Timing Analysis of Real-Time Systems
- multi cores, multi problems -

Reinhard Wilhelm
Universität des Saarlandes
Industrial Needs

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

Sideairbag in car,
Reaction in 4 mSec

Wing vibration of airplane,
sensing every 5 mSec

crankshaft-synchronous tasks
have very tight deadlines, ~45uS
Embedded Systems Go Multicore

• Offer good performance/energy ratio
• Support trends such as
  - Integrated Modular Avionics (IMA) and
  - Automotive Open System Architecture (AUTOSAR)

**Problem:**

• How to derive performance guarantees for embedded systems implemented on multicore architectures?
Working in a Repair Shop

• All the problems treated in this talk result from trends in embedded-systems design, i.e., using desktop COTS processors for safety- and time-critical embedded systems.

Going multicore strengthens this trend
Observations

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 for accesses to memory!
• pipeline modeling is the most complex task in the construction of a timing analyzer!
• adding more cores speeds up waiting!
Structure of the Talk

The problem: determining bounds on execution times

• The single-core setting
  - the problem
  - our solution

• The multi-core setting
  - problems
  - approaches
  - usage scenarios
  - the PROMPT ideas
Deriving Run-Time Guarantees for Hard Real-Time Systems

The simplest problem statement:

Given

1. a terminating software to produce a reaction,
2. a (single-core) hardware platform, on which to execute the software,
3. a required reaction time.

Derive: a guarantee for timeliness (for non-interrupted execution)
Schedulability Analysis

- A set of tasks: $T_1, T_i, T_n$
- Timing Analysis
  - Period, deadline, etc.
  - WCETs

- Schedulability Analysis
  - No or yes + schedule
The Mapping Problem

set of tasks

\( T_1 \)

\( T_i \)

\( T_n \)

period, deadline, comm. behav., redund. req. etc.

Timing Analysis

\( WCETs \)

Mapping + Schedulability Analysis

\( T_n \) multi-core platform

\( T_1, T_i \)
Timing Analysis

• Sounds methods 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.

1. I will show how this huge state space originates.
2. How and how far we can cope with this huge state space.

Decidability is not the problem! - It’s Complexity!
Timing Analysis – the Historic Search Space

• All control-flow paths (through the binary executable) - depending on the possible inputs.

• Feasible as search for a longest path if
  - Iteration and recursion are bounded,
  - Execution time of instructions are (positive) constants.

• Elegant method:
  Timing Schemata (Shaw’89, Puschner/Koza’89) – inductive calculation of upper bounds.

\[
ub \ (if \ b \ then \ S1 \ else \ S2) := ub \ (b) + max \ (ub \ (S1), \ ub \ (S2))
\]
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

\[ x = a + b; \]

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

- **Execution time** of an instruction is a function of the execution **state** → timing schemata no more applicable.
- Execution state results from the execution history.

**state**
- **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

Execution states for paths reaching this program point:

- Instruction in I-cache
  - 1
- Instruction not in I-cache
  - bus occupied: ≥ 40
  - bus not occupied

mul rD, rA, rB

Small operands: 1

Large operands: 4

Bus occupied: ≥ 40

Bus not occupied
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, and vice versa. Ex.: a cache miss preventing a branch misprediction
- Caused by the interference between processor components: Ex.: cache hit/miss influences branch prediction; branch prediction causes prefetching; prefetching pollutes the I-cache.

**Timing anomalies preclude many simple solutions!**
State Space Explosion in Timing Analysis

- Constant execution times
- State-dependent execution times
- Out-of-order execution
- Preemptive scheduling

Multi-core with shared resources: interweavings of several threads
Superscalar processors: interleavings of all schedules
Caches, pipelines, speculation, combined cache and pipeline analysis

Timing schemata

- Static analysis

Years + methods

- ~1995
- ~2000
- 2010+

~1995
~2000
2010+
**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, the A350, and the M400 have been certified using aiT; aiT is the only validated tool for these applications.
Tremendous Progress during the 10 years from 1998 to 2008

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

Abstract Interpretations
- Derives invariants about architectural execution states, computes bounds on execution times of basic blocks
- Determines loop bounds
- Determines enclosing intervals for the values in registers and local variables

Abstract Interpretation

Integer Linear Programming
- Determines a worst-case path and an upper bound
- Determines infeasible paths
- Determines a worst-case path and an upper bound

Combined cache and pipeline analysis

Binary Executable

CFG Reconstruction

Control-flow Graph

Value Analysis

Loop Bound Analysis

Annotated CFG

Basic Block Timing Info

Global Bound Analysis

Legend:
- Data
- Phase
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.

*Murphy’s invariant*

Fastest | Variance of execution times | Slowest
Architectural Complexity implies Analysis Complexity

Every hardware component whose state has an influence on the timing behavior

- must be conservatively modeled,
- contributes to the size of the search space, most of the time exponentially in some architectural parameters

- 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
Recipes for Success

- **Abstraction**: identify abstract domains that
  - have compact representations,
  - are precise, and
  - efficient
Recipes for Success II

• **Decomposition**: separate different aspects of the semantics and use precomputation

```
mul rD, rA, rB
```

Diagram:
- **Pipeline**
  - What it computes
  - How long it takes

```
Value Analysis
Pipeline Analysis
```
Abstraction and Decomposition

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

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

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

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

This has worked for caches and cache-like devices.

Find abstractions $C_{11}^#$ and $C_{12}^#$ factor out $C_{11}^#$ and transform rest into $2^{C_{12}^# \times C_2 \times \cdots \times C_k}$

This has worked for the arithmetic of the pipeline.

program $\rightarrow$ $C_{11}^#$ $\rightarrow$ program with annotations $\rightarrow$ $2^{C_{12}^# \times \cdots \times C_k}$

value analysis $\rightarrow$ microarchitectural analysis
Cyclic Dependences

• may require costly fixed-point computation
• we will encounter cycles that preclude effective solutions
Asynchronous Events

• Our timing analysis is based on the analysis of events triggered by the program under analysis and bound to \textit{(synchronous)} transitions in the HW,

• \textit{asynchronous} events like DRAM refresh, DMA need different (complex) argumentation,

• \textit{asynchronously} working \textit{periphery} leads to case splitting.

• \textit{Going} multi-core introduces many more \textit{asynchronous} events
  - events produced by other threads,
  - events triggered by cache-coherence protocols
Multi-Core Platforms

Interferences on shared resources reduce **performance** and **performance predictability**:

- concurrently executed tasks/threads share hardware resources.
- **Problem: Inter-task interference:**
  - **bandwidth interference:** conflicting requests for a resource with exclusive access, e.g. buses
  - **state interference:** one task changes the state of a shared resource with impact on the performance of another task, e.g. cache, banked memory
Impact on Performance

- Certainty project: Slow down when synchronously executing the same programs, approx. 1.6
- Nowotsch/Paulitsch’12: Slow down of 18 when concurrently accessing DDR memory.
- cf. Survey in the proceedings
Measurement-Based Assurance?
Resource-Stressing Benchmarks

- **Goal:**
  Estimate the slow-down factor caused by the interference on a shared resource
  - identify the worst impact of the interference on a shared resource
  - Run the $n$ resource-stressing benchmarks, each on one core and the application on one core

Result could be used as a **safety margin** in a measurement-based or a static approach to timing analysis
Some Experiments

• Radojkovic et al. 2012:
  - design resource-stressing benchmarks,
  - let them run concurrently with an application,
  - determine the (assumed) maximal slow-down.

• Nowotsch/Paulitsch 2012:
  - similar approach,
  - determine the influence of concurrent accesses on network and memory,
  - determine the overhead for memory coherence.
Experimental Results

Radojkovic et al.’12:
- almost no slow down when sim. running several applications,
- noticeable slow down when sim. running an application with resource-stressing benchmarks, up to 1.2.
- high slow down when sim. running several resource-stressing benchmarks, approx. 14 for L2-cache-stress. and memory-bandwidth benchmark.

Certainty project:
Slow down when synchronously executing the same programs, approx. 1.6

Nowotsch/Paulitsch’12:
Slow down of 18 when concurrently accessing DDR memory.
Methodological Challenges

- How to design a benchmark that puts the maximal stress on a shared resource?
- Is there a systematic way of deriving such benchmarks?
- Can one prove the maximality of the stress?
- How to combine stress for several shared resources? Consider resource inter-dependence!
How Bad Can Things Get for a Particular (Set of) Task(s)?

• Real-life tasks are not designed to put maximal stress on shared resources!
• How to determine the impact of concurrent execution of applications?
• Resource-stressing benchmarks only depend on the platform, ignore the application
  There could exist a worse companion for an application

Conclusion:
• High over-estimation
• No guarantee – independent of the application

Recommendation to the EASA by Thales:
Use measurement-based method combined with a safety margin both unsafe!
Approaches to Performance Prediction

- **Performance isolation**
  - **Pros:**
    - provides timing composability
    - allows (more or less) use of single-core analyses
  - **Cons:**
    - (may be) wasteful on resources

- **Analysis of mutual performance impacts of co-running tasks**
  - **Problem:**
    - complexity
    - needs abstraction
Performance Isolation through Partitioning

- Realized in Integrated Modular Avionics (IMA), ARINC 653:
  - temporal partitioning for bandwidth resources and CPUs:
    static periodic schedule of applications
  - spatial partitioning for memory
  - temporal and spatial partitioning for shared resources, e.g. caches

- Cons:
  - high effort for flushing shared resources to achieve timing composability
  - doubtful soundness; hard to flush the whole HW state
Interference on Bandwidth Resources

Arbitration:

- resolves the competition for shared resource,
- may extend execution time,
- **state-independent protocols**, e.g. TDMA:
  - simple, but high overestimation due to the bad worst case
  - static analysis may be used to determine good slot assignments, cf. work by P. Eles et al.
  - however, has to solve complex cyclic dependence
- **state-dependent protocols**, e.g. FCFS, round robin: allow for/require analysis to more precisely bound delays

Need to consider the precision of the upper bound on the access delay and the access frequency together
Estimating the Interference Costs

The Murphy approach: worst-case delays → Hopeless overestimation

Determining a slow-down factor by measurement (Resource-stressing benchmarks) → Overly pessimistic + unsound

Static analysis of set of co-running tasks → Complexity!

Need good abstractions for the resource behaviour of co-running tasks
Complexity Bites More Heavily

• Consider 2 threads with $n$ and $m$ accesses to 
  something shared
• $(n + m)! / (n! + m!)$ possible interleavings
• Verification of concurrent threads:
  - accesses are to global variables
  - reduction methods, e.g. partial order reduction, reduce
    the number of interleavings to consider

• Timing analysis:
  - accesses to shared bus, shared memory, shared caches
  - reduction methods do not exist so far!
An First Naïve Approach

• Threads with domains of (complex) states $C_1, C_2, \ldots, C_k$

• Combined domain $C_1 \times C_2 \times \ldots \times C_k$

• Use the powerset domain $2^{C_1 \times C_2 \times \ldots \times C_k}$ to analyze the combined behavior of all co-running threads

• Problem: Complexity, i.e., \#interleavings to consider
Following the Recipe for Success

• Threads with domains of (complex) states $C_1, C_2, \ldots, C_k$

• Analyze $C_1^# \times \ldots \times 2^{C_i} \times \ldots \times C_k^#$

  i.e., the behavior of one task in the context of abstractions of the co-running tasks

• Open question: what is the right level of abstraction?
One Example Abstraction

• One such abstraction is the following task model:
  • Task = a sequence of superblocks,
  • each superblock is characterized by
    - an upper bound on the amount of execution time, exec_i
      - ignoring bus access time
    - an upper bound on the number of resource accesses, r_i

\[
\begin{array}{c|c|c|c|}
\text{exec}_1 &= 40, \quad r_1 &= 21 \\
\text{exec}_1 &= 68, \quad r_1 &= 12 \\
\text{exec}_1 &= 43, \quad r_1 &= 33 \\
\end{array}
\]

Quite attractive and realistic: Read-Execute-Write (REW) task model:
\[
\text{exec}_1 = \text{exec}_3 = 0, \quad r_2 = 0
\]
A 2-Phase Approach

1. determine bound on the execution time of each task in isolation and a characterization of its resource-access behaviour
2. use this characterization to bound the impact of interference on the execution time

The sum of the two bounds yields an estimate of the task's worst-case execution time.

See work by L. Thiele et al.

Assumes timing-compositional architecture, no timing anomalies, domino effects
Interference on Shared Caches

Resource accesses may be

- **explicit**, i.e., occur in the program,
  - easy subjects for static analysis
- **implicit**, i.e., caused by HW/SW mechanisms, e.g., cache reloads, prefetching, cache-coherence protocols
  - hard to analyse because (shared-) cache behaviour of a task depends on that of all other sharing tasks

Badly analysable implicit resource accesses ruin the usability of the described task model
The Unpredictability of Shared Caches

• The state of the cache depends on the precise interleaving of accesses from the different cores.
• Their interleaving depends on the relative execution speeds of the applications running on these cores,
• the relative execution speeds depend on their cache performance,
• the cache performance depends on the cache state.

Conclusion: Either only private caches or partitioned shared caches
Cache-Aware Scheduling

Wang Yi et al. propose:

**Cache-aware non-preemptive fixed priority scheduling** associates with each task a required cache-partition size

Open problems:

- determine partitioning to optimize schedulability
- cache footprint of tasks may vary over time
- cache reloads use the memory bus -> interference on a bandwidth resource
Taking Constructive Influence  
- the PROMPT Approach -

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

- Reduced interference on shared resources $\Rightarrow$ fewer interleavings $\Rightarrow$
  - smaller analysis effort,
  - higher precision.
Traditional System Design Process

Design of execution platform

One application as a set of tasks

Software development

Timing Analysis

Schedulability Analysis

Yes

No
System Design Process with Integration of Applications

1. Design of distributed execution platform
2. Software development
3. Timing Analysis
4. Integration: Mapping and Schedulability

- Several applications as sets of sets of tasks

Decision:
- No
- Yes
Usage Scenarios I

- Architectures for safety- and time-critical avionics and automotive systems
- System characteristics:
  - combination of control loops and finite-state control
  - state transitions influence control parameters,
  - control loops trigger state transitions
Usage Scenarios II

- system characteristics (ctd.):
  - each control loop fully contained in one application
  - little shared code
  - global state partly shared between applications
  - 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
Usage Scenarios II

• Similar integration trends, IMA and AUTOSAR, integrating applications on powerful platforms instead of 1-application-per-platform/ECU

• More complex development process Mapping 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,
  - Primitive (and probably incomplete) solution to the problem of *resource interference*: attempt of full *task isolation* ⇒ loss in performance

• AUTOSAR
  - composability only achievable on predictable platforms
Dealing with Shared Resources

Alternatives:

• Avoiding them,
• Bounding their effects on timing variability
The PROMPT Principle: Architecture Follows Application

Starting with a generic multi-node 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
  - clusters with shared resources if performance requires,
• nodes on each hierarchy level should be predictable
• start with predictable cores, cf. Kalray MPPA

Approach similar to the NXP CompSoc architecture
The PROMPT Design Process

The generic PROMPT architecture is **instantiated** for a given set of applications with their **resource requirements and sharing properties**.

The **design process** works in phases:

1. hierarchical privatization
2. sharing of lonely resources
3. controlled socialization
Principles for the PROMPT Architecture and Design Process

- **Shared resources** only
  - where needed for performance,
  - for cost reasons

- **Harmonious integration of applications:**
  Mapping should not introduce sharing not existing in the applications.
The PROMPT System Design Process

Generic PROMPT architecture

Core Design

Implement Timing Analysis

Timing Analysis

Derivation of Timing Guarantees

Software development

Sets of applications as sets of set of tasks

Analysis of Applications

Multi-core Design
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 data
   - 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, if necessary
   - 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.
Concurrent execution of tasks on different cores:
• leads to access collisions and damages on the state of shared resources
• which cause a slow-down of the tasks, i.e. increases the execution times.

• Static timing analysis: The uncertainty introduced by abstraction increases the estimated worst-case bound.

• Arbitration protocols: reduce uncertainty, but introduce (bounded) delays.
Open Problems

• Right level of abstraction of co-running tasks
• Realization of the PROMPT design
• Cost of virtualization
  - dynamically managed virtual machines
  - statically compiled, e.g. AUTOSAR
Conclusion

- We have met many problems associated with the performance prediction for embedded systems running on multi-core platforms with shared resources.
- The developer is unhappy because there is no sound and precise way of deriving performance guarantees.
- The researcher is happy because he has something interesting to work on.
More Future Work

- Designing architectures reconciling (worst-case) **Predictability** with (average-case) **Performance**
  - modular analysis
- Support standardized software architectures, e.g. IMA, AUTOSAR - **compositionality** of the timing behavior
- TA for **concurrent threads** on multi-core platforms with **shared resources**
- Using the **compiler** to simplify timing analysis
  - code generator to guarantee **invariants**
- **Operating-mode** specific timing analysis
  - partition the set of control-flow graphs


A. Hansson, K. Goossens, M. Bekooij, J. Huisken: CoMPSoC: A template for composable and predictable multi-processor system on chips, TODAES 2009

Guan, N., Stigge, M., Yi, W., Yu, G.: Cache-aware scheduling and analysis for multicores, EMSOFT 2009
Some Relevant Publications from my Group

- C. Ferdinand et al.: Reliable and Precise WCET Determination of a Real-Life Processor, EMSOFT 2001
- St. Thesing: Modeling a System Controller for Timing Analysis, EMSOFT 2006
- D. Grund, J. Reineke: Precise and Efficient FIFO-Replacement Analysis Based on Static Phase Detection. ECRTS 2010
- Daniel Grund, Jan Reineke: Toward Precise PLRU Cache Analysis. WCET 2010
- S. Altmeier, C. Maiza, J. Reineke: Resilience analysis: tightening the CRPD bound for set-associative caches. LCTES 2010
Common Problems

• Question to answer by static analysis:
  When does competing task $T$ attempt to access shared resource $R$?
• The answers to this question for all tasks would allow to predict the protocol state.
• However, static analysis determines a (safe) over-approximation of the access times, i.e. intervals for the access times.
• Disjoint and singleton intervals represent definite knowledge,
• all others contain uncertainty.