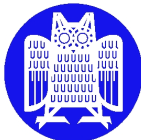
Static Timing Analysis for Hard Real-Time Systems

Sebastian Altmeyer

Saarland University

KPS 2009, Maria Taferl



Hard Real-Time System

- 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

Timing Analysis

Given:

1. required reaction time
2. task (binary)
3. a hardware platform, on which to execute the software

Derive: bound on the worst case execution time (WCET)

Requirements:

1. safe upper bound (no underestimation)
2. tight (close to real worst-case execution time)
3. tolerable analysis effort

What does Execution Time Depend on? (Task)

- the input,
- the initial execution state of the platform (caches, pipeline, branch target buffer, etc.), and
- interferences from the environment (preemptive scheduling, interrupts, shared caches).

Problem: exhaustive measurements not possible

What does Execution Time Depend on? (Instruction)

Modern processors increase performance by using: Caches, Pipelines, Branch Prediction, Speculation
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

Example

$$x = a + b$$

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

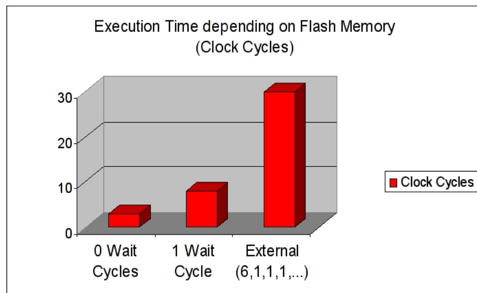


Figure: MPC 5xx

Example

$$x = a+b$$

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

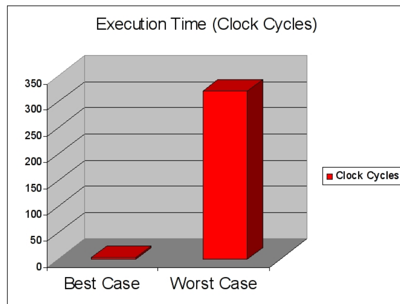


Figure: PPC 755

Execution Time is History-Sensitive

Contribution of the execution of an instruction to a programs 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 to Timing Analysis

Static Analysis of behavior of programs on the execution platform

- invariants about the set of execution states at all program points
- safety properties from these invariants: certain timing accidents never happen

Example:
At program point p, instruction fetch will never cause a cache miss.

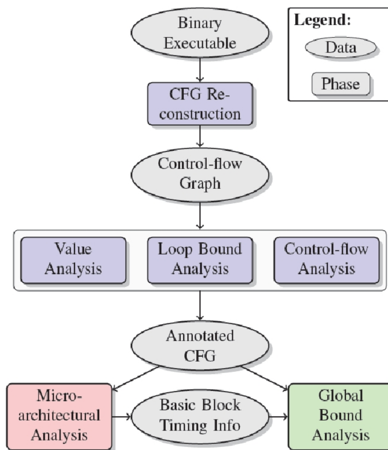
$\Rightarrow$

The more accidents excluded, the lower/tighter the upper bound.

Structure of the Timing Analysis

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

Structure of the Timing Analysis



Example: Cache Analysis

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

Cache Hit memory block a contained in the cache
data available in the next cycle

Cache Miss memory block a not contained in the cache
a transfered from main memory to cache
may replace other cached memory blocks (depending on replacement strategies: LRU, PLRU, FIFO, ...)

Example: Cache Analysis

How to statically precompute cache contents:

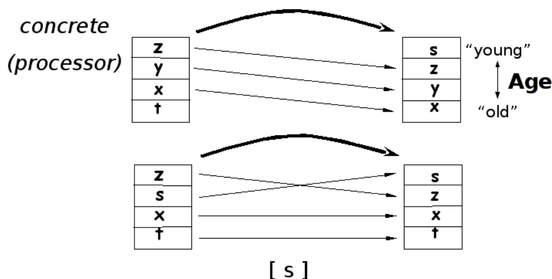
Must Analysis: For each program point, which blocks are definitely in the cache $\rightarrow$ predicts cache-hits.

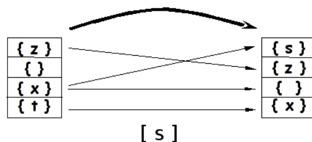
May Analysis: For each program point, which blocks may be in the cache. Complement says what is definitely not in the cache $\rightarrow$ predicts cache-misses.

Cache Analysis can not keep track of concrete cache state. Abstract cache semantics needed (set of memory blocks).

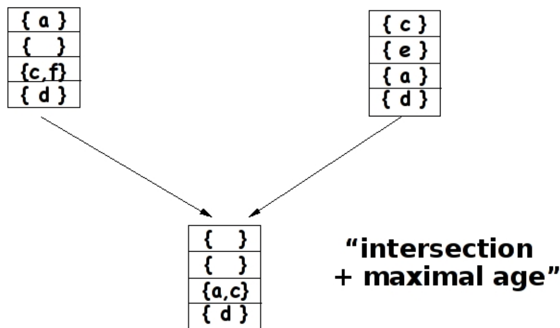
Example: Must-Cache Analysis - Transfer

Access to memory block s:





Example: Must-Cache Analysis - Join

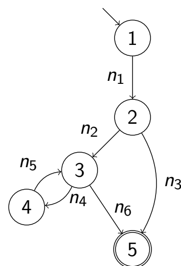


Pipeline Analysis

- 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
- However, model only contains components influencing the timing

Path Analysis - IPET

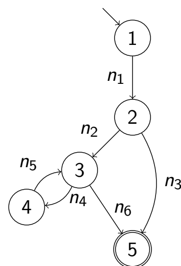
Searching for path with highest execution time by using
Implicit Path Enumeration Technique (IPET)



variables n_i denote how often edge i is traversed

Path Analysis - IPET

Searching for path with highest execution time by using Implicit Path Enumeration Technique (IPET)

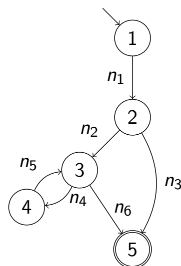


$n_1 = 1;$

first node is entered exactly once

Path Analysis - IPET

Searching for path with highest execution time by using Implicit Path Enumeration Technique (IPET)

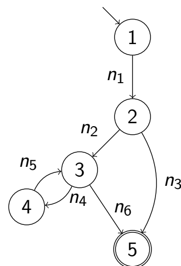


$$
\begin{aligned}
n_1 &= 1; \\
n_1 &= n_2 + n_3; \\
n_2 + n_5 &= n_4 + n_6; \\
n_4 &= n_5;
\end{aligned}
$$

sum of successors traversals equals sum of predecessor traversals

Path Analysis - IPET

Searching for path with highest execution time by using Implicit Path Enumeration Technique (IPET)

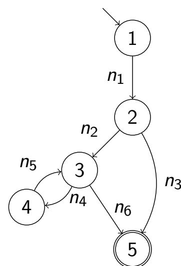


$$n_1 = 1;$$
$$n_1 = n_2 + n_3;$$
$$n_2 + n_5 = n_4 + n_6;$$
$$n_4 = n_5;$$
$$n_4 <= b_L n_2;$$

loop L is executed b_L times as often as it is entered
(b_L is the loop bound)

Path Analysis - IPET

Searching for path with highest execution time by using Implicit Path Enumeration Technique (IPET)

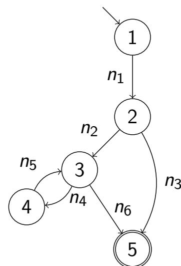


$$
\begin{aligned}
n_1 &= 1; \\
n_1 &= n_2 + n_3; \\
n_2 + n_5 &= n_4 + n_6; \\
n_4 &= n_5; \\
n_4 &<= b_L n_2; \\
n_3 + n_6 &= 1;
\end{aligned}
$$

last node is entered exactly once

Path Analysis - IPET

Searching for path with highest execution time by using Implicit Path Enumeration Technique (IPET)



$$
\begin{aligned}
n_1 &= 1; \\
n_1 &= n_2 + n_3; \\
n_2 + n_5 &= n_4 + n_6; \\
n_4 &= n_5; \\
n_4 &<= b_L n_2; \\
n_3 + n_6 &= 1;
\end{aligned}
$$

$$\max : \sum_i \left(\sum_{\forall j : n_j \text{ enters } B_i} c_i n_j \right)$$

objective function: maximize execution time by maximizing $c_i n_j$
(c_i = WCET of basic block n_J enters)

Conclusions

- Timing analysis possible, using abstract semantics of processor/task and ILP
- Tool available under http://www.absint.com/ (not free)
- successfully used in practise, for instance for Airbus A380

Ongoing work/Open problems

- Incorporation of preemption-caused costs
- Semi-automatic derivation of abstract processor models
- Timing analysis of heap-manipulating programs
- Timing analysis for multicores

Still some time left? Questions?

CAMA - Cache Aware Memory Allocation

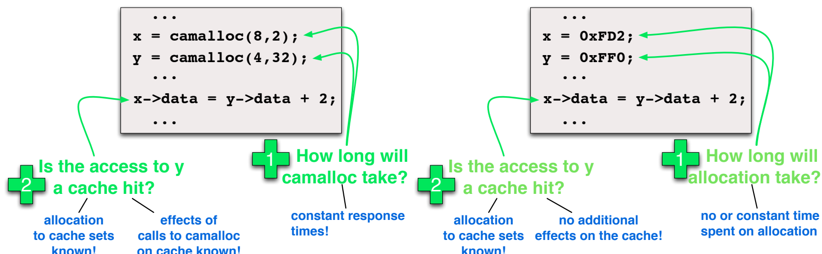
Current WCET analyses fail to give precise WCET bounds for programs that use dynamic memory allocation!

CAMA - Cache Aware Memory Allocation

We are investigating two approaches to enable precise WCET analyses on programs that dynamically allocate memory:

(1) Using a predictable memory allocator that allocates blocks mapped to a given cache set

(2) Automatically transform dynamic memory allocation into static allocation



http://rw4.cs.uni-sb.de/people/jherter

Determination of context-switch costs (CSC)

A memory block m at a program point P is useful cache block (UCB), if

a) it may be cached at P

b) it may be reused at program point Q reached from P without being evicted on this path

Data-flow analyses:

- a) Reaching memory block (forward)
- b) Live memory block (backward)

Schedulability analysis: WCET + CSC

- UCB analysis safely overapproximates context switch costs
- WCET analysis safely overapproximates execution time

$\Rightarrow$ *very pessimistic results if combined*

Some accesses are accounted for as a cache-miss by
WCET analysis **and** *UCB analysis*

Definitely Cached UCB (DC-UCB)

A memory block m at a program point P is useful cache block, if it

a) must be cached at P and on the path to its reuse

b) may be reused at program point Q reached from P

- UCB analysis possibly underapproximates context switch costs
- No cache-miss counted twice
- Overapprox. (WCET) subsumes underapprox. (UCB)

$\Rightarrow$ *tight and safe results if combined*

Sebastian Altmeyer, Claire Burguière

Motivation

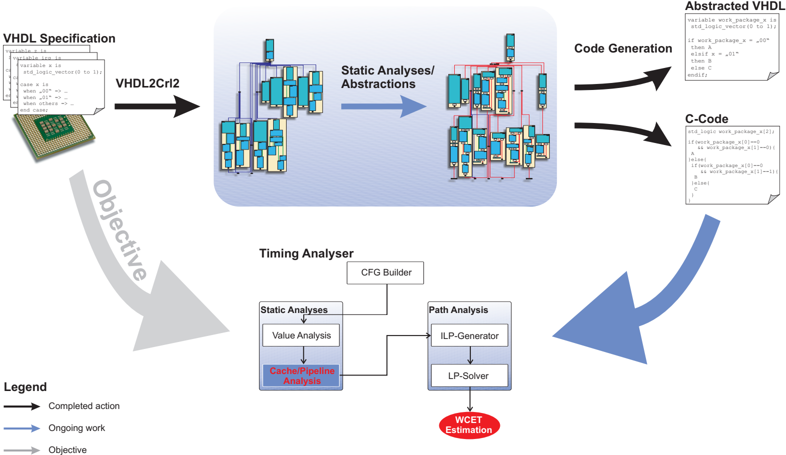
Problems

- [Availability/Accessibility of hardware specification]
- Processor specification too large to be used in aiT
- Specification needs to be abstracted

Idea

- Use of static methods to derive an abstracted model that is suitable for use in aiT

Overview of the derivation process



Markus Pister, Marc Schlickling, Mohamed Abdel Maksoud

Parametric Timing Analysis

- timing analysis essential for hard real-time systems
- many systems depend on input parameters
  (operating system schedulers, etc.)
- only two possible solutions:
  1. assume upper bounds on the unknown parameters
     $\Rightarrow$ highly overapproximated WCET
  2. restart the analysis for all parameter assignments
     $\Rightarrow$ very high analysis time
- parametric timing analysis delivers timing formula instead of a numeric value

Sebastian Altmeyer

Architecture

Predictability:

- Hard to quantify
- Predictable cores are a prerequisite for predictable multi-cores
- General problem: *Sharing* of ressources
  - Main memory, caches
  - Busses
  - I/O
  - Flash memory
- Sharing may be
  - fundamental (necessary access to application global variables)
  - incidental (processors happen to use the same bus for access to non-shared devices)

Predictability of Caches

Several new notions regarding cache replacement policies:

- *Predictability*:
  quantitative measure of how fast information about cache state can be gained
- *Competitiveness*:
  quantitative measure of how numbers of hits and misses of different policies relate
- *Sensitivity*:
  quantitative measure of how the number of hits and misses are influenced by intial cache state

Gives a sound and precise quantitative definition of predictability of caches

Predictable Architectures

Classification of architectures:

- *Timing compositional* (e. g., ARM7):
  No timing anomalies present,
  local worst-case behaviour safely approximates global worst-case behaviour
- *Compositional with bounded effects* (e. g., TriCore (probably)):
  No timing anomalies present,
  local worst-case behaviour safely approximates global worst-case behaviour up to a constant, additive factor
- *Non-compositional architectures* (e. g., PPC 755):
  Timing anomalies, domino effects,
  all global paths have to be considered

PROMPT

Minimise sharing in multi-processor architectures:

- Do not incidentally introduce sharing
- When introducing sharing, minimise its influence

PROMPT (Predictability Of Multi-Processor Timing)

- Start with a generic, parameterizable architecture with predictable (fully timing compositional) cores
- Instantiate architecture for given set of applications, based on their resource requirements