Timing Analysis and Compilers

Sebastian Hack

Overview

1 Timing Analysis
- Introduction
- Challenges for Predictability
- Predictable Architectures

2 Multicores
- Composable SoCs
- . . . and Timing Analysis

3 PROMPT — A proposal for a more predictable multi-core platform

4 Compilers
- Compilers in the Multi-Core Age
- Correct Compilers

Introduction

SAARLAND UNIVERSITY
COMPUTER SCIENCE

- Safety-critical embedded systems should be formally verified
- "Testing can never prove the absence of a bug, only its presence" (Dijkstra)
- Verification of a property means formally proving the presence of that property
- Europe is leading in formal verification research and practice
- Formal verification tools have been successfully commercialized
- For example:
  Airbus successfully uses formal verification tools in the development process of avionics software products since 2001

Timing Analysis

SAARLAND UNIVERSITY
COMPUTER SCIENCE

- Verification of the temporal behavior of an application
- Why is this needed?
- Modern embedded systems execute many tasks on a single processor
- Some of these tasks has hard real-time constraints
- They have to be completed before a certain deadline
- The tasks are arranged in a schedule such that every task can meet his deadline
- To this end, we need to know the worst-case execution time (WCET) of every task

Key Messages of this Talk

- Timing analysis is crucial for safety-critical hard real-time systems
- Most modern processor architectures make timing analysis very hard
- Being hard for single cores, it is even worse for upcoming multi cores
- We urgently need more predictable architectures
- We need to make hardware manufacturers aware of timing predictability
- SoCs have to be composable to ensure predictability

Deriving Run-Time Guarantees for Hard Real-Time Systems

The Problem

Given:

- required reaction time
- a software to produce the reaction
- a hardware platform on which the software is executed

Goal:

- Derive a guarantee for timeliness

What does execution time depend on?

SAARLAND UNIVERSITY
COMPUTER SCIENCE

- The input . . . as usual
- The initial execution state
  - not important for the computed result (functional correctness)
  - but for timeliness
  - caused by caches, pipelines, speculation, etc.
  - hardware state is like an additional input
- Interferences from the environment
  - preemptive scheduling
  - interrupts

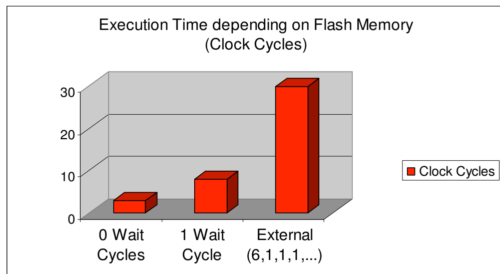
An Example

SAARLAND UNIVERSITY
COMPUTER SCIENCE

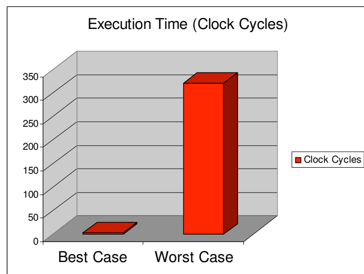
x = a + b;

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

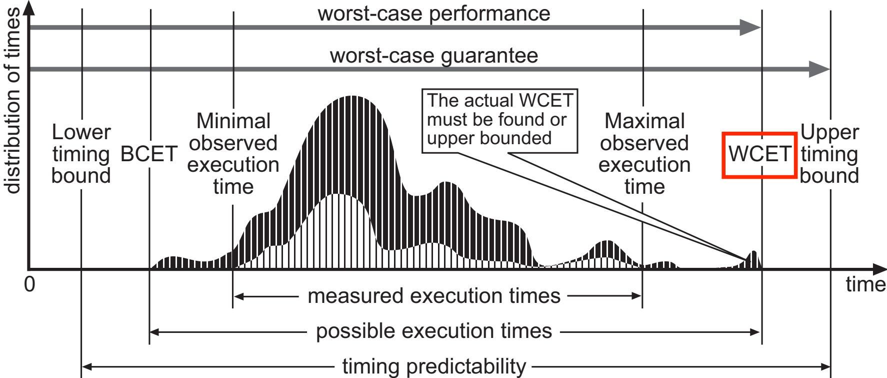
MPC 5xx



PPC 755



Notions in Timing Analysis



The measurement-based approach

SAARLAND UNIVERSITY
COMPUTER SCIENCE

- Methodology:
  - Measure execution times of each basic block
  - Try to use a comprehensive set of test input data
  - Try to combine basic-block measurements to a WCET for the procedure

The measurement-based approach

- Methodology:
  - Measure execution times of each basic block
  - Try to use a comprehensive set of test input data
  - Try to combine basic-block measurements to a WCET for the procedure
- Open questions:
  - How can you account for every possible hardware state?
  - How do you safely combine the basic-block times to a procedure WCET?

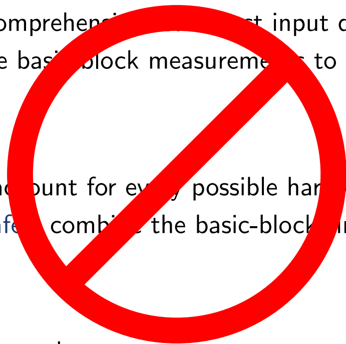
The measurement-based approach

- Methodology:
  - Measure execution times of each basic block
  - Try to use a comprehensive set of test input data
  - Try to combine basic-block measurements to a WCET for the procedure
- Open questions:
  - How can you account for every possible hardware state?
  - How do you safely combine the basic-block times to a procedure WCET?
- Conclusions:
  - Completely unsound
  - No safe timing guarantees can be derived

The measurement-based approach

SAARLAND UNIVERSITY
COMPUTER SCIENCE

- Methodology:
  - Measure execution times of each basic block
  - Try to use a comprehensive set of test input data
  - Try to combine basic block measurements to a WCET for the procedure
- Open questions:
  - How can you account for every possible hardware state?
  - How do you safely combine the basic-block times to a procedure WCET?
- Conclusions:
  - Completely unsound
  - No safe timing guarantees can be derived



The Static-Analysis Approach

- Methodology:
  - Have a conservative machine model in software
  - Use this and abstract interpretation to determine a WCET bound for each basic block
  - This WCET bound is provably larger or equal to the real WCET
  - Use integer linear programming to derive a WCET bound for the procedure

The Static-Analysis Approach

- Methodology:
  - Have a conservative machine model in software
  - Use this and abstract interpretation to determine a WCET bound for each basic block
  - This WCET bound is provably larger or equal to the real WCET
  - Use integer linear programming to derive a WCET bound for the procedure
- Comments:
  - The art is to derive tight bounds
  - Complex machines result in complex machine models

The Static-Analysis Approach

- Methodology:
  - Have a conservative machine model in software
  - Use this and abstract interpretation to determine a WCET bound for each basic block
  - This WCET bound is provably larger or equal to the real WCET
  - Use integer linear programming to derive a WCET bound for the procedure
- Comments:
  - The art is to derive tight bounds
  - Complex machines result in complex machine models
- Conclusions:
  - Provably sound, and precise WCET bounds
  - Proven also in practice: flies in A380

The Static-Analysis Approach

- Methodology:
  - Have a conservative machine model in software
  - Use this and abstract interpretation to determine a WCET bound for each basic block
  - This WCET bound is provably larger or equal to the real WCET
  - Use integer linear programming to derive a WCET bound for the procedure
- Comments:
  - The art is to derive tight bounds
  - Complex machines result in complex machine models
- Conclusions:
  - Provably sound, and precise WCET bounds
  - Proven also in practice: flies in A380

The Architectural Abstraction

Timing analyzer

Architectural abstractions

Value Analysis, Control-Flow Analysis, Loop-Bound Analysis

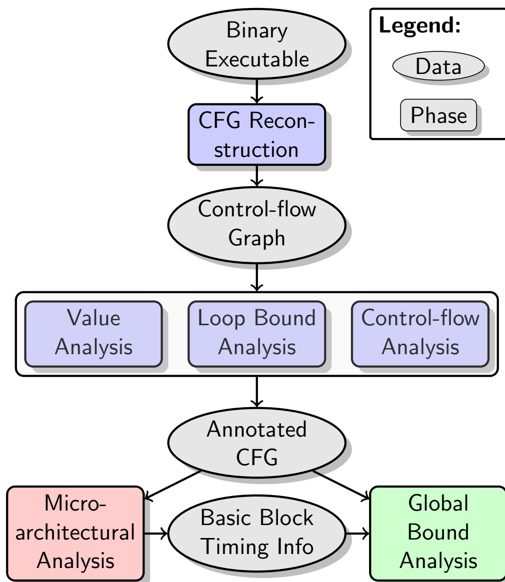
Cache Abstraction

Pipeline Abstraction

The Tool aiT

- Started as a research project at Saarland University
- Abstract-interpretation based WCET verification on machine code
- Commercialized by AbsInt startup
- aiT-verified software in production use

Tool Architecture: aiT



Timing Analysis

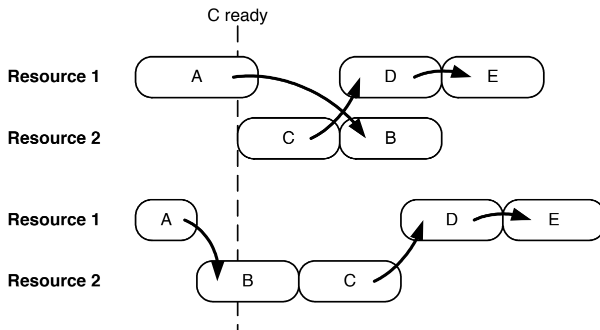
- Sound methods determine upper bounds for all execution times
- They have to explore a huge space of transition paths
  - All possible control-flow paths stemming from possible inputs
  - All paths through the architecture:
    Resulting from the initial execution states
    forced by timing anomalies
- Variability in timing often results from the interference on shared resources!
  - Memory, Caches, Pipelines, Buses, I/O Ports

Timing Anomalies

- Local worst case does not contribute to global worst case
- Consequence: We need to consider all paths through the hardware

Timing Anomalies

- Local worst case does not contribute to global worst case
- Consequence: We need to consider all paths through the hardware



Predictability on the Single-Core Level

Classification of pipelined architectures:

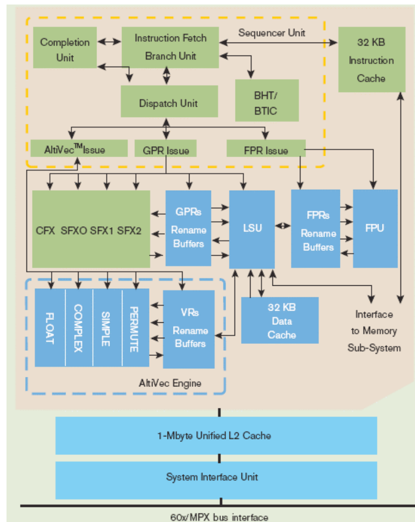
- Fully timing-compositional architectures:
  - no timing anomalies
  - analysis can safely follow local worst-case paths only
  - example: ARM7
- Compositional architectures with constant-bounded effects:
  - exhibit timing anomalies, but no domino effects
  - example: Infineon TriCore
- Non-compositional architectures
  - exhibit domino effects and timing anomalies
  - timing analysis always has to follow all paths
  - example: PowerPC 755

See Wilhelm et al. *Memory Hierarchies, Pipelines, and Buses for Future Architectures in Time-critical Embedded Systems*

Processor Features of the MPC 7448

... just to show how bad things are getting

- 32KB L1 data and instruction cache
- 1MB unified L2 cache with ECC
- Up to 12 instructions in instruction queue
- Up to 16 instructions "in flight"
- 7 pipeline stages
- 3 issue queues GPR, FPR, AltiVec
- 11 independent execution units



Challenges for Predictability

- Out-of-order execution
- Up to 3 levels of speculation due to unknown branch prediction
- Caches
  - Different pipeline paths for L1 cache hits/misses
  - PLRU replacement policy
- Arbitration between different functional units
  - Instructions have different exec times in different integer units
- Connection to the memory subsystem
  - Up to 8 parallel accesses on the MPX bus
- Several clock domains

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
- Depends on how we can abstract from the component
- Successful abstraction for caches [Reineke'08]:
  - For some replacement policies, good abstractions allowing precise analyses exist (LRU)
  - Some have abstractions, but rather imprecise analyses
- No efficient abstraction for pipelines

Complex Architectures in Practice

SAARLAND UNIVERSITY
COMPUTER SCIENCE

- In an experiment, engineers reduced the clock speed of the mentioned PowerPC
- But the WCET did not change!
- Why?

Complex Architectures in Practice

SAARLAND UNIVERSITY
COMPUTER SCIENCE

- In an experiment, engineers reduced the clock speed of the mentioned PowerPC
- But the WCET did not change!
- Why?

Several reasons:

- Their certified C compiler has optimizations disabled!
  - No register allocation, everyting goes to caches
- But, parts of the caches are disabled due to PLRU replacment policy
- L1 cache has no ECC!
- Because there is no ECC near the CPU, the engineers do not want to have data in the caches
- The code generated by the design tools is deterimental for I-cache performance
  - Many unrolled loops, code duplication, etc

Complex Architectures in Practice

Conclusions

- Applications are memory intensive
- Definitely no need for an out-of-order processor
- The processor basically waits for the memory
- Modern CPUs optimize for average-case performance
- Not for predictability of the worst case
- Features are not exploited and timing analysis needlessly complicated

But. . .

- Certification authorities surrender to the (wrong) architectural developments
- proving the correctness of modern high-performance processors used in safety-critical systems is infeasible
- correctness of compilers proved by practice
- current discussion:
  "liberalization" of requirements for proving timing correctness

But. . .

- Certification authorities surrender to the (wrong) architectural developments
- proving the correctness of modern high-performance processors used in safety-critical systems is infeasible
- correctness of compilers proved by practice
- current discussion:
  "liberalization" of requirements for proving timing correctness

We cannot weaken the verification requirements. Instead, we need:

- Simpler, more predictable CPUs
- Verified optimizing compilers [Leroy '06]
- Architecture-aware code generation techniques

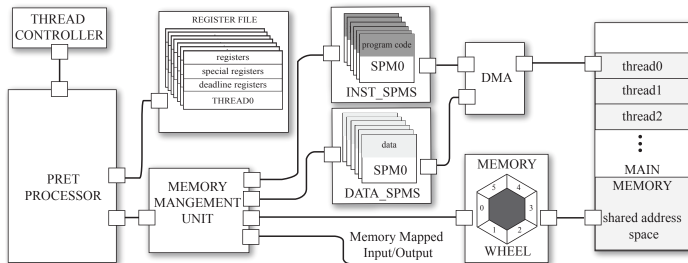
Proposals for more predictable Architectures

- Most proposals only concern some architectural feature, e.g.:
  - Making the pipeline more predictable [Sainrat,Rochange]
  - Freezing (parts of) the cache [Puaut]
  - Single-path paradigm [Puschner]
  - Deterministic bus protocols w/o consideration of application characteristics
- Most proposals entail a serious performance loss
- Overall designs: PRET architecture and the JOP Java processor

The PRET Architecture

SAARLAND UNIVERSITY
COMPUTER SCIENCE

Edwards/Lee et al.



- scratchpad memories instead of caches
- thread-interleaved pipelines with no bypasses
- explicit timing control at the ISA level by deadline instructions
- time-triggered communication with global time synchronization
- high-level languages with explicit timing

JOP Architecture

Schöberl

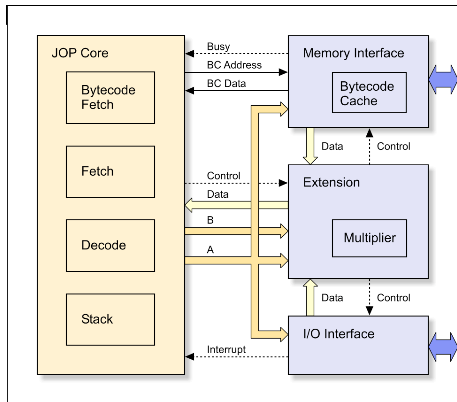
SAARLAND UNIVERSITY
COMPUTER SCIENCE

Instruction set:

- Stack-oriented
- Compact, constant length
- Single cycle

Caching:

- Full method cached
- Cache fill on call and return
- Relative addressing
- No fast tag memory



Timing-aware Hardware Design

- Hardware architects now understand that they need to consult compiler developers when they develop new architectures
- For embedded systems, they should also consult people in timing analysis
- Often, some hardware-component choice impedes timing analysis heavily
- We build abstract models for these components and can provide information about their predictability
- An example:
  - FIFO caches are much harder to analyze than LRU
  - PLRU is even worse

$\Longrightarrow$ For predictability, always use LRU

Overview

SAARLAND UNIVERSITY
COMPUTER SCIENCE

1 Timing Analysis
- Introduction
- Challenges for Predictability
- Predictable Architectures

2 Multicores
- Composable SoCs
- ... and Timing Analysis

3 PROMPT — A proposal for a more predictable multi-core platform

4 Compilers
- Compilers in the Multi-Core Age
- Correct Compilers

Multicores

- SoC grow in complexity:
  increasing number of applications integrated on a single chip
- Individual applications can have different real-time requirements
- Some may have hard, some soft real-time constraints
- To verify functional and temporal behavior of a single application, the architecture, the middleware and the mapping has to be modelled
- The system designer has to integrate all applications and verify the combined behavior
- Traditionally, verification cannot be done on isolated applications due to the interference on shared resources, e.g. interconnect and memory
- Two ways of coping with the complexity of system design: Abstraction and Partitioning

Abstraction and Partitioning

Abstraction:

- trades analysis feasibility for accuracy
- difficult to find a single abstraction for the whole systems
- difficult to find efficient abstractions that do not sacrifice to much accuracy

Partitioning:

- Split system into independent parts
- Each is simpler to understand than the whole
- also known as “divide and conquer”
- We aim for composability

Composability

SAARLAND UNIVERSITY
COMPUTER SCIENCE

Definition

A system is composable of the functional and temporal behavior of an application does not dependent on the presence or absence of other applications in the system

Composability

Definition

A system is composable of the functional and temporal behavior of an application does not dependent on the presence or absence of other applications in the system

- Eliminates interferences between applications
- Enables incremental design, integration, and verification
- Composability well known in the automotive and avionics industry
  Every application runs on one ECU $\implies$ no sharing
- How do we build SoCs that enable composability?

Composability

Definition

A system is composable of the functional and temporal behavior of an application does not dependent on the presence or absence of other applications in the system

- Eliminates interferences between applications
- Enables incremental design, integration, and verification
- Composability well known in the automotive and avionics industry
  Every application runs on one ECU $\Longrightarrow$ no sharing
- How do we build SoCs that enable composability?

Conclusion

We need architectural building blocks that enable composability and tools that integrate applications by managing the shared resources

IMA and AUTOSAR

- Automotive and avionics industry experience similar intergration trends: AUTOSAR and IMA
- Integration of many applications on a powerful platform instead of of one application per platform/ECU
- More complex development process:
  mapping problem: assign set of apps to nodes of the platform
- Expectations:
  - IMA: incremental qualification, i.e. modification of one application integrated with a set of other applications only requires re-certification of the modified component.
  - AUTOSAR: component-based design requiring composability: timing behavior of one task is independent from others

Considering Timing Analysis

- We have seen that timing analysis is demanding for modern single cores
- To derive safe WCET bounds we also need to model access to resources outside of the CPU: Buses for memory, I/O
- For a single core this is more or less tractable since there is only one program running
- For a concurrent system with multiple threads on multiple CPUs arbitrarily competing for shared resources, it becomes impossible
- We need composability for timing predictability

Overview

SAARLAND UNIVERSITY
COMPUTER SCIENCE

1 Timing Analysis
- Introduction
- Challenges for Predictability
- Predictable Architectures

2 Multicores
- Composable SoCs
- . . . and Timing Analysis

3 PROMPT — A proposal for a more predictable multi-core platform

4 Compilers
- Compilers in the Multi-Core Age
- Correct Compilers

Application Characteristics

- Combination of control loops and finite-state control
- Each control loop fully contained in one application
- little shared code
- global state partly shared between applications
- state transitions influence control parameters
- control loops trigger state transitions
- access to shared state only at beginning end end of task activations
- some applications require high performance but have no sharing with control applications

The PROMPT Principle

Architecture follows Application

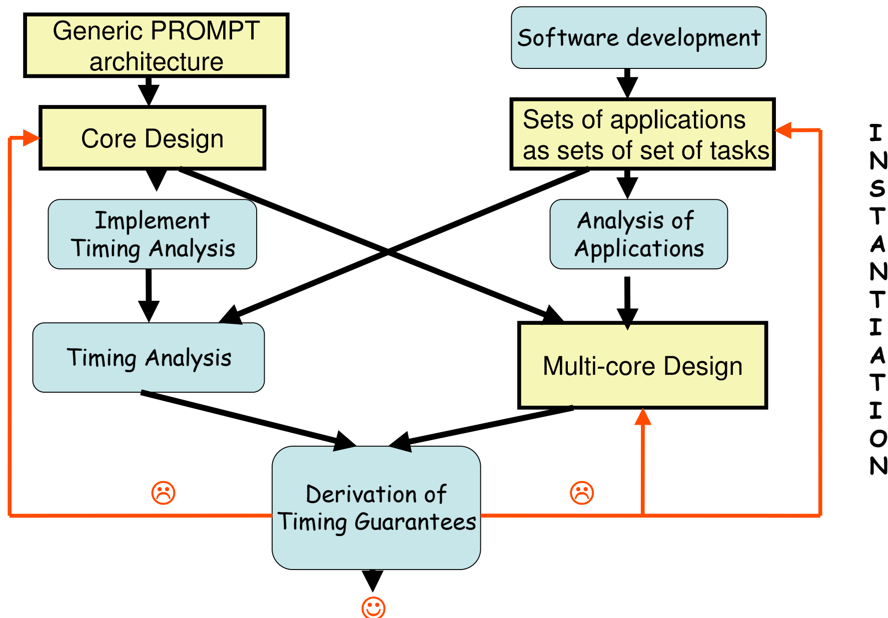
- Starting with a generic multi-node architecture the PROMPT architecture is parametric in
  - the ISAs
  - the hierarchy of nodes
  - the memory hierarchies
  - the interconnect
- Nodes may:
  - have completely private resources
  - shared resources if performance requires it
- Nodes on each hierarchy level should be predictable
- We start with predictable cores

The PROMPT Principle

cont'd

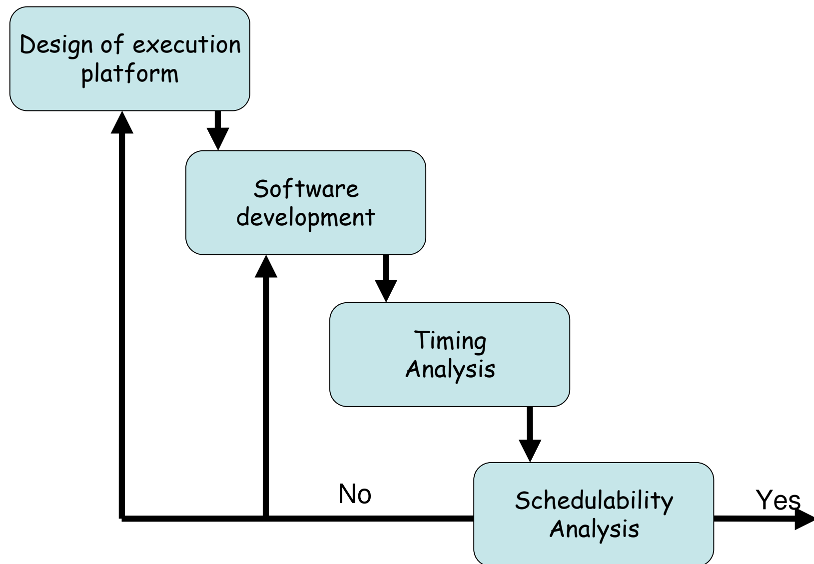
- No interference on shared resources where not needed for performance
- If needed, isolate interfering nodes in a new subnode from the rest
- Harmonious integration of applications without introducing interferences on shared resources not existing in the applications

The PROMPT System Design Process



The Traditional Design Process

SAARLAND UNIVERSITY
COMPUTER SCIENCE



Principles of the PROMPT Design Process

- 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
  - sound (and precise) determination of delays for accesses to the shared global state
- Controlled Socialization
  - introduction of sharing to reduce costs
  - controlling loss of predictability
- Sharing of lonely resources: rarely accessed resources, e.g. I/O devices
  - Costly lonely resources will be shared
  - Access rate is low compared to CPU and memory bandwidth
  - analyze the access behavior and determine a TDMA-like (deterministic) access protocol [Rosen et al. '07]

Key Messages of this Talk

- Timing analysis is crucial for safety-critical hard real-time systems
- Most modern processor architectures make timing analysis very hard
- Being hard for single cores, it is even worse for upcoming multi cores
- We urgently need more predictable architectures
- We need to make hardware manufacturers aware of timing predictability
- SoCs have to be composable to ensure predictability

Overview

1 Timing Analysis
- Introduction
- Challenges for Predictability
- Predictable Architectures

2 Multicores
- Composable SoCs
- ... and Timing Analysis

3 PROMPT — A proposal for a more predictable multi-core platform

4 Compilers
- Compilers in the Multi-Core Age
- Correct Compilers

What Compilers can do

- We are quite good at generating machine code from programs in low-level languages (like C) for single cores
- No wonder: we do research on that for over 50 years
- Compilers employ many machine-independent optimizations
- Basically to:
  - Remove redundant computations
  - Remove memory accesses
- Machine-code selection well understood for “standard architectures”
  $\Longrightarrow$ can be done systematically from descriptions
- Special-case “hacking” for more exotic architectures
- Still room for improvement on “exotic” DSPs, VLIWs

What Compilers do not so well

- Perform optimizations sensitive to the memory hierarchy
  - Very important
  - The multi-core problem is also a locality problem!
- Gain control over data layout
  - All optimizations are very code-centric
  - Layout of data is fixed and dictated by the programmer
    $\Longrightarrow$ bad for multi-cores
- Deep data-dependence analysis
  - needed for parallelization
  - Is intractable for languages like C
- Auto-tune themselves to new platforms (adaptivity)
- Produce verified code

Compiler Construction is hard

- Almost every important problem in compilers is at least NP-complete
- Especially memory hierarchy related optimizations
- Basically stems from discreteness:
  - Compiling a function into 4096 or 4097 bytes might make a huge difference!
  - However, 4095 or 4096 might make no difference
  - Due to fixed size of caches
- Compilers have to solve discrete optimization problems all the time
- Much in compilers is about developing efficient (sub-quadratic!) heuristics for very hard problems

Compilers in the multi-core age

- Most compiler optimizations focus on single-threaded programs
- Concurrency/Parallelization was largely thought of as a programmer's task
- With the advent of multi-cores this changed
- There was research on automatic parallelization that is now revived
- However, to find the parallelism, these techniques impose severe restriction on the programs
- Furthermore, the multi-core problem is also a locality/granularity problem:
- Many cores share resources, especially memory
- Working in parallel is only efficient if the data can be partitioned such that every processor is utilized

Compilers in the multi-core age

- The research of the last years suggest that all this is more a language problem
- Languages like C are too low-level to permit automatic parallelization/mapping to multi-cores
- They give the programmer too much freedom to access shared state
- Therefore, data dependences are hidden from the compiler
- Data structures and memory layout have to be mapped by the programmer
- They cannot be touched by the compiler anymore (the semantics of the language forbids it)

Compilers in the multi-core age
cont'd

- Parallelism has to come from more abstract programming paradigm
  - Stream programming
  - Data-flow programming
  - Functional programming
- Can we convince programmers to use such languages?
- How can we incorporate common programming techniques to such languages
- Can we depart from the human, stateful concept of algorithms?

Correct Compilers

- Often ignored problem: Can you trust your compiler?
- Is the generated machine code semantically equivalent to your source program?
- Some companies need to disable optimizations to get a compiler certified $\Longrightarrow$ That is not the way to go
- We need formally verified code generators and optimizers like [Leroy '06]
- Therefore, you need a formal semantics of the machine and the language
- Hence, it would be desirable to
  - Describe the semantics of the machine and generate a correct code-generator automatically
  - Correct by construction

Conclusions

- C is not the language of the multi-core age
- Compilers need to become more sensitive to the memory hierarchy
- We need to shape new languages that allow compilers to efficiently exploit multi-core systems
- We need correct, formally verified compilers to provide compiler optimizations to safety-critical systems