Timing Analysis and Timing Predictability

Reinhard Wilhelm
Saarbrücken

AbsInt
Angewandte Informatik GmbH

UNIVERSITÄT DES SAARLANDES

Structure of the Talk

1. Timing Analysis – the Problem
2. Timing Analysis – our Solution
   - the overall approach, tool architecture
   - cache analysis
3. Results and experience
4. The influence of Software
5. Design for Timing Predictability
   - predictability of cache replacement strategies

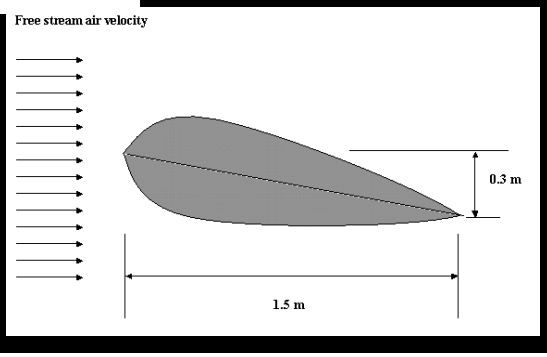
Industrial Needs

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)

Modern Hardware Features

- Modern processors increase performance by using: Caches, Pipelines, Branch Prediction, Speculation
- These features make WCET 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

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

How to Deal with Murphy's Law?

Essentially three different answers:

- **Accepting**: Every timing accident that may happen will happen
- **Fighting**: Reliably showing that many/most Timing Accidents cannot happen
- **Cheating**: monitoring "enough" runs to get a good feeling

Accepting Murphy's Law

like guaranteeing
a speed of 4.07 km/h
for this car

because of the variability of execution times on modern processors

Cheating to deal with Murphy's Law

- measuring "enough" runs to feel comfortable
- how many runs are "enough"?
- Example: Analogy – Testing vs. Verification

  AMD was offered a verification of the K7.
  They had tested the design with 80 000 test vectors, considered verification unnecessary.
  Verification attempt discovered 2 000 bugs!

The only remaining solution: Fighting Murphy's Law!

Execution Time is History-Sensitive

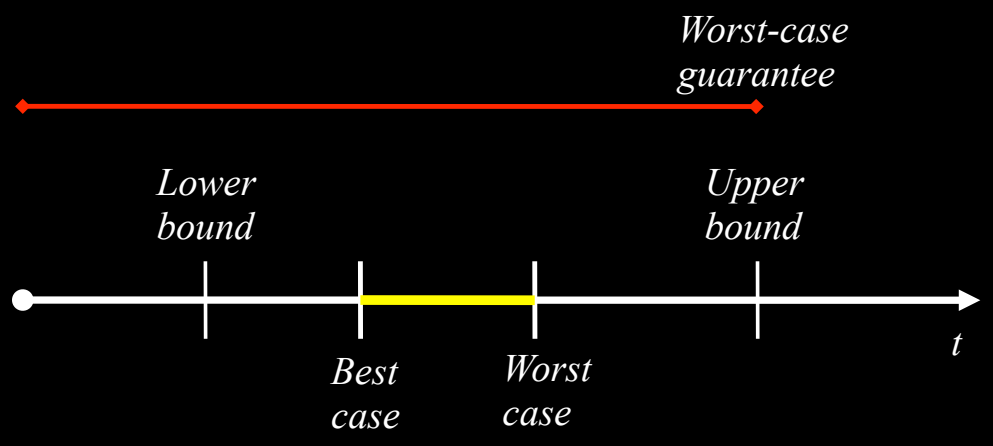
Contribution of the execution of an instruction to a program's execution time

- depends on the execution state, i.e., on the execution so far,
- i.e., cannot be determined in isolation

Deriving Run-Time Guarantees

- Static Program Analysis derives **Invariants** about all execution states at a program point.
- Derive **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 (and the more accidents **predicted**, the **higher** the **lower** bound).

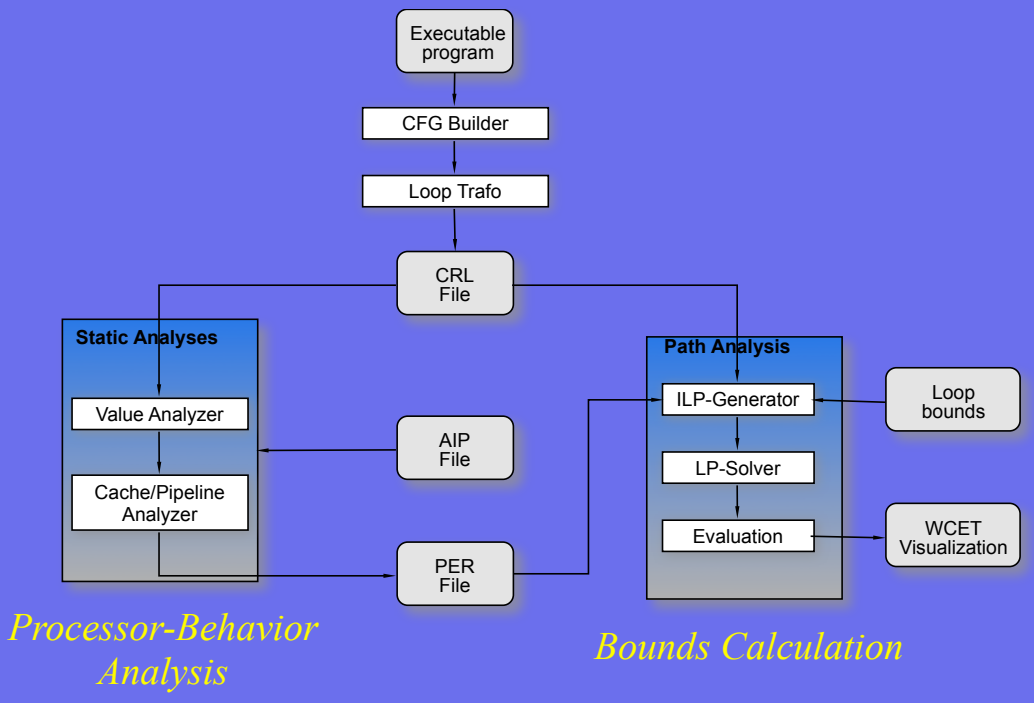
Basic Notions



Overall Approach: Natural Modularization

1. Processor-Behavior Analysis:
   - Uses Abstract Interpretation
   - Excludes as many Timing Accidents as possible
   - Determines upper bound for basic blocks (in contexts)
2. Bounds Calculation
   - Maps control flow graph to an integer linear program
   - Determines upper bound and associated path

Overall Structure



Static Program Analysis Applied to WCET Determination

- Upper bounds must be safe, i.e. not underestimated
- Upper bounds should be tight, i.e. not far away from real execution times
- Effort must be tolerable

Abstract Interpretation (AI)

- semantics-based method for static program analysis
- Basic idea of AI: Perform the program's computations using value descriptions or abstract values in place of the concrete values, start with a description of all possible inputs
- AI supports correctness proofs
- Tool support (Program-Analysis Generator PAG)

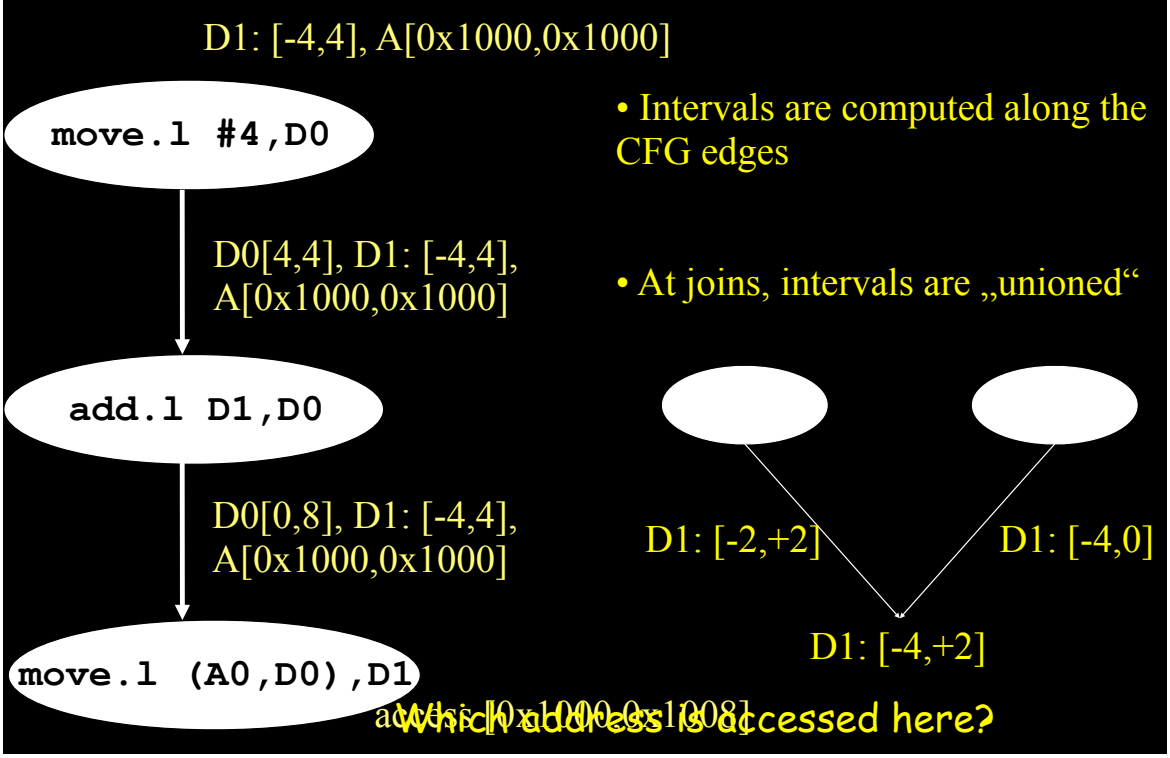
Abstract Interpretation – the Ingredients and one Example

- **abstract domain** – complete semilattice,
  related to concrete domain by **abstraction** and **concretization functions**,
  e.g. intervals of integers (including $-\infty$, ∞) instead of integer values
- **abstract transfer functions** for each statement type – abstract versions of their semantics
  e.g. arithmetic and assignment on intervals
- a **join function** combining abstract values from different control-flow paths – lub on the lattice
  e.g. "union" on intervals
- Example: Interval analysis (Cousot/Halbwachs78)

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, as in the example above
  for the values occurring in the machine program
  (addresses, register contents, local and global variables)

Value Analysis II



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

Good means less than 16 cache lines

Caches: Fast Memory on Chip

- Caches are used, because
  - Fast main memory is too expensive
  - The speed gap between CPU and memory is too large and increasing
- 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

Caches: How the work

CPU wants to read/write at memory address **a**,
sends a request for **a** to the bus

Cases:

- Block **m** containing **a** in the cache (hit):
  request for **a** is served in the next cycle
- Block **m** not in the cache (miss):
  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
- Several replacement strategies: LRU, PLRU, FIFO,...
  determine which line to replace

Cache Analysis

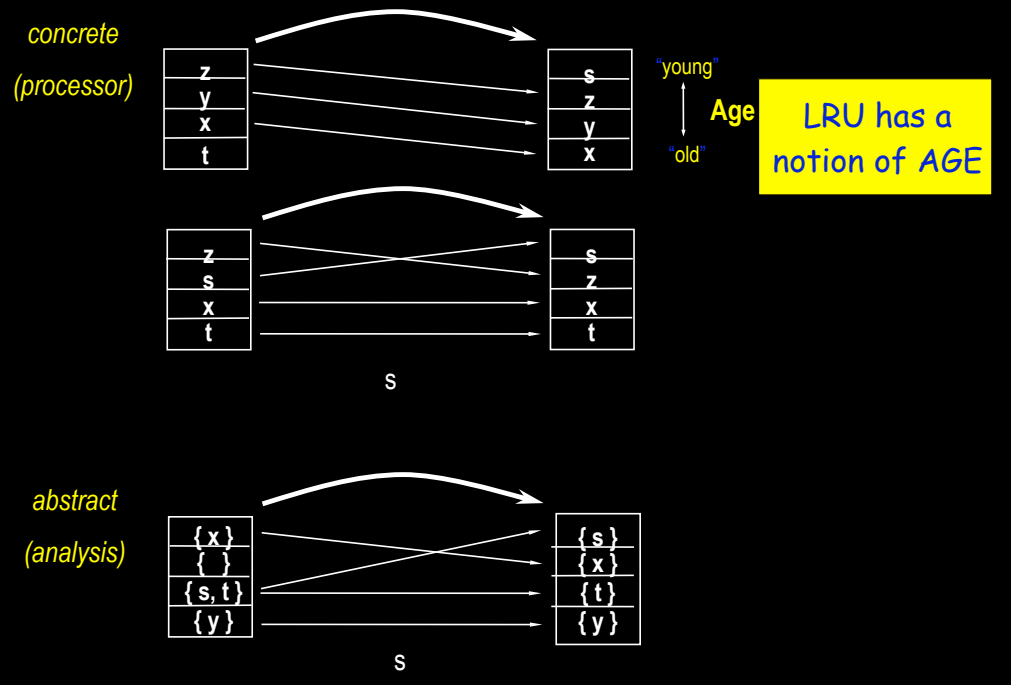
How to statically precompute cache contents:

- Must Analysis:
  For each program point (and calling context), find out which blocks are in the cache
- May Analysis:
  For each program point (and calling context), find out which blocks may be in the cache
  Complement says what is not in the cache

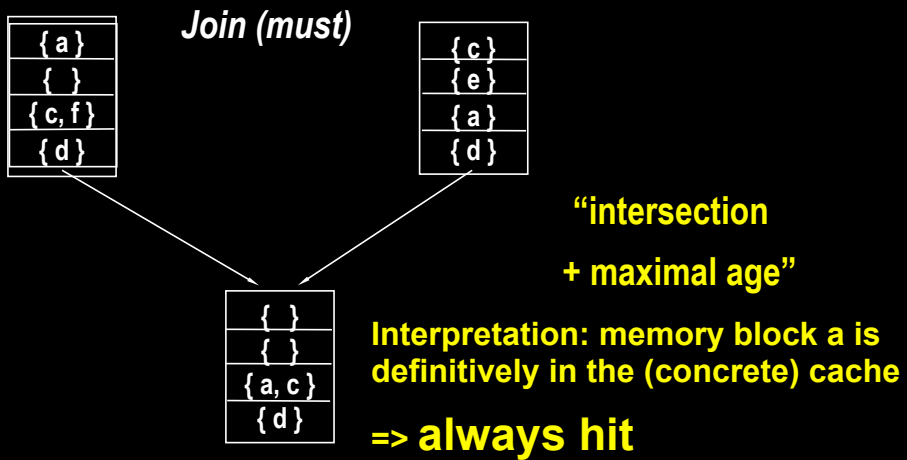
Must-Cache and May-Cache- Information

- Must Analysis determines safe information about cache hits
  Each predicted cache hit reduces upper bound
- May Analysis determines safe information about cache misses
  Each predicted cache miss increases lower bound

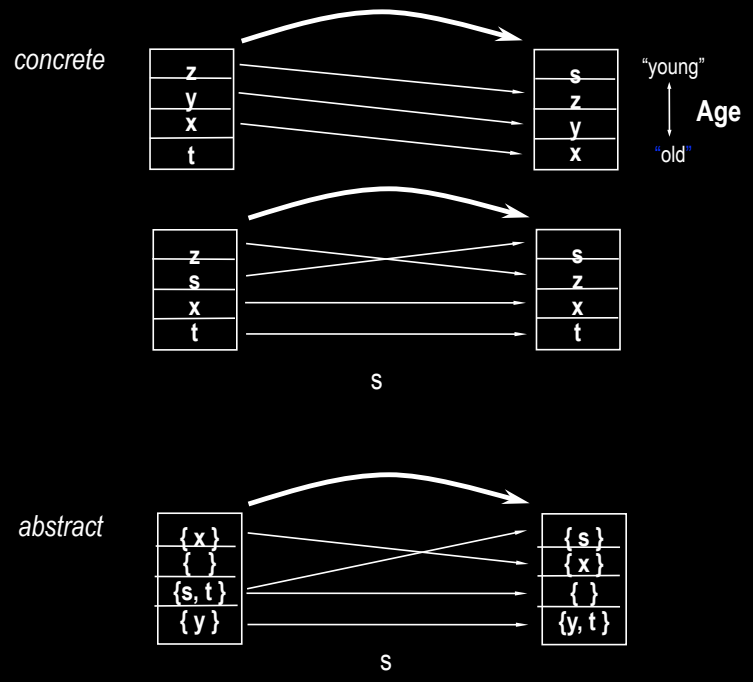
Cache with LRU Replacement: Transfer for must



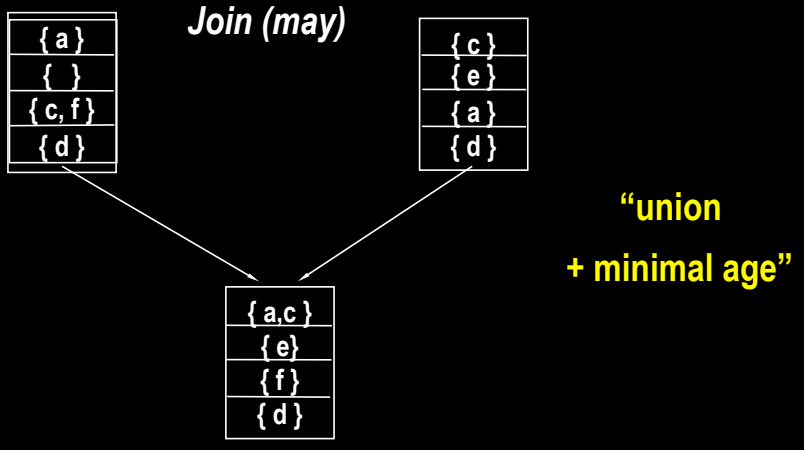
Cache Analysis: Join (must)



Cache with LRU Replacement: Transfer for may



Cache Analysis: Join (may)



"union

+ minimal age"

Penalties for Memory Accesses
(in #cycles for PowerPC 755)

cache miss	cm = 40
cache miss + write back	cm + wb = 80 (wb = 40)
TLB-miss and loading	tlb = 12 cm + 1 wb = 520
Memory-mapped I/O	mm = 800
Page fault	pf = 2000

Penalties have to be assumed for uncertainties!

Tendency increasing, since clocks are getting faster faster than everything else

Cache Impact of Language Constructs

- Pointer to data
- Function pointer
- Dynamic method invocation
- Service demultiplexing CORBA

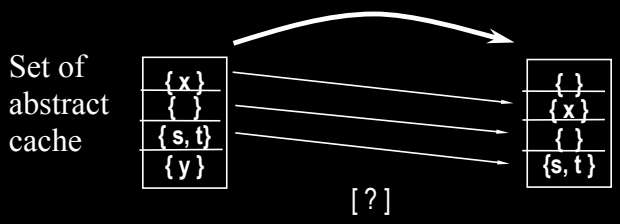
The Cost of Uncertainty

Cost of statically unresolvable dynamic behavior.

Basic idea: What does it cost me if I cannot exclude a timing accident?

This could be the basis for design and implementation decisions.

Cache with LRU Replacement:
Transfer for must under unknown access,
e.g. unresolved data pointer



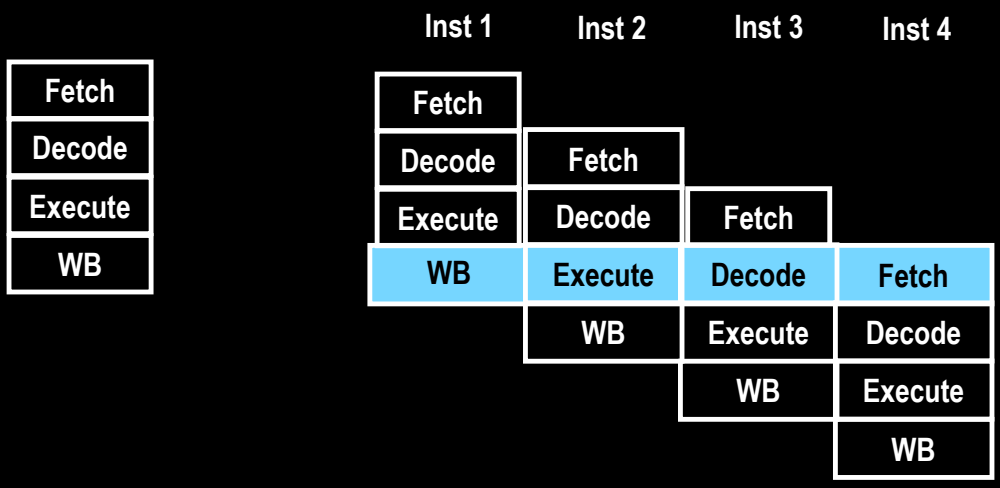
If address is undetermined, loss of information in every cache set!

Analogously for multiple unknown accesses, e.g. unknown function pointer; assume maximal cache damage

Dynamic Method Invocation

- Traversal of a data structure representing the class hierarchy
- Corresponding worst-case execution time and resulting cache damage
- Efficient implementation [WiMa] with table lookup needs 2 indirect memory references;
  if page faults cannot be excluded:
  2 x pf = 4000 cycles!

Pipelines



Ideal Case: 1 Instruction per Cycle

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

More Threats

- Out-of-order execution — Consider all possible execution orders
- Speculation — ditto
- Timing Anomalies — Considering the locally worst-case path insufficent

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

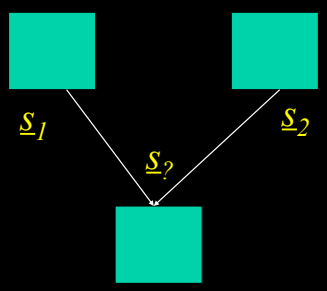
function exec (b : **basic block**, s : **abstract pipeline state**) t: **trace**

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.
- Assume local worst cases is safe
  (in the case of no timing anomalies)
- 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*

How to Create a Pipeline Analysis?

- Starting point: Concrete model of execution
- First build reduced model
  - E.g. forget about the store, registers etc.
- Then build abstract timing model
  - Change of domain to abstract states, i.e. sets of (reduced) concrete states
  - Conservative in execution times of instructions

Defining the Concrete State Machine

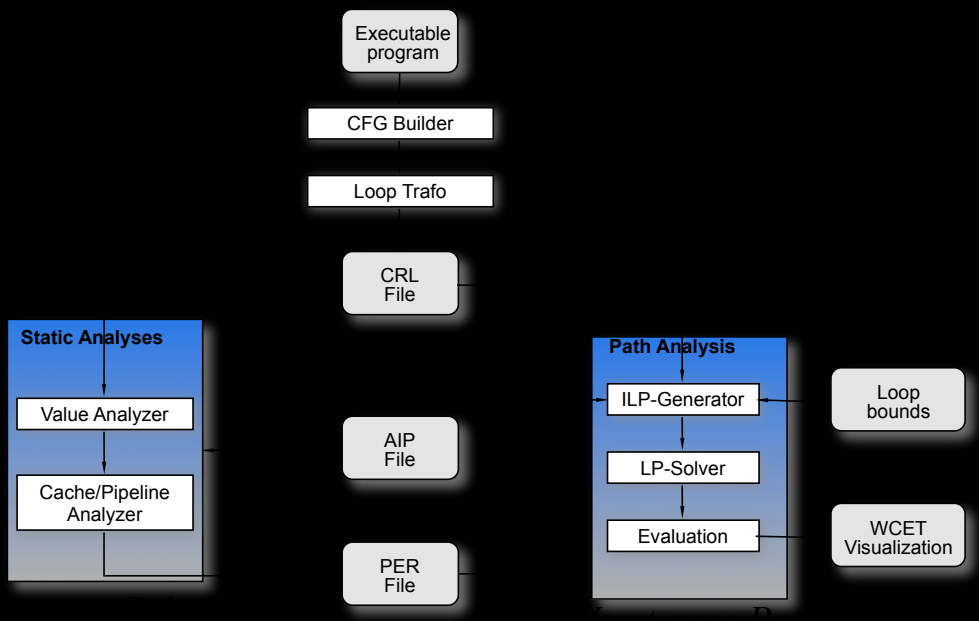
How to define such a complex state machine?

- A state consists of (the state of) internal components (register contents, fetch/ retirement queue contents...)
- Combine internal components into units (modularisation, cf. VHDL/Verilog)
- Units communicate via signals
- (Big-step) Transitions via unit-state updates and signal sends and receives

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

Overall Structure



Path Analysis

by Integer Linear Programming (ILP)

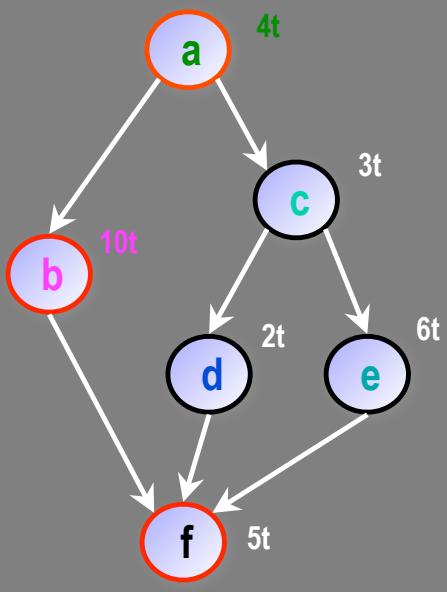
- Execution time of a program =

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

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

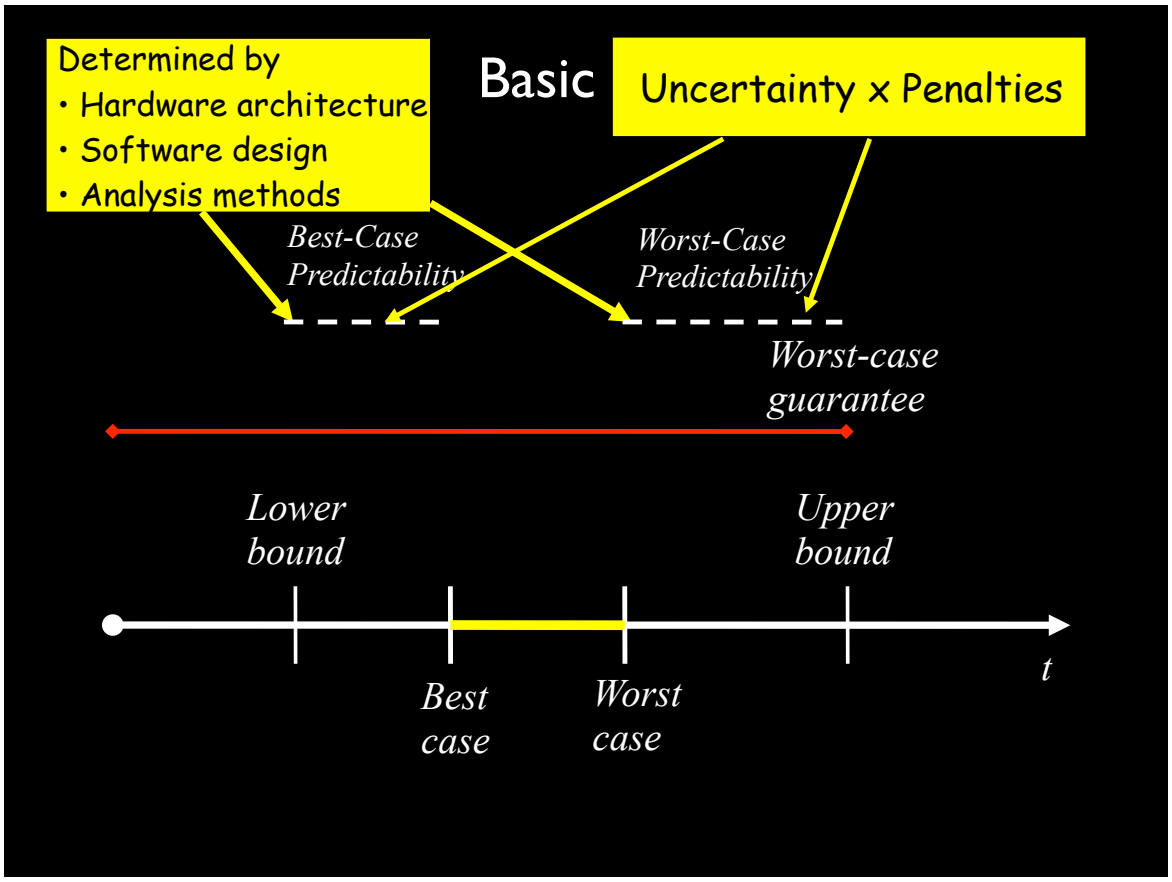
where $x_a = x_b + x_c$

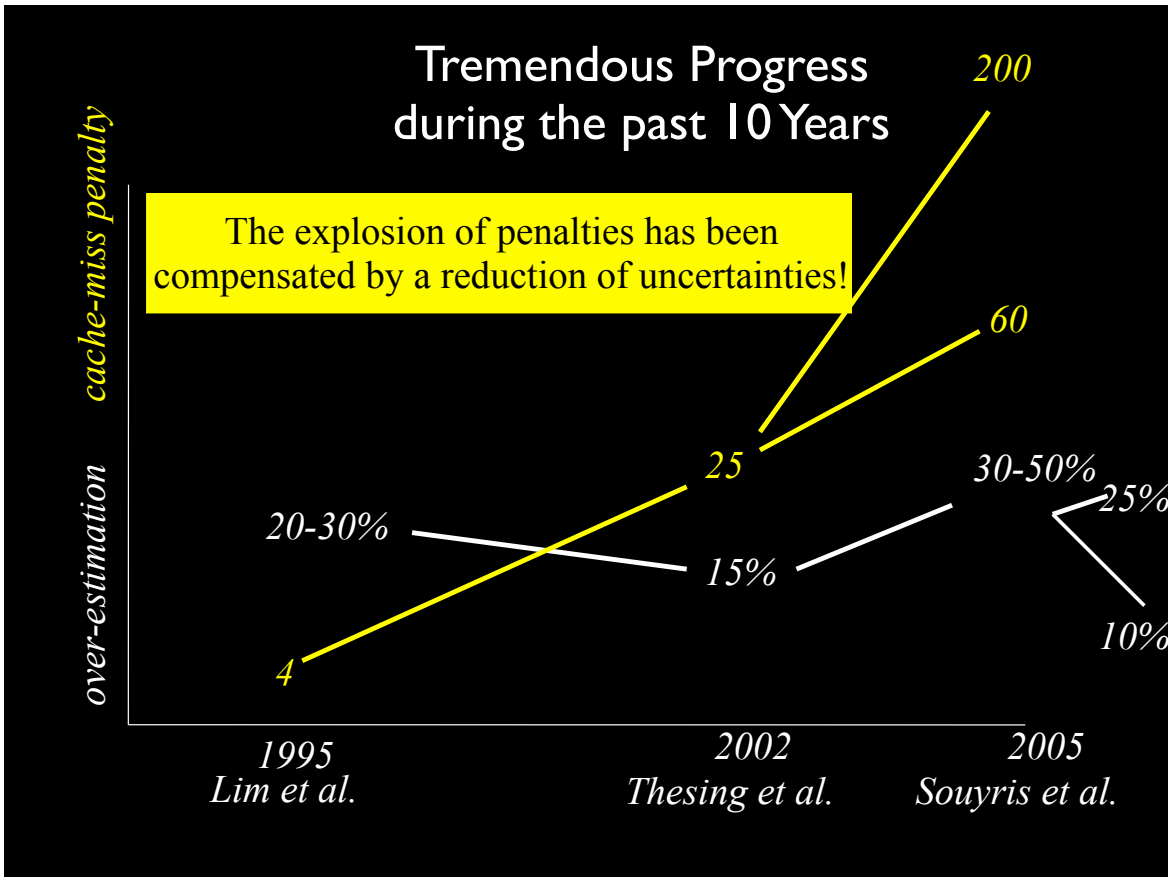
$x_c = x_d + x_e$

$x_f = x_b + x_d + x_e$

$x_a = 1$

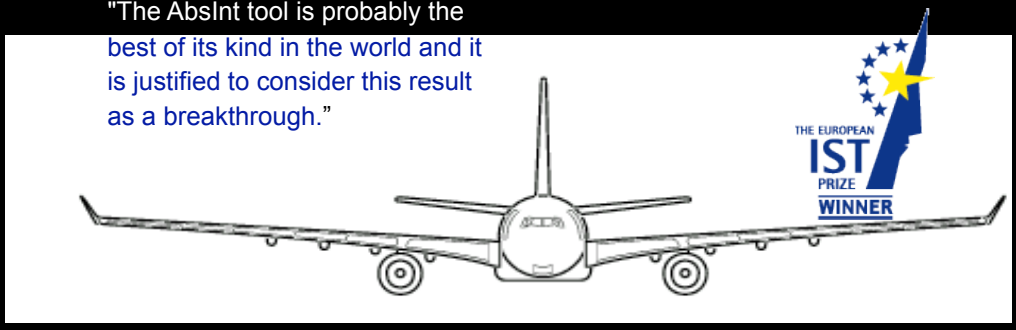
Value of objective function: 19	
x_a	1
x_b	1
x_c	0
x_d	0
x_e	0
x_f	1


Determined by
• Hardware architecture
• Software design
• Analysis methods
Basic
Uncertainty x Penalties
Best-Case Predictability
Worst-Case Predictability
Worst-case guarantee
Lower bound
Upper bound
Best case
Worst case
t


Tremendous Progress during the past 10 Years
The explosion of penalties has been compensated by a reduction of uncertainties!
over-estimation
cache-miss penalty
200
60
25
4
20-30%
15%
30-50%
25%
10%
1995
Lim et al.
2002
Thesing et al.
2005
Souyris et al.

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."



Timing Predictability

- Experience has shown that the precision of results depend on system characteristics both
- of the underlying hardware platform and
- of the software layers

Timing Predictability

- System characteristics determine
  - size of penalties – hardware architecture
  - analyzability – HW architecture, programming language, SW design
- Many "advances" in computer architecture have increased average-case performance at the cost of worst-case performance

Conclusion

- The determination of safe and precise upper bounds on execution times by static program analysis essentially solves the problem
- Usability, e.g. need for user annotation, can still be improved
- Precision greatly depends on predictability properties of the system
- Integration into the design process is necessary