ARTIST2 Summer School 2008 in Europe
Autrans (near Grenoble), France
September 8-12, 2008

Timing Analysis and Timing Predictability

Reinhard Wilhelm
Saarland University

AbsInt Angewandte Informatik GmbH

http://www.artist-embedded.org/

UNIVERSITÄT DES SAARLANDES

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)).
- Timing Analysis provides the abstraction for Scheduling

Deriving Run-Time Guarantees for Hard Real-Time Systems

Given:

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

Derive: a guarantee for timeliness.

Structure of the Talk

1. Timing Analysis – the Problem
2. Timing Analysis – a Sketch of our Approach
3. Results and experience
4. Our Approach in more details
   - the overall approach, tool architecture
   - cache analysis
   - pipeline analysis
5. Architectural and Timing Predictability
   - predictability of cache replacement strategies
   - extending predictability concepts beyond caches
6. Conclusion

What does Execution Tim

causes the DAG-based task representation of the scheduling people

- the input – this has always been so and will remain so,

Caused by caches, pipelines, speculation etc.

- the initial execution state of the platform – this is (relatively) new,

Explosion of the space of inputs **and** initial states
$\Rightarrow$ measurement infeasible

- interferences from the environment – this depends on whether the system design admits it (preemptive scheduling, interrupts).

"external" interference as seen from analyzed task

Modern Hardware Features

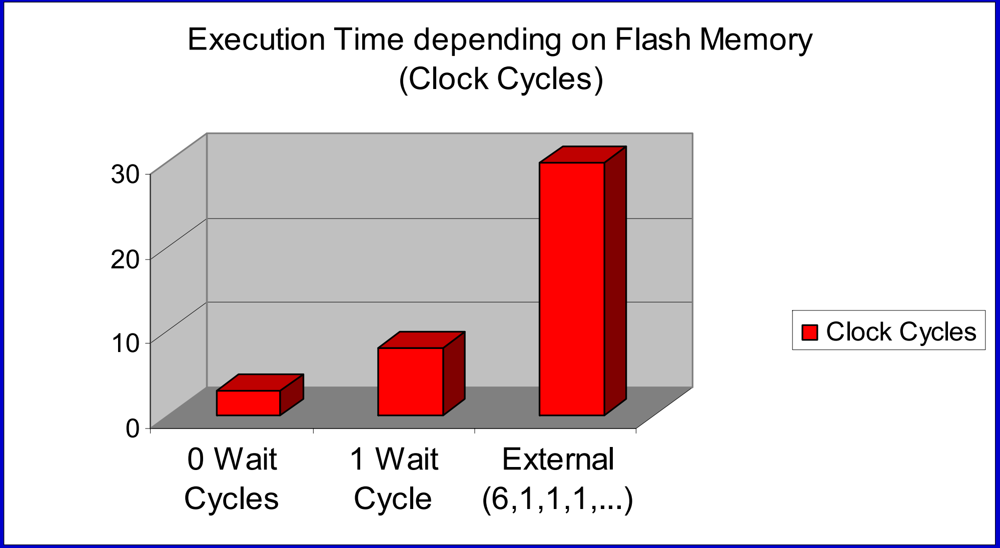
- Modern processors increase performance by using: Caches, Pipelines, Branch Prediction, Speculation
- These features make bounds computation difficult: Execution times of instructions vary widely
  - Best case - everything goes smoothly: no cache miss, operands ready, needed resources free, branch correctly 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

Access Times

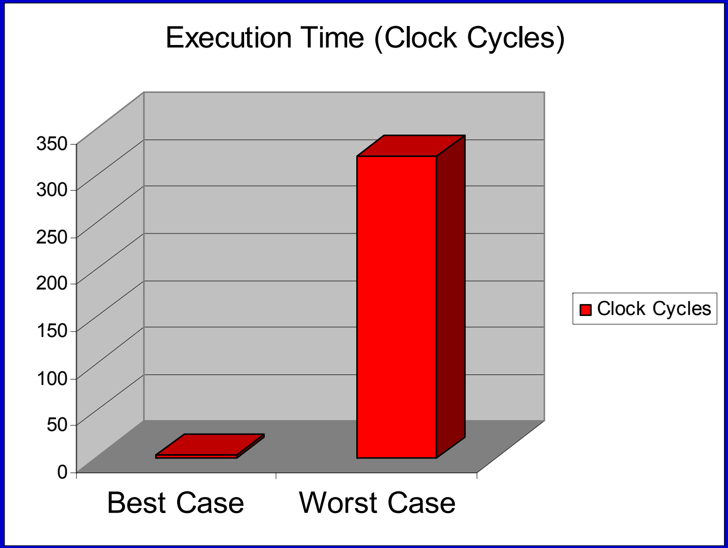
x = a + b; →

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

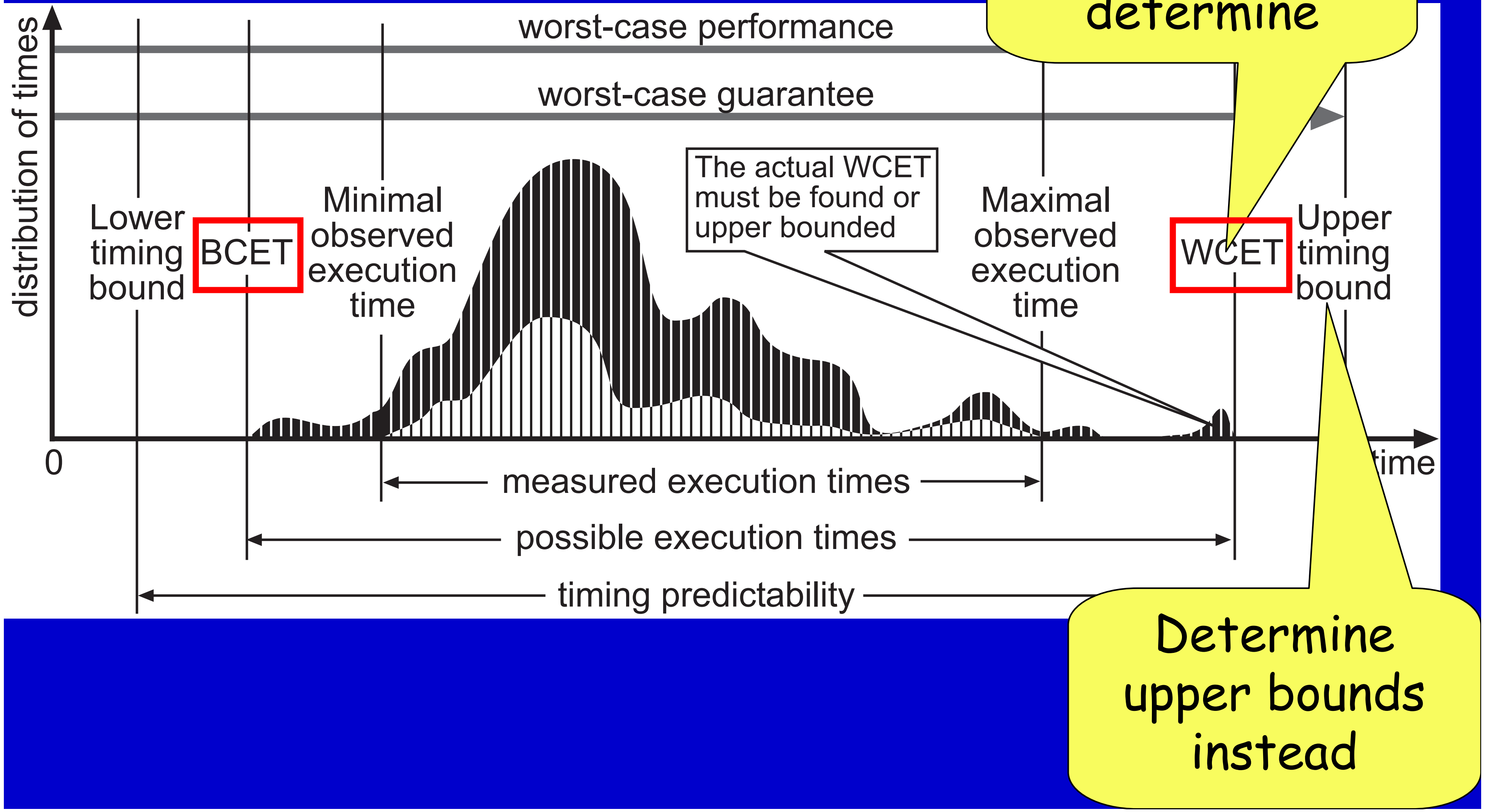
MPC 5xx



PPC 755



Notions in Timing Analysis



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

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, i.e., cannot be determined in isolation

Timing Accidents and Penalties

Timing Accident – cause for an increase of the execution time of an instruction

Timing Penalty – the associated increase

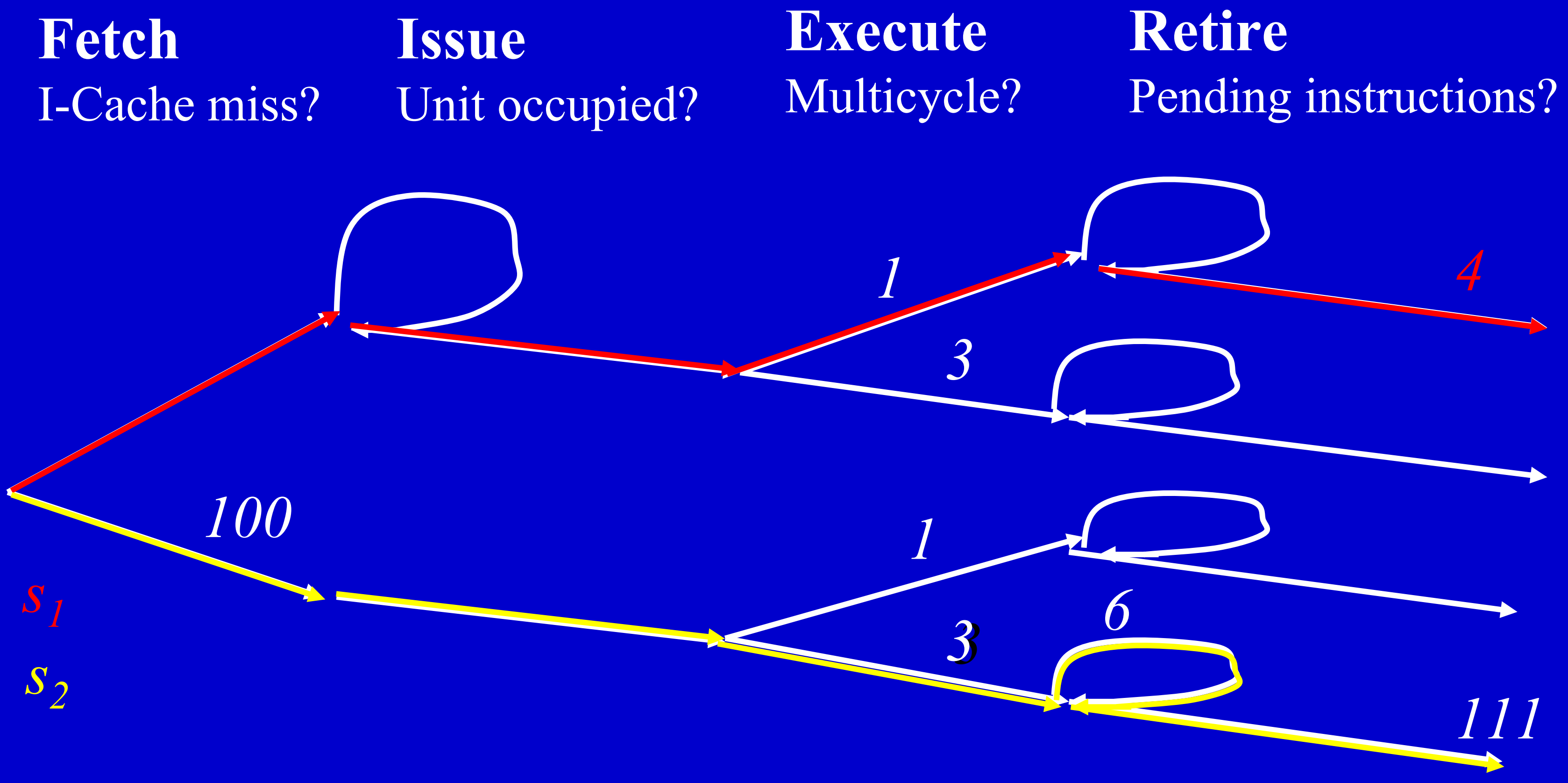
- Types of timing accidents
  - Cache misses
  - Pipeline stalls
  - Branch mispredictions
  - Bus collisions
  - Memory refresh of DRAM
  - TLB miss

Our Approach

- Static Analysis of Programs for their behavior on the Execution platform
- Static program analysis computes invariants about the set of possible execution states at all program points

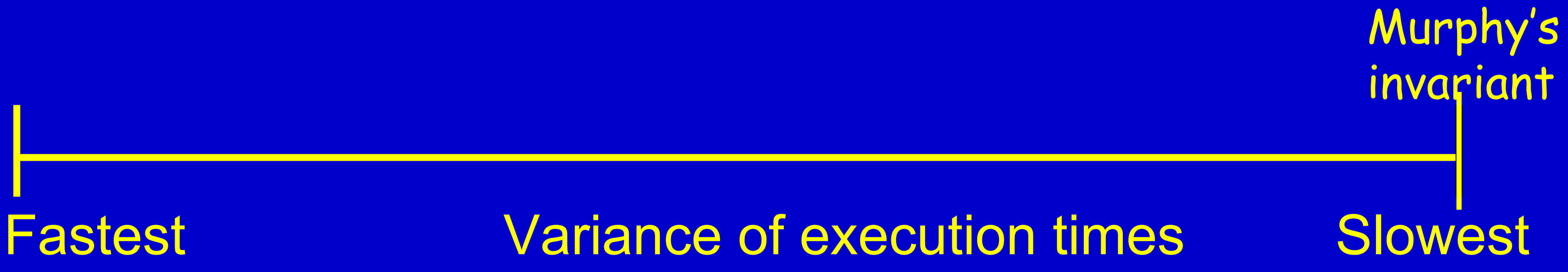
(Concrete) Instruction Execution

`mul`



Deriving Run-Time Guarantees

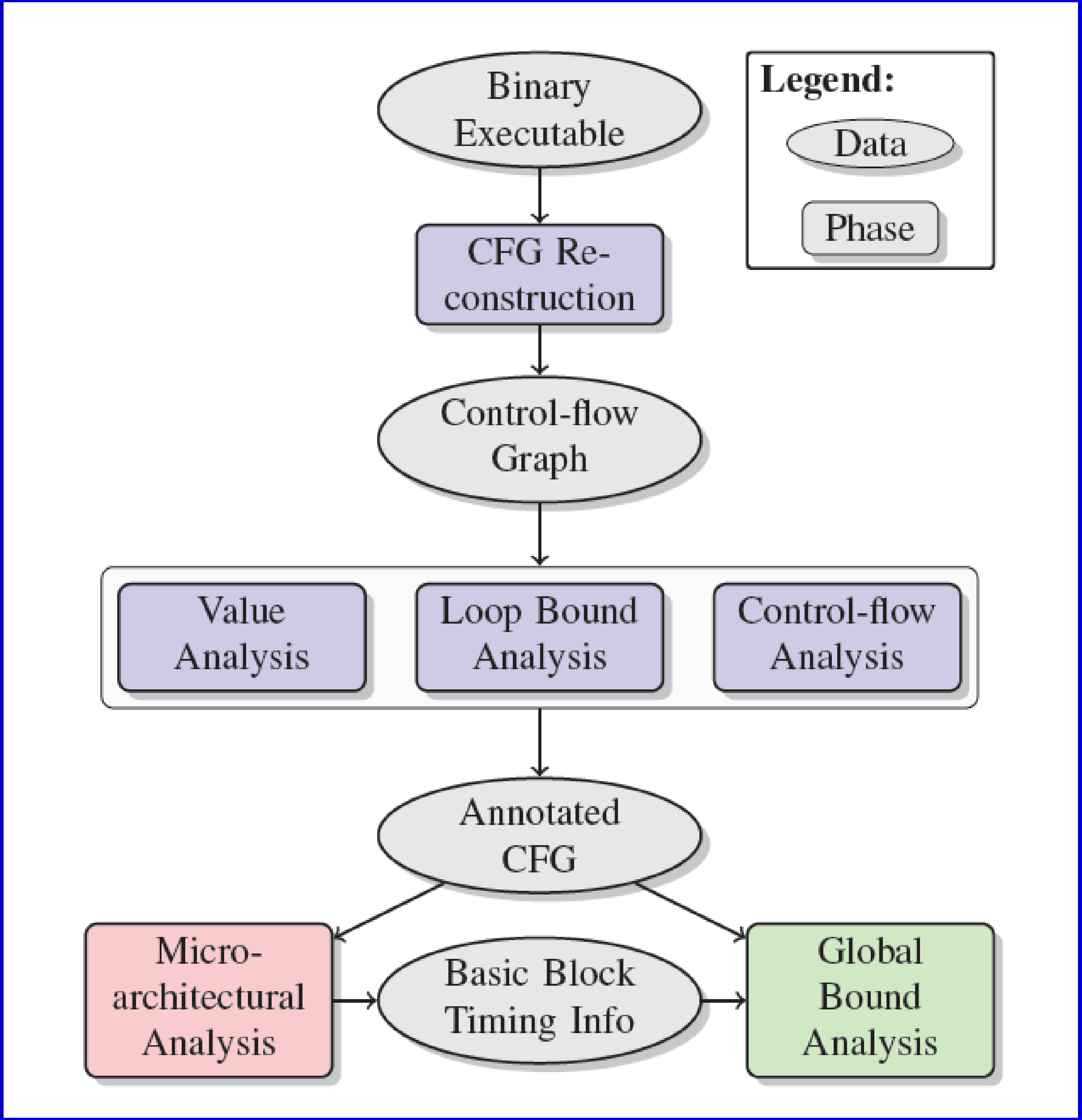
- Our method and tool derives Safety Properties from these invariants : Certain timing accidents will never happen. Example: At program point p, instruction fetch will never cause a cache miss.
- The more accidents **excluded**, the **lower** the **upper** bound.



Overall Approach: Natural Modularization

1. Control-Flow Analysis
   - determines infeasible paths,
   - computes loop bounds,
   - missing information as annotation by user
2. Micro-architecture Analysis:
   - Uses static program analysis
   - Excludes as many Timing Accidents as possible
   - Determines upper bounds for basic blocks
3. Worst-case Path Determination
   - Maps control flow to integer linear program
   - Determines upper bound for the whole program and an associated path

Tool Architecture



Abstract Interpretations

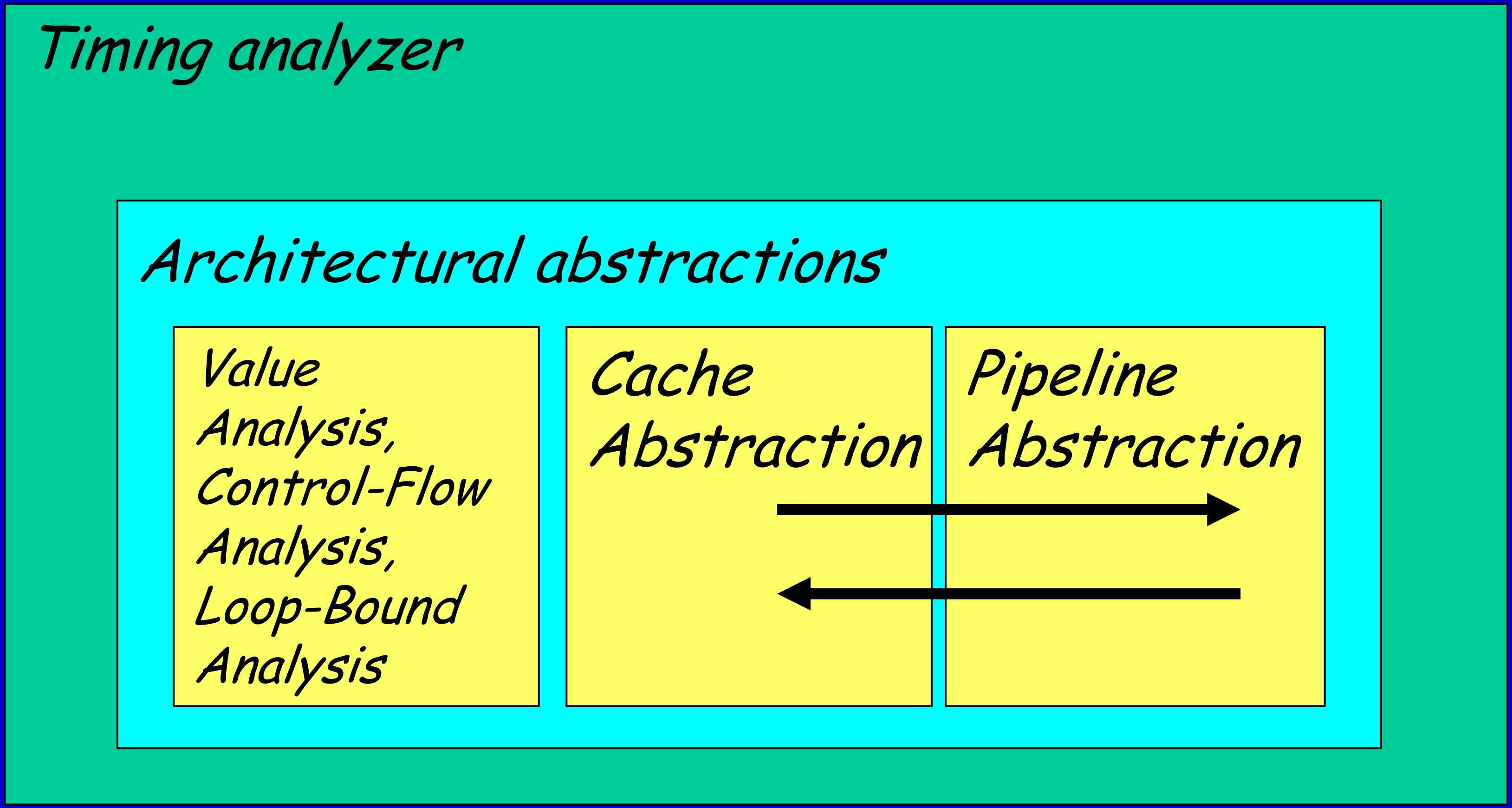
Abstract Interpretation

Integer Linear Programming

Semantics for Timing Analysis

- Abstract Interpretation uses an abstraction of the semantics of the language.
- Timing Analysis:
  - Analyzes executables; source programs don't talk about the machine, machine cycles, etc.
  - We need concrete semantics of the Instruction Set Architecture (ISA), more precisely, one semantics for each realization (processor, even fabrication) of the ISA.
  - The abstract semantics must contain an abstract architecture model that is conservative with respect to the timing behavior.

The Architectural Abstraction inside the Timing Analyzer


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

Variability of Execution Times

- caused by dependence on input and initial state,
- does not permit exhaustive measurements.
- Non-exhaustive end-to-end measurements are in general unsafe.

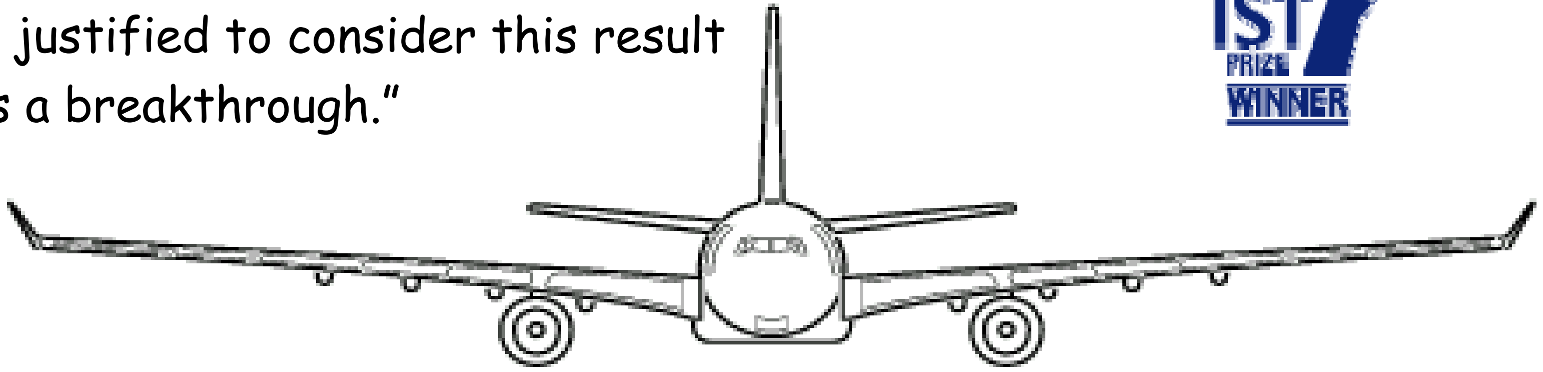
What is the alternative?

- Determine safe bounds on instructions/basic blocks and derive safe bounds for the whole program.
- But, variability of execution times appears on all levels, individual memory access, instruction, arithmetic operation, etc.

aiT WCET Analyzer

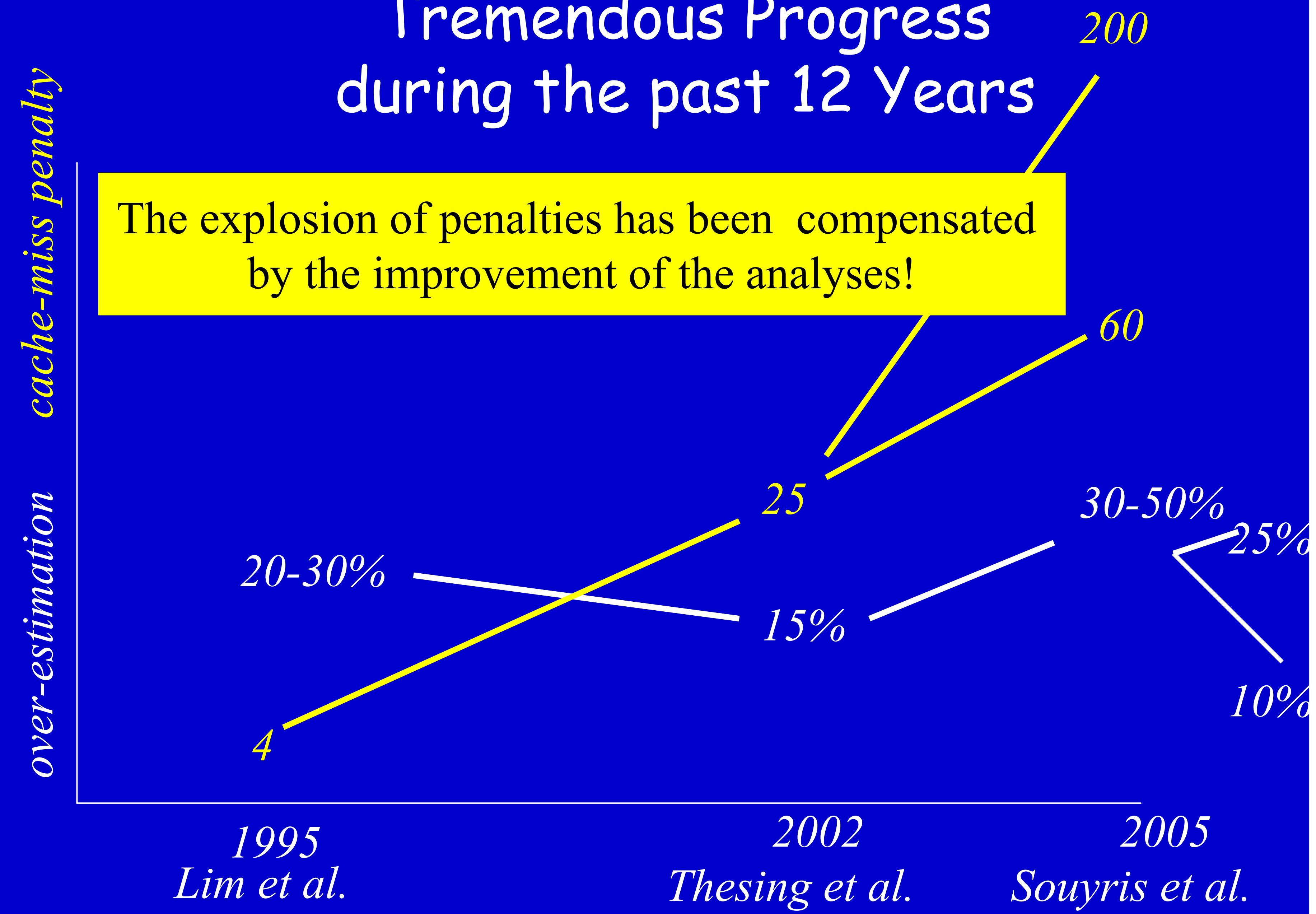
IST Project DAEDALUS final review report:

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



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 12 Years



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

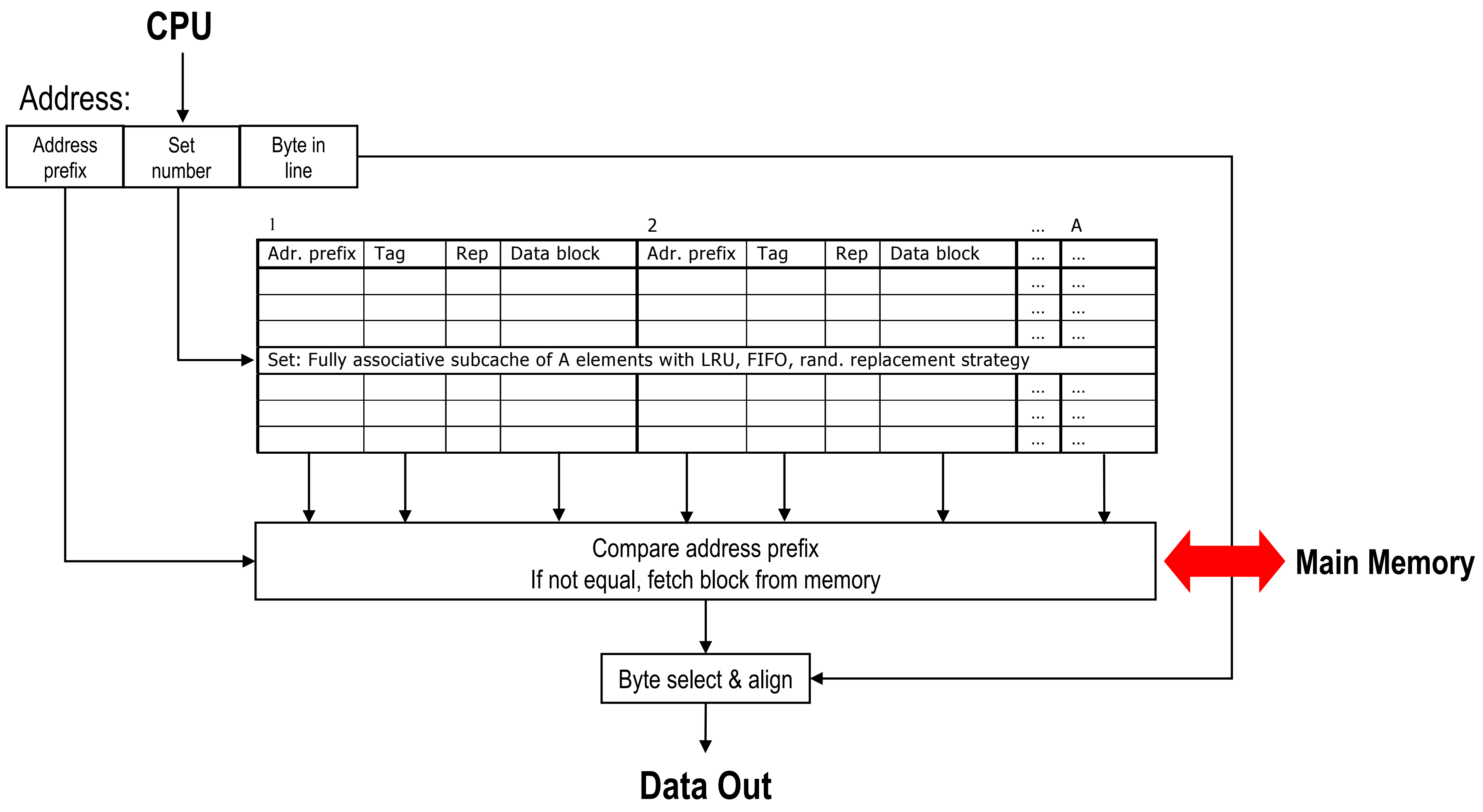
Caches: How the work

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

k-Way Set Associative Cache



Caches

- Static cache analysis, i.e. the sound prediction of must- and may-caches at program points is possible.
- Compact representation and efficient analysis.
- Precision depends on the replacement strategy – LRU is best.
- Caches are sensitive to the initial contents – LRU is best.
- Alternative is compiler-controlled scratchpad, however, sensitive to interference.

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

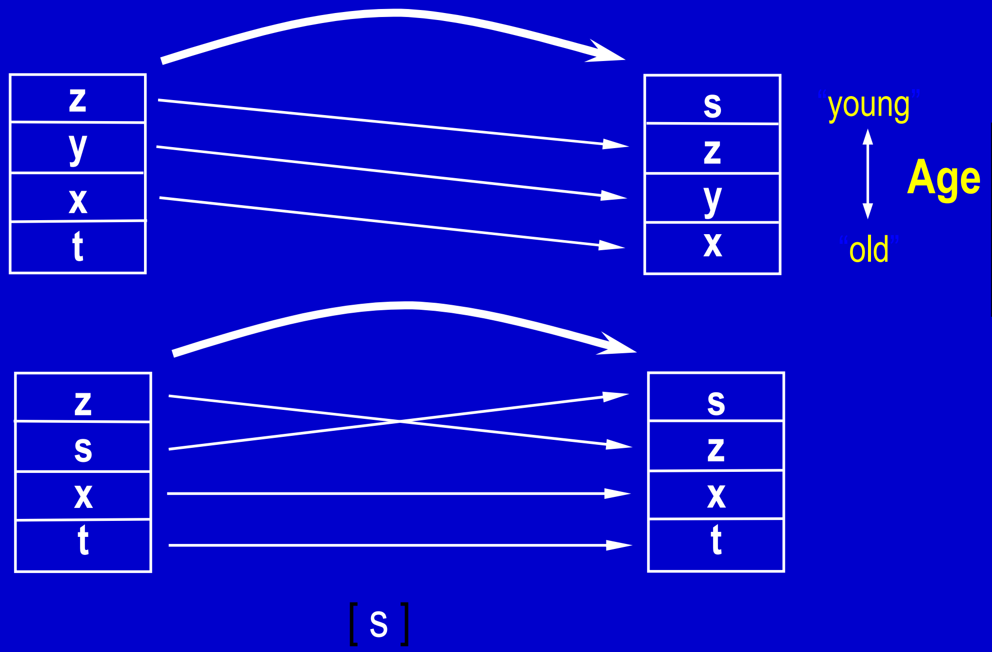
Must-Cache and May-Cache- Information

- 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

concrete

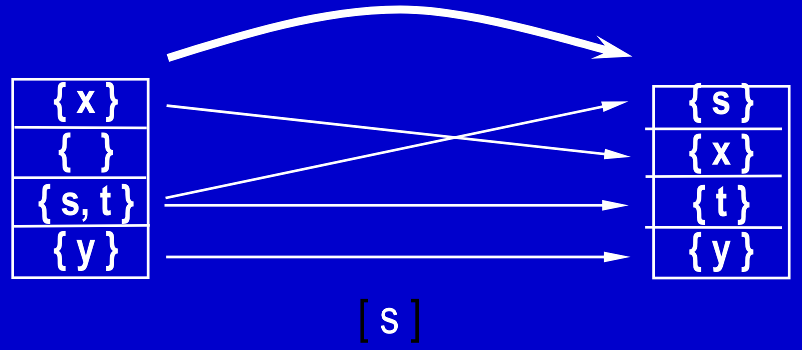
(processor)



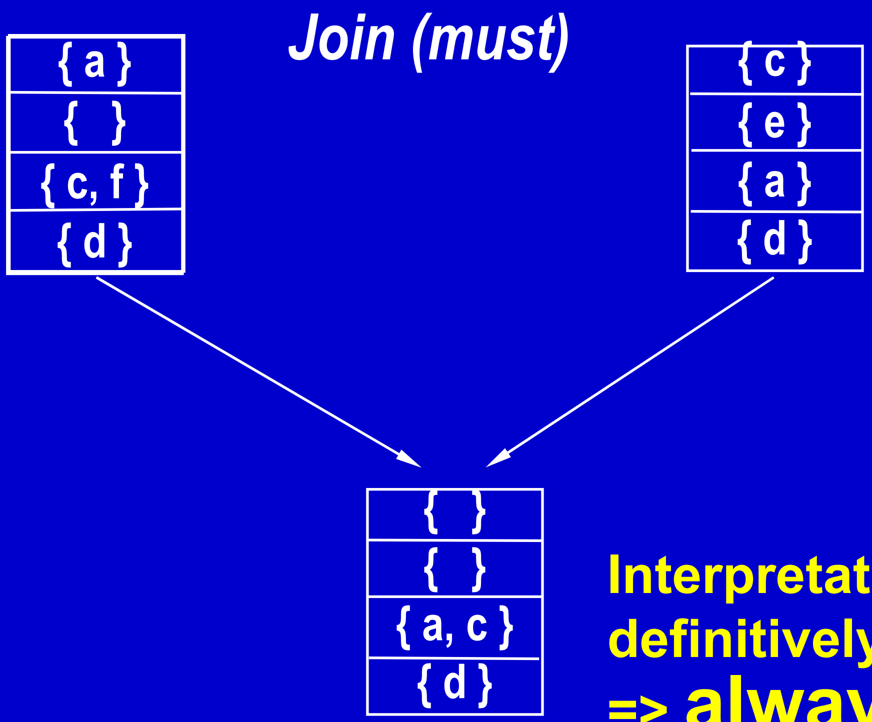
LRU has a notion of AGE

abstract

(analysis)



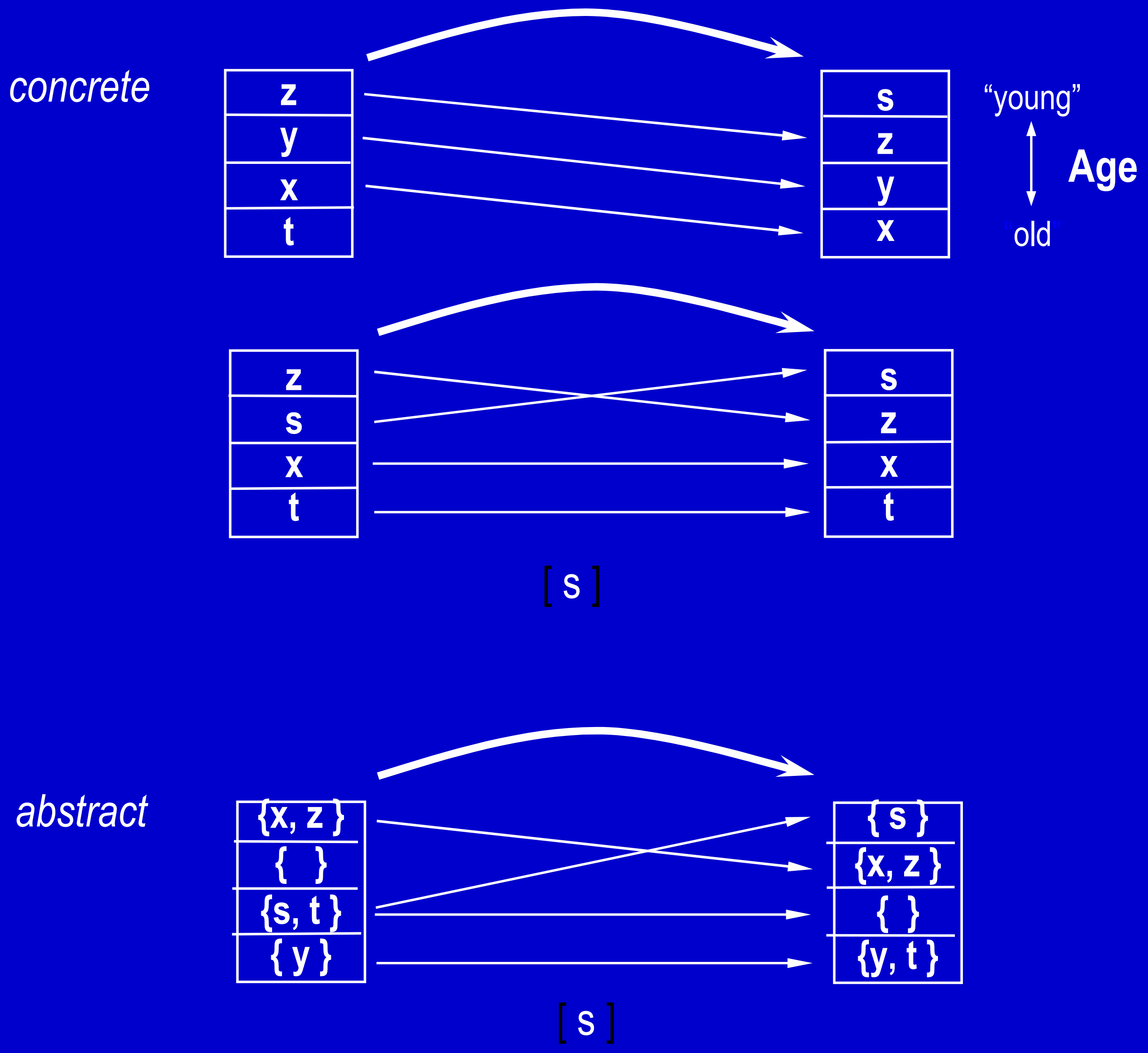
Cache Analysis: Join (must)



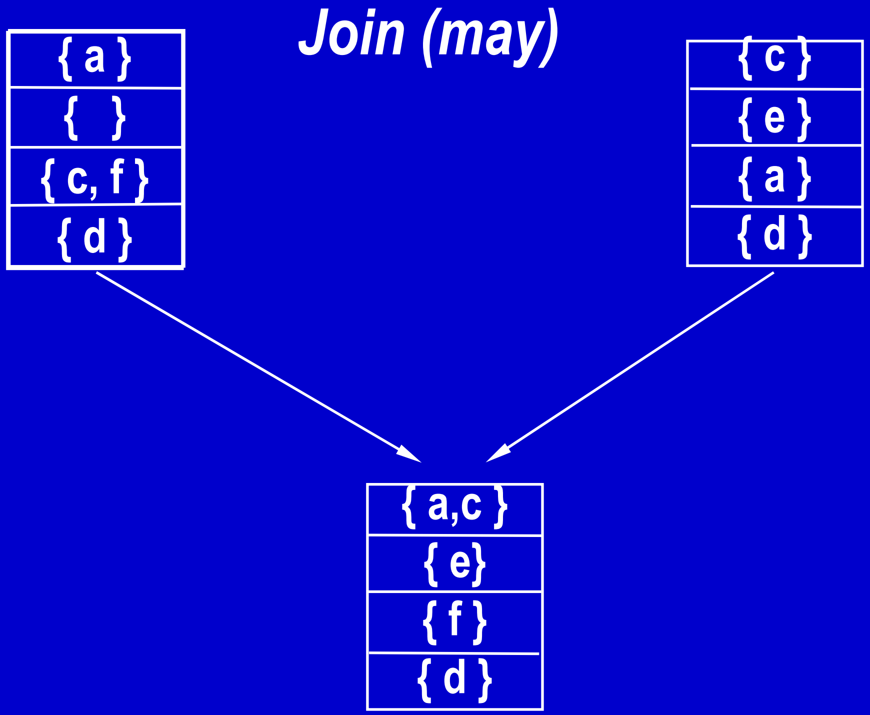
"intersection + maximal age"

Interpretation: memory block a is definitively in the (concrete) cache => always hit

Cache with LRU Replacement: Transfer for may



Cache Analysis: Join (may)



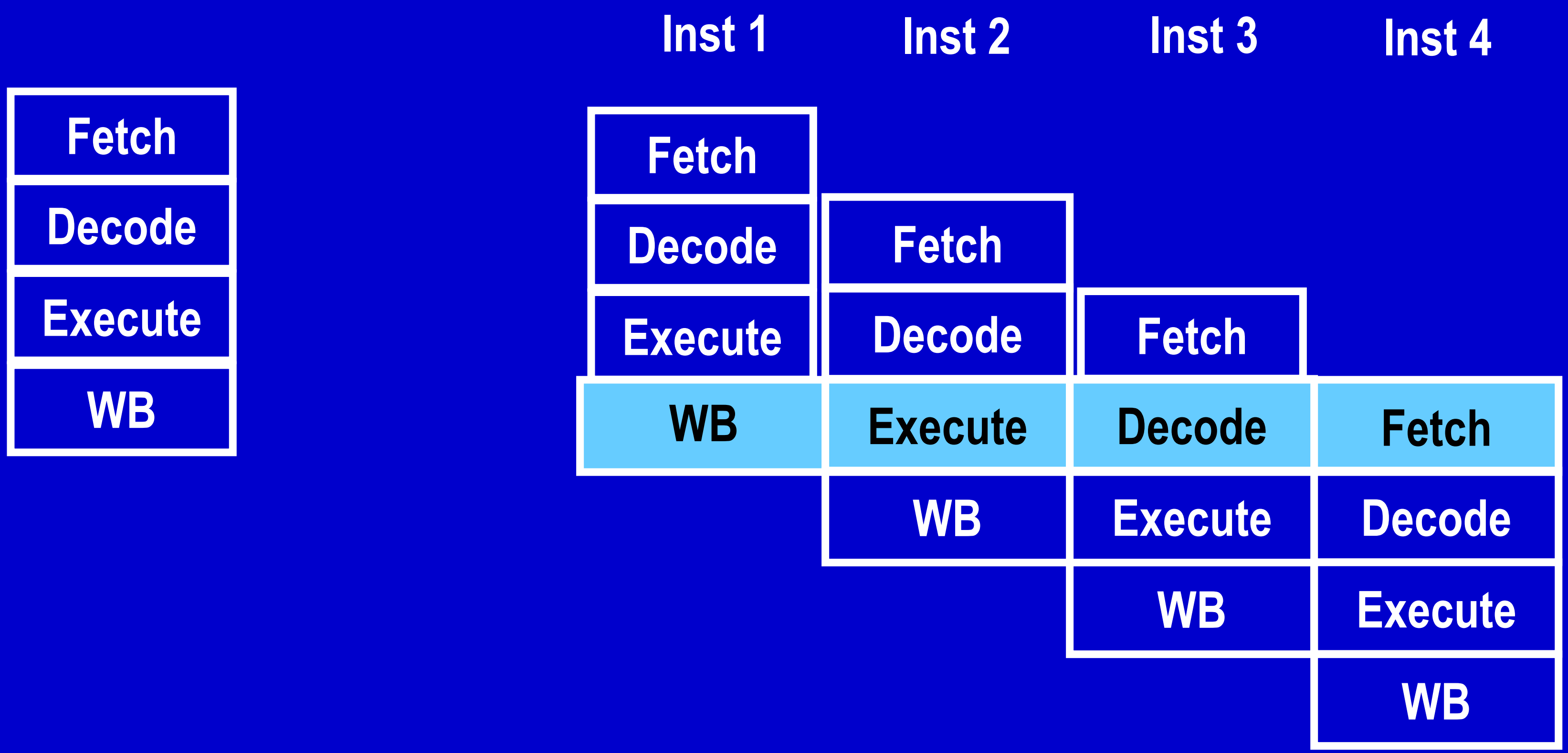
"union
+ minimal age"

Cache Analysis

Approximation of the Collecting Semantics

$$\begin{tikzcd}
\text{the semantics} \arrow[r, "\text{determines}"] & \text{set of all cache states for each program point} \\
\text{“cache” semantics} \arrow[r, "\text{determines}"] & \text{set of all cache states for each program point} \arrow[u, phantom, "\rotatebox{90}{$\supseteq$}"] \\
\text{abstract semantics} \arrow[r, "\text{determines}", "\text{PAG}"'] & \text{abstract cache states for each program point} \arrow[u, phantom, "\rotatebox{90}{$\supseteq$}"' near start, "conc" near end]
\end{tikzcd}$$

Pipelines



Ideal Case: 1 Instruction per Cycle

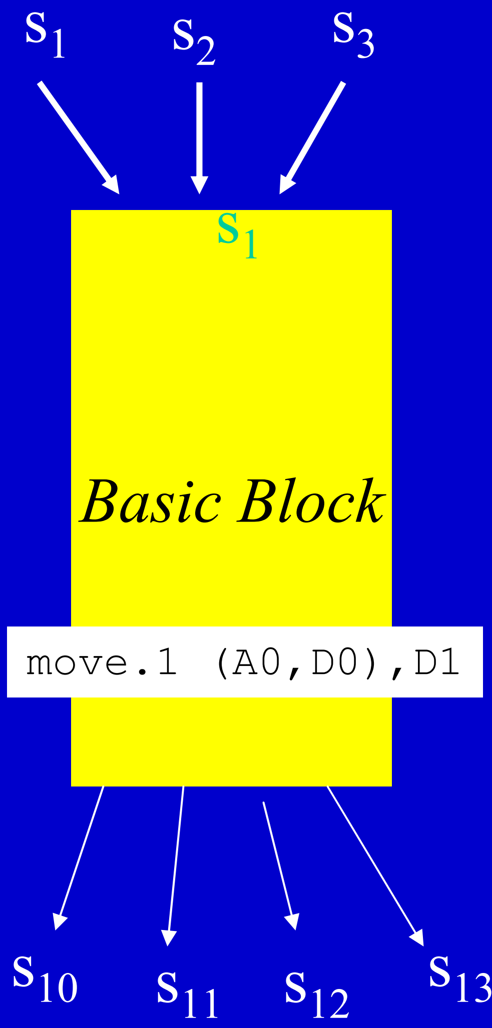
CPU as a (Concrete) State Machine

- 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

Pipeline Analysis

- simulates the concrete pipeline on abstract states
- counts the number of steps until an instruction retires
- non-determinism resulting from abstraction and timing anomalies require exhaustive exploration of paths

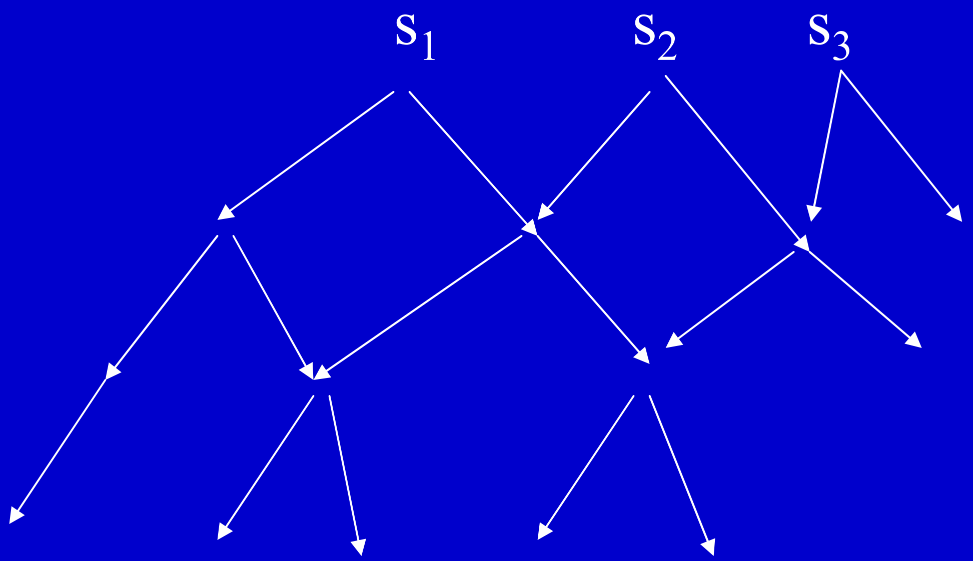
Integrated Analysis: Overall Picture



```
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



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 bound
- From this, we can obtain bounds for basic blocks

Classification of Pipelined Architectures

- 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

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

Path Analysis

by Integer Linear Programming (ILP)

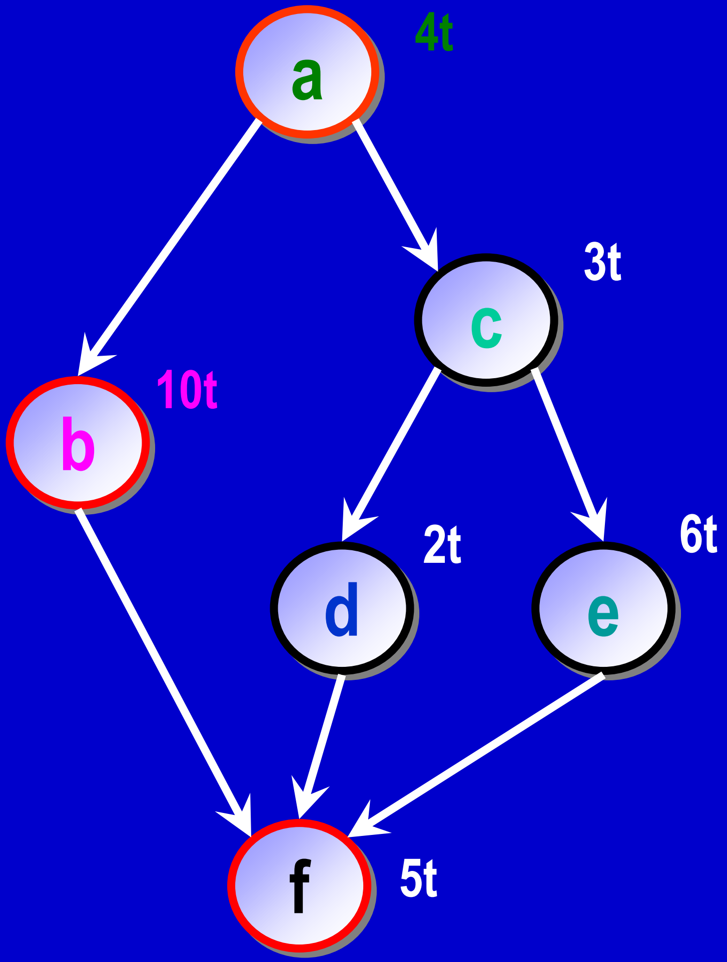
- Execution time of a program =

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

- 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

Structure of the Talk

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?

Making Life Easier

PREDATOR

Goal: Reconcile (average-case) performance with (worst-case) predictability.

Simplify the semantics, more precisely the architecture, if it is too complex:

- hard to provide sound timing analyses for ever more complex architectures,
- they are optimized for the wrong target, anyway.

Scalability of analyses and precision of the results are often correlated.

Objectives of PREDATOR

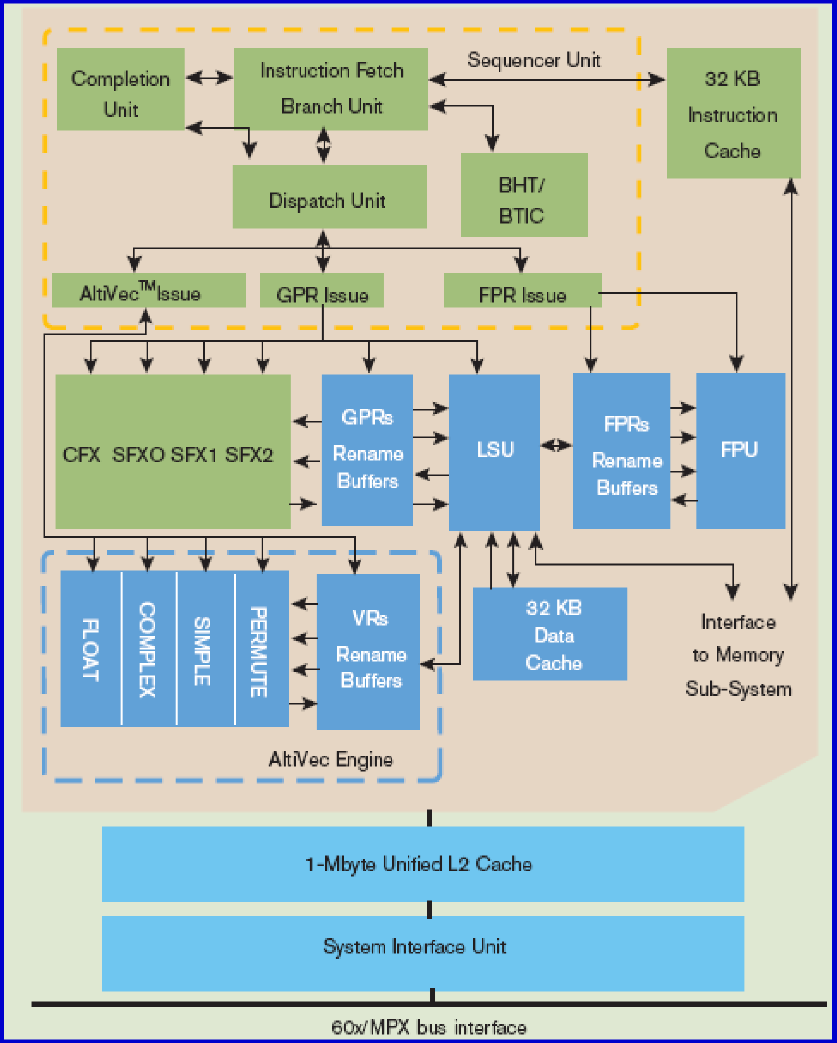
Identify good points in the 3-dimensional space of

- predictability (of the worst case),
- performance (in the average case),
- efficiency of verification methods.

Develop design methods for timing-predictable and performant systems

Processor Features of the MPC 7448 (just to show how bad things are getting)

- 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

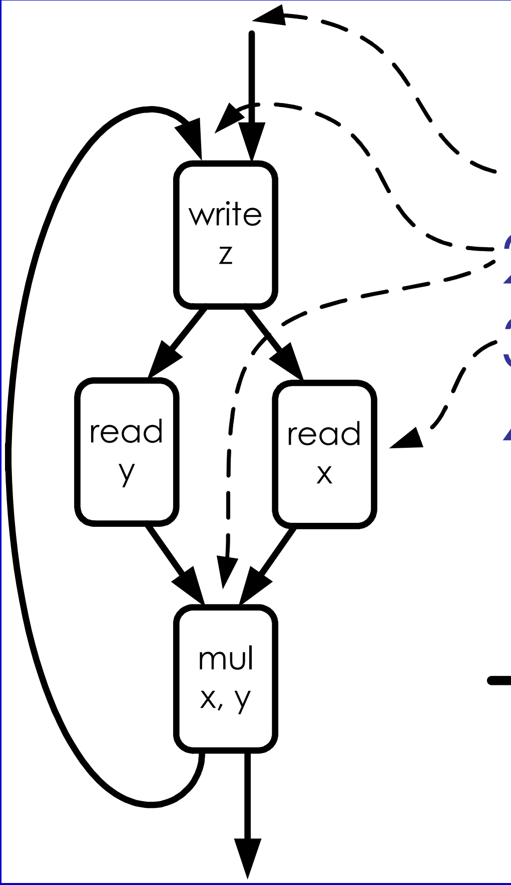
Architectural Complexity implies Analysis Complexity

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

Predictability of Cache Replacement Policies

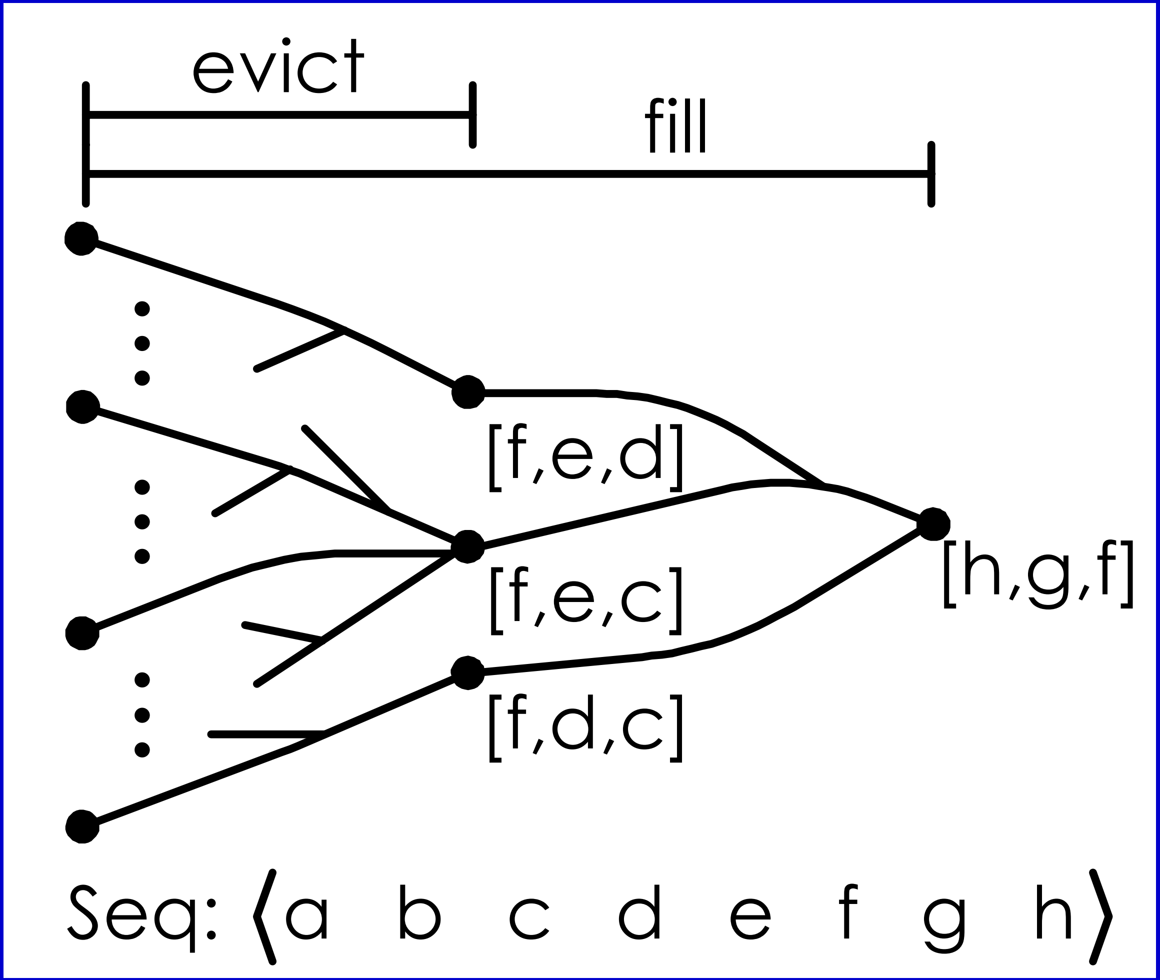
Uncertainty in Cache Analysis



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

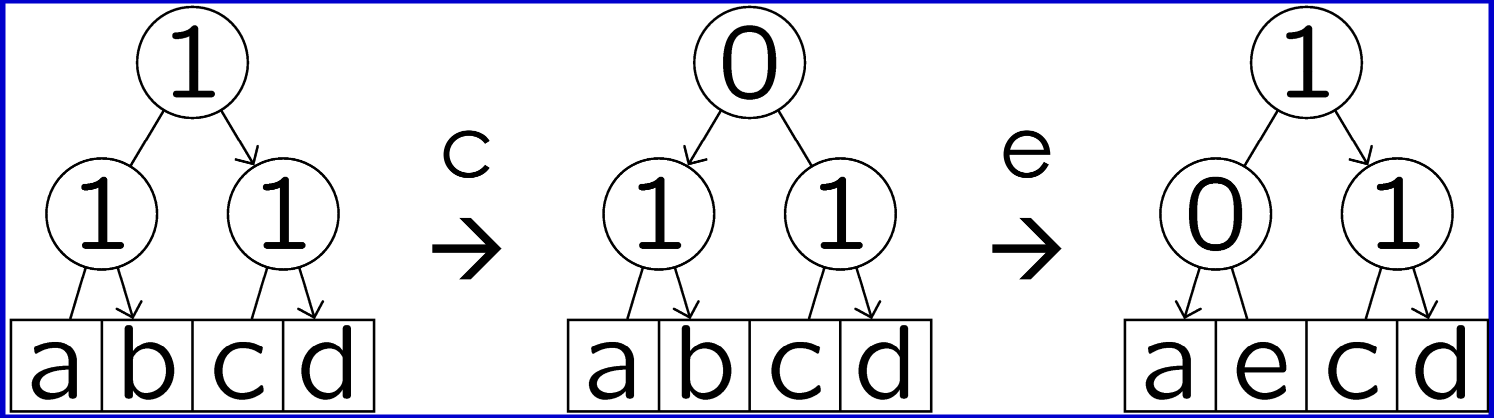
$$\begin{tikzcd} {[a,b,c,d]_{0101}} \arrow[d, bend right, "e"'] \\ {[e,b,c,d]_{1101}} \arrow[d, bend right, "c"'] \arrow[loop right, "{b,d}"] \\ {[e,b,c,d]_{0010}} \end{tikzcd}$$

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

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

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?

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

Extended the Predictability Notion

- The cache-predictability concept applies to all cache-like architecture components:
- TLBs, BTBs, other history mechanisms
- It does not cover the whole architectural domain.

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

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

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

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

Memory Hierarchy Example (S. Baruah)

The highest shared level in the memory hierarchy determines

- efficiency for accessing shared state and shared code or for migrating tasks
- the effort needed for timing analysis and the precision of the results

$\Rightarrow$ conflict: average-case performance of access will be better **but** bounds will be worse

Predictability - the Inner View

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

Variability of Execution Times

- often caused by the interference on shared resources
  - instructions interfer on the caches
  - bus masters interfer on the bus
  - several threads interfer on shared caches

Two Sources of Variability

caused by

- the input
- the execution-state

The PRET-machine people, Ed Lee and Stephen Edwards, attempt to balance out both – all executions of the tasks should take the same time.

PROMPT Design Principles for Predictable Systems

- reduce interference on shared resources in architecture design
- avoid introduction of interferences in mapping application to target architecture

Applied to Predictable Multi-Core Systems

- Private resources for non-shared components of applications
- Deterministic regime for the access to shared resources

Conclusions

- The determination of safe and precise upper bounds on execution times by static program analysis and Integer Linear Programming essentially solves the problem.
  Ongoing work:
  - Incorporation of preemption-caused costs,
  - timing analysis of heap-manipulating programs,
  - semi-automatic derivation of abstract processor models
- Precision greatly depends on predictability properties of the system
  - notion needs further clarification, criteria to be used in design

Relevant Publications

- *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
- *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
- *L. Thiele, R. Wilhelm: Design for Timing Predictability,* Real-Time Systems, *Dec. 2004*
- *R. Wilhelm: Determination of Execution Time Bounds,* Embedded Systems Handbook, CRC Press, 2005
- *St. Thesing: Modeling a System Controller for Timing Analysis,* EMSOFT 2006
- *J. Reineke et al.: Predictability of Cache Replacement Policies,* Real-Time Systems, Springer, 2007
- *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, accepted by* IEEE TCAD