

# The Limits to Testing

Reinhard Wilhelm

Universität des Saarlandes



UNIVERSITÄT  
DES  
SAARLANDES



# Reinhard Wilhelm

- studied math, physics and mathematical logic at Westfälische Wilhelms-Universität Münster and Informatics at Technische Universität München and Stanford University
- 1977 Dr. rer. nat TU Munich
- **1978 – 2014 Professor at Saarland University**
- 1990 – 2014 Scientific Director of the Leibniz Center for Informatics at Schloss Dagstuhl
- **1998 cofounder of AbsInt**
- 2000 Fellow of the ACM
- 2006 Alwin-Walther medal from TU Darmstadt and Fraunhofer-Institut für Graphische Datenverarbeitung
- 2007 Prix Gay-Lussac-Humboldt from the French Minister of Education and Research
- 2008 Member of the European Academy of Sciences (Academia Europaea)
- 2008 Honorary doctorates from RWTH Aachen and Tartu University
- 2009 Konrad Zuse Medal from Gesellschaft für Informatik
- 2010 Bundesverdienstkreuz am Bande (Federal Order of Merit)
- 2010 ACM Distinguished Service Award
- 2013 Member of the German National Academy of Sciences Leopoldina

# Background

- Time drift in Patriot rockets in 1991 [Rounding error]
- Crash of railway switch controller 1995 in Hamburg-Altona [Stack overflow]
- Explosion of Ariane 5 rocket 1996 [Arithmetic overflow]
- AECL Medical Therac-25 incident (1985-1987) [Arithmetic overflow]
- Toyota Unintended Acceleration Accidents (2000 – 2010) [Stack overflow]

# Structure of the Talk

- Formal Verification Methods
- Testing
- Limits to Testing
- Performance Testing vs. Verification of Real-Time Requirements

# Formal Verification of Safety-Critical Systems

Given the task to verify some **correctness statement** about a program

- the **functional correctness** of a program,
- the absence of **run-time errors**,
- the satisfaction of **space constraints**, or
- the satisfaction of **timing constraints**

non-functional  
correctness  
properties

**Verification** is used here in a strong sense!

We look for **guarantees**.

No doubts about verified correctness claims!

- What are the alternatives?
- Question: What are you doing in practice?

# Approaches to Functional Verification

Starting Point:

**Correctness statements**

concern all (or some  
specified subset of)  
behaviours of  
the program



all behaviours of  
the program

# Deductive Verification

Functional correctness proved by **Theorem Proving**

- done for highly safety-critical systems, e.g.

- PKI key gen.,
- Software of Paris Metro,
- CompCert, verified C compiler

**Problem: not automatable**

requires complex user interaction, realistic in academic setting or with highly skilled verification specialists



all behaviours of  
the program

a correct compiler is the mother of all verification  
and testing activities on the source level!

# Testing

(for functional and non-functional properties)

- Test the program on a number of inputs

- **coverage criteria**

increase the  
chance to  
find bugs

Question:

Program free of a certain type of error?

- **false positive**: answer wrongly “Yes”
- **false negative**: answer wrongly “No”

all behaviours of  
the program

test cases

Undetected fault:  
False positive

Detected fault

## Limitation:

Program testing  
can be used to  
show the  
presence of bugs,  
but never to show  
their absence!  
Dijkstra (1970)



# Bug Chasing

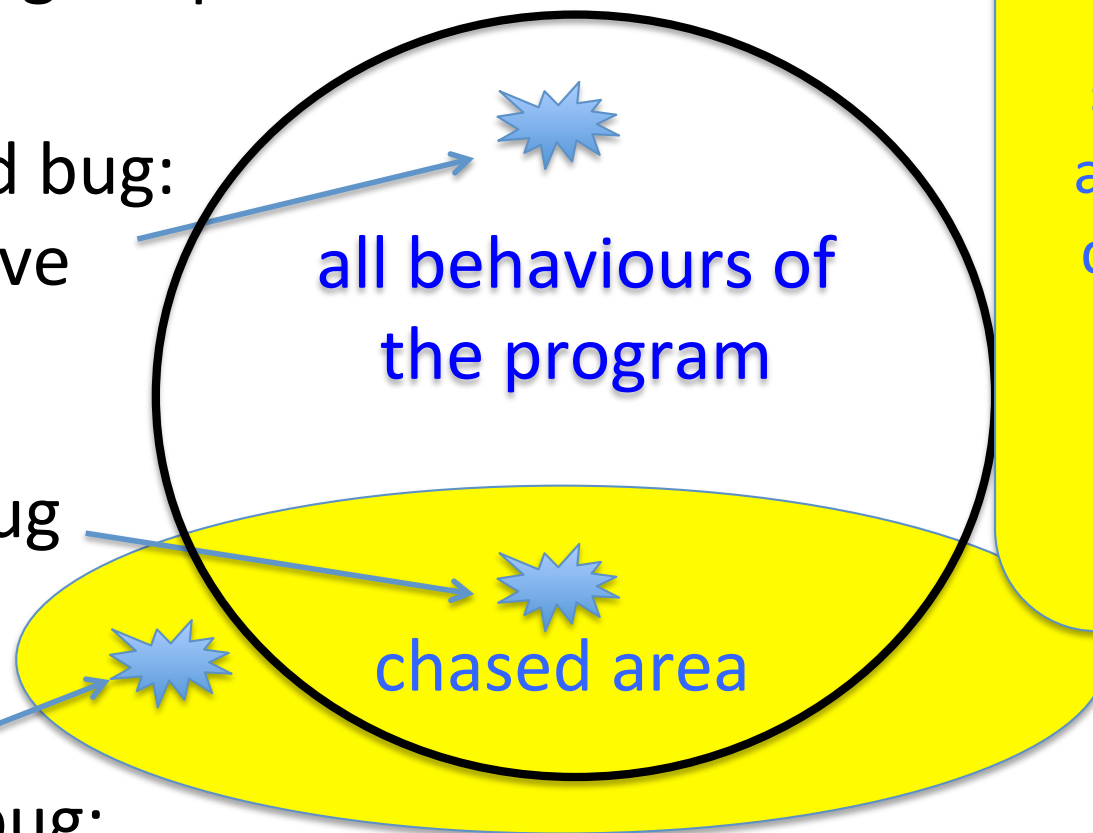
Question: Is program free of bugs?

Attempt to (quickly) find  
as many bugs as possible

Undetected bug:  
False positive

Detected bug

Reported bug:  
False negative (false alarm)



Bug chasing tools,  
often called  
**static analysers**  
are **unsound**, i.e.  
do not detect all  
bugs, and often  
**imprecise**, i.e.,  
produce many  
false negatives

# Model Checking

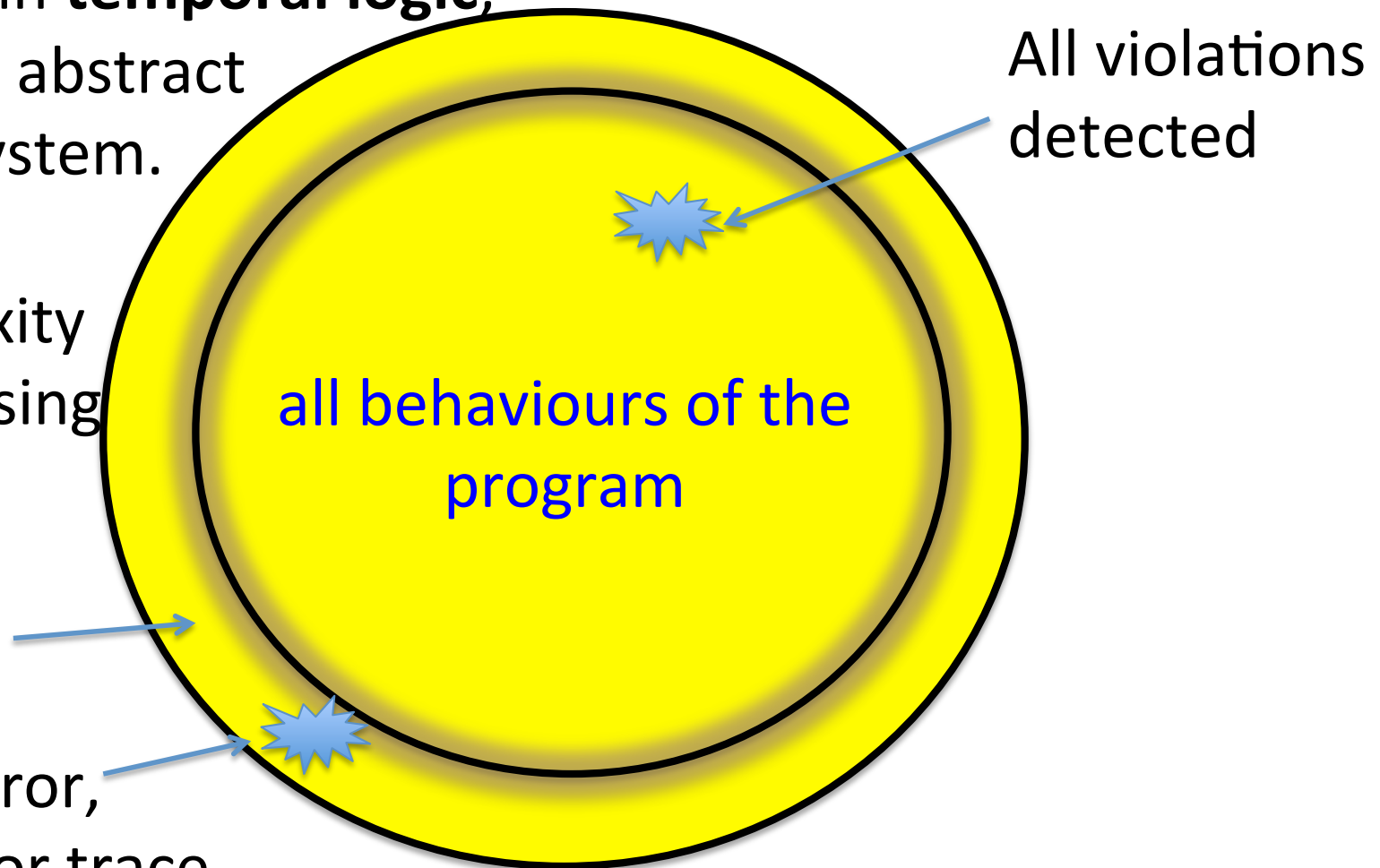
**Safety properties**, something bad cannot happen,  
formalized in **temporal logic**,  
checked on abstract  
model of system.

Problem:

- Complexity
- Bug chasing instead

Infeasible  
behaviours

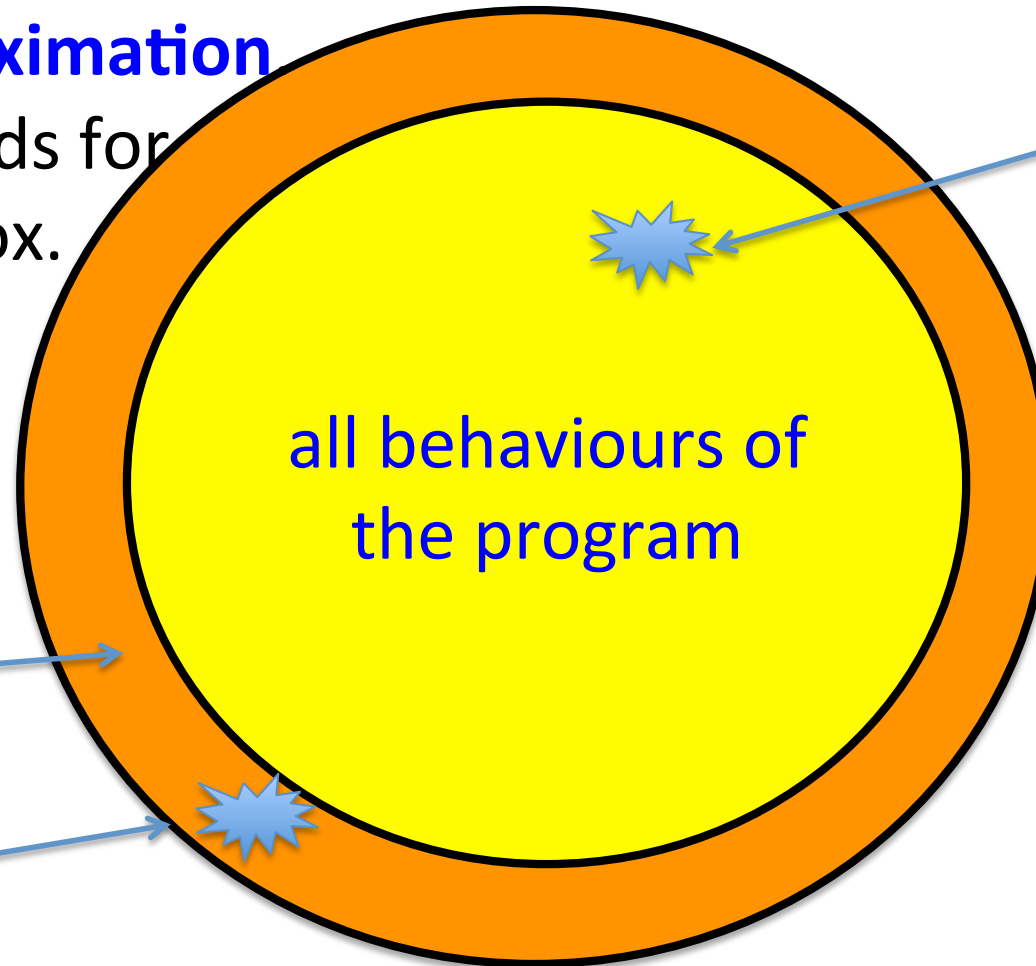
Spurious error,  
output: error trace



# Sound Static Program Analysis (Abstract Interpretation)

Verify **safety properties**, something bad cannot happen,  
by **over-approximation**

If property holds for  
the over-approx.  
it holds for all  
behaviours.



All violations  
detected

Infeasible  
behaviours

Warning:  
False Negative

# Some (Non-Functional) Safety Properties

Absence of **run-time errors** (at program points)

- division never by 0
- sqrt never of a negative number
- arithmetic operation never causes overflow/underflow
- array-index is never out of bounds
- dereference never applied to null-pointer

**Stacks** do never **overflow**

Absence of **timing accidents** (for instructions) for WCET analysis

- memory access never misses cache
- pipeline unit is always available for dispatch of operation
- bus access is not blocked

Derived global program properties

- program **terminates** within a given **deadline**

# Admitted Methods for Automotive Systems: according to ISO-26262

**Table 9 — Methods for the verification of software unit design and implementation**

Methods		ASIL			
		A	B	C	D
1a	Walk-through <sup>a</sup>	++	+	o	o
1b	Inspection <sup>a</sup>	+	++	++	++
1c	Semi-formal verification	+	+	++	++
1d	Formal verification	o	o	+	+
1e	Control flow analysis <sup>bc</sup>	+	+	++	++
1f	Data flow analysis <sup>bc</sup>	+	+	++	++
1g	Static code analysis	+	++	++	++
1h	Semantic code analysis <sup>d</sup>	+	+	+	+
<sup>a</sup> In the case of model-based software development the software unit specification design and implementation can be verified at the model level.					
<sup>b</sup> Methods 1e and 1f can be applied at the source code level. These methods are applicable both to manual code development and to model-based development.					
<sup>c</sup> Methods 1e and 1f can be part of methods 1d, 1g or 1h.					
<sup>d</sup> Method 1h is used for mathematical analysis of source code by use of an abstract representation of possible values for the variables. For this it is not necessary to translate and execute the source code.					

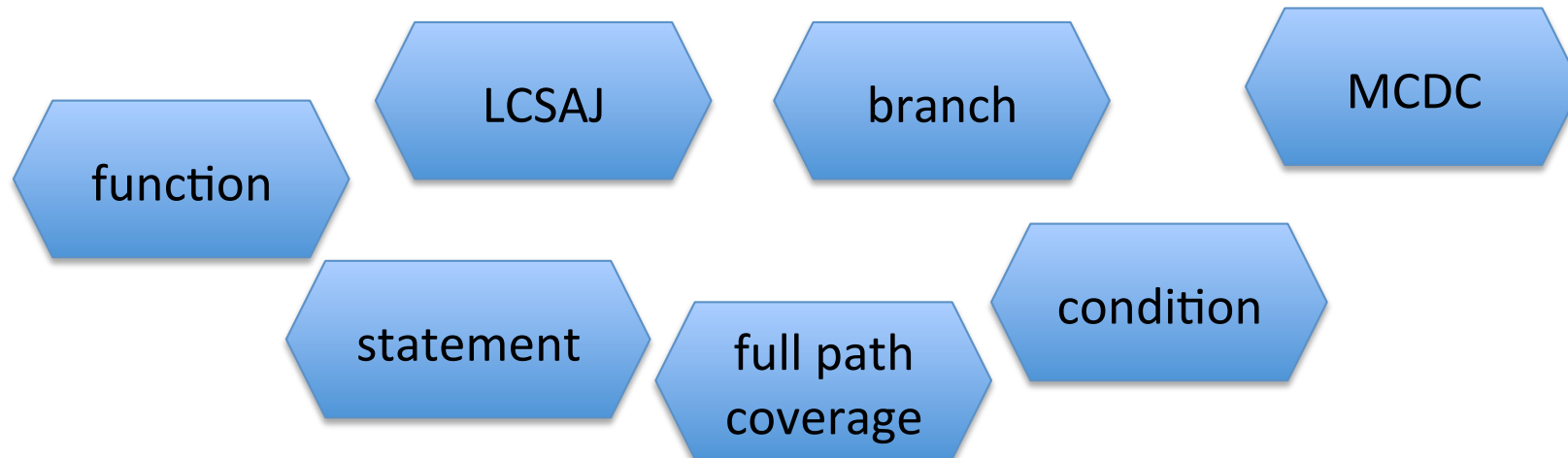
sound, i.e., can  
give a guarantee

-  
-  
-  
+  
-  
-  
-  
+

Excerpt from:  
*Final Draft ISO 26262-6 Road vehicles - Functional safety –*  
*Part 6: Product development: Software Level.*  
*Version ISO/FDIS 26262-6:2011(E), 2011.*

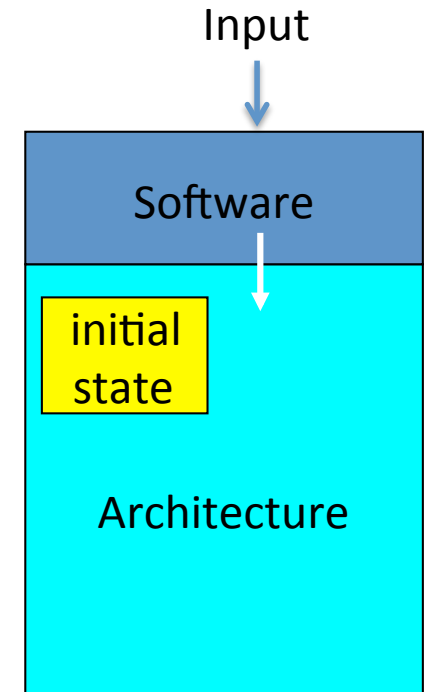
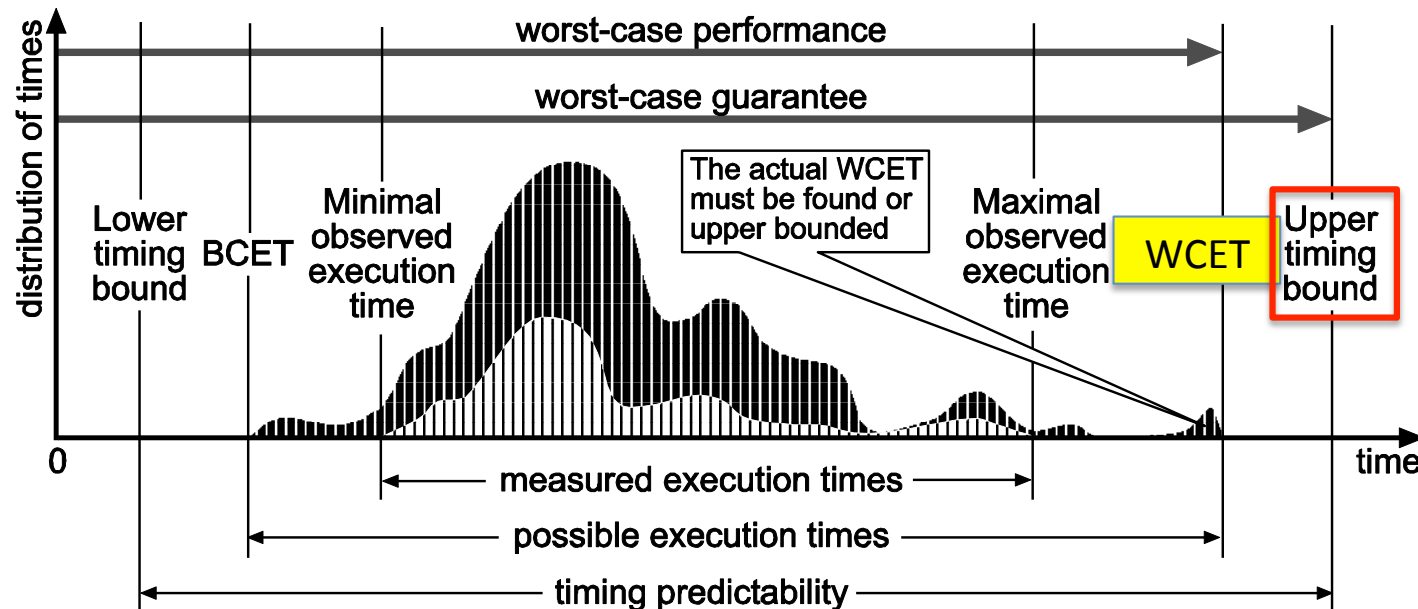
# Testing

- **functional testing** – testing the input-output behavior against the specification
  - often on the basis of a validation suite
- **non-functional testing**
  - in particular **performance testing**
- **testing to locate bugs**
  - coverage criteria to increase the chance to catch bugs



# Verifying Real-Time Requirements

- by using testing
  - special case of performance testing
  - performed by measurement-based methods



- complexity prevents computing exact WCET
- no guarantees!

2. limit of testing

Sound timing analysis determines upper bound instead

# Loops

- Programs spend their time in loops (and in recursion)

Assumption:  $ET(\text{loop}) = \#iterations \times (ET(\text{cond}) + ET(\text{body}))$

- How do we know  $\#iterations$ ?

– Program/Developer/Model/Static Analysis

3. limit of testing

for automation: AbsInt's value analysis

- What is the impact of coverage criteria on WCET determination?

none!

- Even path coverage executes loop body only once
- But:  $ET(\text{body})$  varies heavily – first iteration loads the cache, consecutive iterations profit → considering first iteration leads to overestimation

4. limit of testing



# How to do (Sound) Timing Analysis

## The Problem

Given:

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

Derive: a **guarantee for timeliness**.

# Timing (WCET) Analysis

- Sound methods that determine upper bounds for all execution times,
- can be seen as the **search for a longest path**,
  - through different types of graphs,
  - through a huge space of paths.

I will show

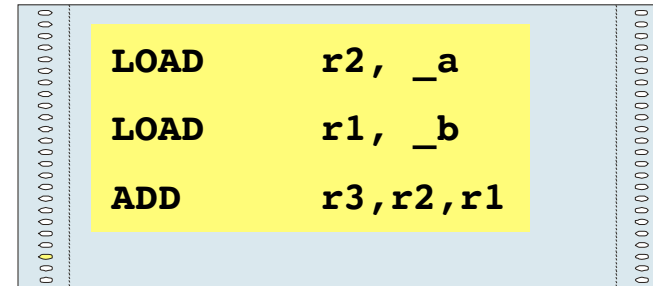
1. how this huge state space originates,
2. how and how far we can cope with this huge state space.

# High-Performance Microprocessors

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

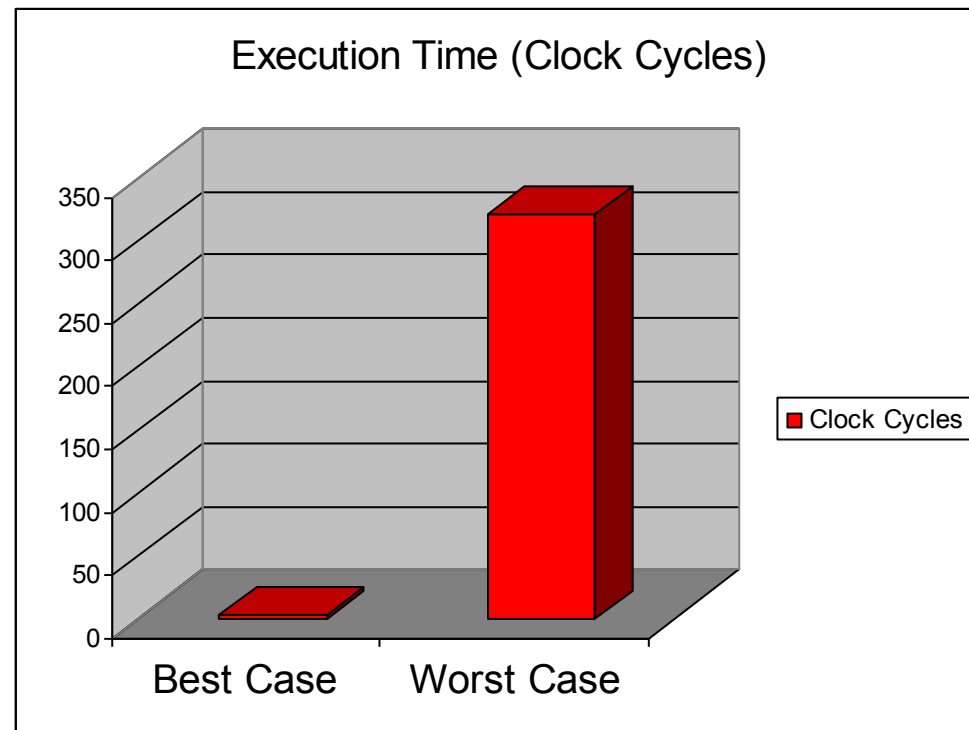
# Variability of Execution Times

$x = a + b;$



PPC 755

In most cases, execution will be fast.  
So, assuming the worst case is safe, but very pessimistic!



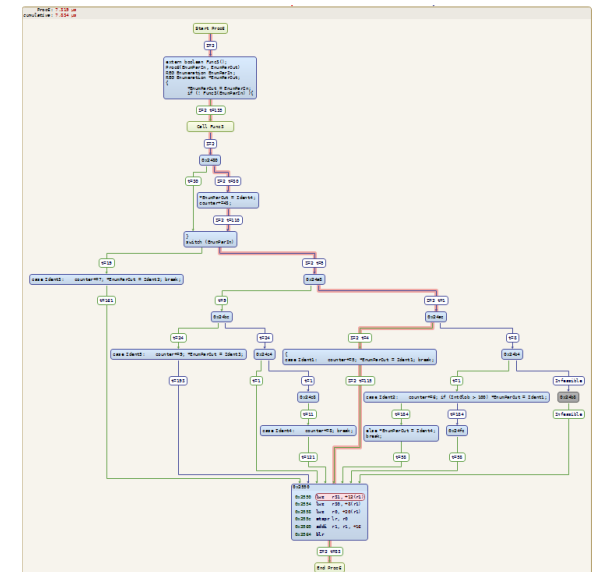
# State-dependent Execution Times

state

- Execution time of an instruction depends on the execution state, e.g.
  - execution of instruction fetch depends on cache state and on bus occupation and on memory state
- Execution state results from the execution history
  - knowing the initial state and the execution history we would know the execution state and the execution time of the instruction
  - however, there may be many execution histories leading to the execution of the instruction

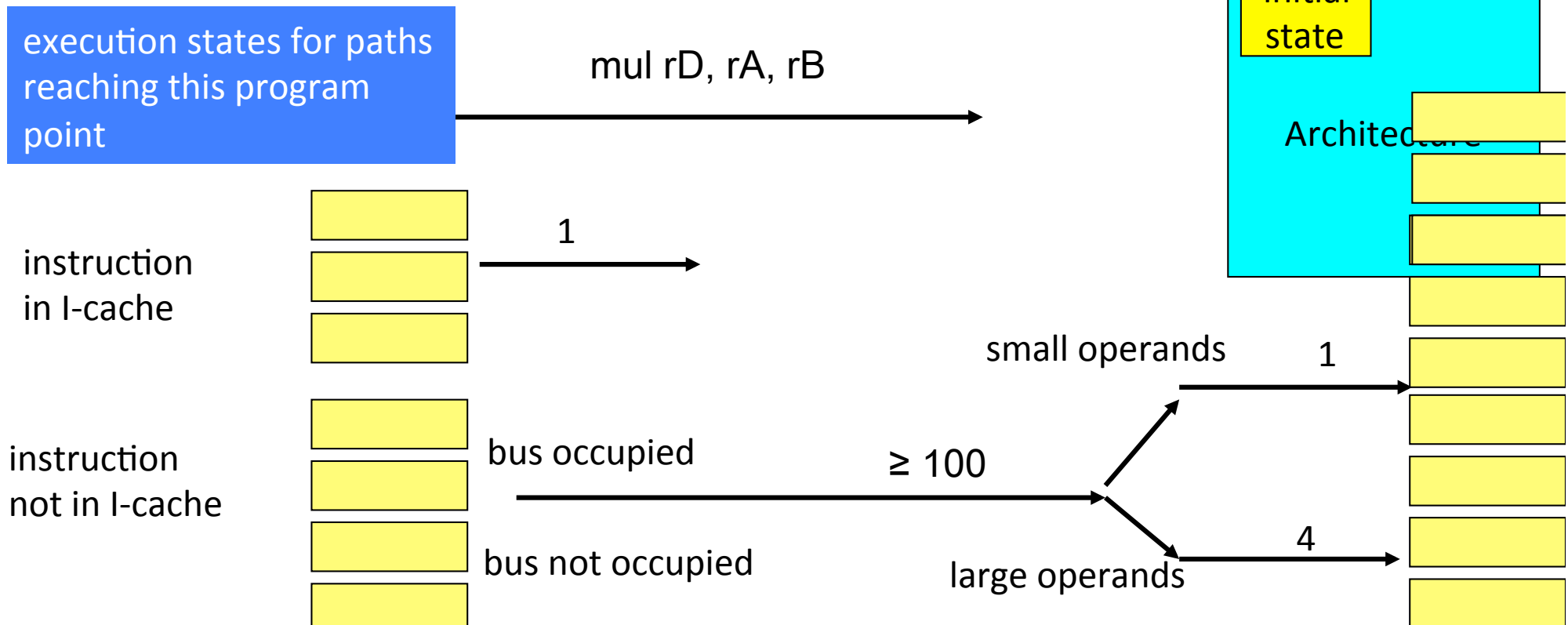
**semantics state:**  
values of variables

**execution state:**  
occupancy of  
resources



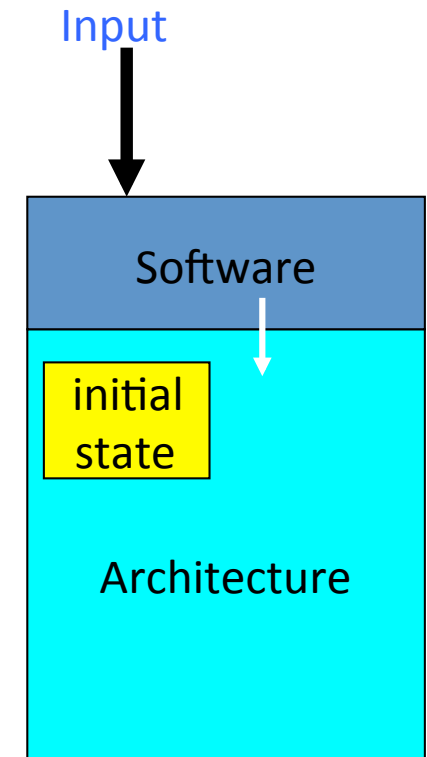
# Timing Analysis – the Search Space with State-dependent Execution Times

- all control-flow paths – depending on the possible **inputs**
- all paths through the architecture for potential **initial states**



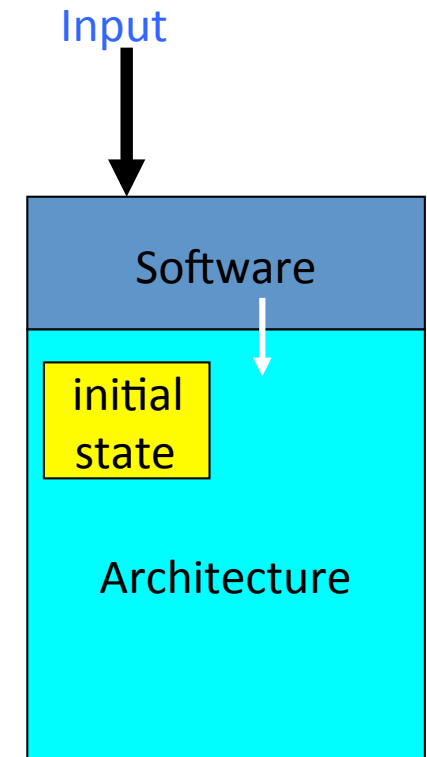
# Timing Analysis – the Search Space with out-of-order execution

- all control-flow paths – depending on the possible inputs
- all paths through the architecture for potential initial states
- including different schedules for instruction sequences



# Timing Analysis – the Search Space with multi-threading

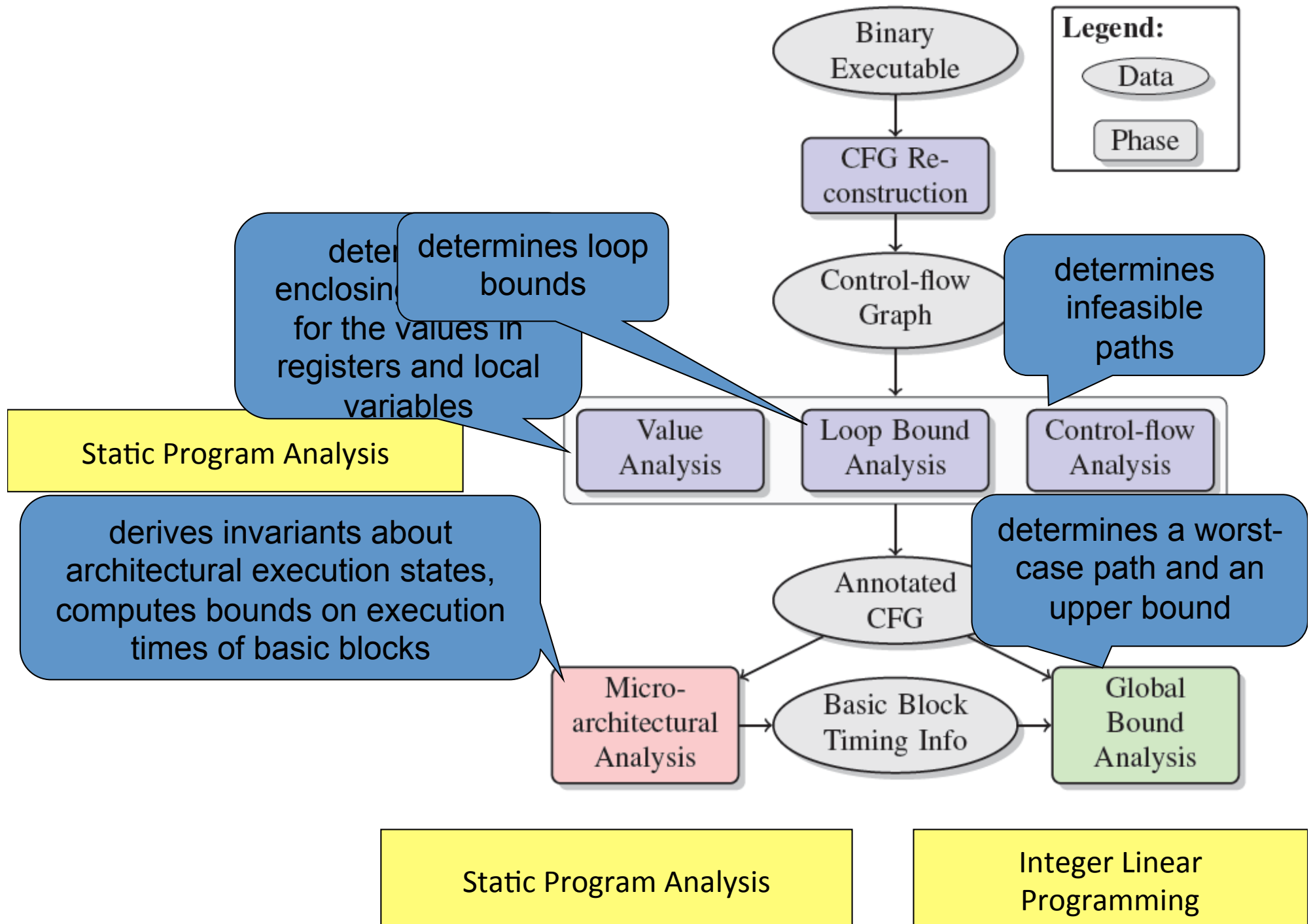
- all control-flow paths – depending on the possible **inputs**
- all paths through the architecture for potential **initial states**
- including **different schedules** for instruction sequences
- including **different interleavings of accesses to shared resources**





# Why Exhaustive Exploration?

- Naive attempt: follow local worst-case transitions only
- Unsound in the presence of **Timing Anomalies**:  
A path starting with a local worst case may have a lower overall execution time,  
Ex.: a cache miss preventing a branch mis-prediction
- Caused by the interference between processor components:  
Ex.: **cache** hit/miss influences **branch prediction**;  
**branch prediction** causes **prefetching**; **prefetching** pollutes the **I-cache**.



# 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

# Our Approach

- Execution time of an instruction depends on the execution state
- Execution state results from the execution history
- We use Static Analysis of Programs for their behavior on the execution platform
- It computes invariants about the set of all potential execution states at all program points,
- by exploring all execution histories in parallel

## state

**semantics state:**  
values of variables

**execution state:**  
occupancy of  
resources

# Deriving Run-Time Guarantees

- Our method and tool 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.



# Tremendous Progress

cache-miss penalty

over-estimation

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

1995  
Lim et al.

2002  
Thesing et al.

2005

20-30%

25

15%

30-50%  
25%

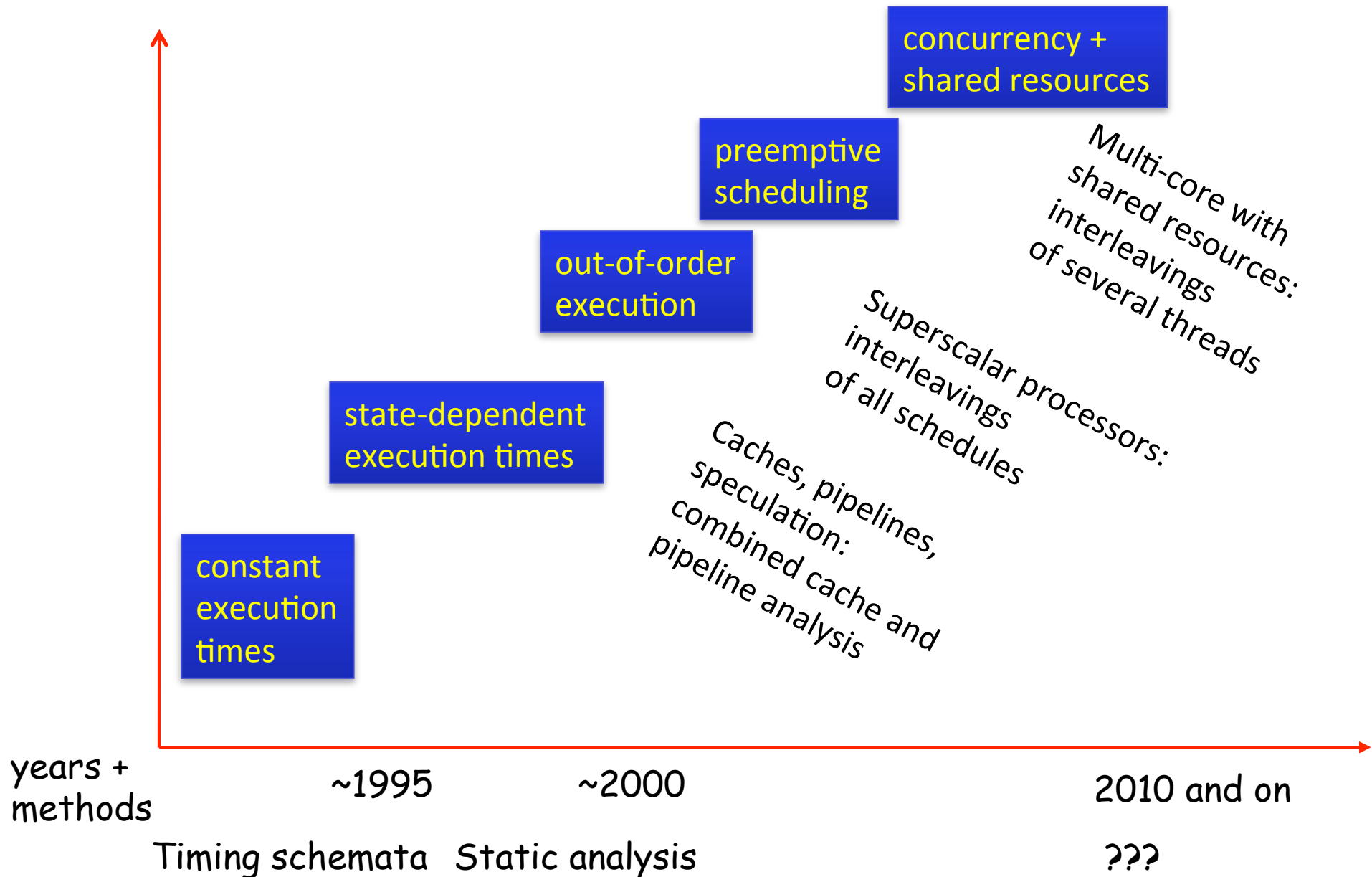
10%

4

200

60

# State Space Explosion in Timing Analysis



# Is All Lost?

How could one get (halfway) precise timing estimates?

- No guarantees, though
- Based on the facility to extract traces including time stamps, e.g. NEXUS
- Needs loop bounds from Value Analysis
- Rough idea: determine end-to-end execution times of traces
- identify associated program paths,
- Map control flow to an Integer Linear Program (ILP)
- Determine worst-case path by solving the ILP



# But

- Often bandwidth problems for tracing HW
- Unclear contribution of asynchronous events, e.g. DRAM refresh
- Complex control-flow structure compared to basic-block graph

# Conclusions

Performance testing in the form of **measurement-based timing analysis** **cannot derive guarantees**

**Sound timing analysis** delivers **guarantees** for quite complex processors

Current architectural trends

- **unpredictable** cores
- **multi-core architectures** with **shared resources**
- **reduced observability**

do not allow

- **sound timing analysis** with acceptable **effort** and desired **precision**
- reduce precision for **performance estimation**

- The problem remains
- The possibility to solve it disappears

# Literature

- *C. Ferdinand et al.: Reliable and Precise WCET Determination of a Real-Life Processor*, EMSOFT 2001
- *R. Wilhelm et al.: The Determination of Worst-Case Execution Times - Overview of the Methods and Survey of Tools*. ACM Transactions on Embedded Computing Systems (TECS) 7(3), 2008.
- *R. Wilhelm et al.: Memory Hierarchies, Pipelines, and Buses for Future Architectures in Time-critical Embedded Systems*, IEEE TCAD, July 2009
- *Mingsong Lv et al. : A Survey on Cache Analysis for Real-Time Systems*, LITES 2016
- <https://www.absint.com/ait/>