ARTIST Summer School in Europe 2009

Autrans (near Grenoble), France

September 7-11, 2009

Timing Analysis and Timing Predictability

Reinhard Wilhelm and Jan Reineke

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).
- Deadlines are often in the order of mS and down to μS
- Timing Analysis provides the abstraction for Scheduling.

Deriving Run-Time Guarantees for Hard Real-Time Systems

Given:

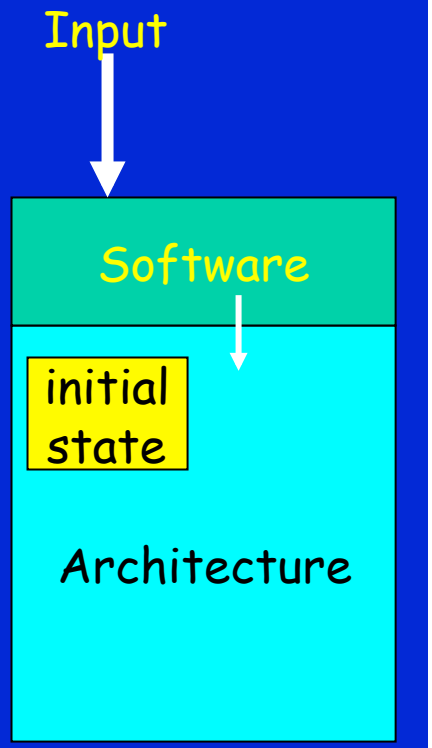
1. required reaction time,
2. a software to produce the reaction,
3. a hardware platform, on which to execute the software.

Derive: a guarantee for timeliness

What does Execution Time Depend on?

- the input,
- the initial and intermediate execution states of the platform,
- interferences from the environment – this depends on whether the system design admits it (preemptive scheduling, interrupts).

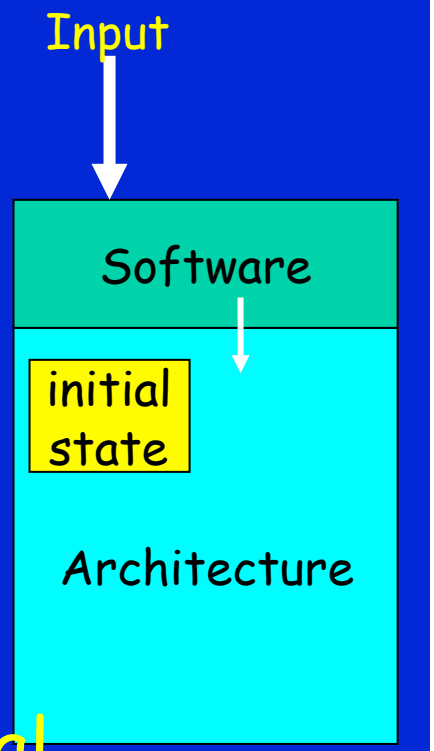
Caused by caches, pipelines, speculation etc. ⇒ Explosion of state space



"external" interference as seen from analyzed task, see Jan's lecture

Timing Analysis

- Sounds methods determine upper bounds for all execution times.
- They have to explore a huge space of transition paths
  - all control-flow paths – stemming from possible inputs
  - all paths through the architecture – resulting from the potential initial and assumed intermediate architectural states - enforced by the existence of timing anomalies.



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
5. Caches (Jan Reineke)
   - cache analysis,
   - cache predictability,
   - cache sensitivity,
   - cache interference
6. Pipeline analysis
7. Bounds analysis
8. Timing Predictability
9. Predictable multi-core architectures (on the way to…)
10. Conclusion

Timing Analysis – the Alternatives

- End-to-end measurement – execute or simulate the program
  - a couple of times – most of industry's "best practice", but unsafe ☹
  - exhaustively – too costly ☹
- Piecewise measurement and structured composition of the results – how to do it? ☹
- Static analysis – safe ☺
  but costly to implement ☹

Note: we have no decidability problem, but a complexity problem!
The architecture is finite, the input domain is, in general, finite, loops and recursion are bounded.
But the search space is too large to explore it exhaustively!

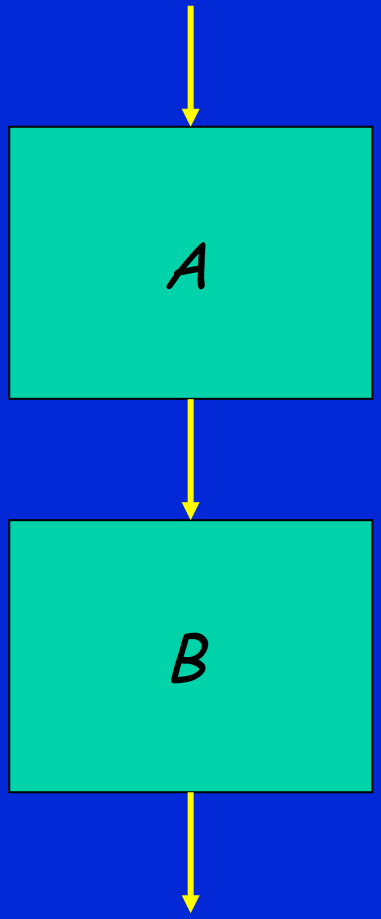
Piecewise Measurement

1. measure A (many times)

not sure to hit the worst case!

2. measure B (many times)

not sure to hit the worst case!

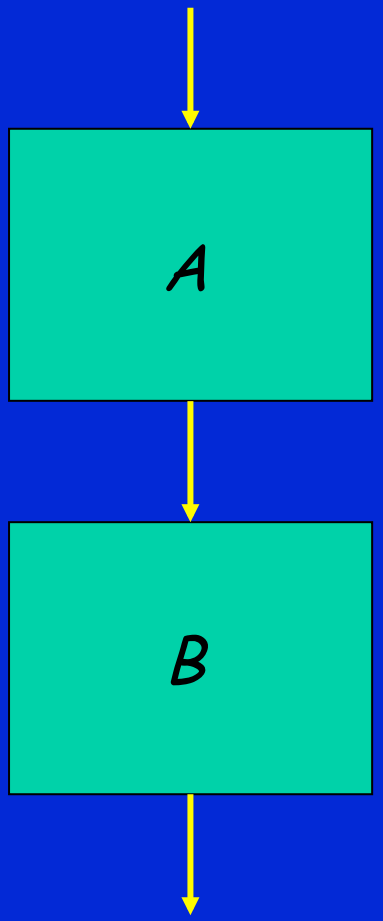


Program snippets A and B have many execution times depending on the execution state

3. How to combine the results?
-add worst-case times: too pessimistic
- alternatives?

Another Piecewise Process

Assume to know Information about all potential execution states before A;
analyze A in this context;
determine how A transforms these states;
analyze B in the new context.



Result: more precise results for A and B; addition of times less pessimistic!

Requires: knowing the transitons between execution states, i.e. an operational semantics including the platform.

What makes the problem hard (and interesting)?

Execution time t of machine instructions i

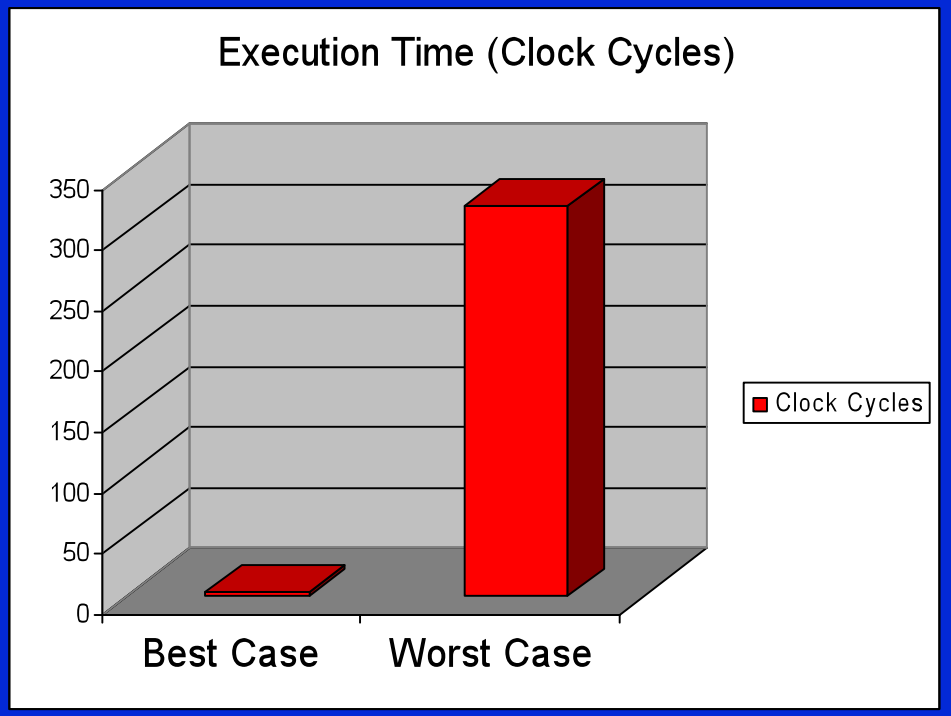
- in the good old times:
  $t(i) = c$ (c to be found in a table)
- in modern, high-performance processors:
  execution time depends on the execution state, so is $t(i,s)$
- The execution times of i may range between $\min\{t(i,s) \mid s \in S\}$ and $\max\{t(i,s) \mid s \in S\}$
- The execution state results from the execution history.

And the Variability of Execution Times is large!

x = a + b;

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

PPC 755

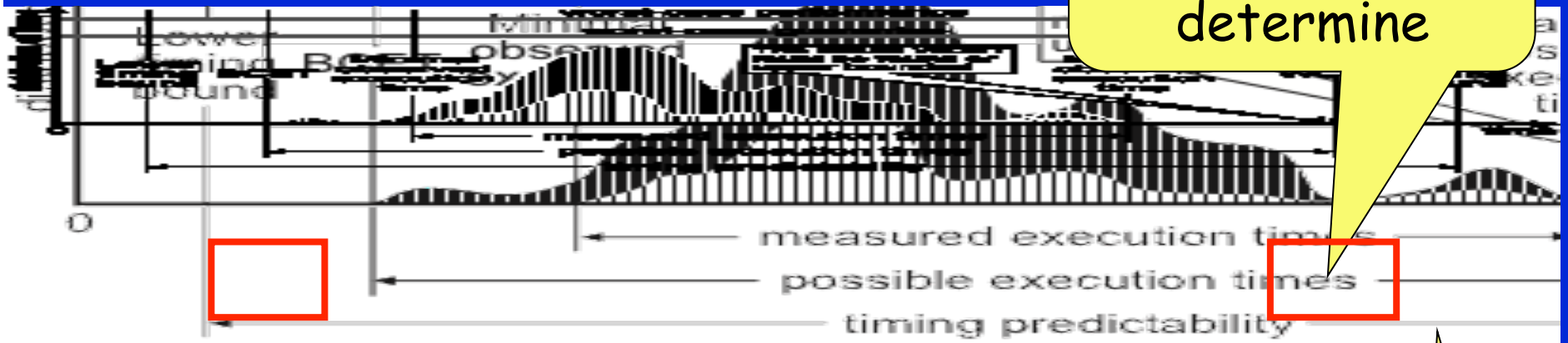


In most cases, execution will be fast.
So, assuming the worst case is safe, but very pessimistic!

Modern Hardware Features

- Modern processors 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, 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

Notions in Timing Analysis



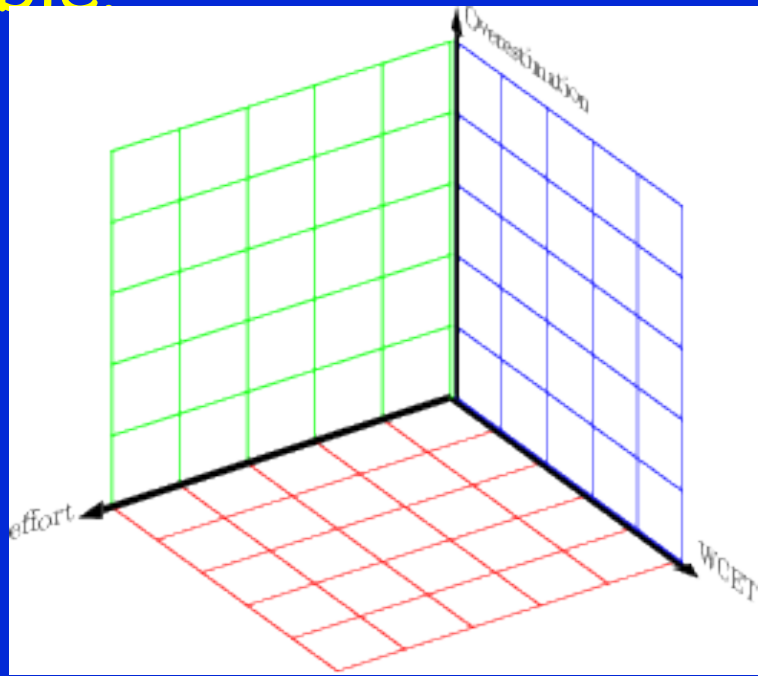
Hard or impossible to determine

Determine upper bounds instead

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.

Later on, we look at the predictability of architectures:
Designs will occupy points in a 3-dimensional space:
worst-case performance,
degree of overestimation,
required analysis effort.



Timing Accidents and Penalties

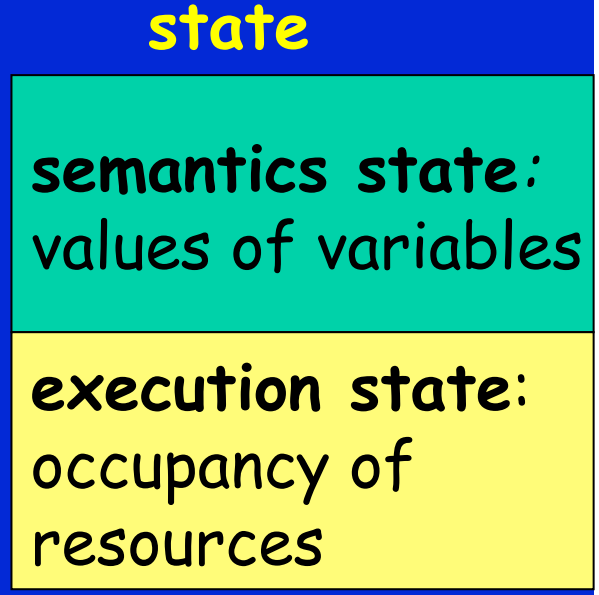
Timing Accident – cause for an increase of the execution time of an instruction

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
- computes invariants about the set of all potential execution states at all program points,
- the execution states result from the execution history,
- static analysis explores all execution histories



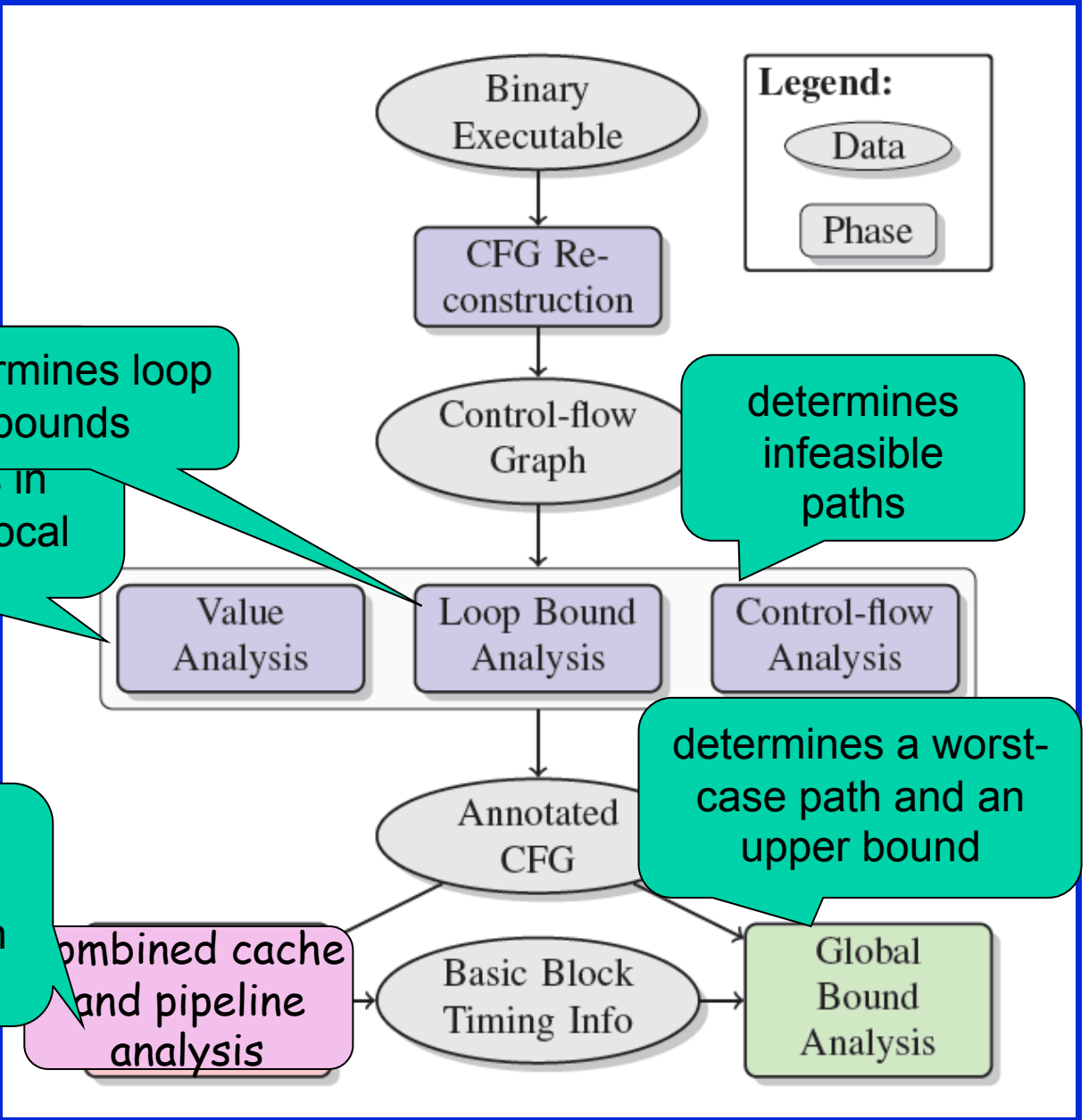
Deriving Run-Time Guarantees

- 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. Global-Bounds Analysis
   - Maps control flow to integer linear program
   - Determines upper bound for the whole program and an associated path

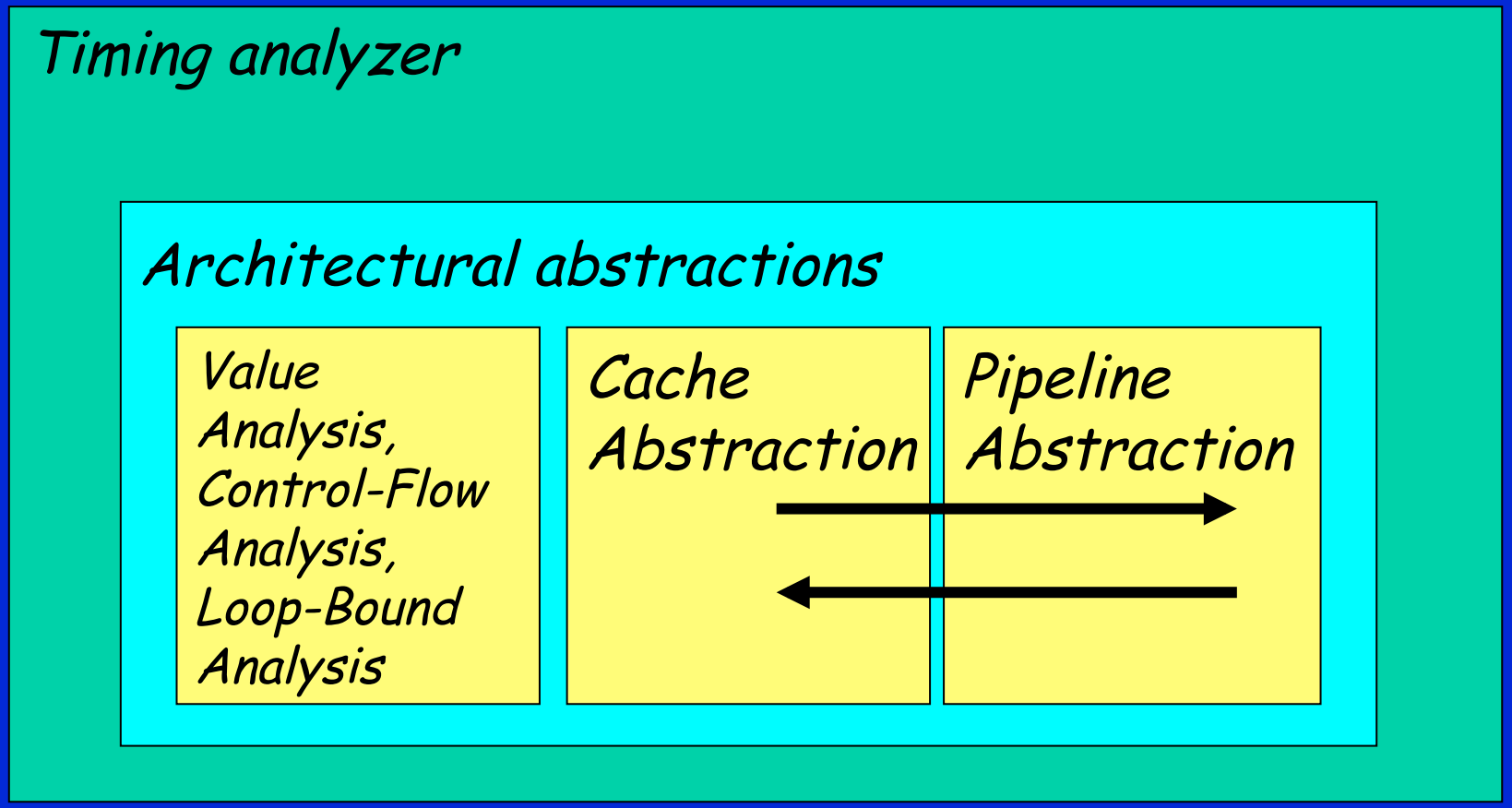
Tool Architecture



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.

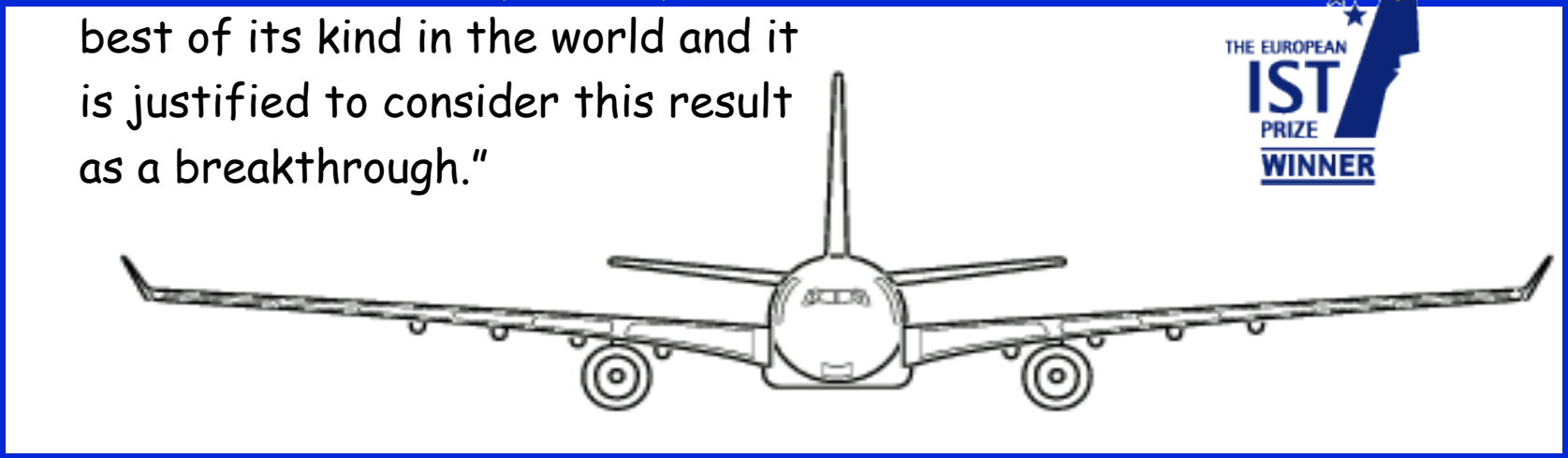
The Architectural Abstraction inside the Timing Analyzer



AbsInt's WCET Analyzer aiT

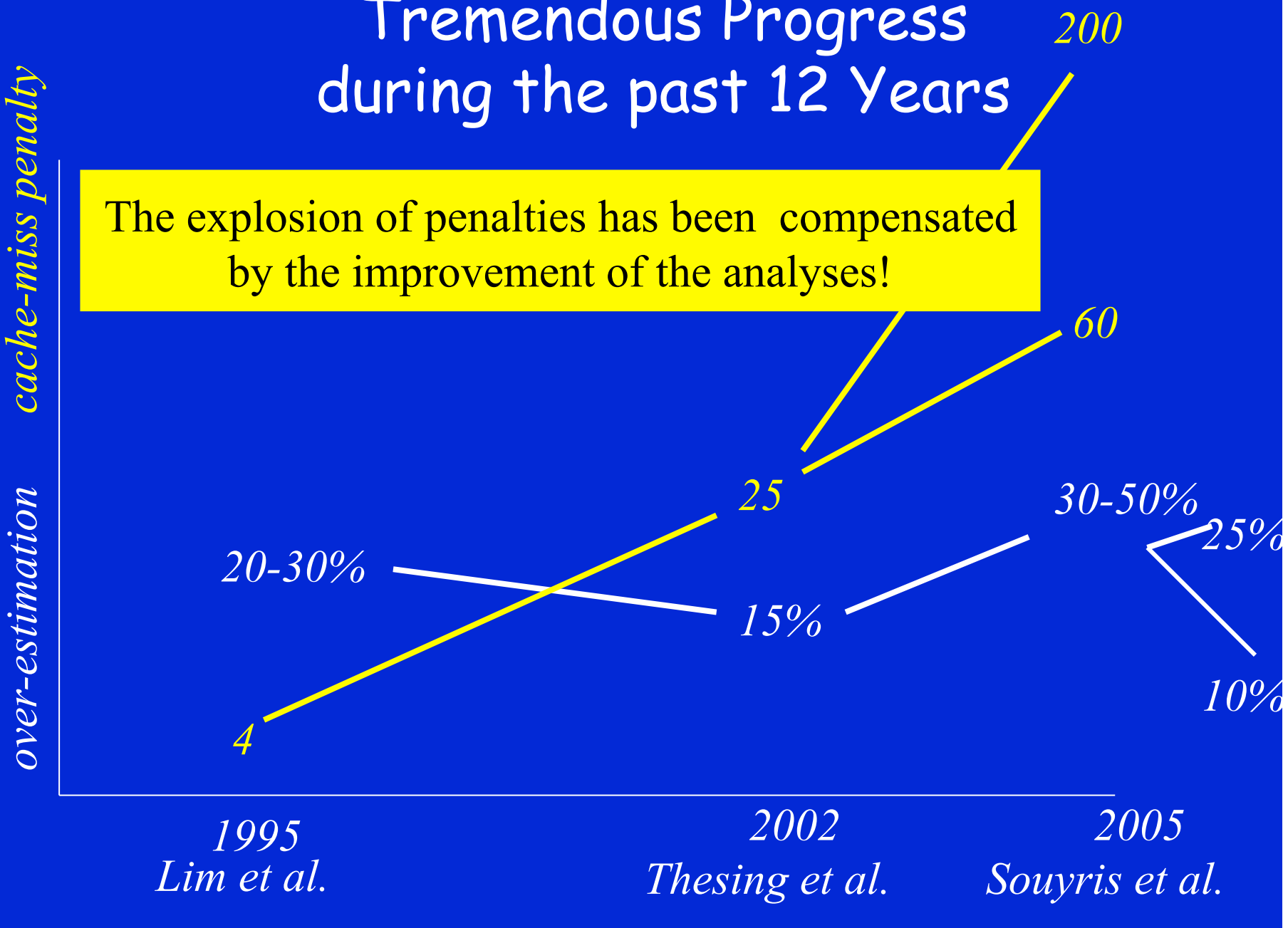
IST Project DAEDALUS final review report:

"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

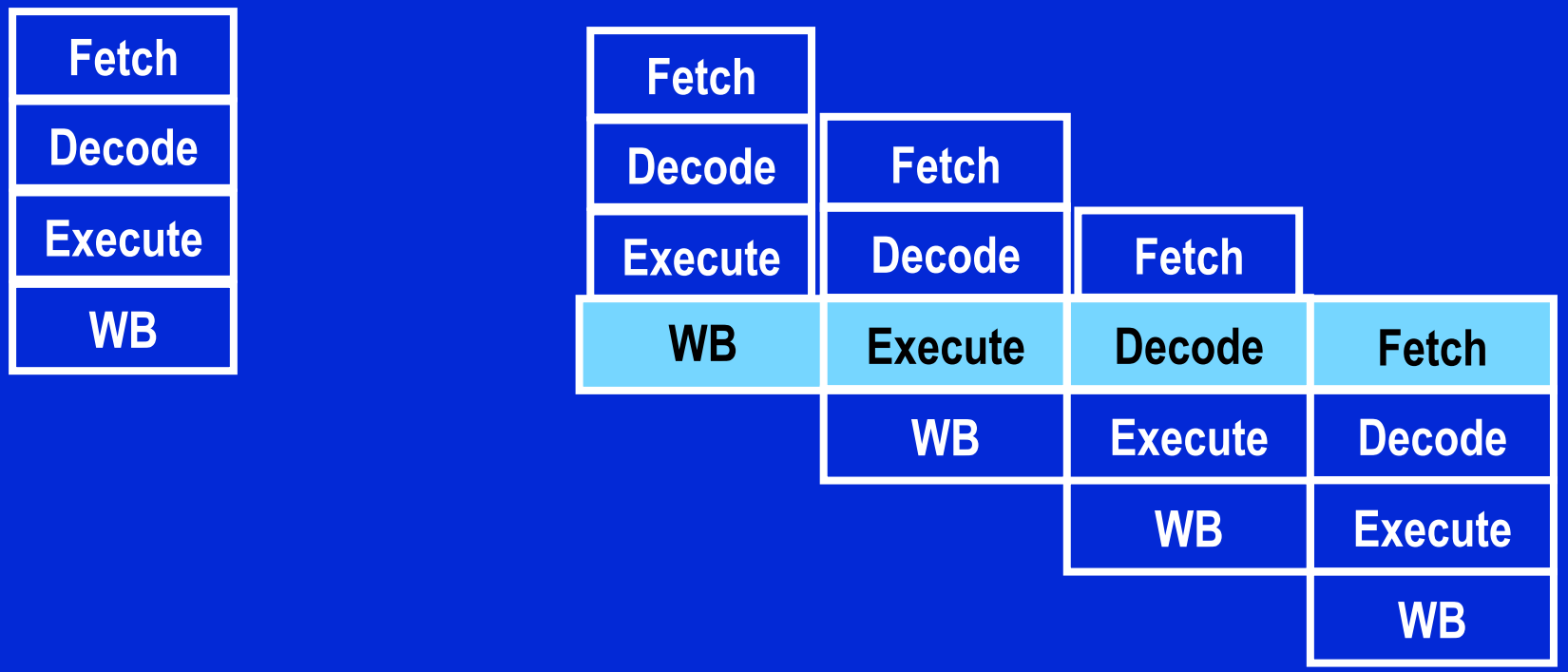


The explosion of penalties has been compensated by the improvement of the analyses!

Everything You (n)ever Wanted to Know about Caches

Jan Reineke

Pipelines



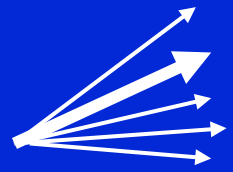
Ideal Case: 1 Instruction per Cycle

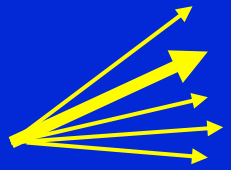
CPU as a (Concrete) State Machine

- 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 results from abstraction – more non-determinism from "stronger" abstractions
- timing anomalies require exhaustive exploration of paths.
- We didn't find nice abstractions as we did for the caches ⇒ large search space





A Concrete Pipeline Executing a Basic Block

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

interprets instruction stream of *b* starting in state *s* producing trace *t*.

Successor basic block is interpreted starting in initial state *last(t)*

length(t) gives number of cycles

An Abstract Pipeline Executing a Basic Block

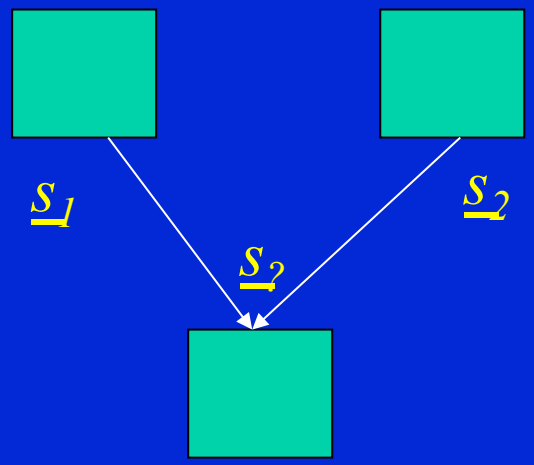
function exec (*b* : **basic block**, *s* : **abstract pipeline state**)
t: **trace**

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

length(t) gives number of cycles

What is different?

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



Alternatives:

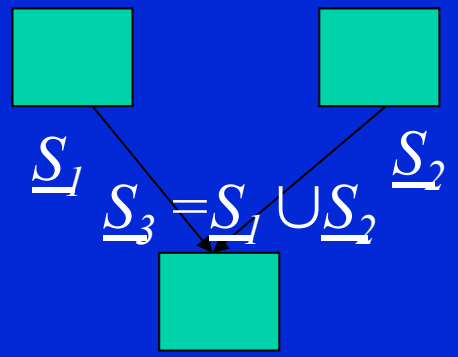
- *sets of states*
- *combine by least upper bound (join), so far none found that*
  - *preserves information and*
  - *has a compact representation.*

Non-Locality of Local Contributions

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

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

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



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

PS = set of abstract pipeline states

CS = set of abstract cache states

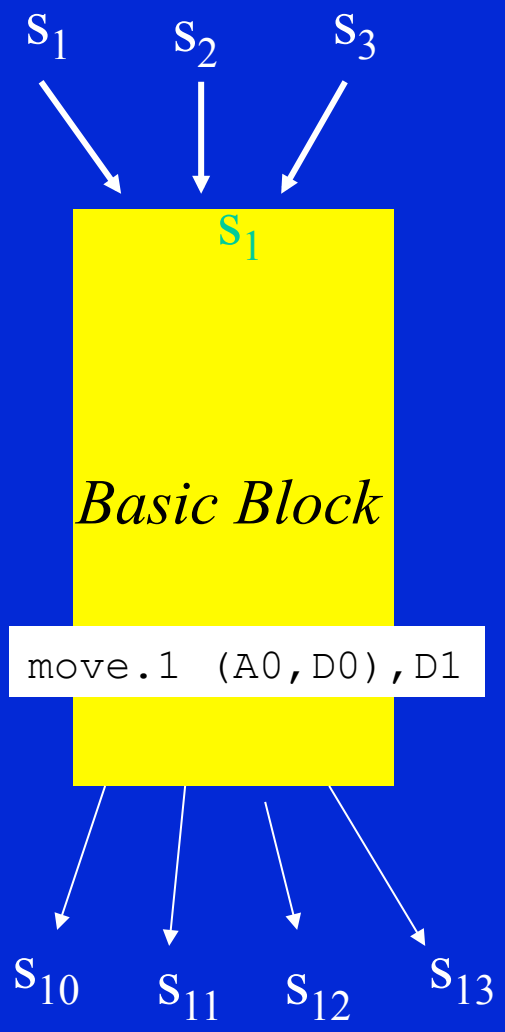
interprets instruction stream of *b* (annotated with cache information) starting in state *S* producing set of traces *T*

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

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

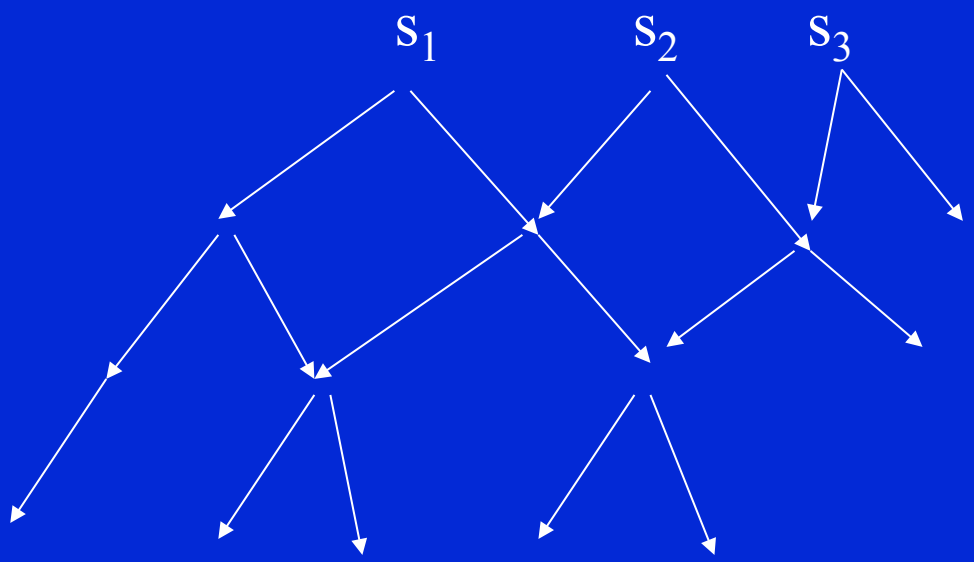
Union for blocks with several predecessors.

Integrated Analysis: Overall Picture



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

Cyclewise evolution of processor model for instruction



Classification of 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

Path Analysis

by Integer Linear Programming (ILP)

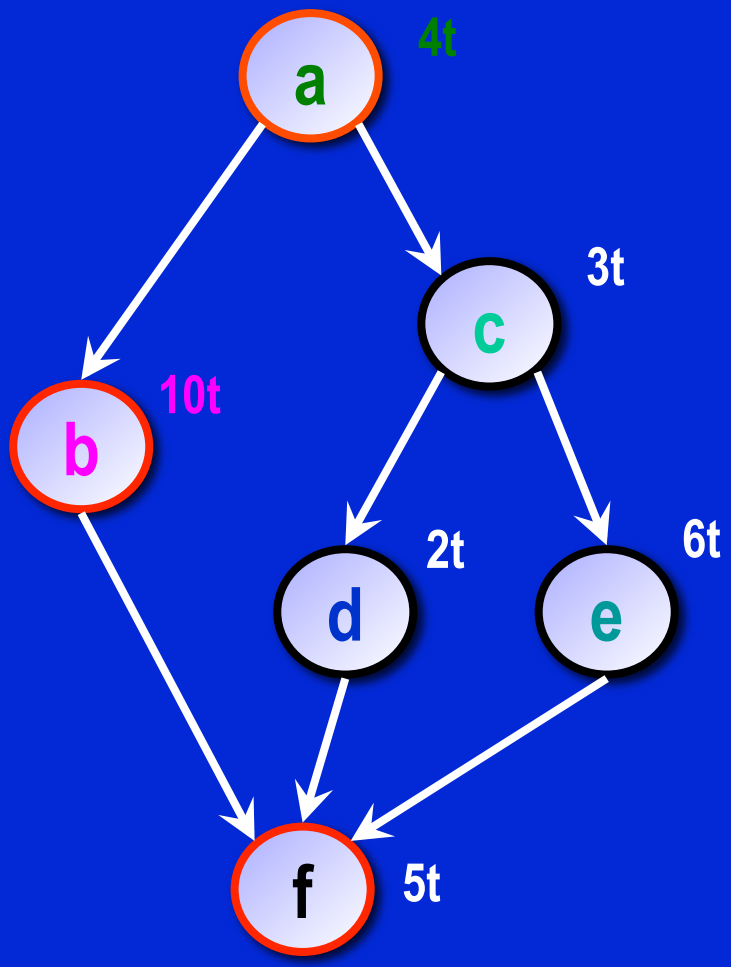
- Execution time of a program =

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

- 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

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
5. Caches (Jan Reineke)
   - cache analysis,
   - cache predictability,
   - cache sensitivity,
   - cache interference
6. Pipeline analysis
7. Bounds Analysis
8. Predictable multi-core architectures (on the way to…)
9. Conclusion

Timing Predictability

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

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

Cache predictability (see Jan's talk) argues over all memory-access sequences, is independent of the software to analyze.

Can we influence the set of access sequences to caches or, in general, to shared resources?
***Design issue**: reduce the set of possible access sequences!*

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

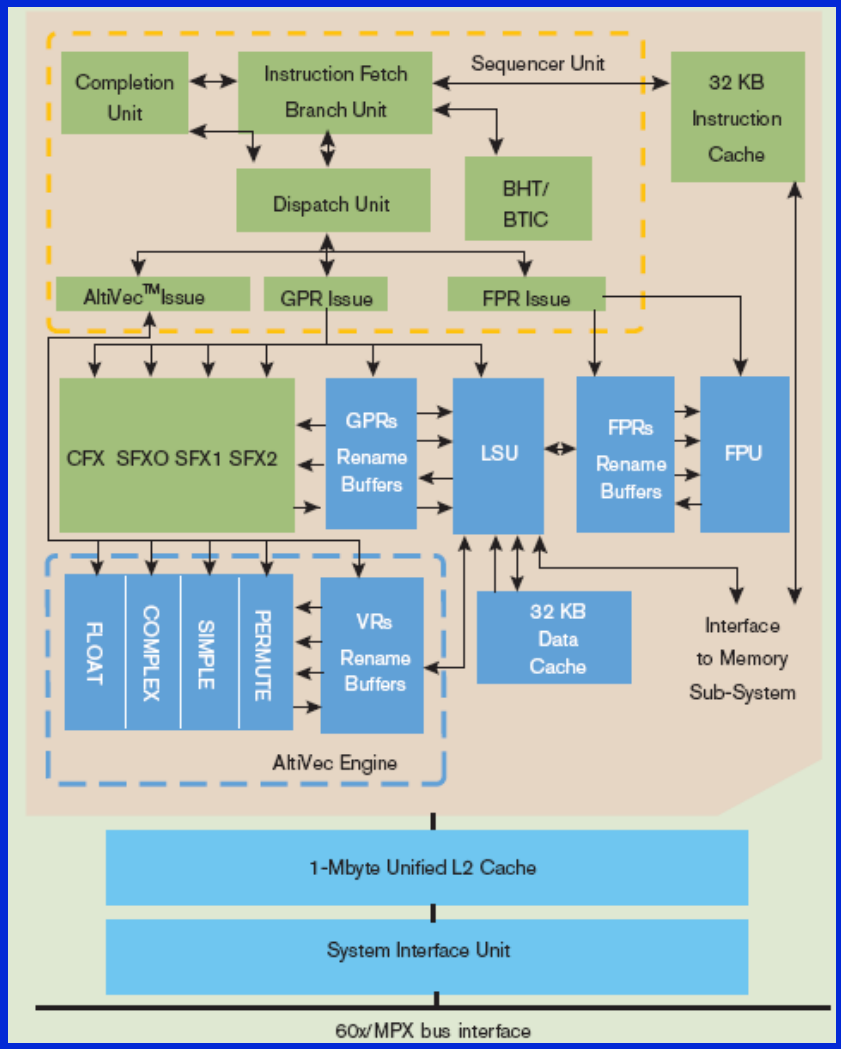
Identify good points in the 3-dimensional space of

- performance (in the worst case),
- efficiency and precision 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,
- may contribute a multiplicative factor to the size of the search space
- Exception: Caches
  - some have good abstractions providing for highly precise analyses (LRU), cf. Diss. of J. Reineke
  - some have abstractions with compact representations, but not so precise analyses

The Predictability Notion

- Hypothesis: Predictability = Analyzability
- Analyzability means
  - efficiently analyzable with
  - precise results, i.e. small overestimation
- How does this match the cache-predictability notion?
  The cache-predictability metrics
  - give bounds on what can be found out,
  - correlate with the existence of compact abstract domains supporting efficient analyses.

Further dimension (beyond precision and efficiency):
Worst-case performance – should not suffer too much.
Yet another dimension: Cost

The Main Culprit: Interference on Shared Resources

- They come in many flavors:
  - instructions interfere on the caches,
  - bus masters interfere on the bus,
  - several threads interfere on shared caches.
- some directly cause variability of execution times, e.g. different bus access times in case of collision,
- some allow for different interleavings of control or architectural flow resulting in different execution states and subsequently different timings.

NB: The problem is not interference changing the semantics, but interference leading to different timing behaviors!

Analysis of the Interference on Caches

- Out-of-order processor executes an instruction sequence ⇒
  - several different memory access sequences
  - with different intermediate and final cache contents and
  - different execution times.
- Preemptive scheduling ⇒
  - many different interleavings of preempted and preempting tasks ⇒
  - uncertainty about cache contents ⇒ large overestimation.
- Multithreading with shared caches ⇒
  - many different interleavings,
  - larger search space,
  - less precision.

Taking Constructive Influence - the PROMPT Approach -

Making applications running on multi-core / multi-processor systems analyzable

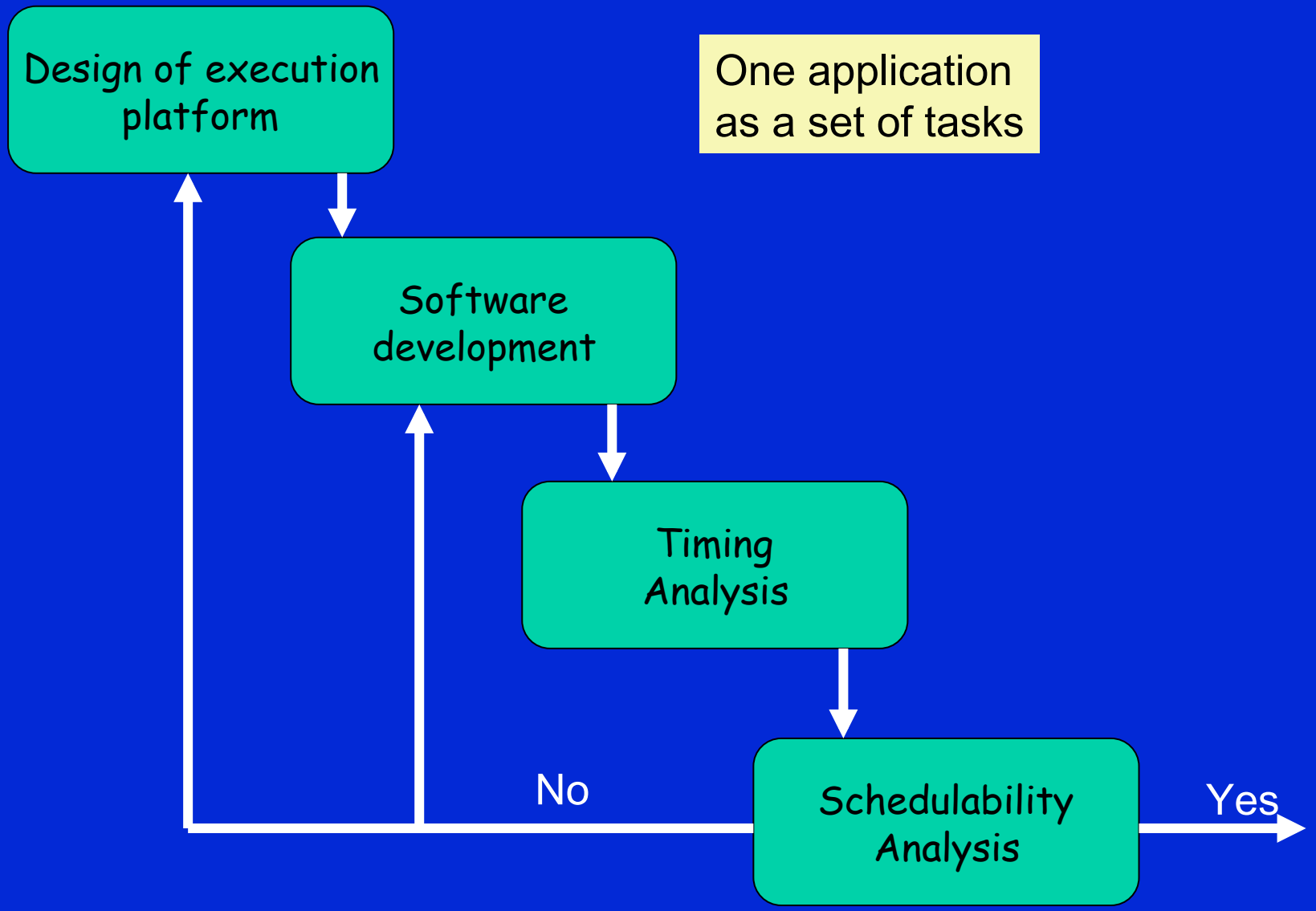
- Remember the metrics for Cache-Predictability: independent of the software to analyze, defined over all memory-access sequences.
- Monotonicity: less access sequences ⇒ better values under these metrics.
- In analogy, reduced interference on shared resources ⇒ less interleavings ⇒
  - smaller analysis effort,
  - higher precision.

Multi-core implementations require mapping applications to cores – one point of attack.

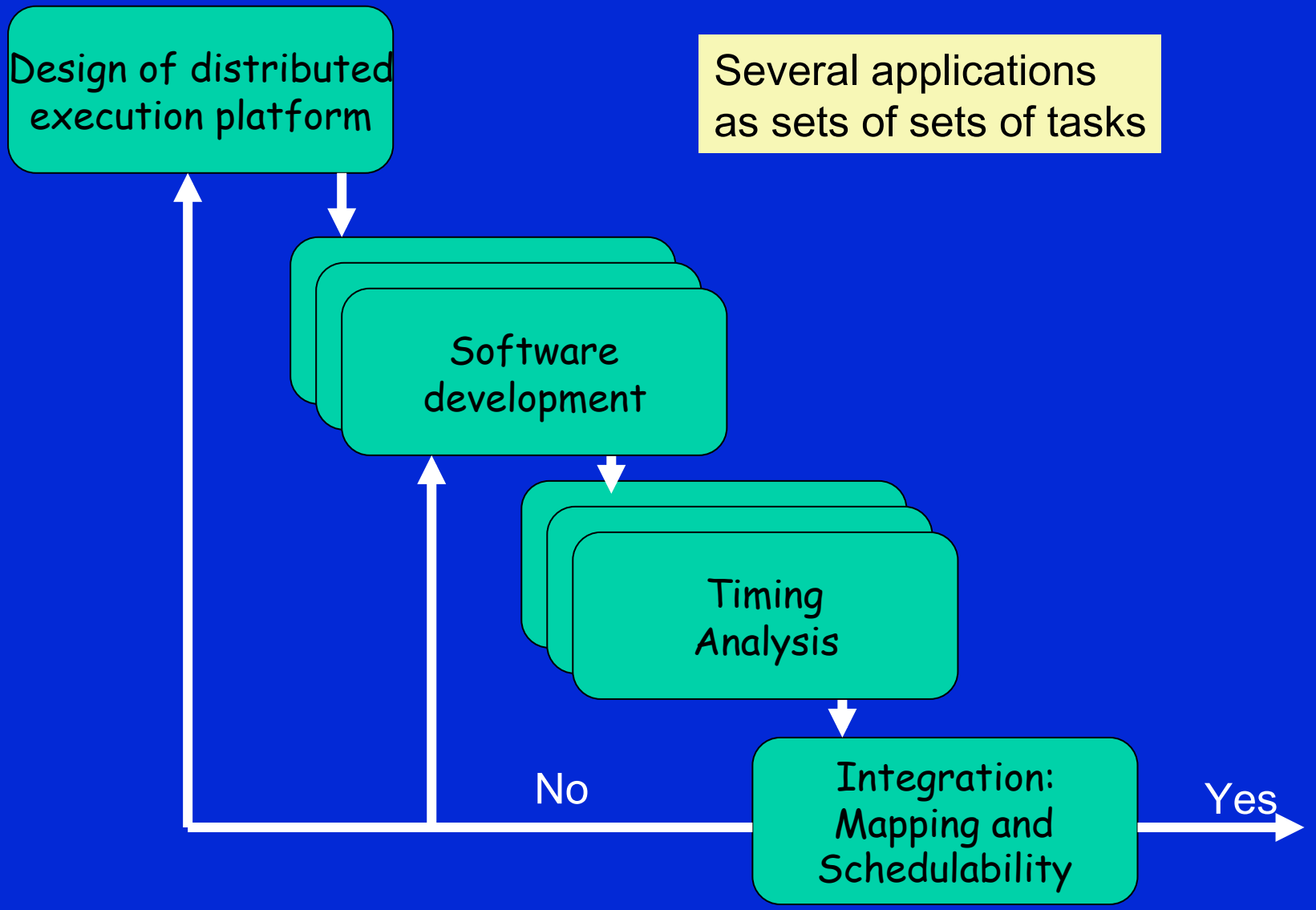
Restriction to Embedded System in the Avionics and Automotive Domains

- Goal is not the general purpose multi-core architecture with good predictability

Traditional System Design Process



System Design Process with Integration of Applications



Application Domains I

- Architectures for safety- and time-critical avionics and automotive systems
- system characteristics:
  - combination of control loops and finite-state control
  - each control loop fully contained in one application
  - little shared code
  - global (finite) state partly shared between applications;
  - state transitions influence control parameters,
  - control loops trigger state transitions
  - reading from and writing to shared state happens only at the beginning and at the end of task activations
  - some applications require high performance, but share little with the control applications

Application Domains II

- Similar integration trends, IMA and AUTOSAR, integrating applications on powerful platforms instead of 1-application-per-platform/ECU
- More complex development process – Mapping a set of applications to nodes of a platform.
- Goal is Composability:
  timing behavior of one task is independent of that of the other tasks integrated on the same platform.
  - IMA: incremental qualification, i.e. modification of one application integrated with a set of other applications only requires re-certification of the modified component.

IMA and AUTOSAR - New safety problems

- IMA
  - ensures logical non-interference by temporal and spatial partitioning,
  - but no consideration of resource interference, ⇒ no incremental qualification!
  - resource interference must be avoided to achieve predictability
- AUTOSAR
  - composability only achievable on predictable platforms

Observations II

Performance of many control computers is dominated by the performance of the memory subsystem

- holds for many safety-critical avionics applications,
- many automotive applications are executed out of FLASH memory, limiting performance.

Consequences:

- extremely complex pipelines, e.g. out-of-order, highly parallel, speculating, essentially wait!
- pipeline modeling is the most complex task in the construction of an instance of aiT!
- adding more cores speeds up waiting!

Dealing with Shared Resources

Alternatives:

- Avoiding them,
- Bounding their effects on timing variability

The PRET Architecture (Edwards/Lee et al.)

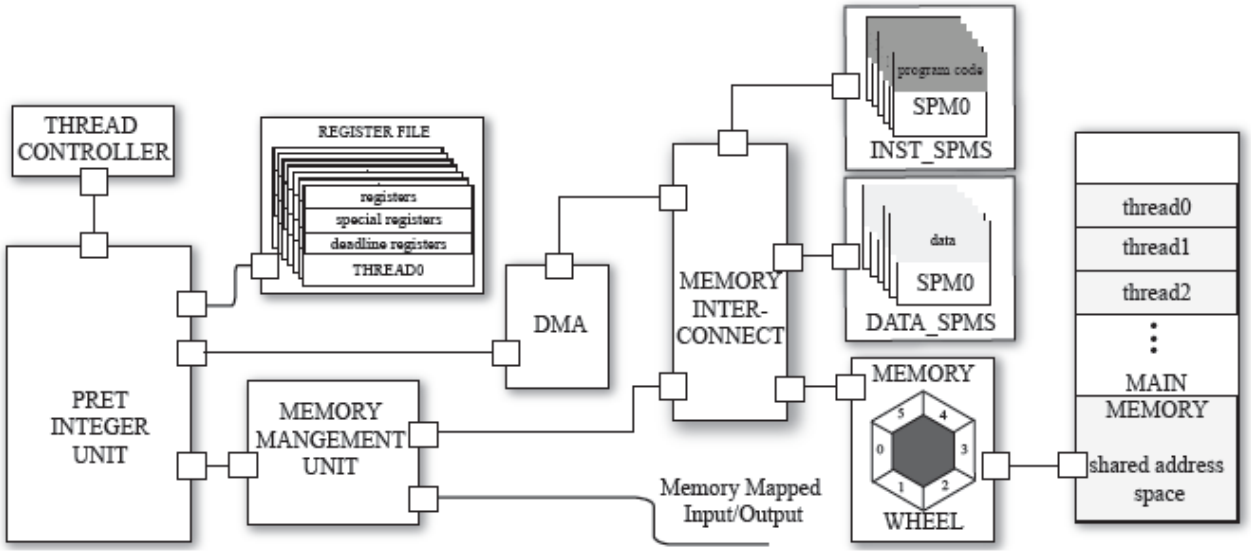


Figure 1: Block Diagram of PRET Architecture

Characteristics:

- software-managed scratchpad memories – no caches!
- thread-interleaved pipelines with no bypassing – predictable timing of instruction execution
- explicit timing control at the ISA level - deadline instruction
- time-triggered communication with global time synchronization
- high-level languages with explicit timing

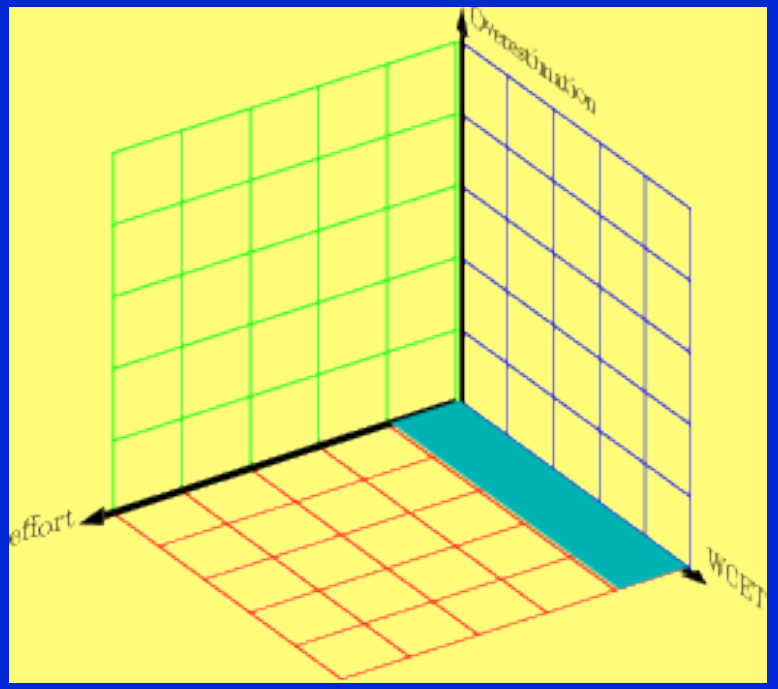
Unclear:

- memory management
- which performance loss

Character of PRET

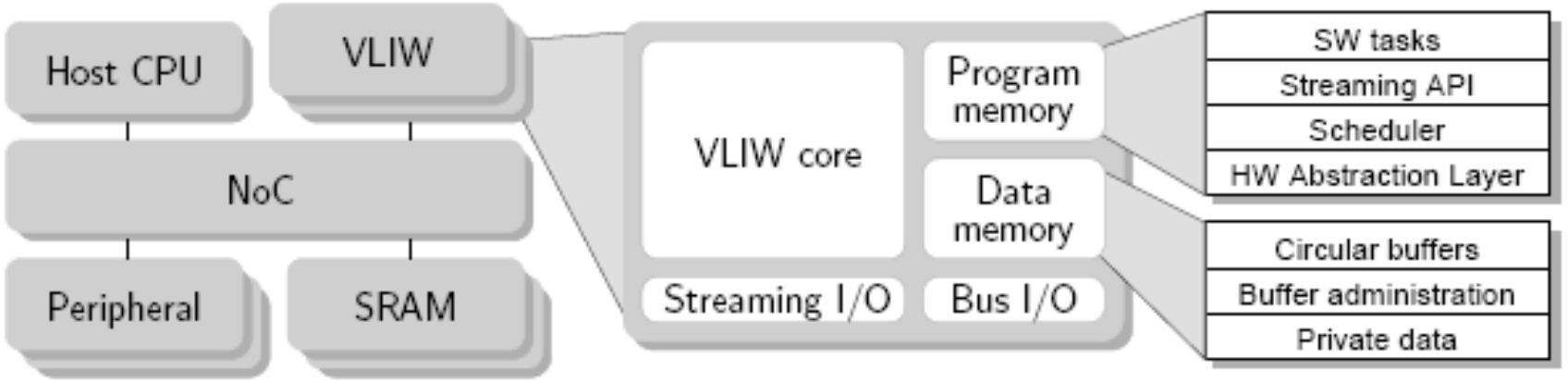
PRET will have

- overestimation 0 – due to predictable/repeatable timing
- small analysis effort – due to local determinism
- (I guess) bad worst-case performance



CoMPSoC (NXP)

- templates for predictable multi-processor-on-chip architectures



The PROMPT Principle: Architecture Follows Application

Starting with a **generic multi-node architecture**, the PROMPT architecture,

- parametric in the ISAs, the hierarchy of "nodes", the memory hierarchies, the interconnect, etc.
- nodes may be
  - atomic processing units with their private resources or
  - if performance requires with shared resources,
- nodes on each hierarchy level should be predictable
- we start with **predictable cores**, i.e., fully compositional architectures

The PROMPT Design Process

The generic PROMPT architecture is instantiated for a given set of applications with their resource requirements

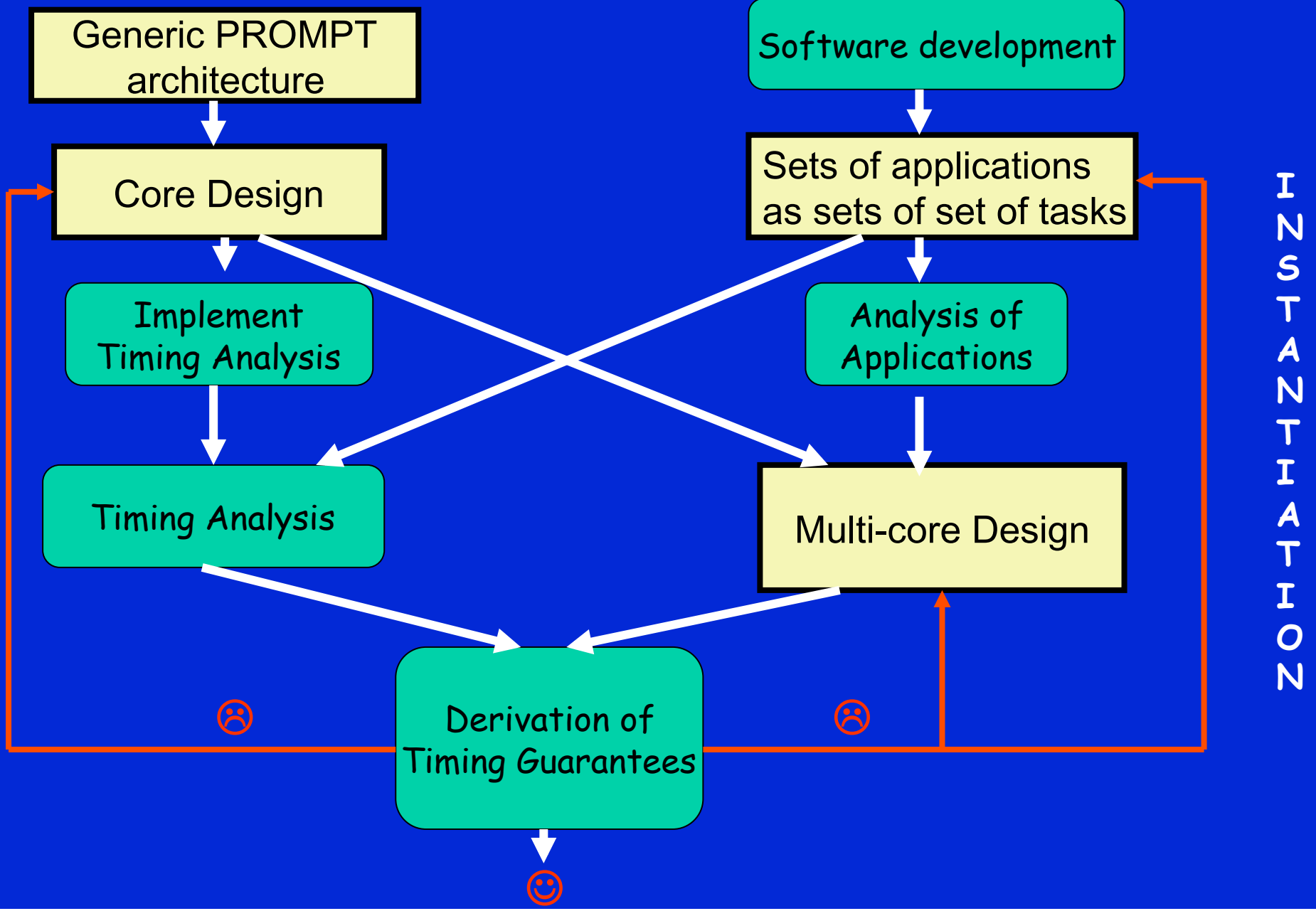
The design process works in multiple phases

1. hierarchical privatization
2. sharing of lonely resources
3. controlled socialization

Principles for the PROMPT Architecture and Design Process

- No shared resources where not needed for performance,
- Harmonious integration of applications: not introducing interferences on shared resources not existing in the applications.

The PROMPT System Design Process



Steps of the Design Process

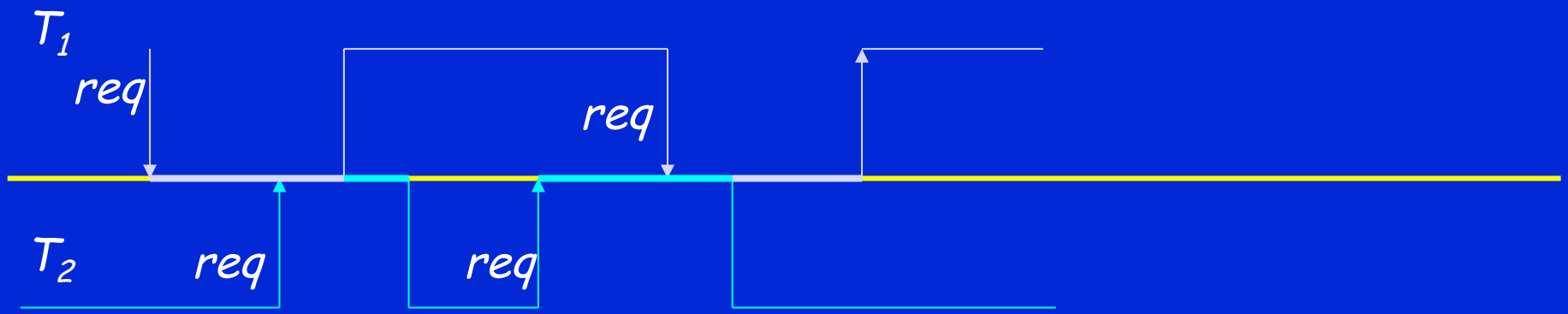
1. Hierarchical privatization
   - decomposition of the set of applications according to the sharing relation on the global state
   - allocation of private resources for non-shared code and state
   - allocation of the shared global state to non-cached memory, e.g. scratchpad,
   - sound (and precise) determination of delays for accesses to the shared global state
2. Sharing of lonely resources – seldom accessed resources, e.g. I/O devices
3. Controlled socialization
   - introduction of sharing to reduce costs
   - controlling loss of predictability

Sharing of Lonely Resources

- Costly lonely resources will be shared.
- Accesses rate is low compared to CPU and memory bandwidth.
- The access delay contributes little to the overall execution time because accesses happen infrequently.

Dealing with Shared Resources

Shared resources may introduce cyclic dependences between threads/tasks:



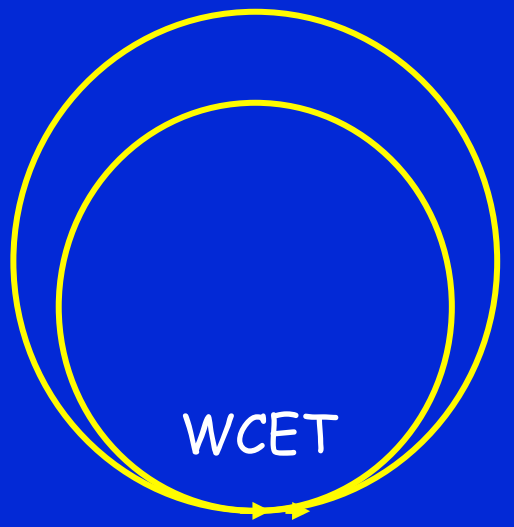
How to deal with the cycle?

- analyze it and determine a TDMA slot assignment,
- abstract the resource consumption of the threads to bound functions and determine bounds on the delays,
- cut it by an arbitration protocol with guaranteed delay bounds

TDMA Protocol

Determine a TDMA access protocol, cf. J. Rosen et al. 2007

1. Nested fixed point iterations:
   - inner loop: WCET analysis, assuming access times,
   - outer loop: determining access times increasing WCET bounds
2. Derivation of a slot assignment for the TDMA protocol



Promising because of the reading/writing bursts at the begin and end of tasks.

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:
  - semi-automatic derivation of abstract processor models
  - extension to multicore platforms
- Precision greatly depends on predictability properties of the system
  - notion needs further clarification, criteria to be used in design

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

Some Relevant Publications from my Group

- *C. Ferdinand et al.: Cache Behavior Prediction by Abstract Interpretation. Science of Computer Programming 35(2): 163-189 (1999)*
- *C. Ferdinand et al.: Reliable and Precise WCET Determination of a Real-Life Processor*, EMSOFT 2001
- *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,* IEEE TCAD, July 2009
- *R. Wilhelm et al.: Designing Predictable Multicore Architectures for Avionics and Automotive Systems, RePP Workshop, Grenoble, Oct. 2009*

Some other Publications dealing with Predictability

- R. Pellizzoni, M. Caccamo: *Toward the Predictable Integration of Real-Time COTS Based Systems*. RTSS 2007: 73-82
- J. Rosen, A. Andrei, P. Eles, and Z. Peng: *Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip*, RTSS 2007
- B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards and E. A. Lee: *Predictable Programming on a Precision Timed Architecture*, CASES 2008
- M. Schoeberl, *A Java processor architecture for embedded real-time systems*, Journal of Systems Architecture, 54/1--2:265--286, 2008
- M. Paolieri et al.: Hardware Support for WCET Analysis of Hard Real-Time Multicore Systems, ISCA 2009
- A. Hansson et al.: CompSoC: A Template for Composable and Predictable Multi-Processor System on Chips, ACM Trans. Des. Autom. Electr. Systems, 2009