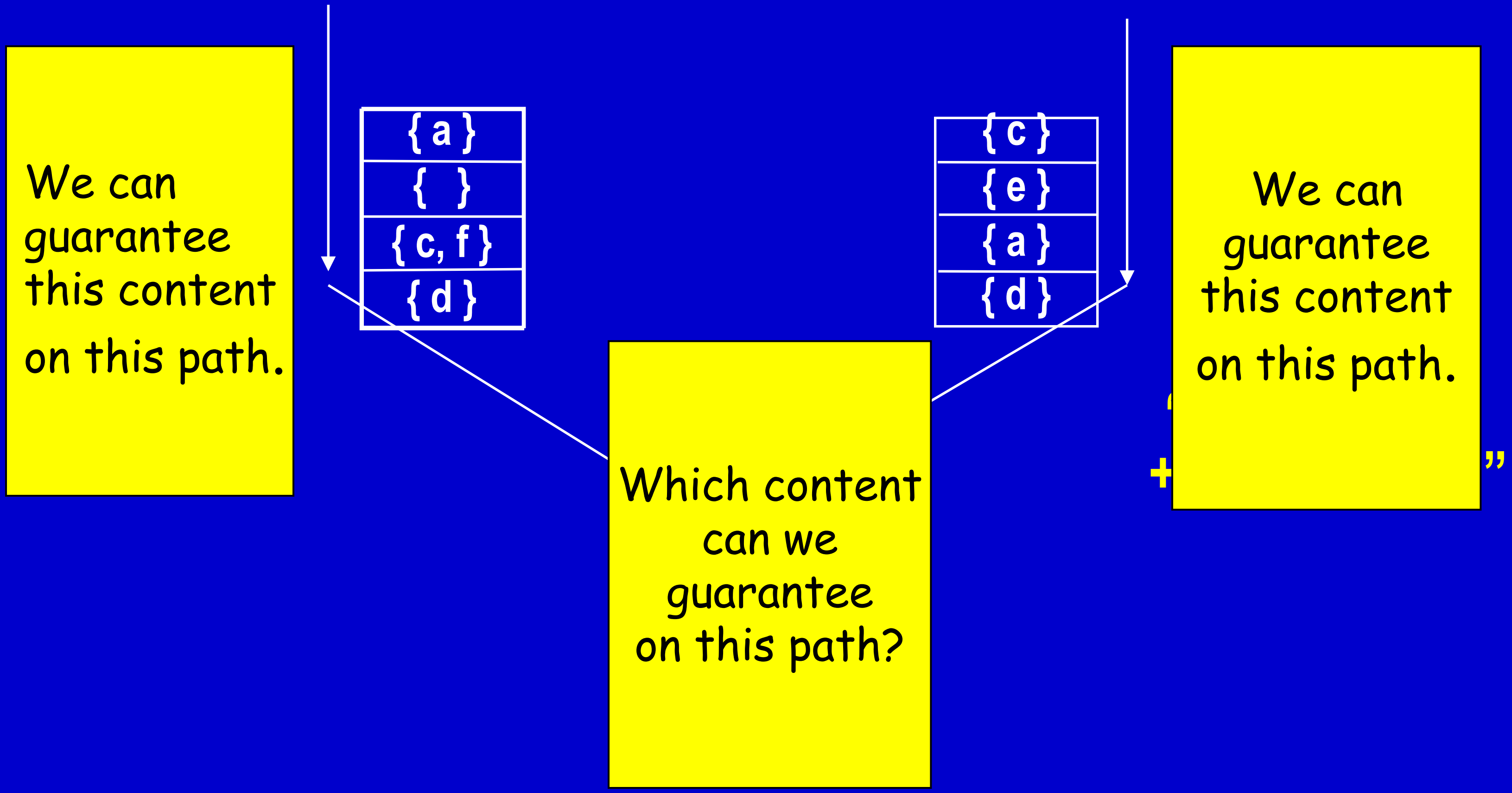
{a}

{b, x}

After the access to a,

a is the youngest memory block in cache,
and we must assume that x has aged.

What happens when control-paths merge?



We can guarantee this content on this path.

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

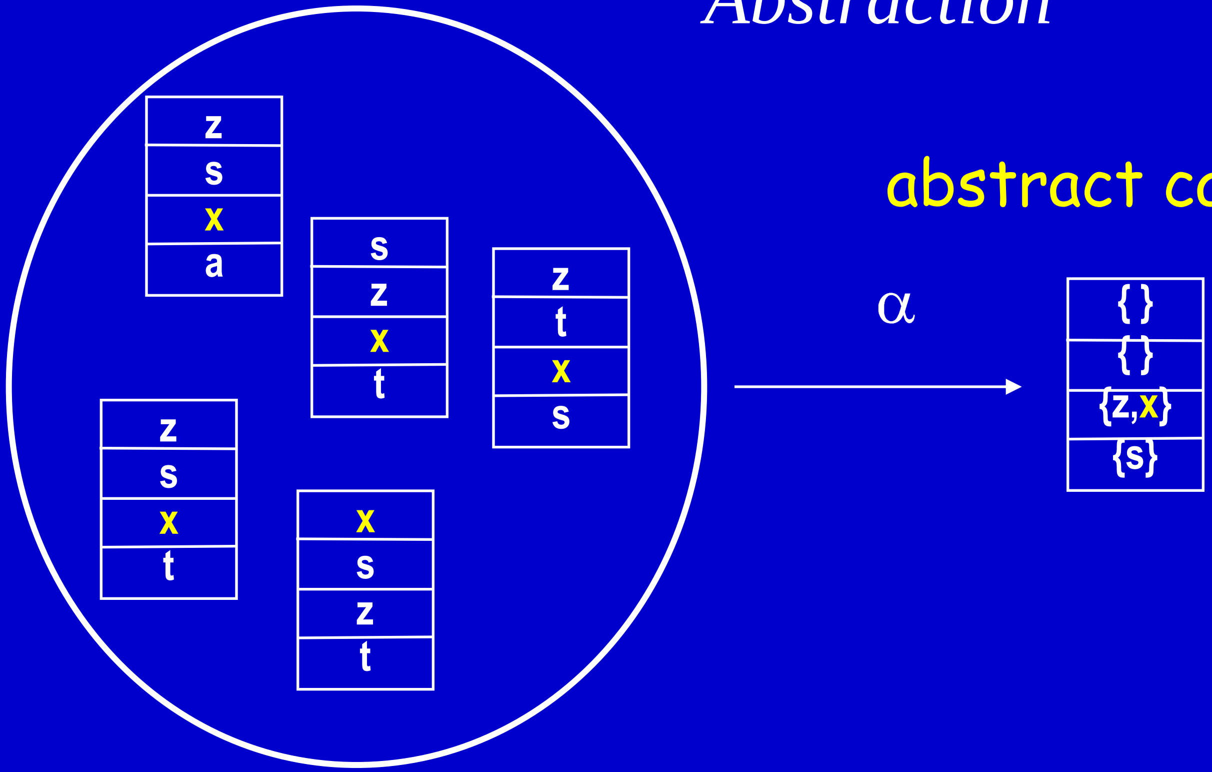
Abstract Domain: Must Cache

Representing sets of concrete caches by their description

concrete caches

Abstraction

abstract cache



z, s, x, a

s, z, x, t

z, t, x, s

z, s, x, t

x, s, z, t

α

{ }, { }, {z,x}, {s}

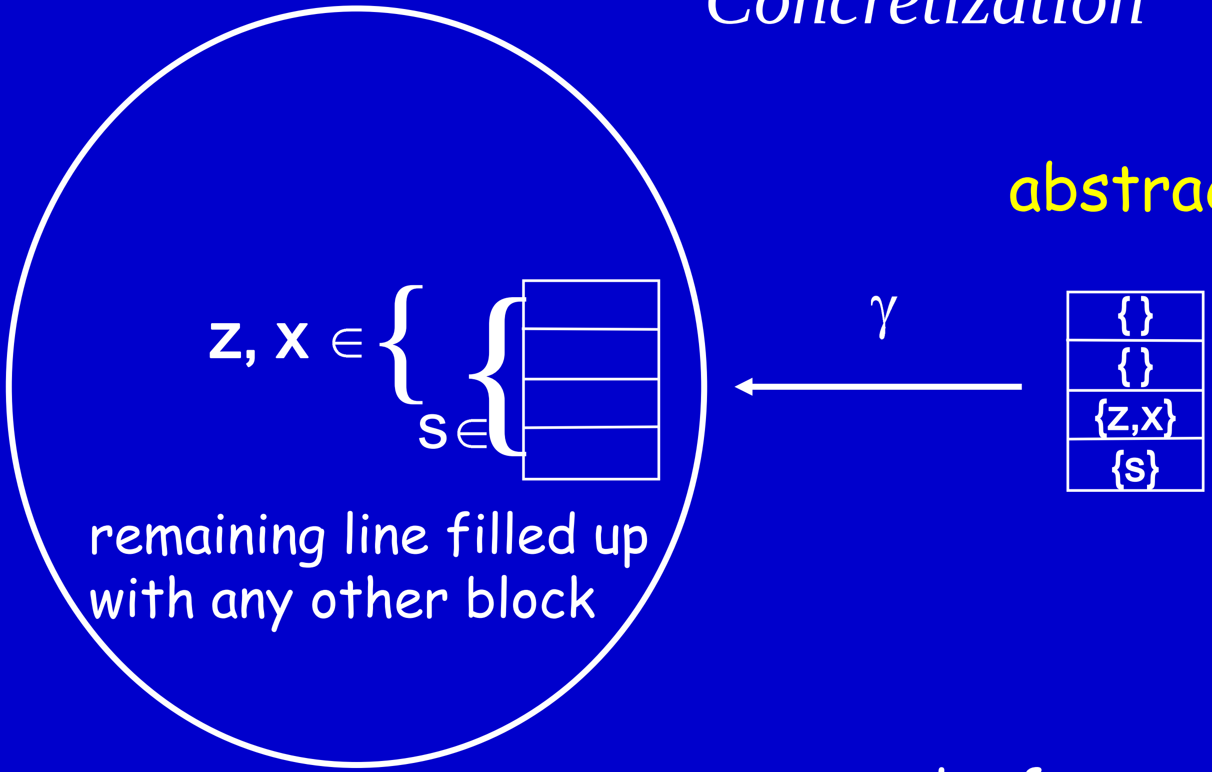
Abstract Domain: Must Cache

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

γ

{ }

{ }

{z,x}

{s}

α and γ form a Galois connection

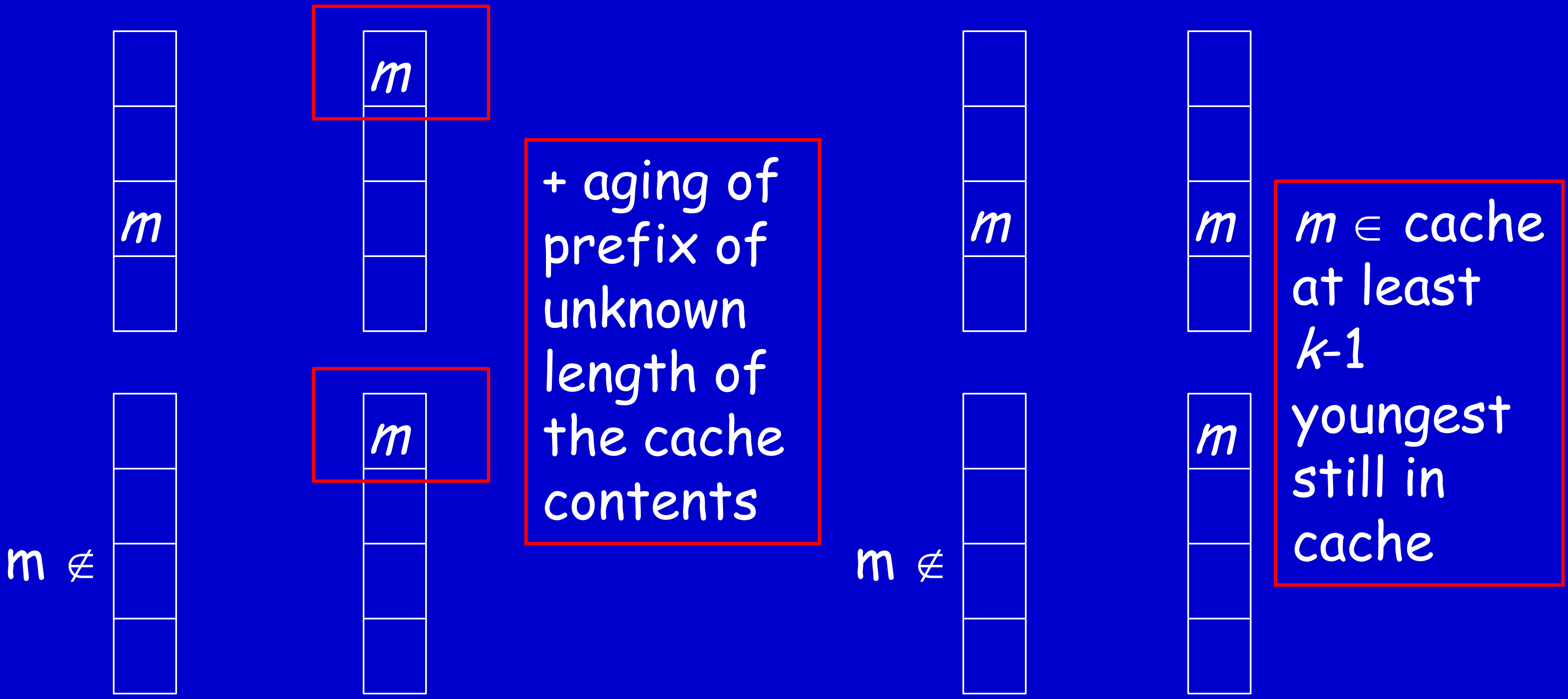
The Influence of the Replacement Strategy

(an excursion into the area of Predictability)

Information gain through access to *m*

LRU:

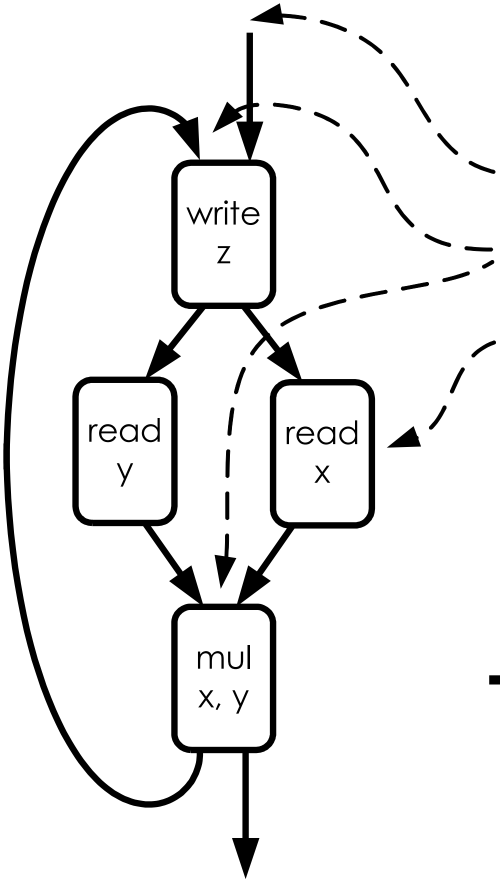
FIFO:



+ aging of prefix of unknown length of the cache contents

m ∈ cache at least *k*-1 youngest still in cache

Predictability of Caches
- Speed of Recovery from Uncertainty



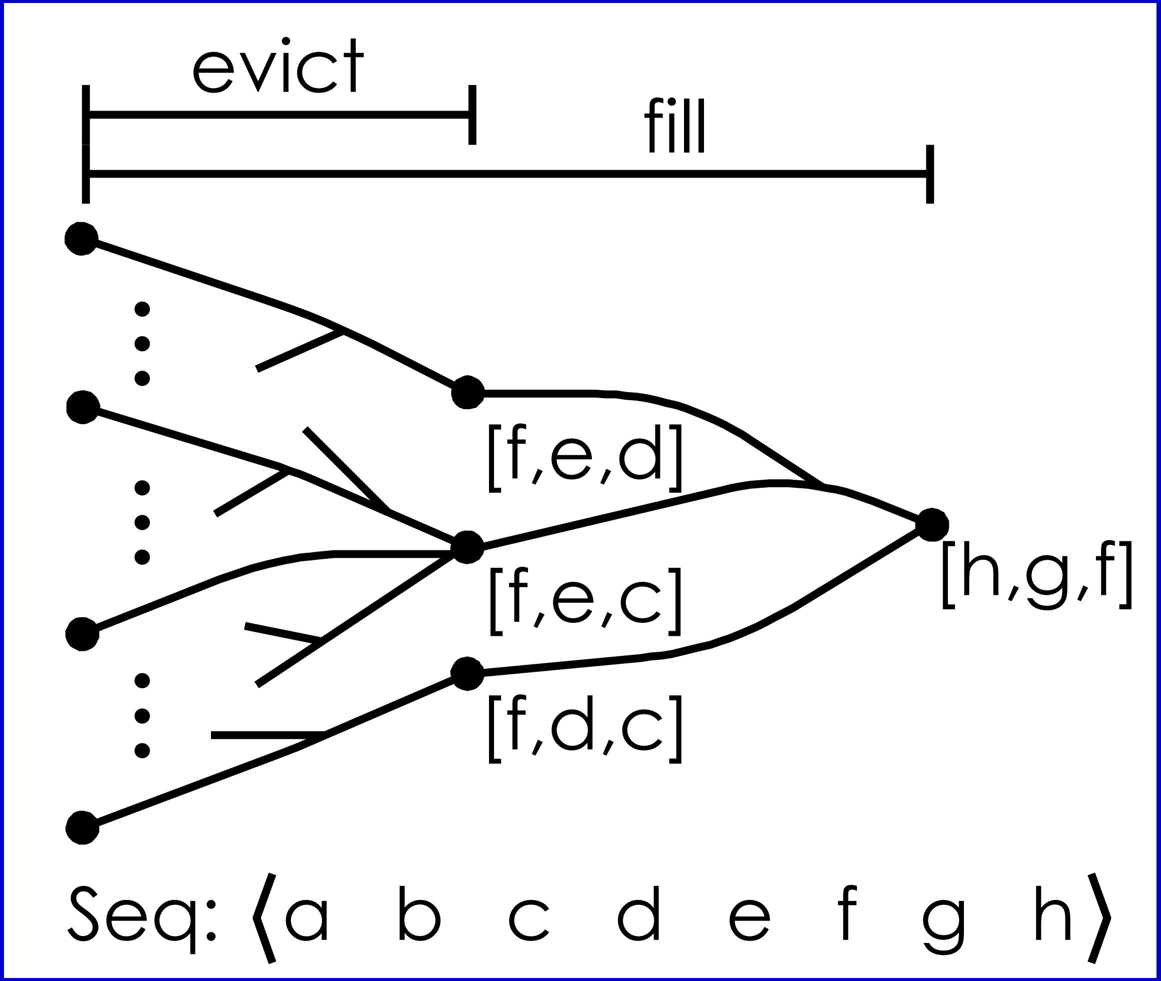
1. Initial cache contents?
2. Need to combine information
3. Cannot resolve address of x...
4. Imprecise analysis domain/
   update functions

→ Need to recover information:
Predictability = Speed of Recovery

J. Reineke et al.: Predictability of Cache Replacement Policies, Real-Time Systems, Springer, 2007

Metrics of Predictability:

evict & fill

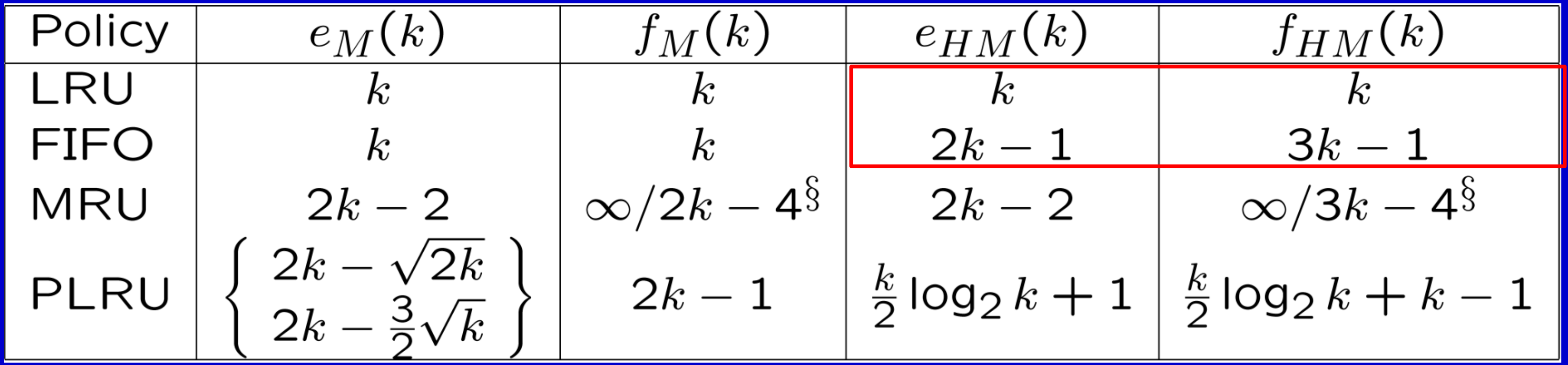


Two Variants:

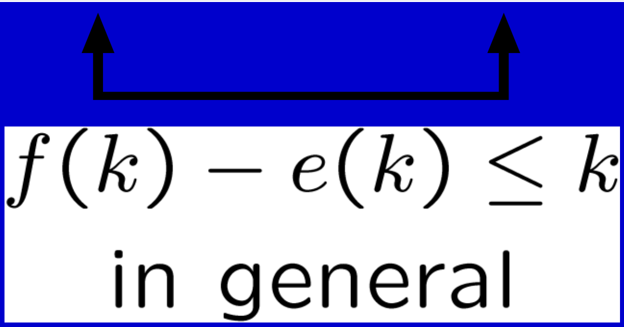
M = Misses Only

HM

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	$\left\{\begin{array}{c} 2k-\sqrt{2k} \\ 2k-\frac{3}{2}\sqrt{k} \end{array}\right\}$	$2k-1$	$\frac{k}{2}\log_2 k+1$	$\frac{k}{2}\log_2 k+k-1$

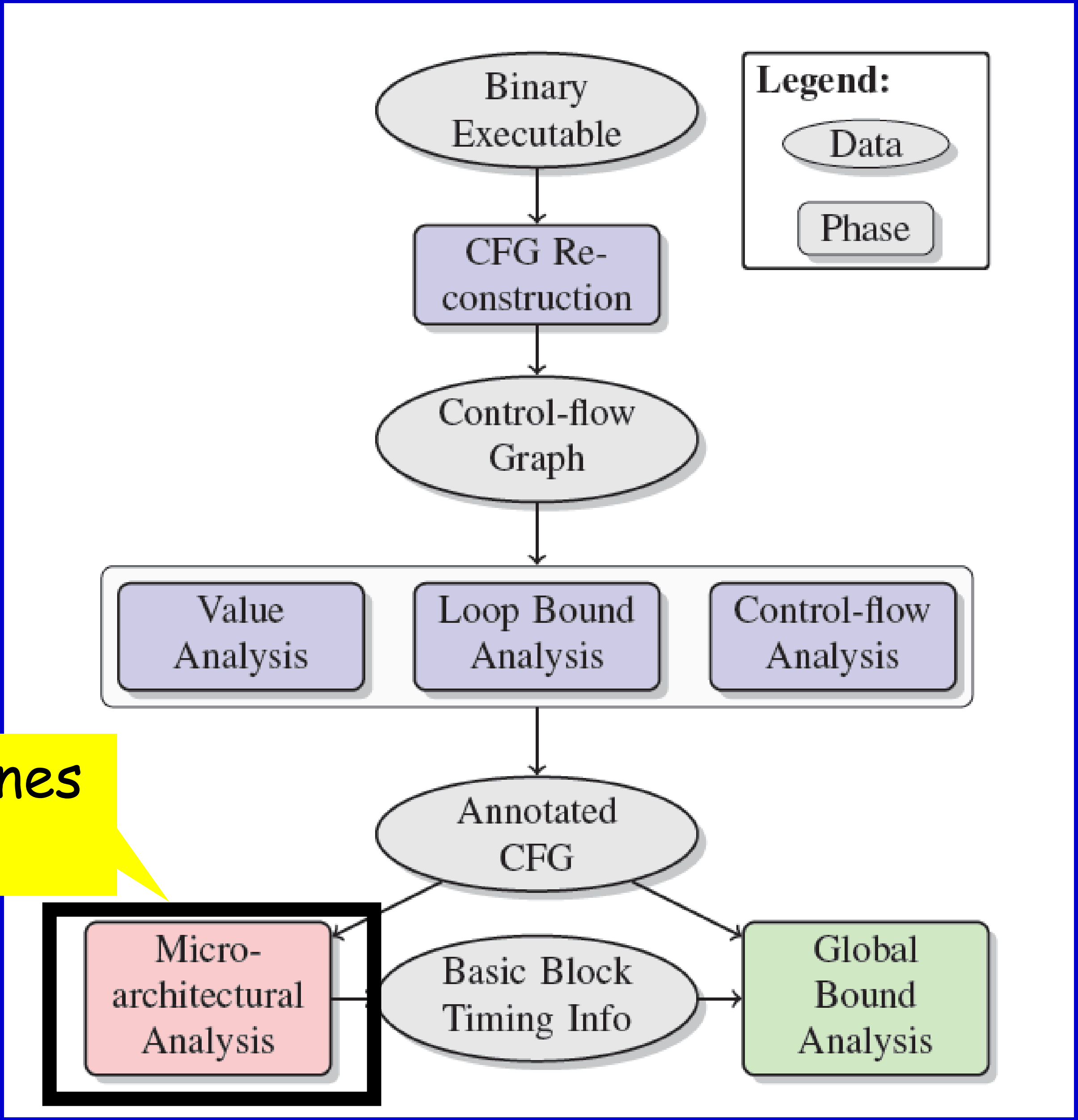


$$f(k)-e(k)\leq k$$

in general

Generic examples prove tightness.

Tool Architecture



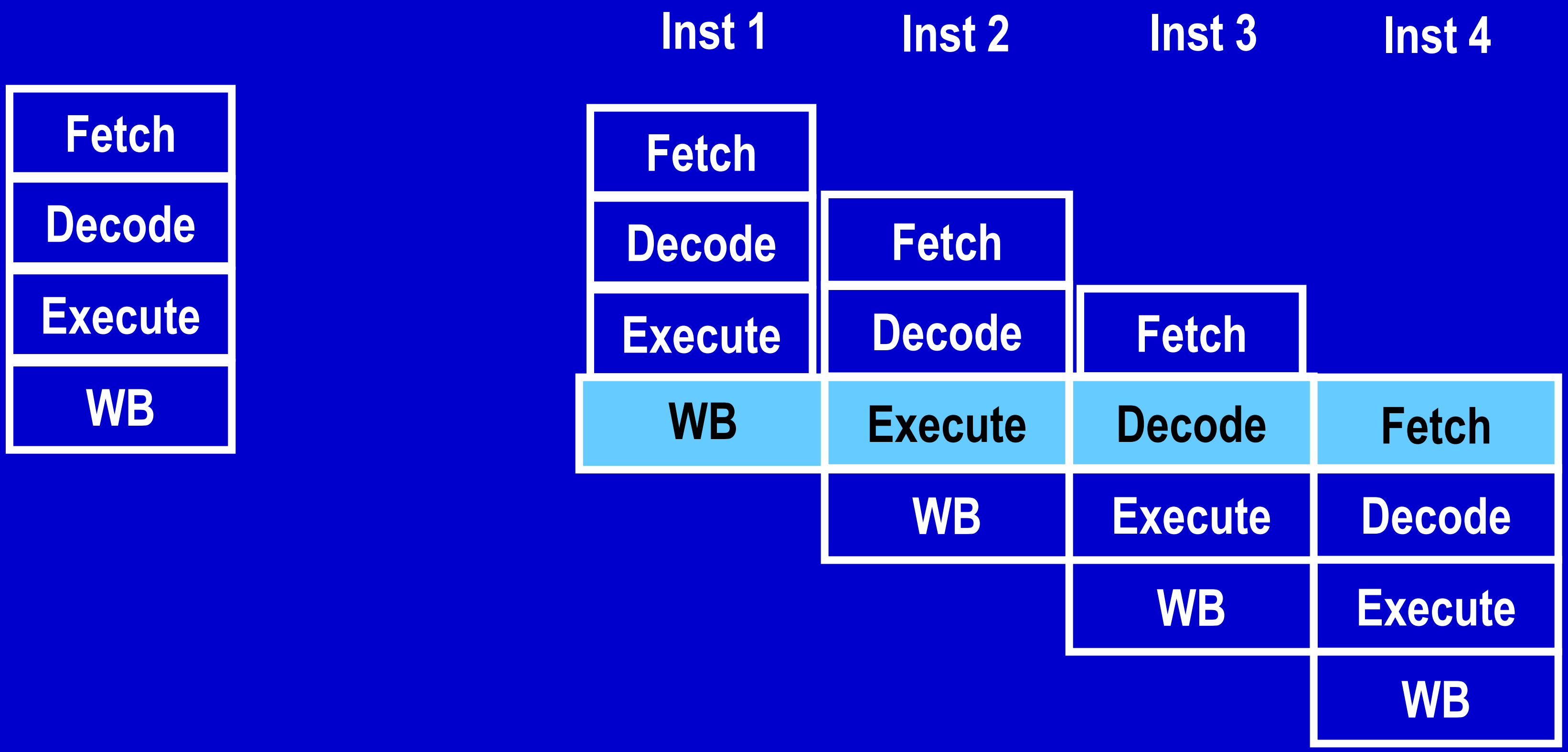
Abstract Interpretations

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 for a 1-issue pipeline: 1 Instruction per Cycle

Non-ideal Case: Pipeline Hazards

Several types:

- Data Hazards: Operands not yet available (Data Dependences)
- Resource Hazards: needed resource not in necessary state
  - Pipeline unit occupied
  - Bus occupied
  - Prefetch queue empty
  - Reorder buffer full
  - …
- Control Hazards: Conditional branch, direction unknown
- Cache Hazards: Instruction or operand fetch causes cache miss

In general different penalties!

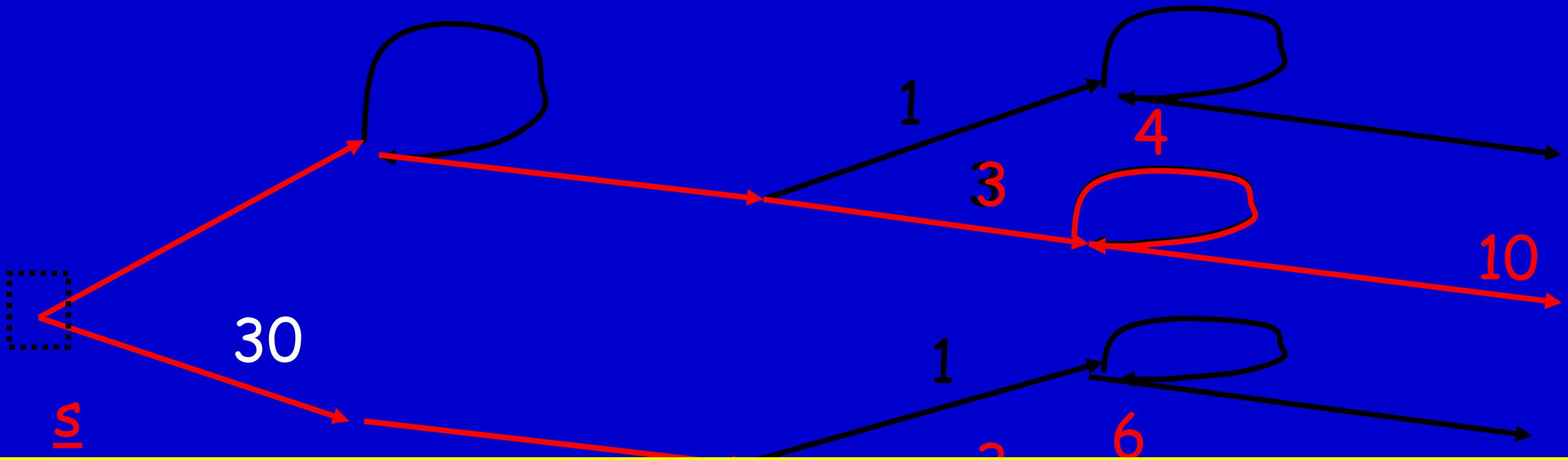
Abstract Instruction-Execution

Fetch
I-Cache miss?

Issue
Unit occupied?

Execute
Multicycle?

Retire
Pending instructions?



Interference between processor components leads to **Timing Anomalies**: Assuming local worst case leads to lower overall execution time and vice versa.
Ex.: Cache miss in the context of branch prediction

Designing a Pipeline-Analysis Domain

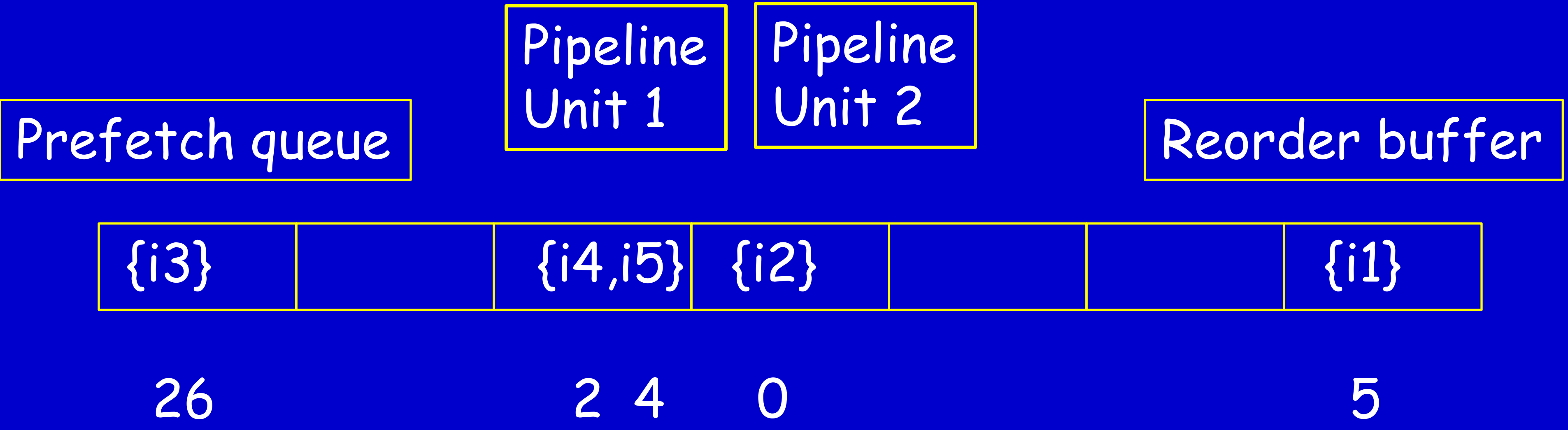
Let's try an analogy to the cache domain:

- Abstract pipeline state to describe sets of concrete pipeline states at a program point,
- should express how far the instructions of the basic block have certainly progressed into the pipeline stages:

{i3}		{i4,i5}	{i2}			{i1}

Abstract Transfer Functions

- What is needed for an abstract transfer?
- A counter for the stall cycles,
- State of prefetch queue and reorder buffer,
- Occupancy of pipeline units
- …



Prefetch queue		Pipeline Unit 1	Pipeline Unit 2			Reorder buffer
{i3}		{i4,i5}	{i2}			{i1}
26		2 4	0			5

Abstract Pipeline Domain ctd.

- Many components of concrete pipeline states have to be represented in abstract pipeline states,
- Component transitions depend on the state of other components,
- all make transitions every cycle,
  ⇒ need to keep track of these states in combination, no modular analysis ☹
- What is different about cache analysis?
  - Abstract caches code in a compact way all that is needed for a transition,
  - Cache analysis can be split off since it makes a transition every memory access and returns Hit/Miss to the pipeline

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
- State-space explosion! Fortunately only on the basic-block level.

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

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

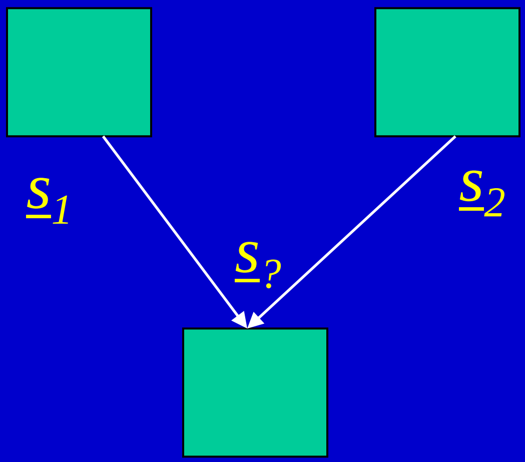
function <u>*exec*</u> (*b* : **basic block,** <u>*s*</u> : **abstract pipeline state)**
    <u>*t*</u>: **trace**

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

length(<u>t</u>) 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.*

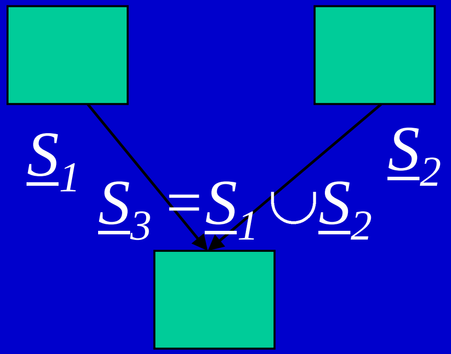
An Abstract Pipeline Executing a Basic Block - processor with timing anomalies -

function **analyze** **(*b* : basic block, $\underline{S}$: analysis state) $\underline{T}$: set of trace**

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

$\underline{PS}$ = set of abstract pipeline states

$\underline{CS}$ = set of abstract cache states



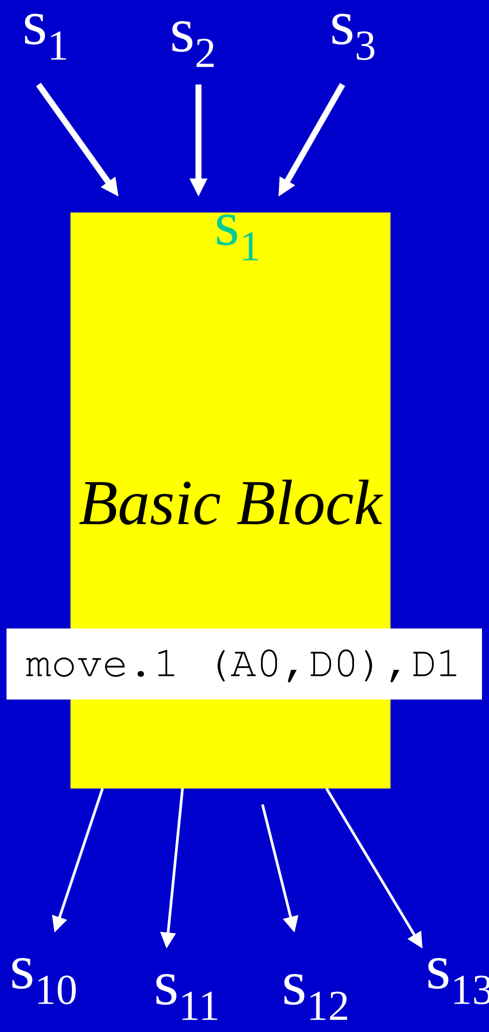
interprets instruction stream of *b* (annotated with cache information) starting in state $\underline{S}$ producing set of traces $\underline{T}$

max(length($\underline{T}$)) - upper bound for execution time

last($\underline{T}$) - set of initial states for successor block

Union for blocks with several predecessors.

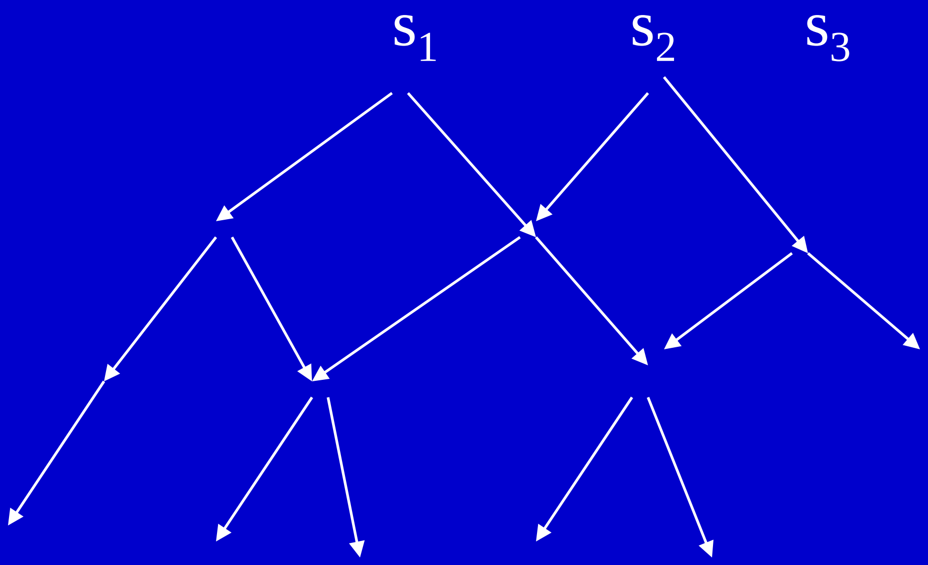
Pipeline Analysis: Overall Picture



```
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 each instruction



Reducing Complex Domains

Components with domains of states $C_1, C_2, \ldots, C_k$

Analysis has to track domain $C_1 \times C_2 \times \ldots \times C_k$

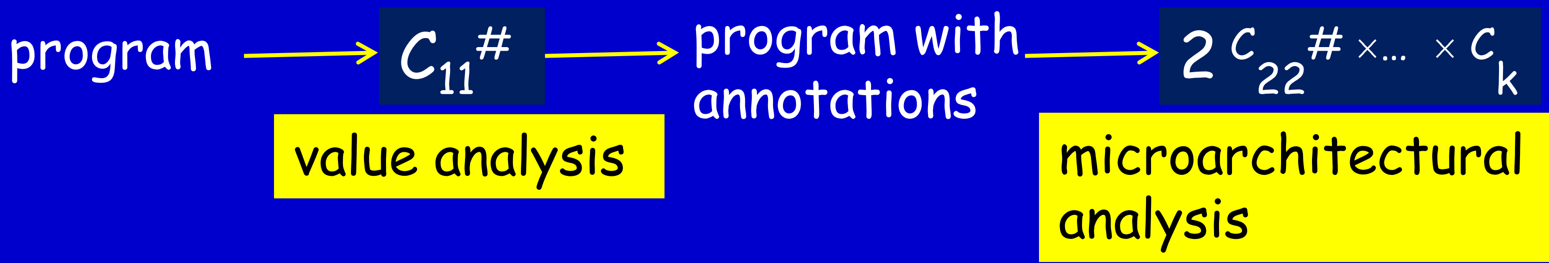
Start with the powerset domain $2^{C_1 \times C_2 \times \ldots \times C_k}$

Find an abstract domain $C_1^\#$ transform into $C_1^\# \times 2^{C_2 \times \ldots \times C_k}$

This has worked for caches and cache-like devices.

Find abstractions $C_{11}^\#$ and $C_{12}^\#$ factor out $C_{11}^\#$ and transform rest into $2^{C_{22}^\# \times \ldots \times C_k}$

This has worked for the arithmetic of the pipeline.

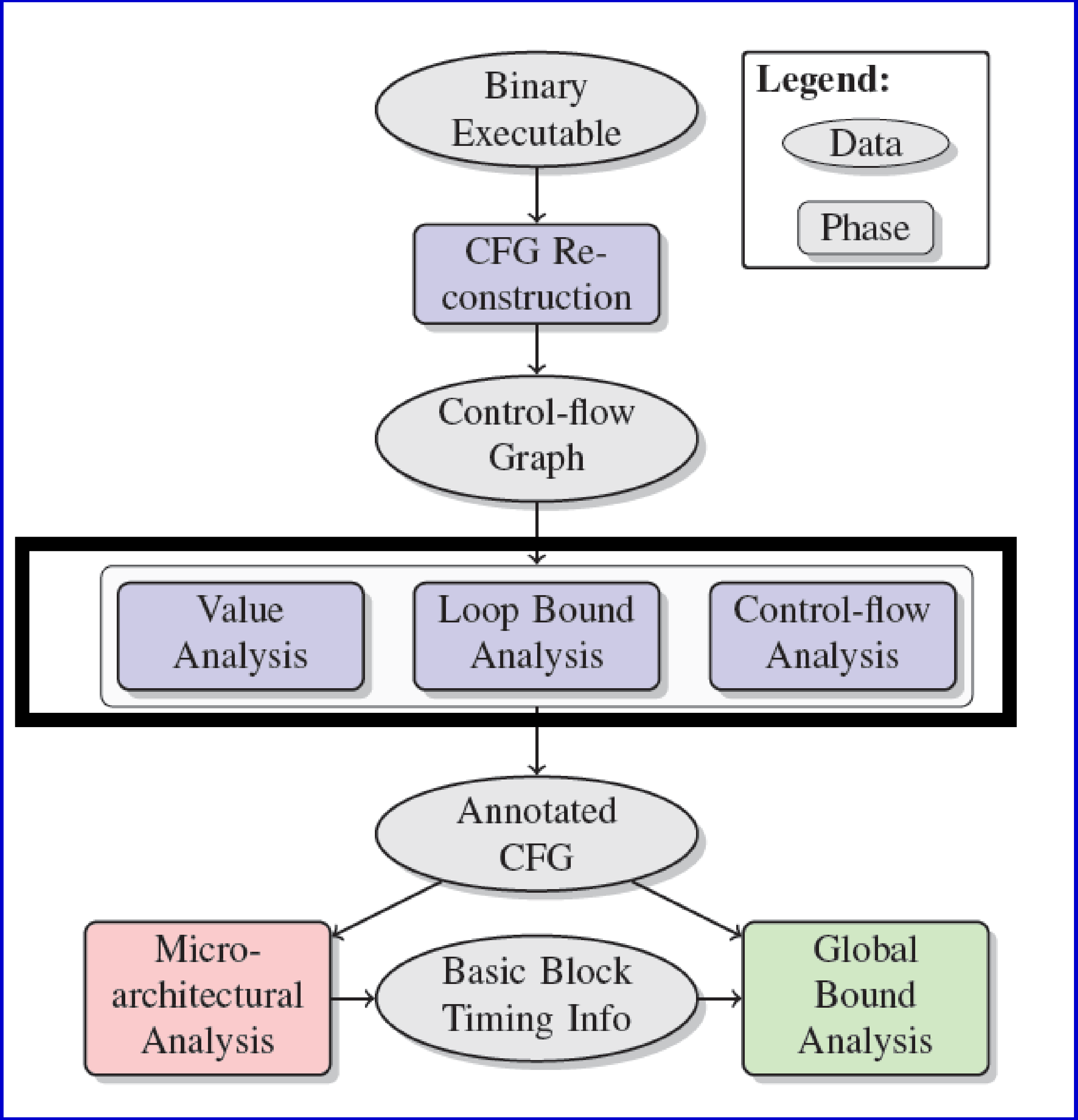


program → $C_{11}^\#$ → program with annotations → $2^{C_{22}^\# \times \ldots \times C_k}$

value analysis

microarchitectural analysis

Tool Architecture



Abstract Interpretations

Abstract Interpretation

Integer Linear Programming

Path Analysis

by Integer Linear Programming (ILP)

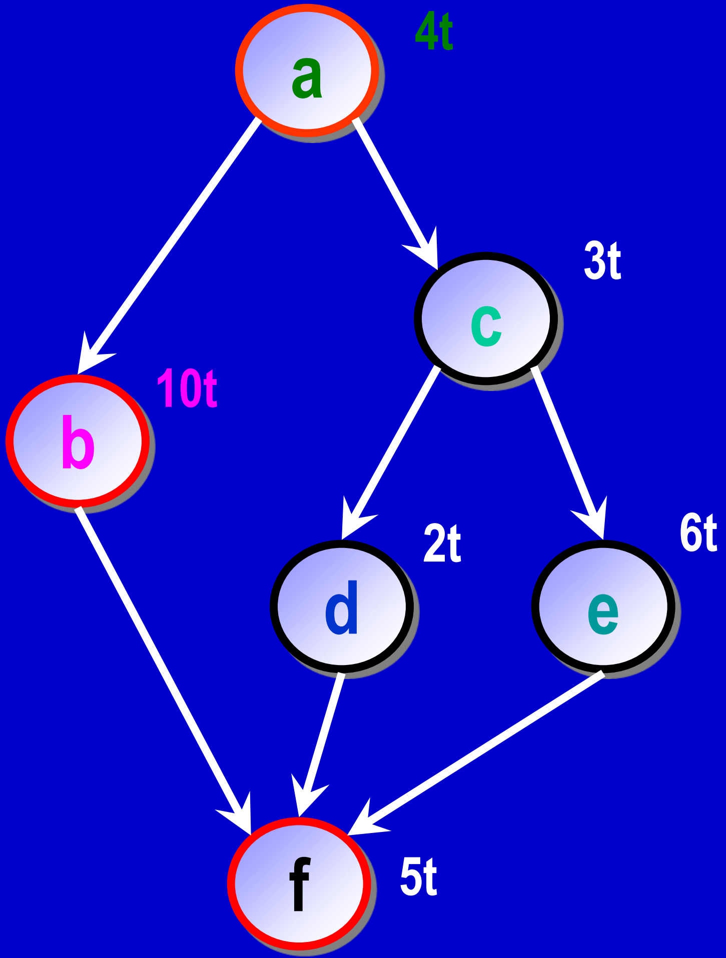
- Execution time of a program =

$$\sum_{\text{Basic\_Block } b} \text{Execution\_Time}(b) \times \text{Execution\_Count}(b)$$

- ILP solver maximizes this function to determine the execution-time bound
- 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
```



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

where

$$x_a = x_b + x_c$$

$$x_c = x_d + x_e$$

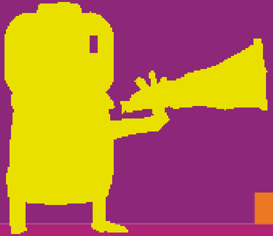
$$x_f = x_b + x_d + x_e$$

$$x_a = 1$$

Value of objective function: 19

x_a	1
x_b	1
x_c	0
x_d	0
x_e	0
x_f	1

HELMUT SEIDL
REINHARD WILHELM
SEBASTIAN HACK



Übersetzerbau

Analyse und Transformation

Dieses Buch behandelt die Optimierungsphase von Übersetzern. In dieser Phase werden Programme zur Effizienzsteigerung transformiert. Damit die Semantik der Programme bei diesen Transformationen erhalten bleibt, müssen jeweils zugehörige Anwendbarkeitsbedingungen erfüllt sein. Diese werden mittels statischer Analyse der Programme überprüft. In diesem Buch werden Analysen und Transformationen imperativer und funktionaler Programme systematisch beschrieben. Neben einer detaillierten Beschreibung wichtiger Optimierungen bietet das Buch eine knappe Einführung in die erforderlichen Konzepte und Methoden zur operationalen Semantik, zu vollständigen Verbänden und Fixpunktalgorithmen.

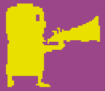
eXamen.press

➔ Technische Informatik
➔ Studierende und Praktiker

ISSN 1614-5216
ISBN 978-3-642-03329-2

9 783642 033292

❯ springer.de



SEIDL
WILHELM
HACK

Übersetzerbau Analyse und Transformation

HELMUT SEIDL
REINHARD WILHELM
SEBASTIAN HACK

Übersetzerbau

Analyse und Transformation

eXamen.press

Springer

Ongoing and Future Work

- Integrate preemption costs into the tools (AbsInt)
- Analyze heap-manipulating programs
- Clarify the notion of Predictability (FP7 IST project PREDATOR)
- Develop a discipline **Design for Predictability**
- Construct a predictable multi-processor platform for embedded systems

Relevant Publications (from my group)

- *C. Ferdinand et al.: Reliable and Precise WCET Determination of a Real-Life Processor*, EMSOFT 2001
- *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
- *M. Langenbach et al.: Pipeline Modeling for Timing Analysis*, SAS 2002
- *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*
- *R. Wilhelm: Determination of Execution-Time Bounds, CRC Handbook on Embedded Systems, 2005*
- *J. Reineke et al.: Predictability of Cache Replacement Policies,* Real-Time Systems, Springer, 2007
- *Reinhard 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 et al.: Memory Hierarchies, Pipelines, and Buses for Future Architectures in Time-critical Embedded Systems,* IEEE TCAD, July 2009