Timing Analysis
- timing guarantees for hard real-time systems-

Reinhard Wilhelm
Saarland University
Saarbrücken
Structure of the Lecture

1. Introduction
2. Static timing analysis
   - the problem
   - our approach
   - the success
   - tool architecture
3. Cache analysis
4. Pipeline analysis
5. Value analysis
6. Worst-case path determination
7. Conclusion
8. Further readings
Industrial Needs

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

Sideairbag in car,
Reaction in <10 mSec

Wing vibration of airplane,
sensing every 5 mSec

crankshaft-synchronous tasks
have very tight deadlines, ~45uS
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)**.
- Analogously, **Best-Case Execution Time (BCET)**.
Static Timing Analysis

Embedded controllers are expected to finish their tasks reliably within time bounds.

The problem:

Given

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

Derive: a **guarantee for timeliness**.
What does Execution Time Depend on?

- the **input** - this has always been so and will remain so,
- the **initial execution state of the platform** - this is (relatively) new,
- **interferences from the environment** - this depends on whether the system design admits it (preemptive scheduling, interrupts).

"external" interference as seen from analyzed task

Explosion of the space of inputs and initial states ⇒ no exhaustive approaches feasible.

Caused by caches, pipelines, speculation etc.
Modern Hardware Features

• Modern processors increase (average-case) performance by using: Caches, Pipelines, Branch Prediction, Speculation

• These features make bounds computation difficult: Execution times of instructions vary widely
  - **Best case** - *everything goes smoothly*: no cache miss, operands ready, needed resources free, branch correctly predicted
  - **Worst case** - *everything goes wrong*: all loads miss the cache, resources needed are occupied, operands are not ready
  - Span may be several hundred cycles
Access Times

The threat:
Over-estimation by a factor of 100 😞

\[ x = a + b; \]

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

Execution Time depending on Flash Memory
(Clock Cycles)

- MPC 5xx
- PPC 755

Execution Time (Clock Cycles)

- Best Case
- Worst Case

Clock Cycles

0 10 20 30

0 Wait Cycles 1 Wait Cycle External (6,1,1,1,...)
Notions in Timing Analysis

Hard or impossible to determine

Determine upper bounds instead
Timing Analysis and Timing Predictability

- **Timing Analysis** derives upper (and maybe lower) bounds
- **Timing Predictability** of a HW/SW system is the degree to which bounds can be determined
  - with acceptable precision,
  - with acceptable effort, and
  - with acceptable loss of (average-case) performance.
- The goal (of the Predator project) is to find a good point in this 3-dimensional space.
Timing Analysis
A success story for formal methods!
aiT WCET Analyzer

IST Project DAEDALUS final review report:
"The AbsInt tool is probably the best of its kind in the world and it is justified to consider this result as a breakthrough."

Several time-critical subsystems of the Airbus A380 have been certified using aiT; aiT is the only validated tool for these applications.
Tremendous Progress during the past 13 Years

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

Note: all analyzed programs are terminating, loop bounds need to be known ⇒ no decidability problem, but a complexity problem!
Our Approach

• **End-to-end measurement** is not possible because of the large state space.

• *We compute bounds* for the execution times of *instructions* and *basic blocks* and determine a longest path in the basic-block graph of the program.

• The *variability of execution times*
  - may cancel out in end-to-end measurements, but this is hard to quantify,
  - exists “in pure form” on the instruction level.
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
Execution Time is History-Sensitive

*Contribution* of the execution of an instruction to a program's execution time

- depends on the execution state, e.g. the time for a memory access depends on the cache state
- the execution state depends on the execution history
- needed: an invariant about the set of execution states produced by all executions reaching a program point.
- We use abstract interpretation to compute these invariants.
Deriving Run-Time Guarantees

- Our method and tool, aiT, 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.
Abstract Interpretation in Timing Analysis

• Abstract interpretation is always based on the semantics of the analyzed language.

• A **semantics** of a programming language that talks about **time** needs to incorporate the **execution platform**!

• **Static timing analysis is thus based on such a semantics.**
The Architectural Abstraction inside the Timing Analyzer

Architectural abstractions

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

Cache Abstraction

Pipeline Abstraction

abstractions of the processor’s arithmetic
Abstract Interpretation in Timing Analysis

Determines

• invariants about the values of variables (in registers, on the stack)
  - to compute loop bounds
  - to eliminate infeasible paths
  - to determine effective memory addresses
• invariants on architectural execution state
  - Cache contents ⇒ predict hits & misses
  - Pipeline states ⇒ predict or exclude pipeline stalls
Tool Architecture

Abstract Interpretations

Abstract Interpretation

Integer Linear Programming
Caches: Small & Fast Memory on Chip

- Bridge speed gap between CPU and RAM
- **Caches work well in the average case:**
  - Programs access data locally (many hits)
  - Programs reuse items (instructions, data)
  - Access patterns are distributed evenly across the cache
- **Cache performance has a strong influence on system performance!**
Caches: How they work

CPU: read/write at memory address $a$,
- sends a request for $a$ to bus

Cases:
- **Hit**: Block $m$ containing $a$ in the cache:
  - request served in the next cycle
- **Miss**: Block $m$ not in the cache:
  - $m$ is transferred from main memory to the cache,
  - $m$ may replace some block in the cache,
  - request for $a$ is served asap while transfer still continues
Replacement Strategies

• Several replacement strategies: LRU, PLRU, FIFO,... determine which line to replace when a memory block is to be loaded into a full cache (set)
LRU Strategy

- Each cache set has its own replacement logic => Cache sets are independent: Everything explained in terms of one set

- LRU-Replacement Strategy:
  - Replace the block that has been Least Recently Used
  - Modeled by Ages

- Example: 4-way set associative cache

<table>
<thead>
<tr>
<th>age</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access $m_4$ (miss)</td>
<td>$m_0$</td>
<td>$m_1$</td>
<td>$m_2$</td>
<td>$m_3$</td>
</tr>
<tr>
<td>Access $m_1$ (hit)</td>
<td>$m_4$</td>
<td>$m_0$</td>
<td>$m_1$</td>
<td>$m_2$</td>
</tr>
<tr>
<td>Access $m_5$ (miss)</td>
<td>$m_1$</td>
<td>$m_4$</td>
<td>$m_0$</td>
<td>$m_2$</td>
</tr>
<tr>
<td></td>
<td>$m_5$</td>
<td>$m_1$</td>
<td>$m_4$</td>
<td>$m_0$</td>
</tr>
</tbody>
</table>
Cache Analysis

How to statically precompute cache contents:

- **Must Analysis:**
  For each program point (and context), find out which blocks are in the cache $\rightarrow$ prediction of cache hits

- **May Analysis:**
  For each program point (and context), find out which blocks may be in the cache
  Complement says what is not in the cache $\rightarrow$ prediction of cache misses

- In the following, we consider must analysis until otherwise stated.
(Must) Cache Analysis

- Consider one instruction in the program.
- There may be many paths leading to this instruction.
- How can we compute whether $a$ will always be in cache independently of which path execution takes?

Question: Is the access to $a$ always a cache hit?
Determine Cache-Information (abstract must-cache states) at each Program Point

youngest age - 0
{x}
{a, b}

oldest age - 3

Interpretation of this cache information:
describes the set of all concrete cache states in which x, a, and b occur
• x with an age not older than 1
• a and b with an age not older than 2,

Cache information contains
1. only memory blocks guaranteed to be in cache.
2. they are associated with their maximal age.
Must-Cache-Information

Cache analysis determines safe information about Cache Hits. Each predicted Cache Hit reduces the upper bound by the cache-miss penalty.

Computed cache information

\[
\{x\} \\
\{a, b\}
\]

load a

Access to a is a cache hit; assume 1 cycle access time.
Cache Analysis - how does it work?

• How to compute for each program point an abstract cache state representing a set of memory blocks guaranteed to be in cache each time execution reaches this program point?

• Can we expect to compute the largest set?

• Trade-off between precision and efficiency – quite typical for abstract interpretation
(Must) Cache analysis of a memory access

After the access to a, a is the youngest memory block in cache, and we must assume that x has aged. What about b?
Combining Cache Information

• Consider two control-flow paths to a program point:
  - for one, prediction says, set of memory blocks $S_1$ in cache,
  - for the other, the set of memory blocks $S_2$.
  - Cache analysis should not predict more than $S_1 \cap S_2$ after the merge of paths.
  - the elements in the intersection should have their maximal age from $S_1$ and $S_2$.

• Suggests the following method: Compute cache information along all paths to a program point and calculate their intersection - but too many paths!

• More efficient method:
  - combine cache information on the way,
  - iterate until least fixpoint is reached.

• There is a risk of losing precision, not in case of distributive transfer functions.
What happens when control-paths merge?

We can guarantee this content on this path.

\{a\}  
\{\}   
\{c, f\} 
\{d\}

Which content can we guarantee on this path?

\{c\}  
\{e\}   
\{a\}   
\{d\}

We can guarantee this content on this path.

“Intersection + Maximal Age”

combine cache information at each control-flow merge point
Must-Cache and May-Cache-Information

- The presented cache analysis is a **Must Analysis**. It determines safe information about cache hits. Each predicted cache hit reduces the upper bound.
- We can also perform a **May Analysis**. It determines safe information about cache misses. Each predicted cache miss increases the lower bound.
(May) Cache analysis of a memory access

Why? After the access to `a` `a` is the youngest memory block in cache, and we must assume that `x, y` and `b` have aged.
Cache Analysis: Join (may)

Join (may)

```
\{ a \}
\{ \}
\{ c, f \}
\{ d \}
```

```
\{ c \}
\{ e \}
\{ a \}
\{ d \}
```

```
\{ a, c \}
\{ e \}
\{ f \}
\{ d \}
```

"union + minimal age"
# Result of the Cache Analyses

## Categorization of memory references

<table>
<thead>
<tr>
<th>Category</th>
<th>Abb.</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>always hit</td>
<td>ah</td>
<td>The memory reference will always result in a cache hit.</td>
</tr>
<tr>
<td>always miss</td>
<td>am</td>
<td>The memory reference will always result in a cache miss.</td>
</tr>
<tr>
<td>not classified</td>
<td>nc</td>
<td>The memory reference could not be classified as ah nor am.</td>
</tr>
</tbody>
</table>
Abstract Domain: Must Cache

Representing sets of concrete caches by their description

<table>
<thead>
<tr>
<th>Concrete caches</th>
<th>Abstract cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>{z, x}</td>
<td>{}</td>
</tr>
<tr>
<td>{s}</td>
<td>{s}</td>
</tr>
<tr>
<td>{z, t}</td>
<td>{}</td>
</tr>
<tr>
<td>{x}</td>
<td>{z, x}</td>
</tr>
<tr>
<td>{t}</td>
<td>{}</td>
</tr>
</tbody>
</table>
Abstract Domain: Must Cache

Sets of concrete caches described by an abstract cache

Concretization

\[ \{z, x\} \subseteq \{s\} \]

remaining line filled up with any other block

\[ \gamma \]

\[ \{\} \]
\[ \{\} \]
\[ \{z, x\} \]
\[ \{s\} \]

\(\alpha\) and \(\gamma\) form a Galois Connection

over-approximation!
Abstract Domain: May Cache

Abstractation

concrete caches

abstract cache

\[\{z, s, x\}\]
\[
\{t\}
\]
\[
\{\}\n\]
\[
\{a\}\n\]
Abstract Domain: May Cache

Concrete caches

\[\in \{z,s,x\}\]
\[\in \{z,s,x,t\}\]
\[\in \{z,s,x,t\}\]
\[\in \{z,s,x,t,a\}\]

Concretization

Abstract cache

\[\{z,s,x\}\]
\[\{t\}\]
\[\{\}\]
\[\{a\}\]

Abstract may-caches say what definitely is not in cache and what the minimal age of those is that may be in cache.
Galois connection - Relating Semantic Domains

- Lattices $C$, $A$
- two monotone functions $\circledast$ and $\circledcirc$
- Abstraction: $\circledast$: $C \rightarrow A$
- Concretization $\circledcirc$: $A \rightarrow C$
- $(\circledast, \circledcirc)$ is a Galois connection if and only if $\circledcirc \circledast w_C id_C$ and $\circledast \circledcirc v_A id_A$

Switching safely between concrete and abstract domains, possibly losing precision
Abstract Domain Must Cache

\[ z, x \in \{ s \mid z, x \in \{z, x\}, t \in \{s\} \} \]

safe, but may lose precision

concrete caches

abstract cache

remaining line filled up with any memory block

\[ \{s\} \{z\} \{z, x\} \]
Lessons Learned

• **Cache analysis**, an important ingredient of static timing analysis, provides for abstract domains,
  • which proved to be sufficiently precise,
  • have compact representation,
  • have efficient transfer functions,
  • which are quite natural.
An Alternative Abstract Cache Semantics: Power set domain of cache states

- Set A of elements - sets of concrete cache states
- Information order \( \subseteq \) - set inclusion
- Join operator \( \cup \) - set union
- Top element \( > \) - the set of all cache states
- Bottom element \( ? \) - the empty set of caches
Power set domain of cache states

- Potentially more precise
- Certainly not similarly efficient
- Sometimes, power-set domains are the only choice you have → pipeline analysis
Problem Solved?

• We have shown a solution for LRU caches.
• LRU-cache analysis works smoothly
  - Favorable „structure“ of domain
  - Essential information can be summarized compactly
• LRU is the best strategy under several aspects
  - performance, predictability, sensitivity
• ... and yet: LRU is not the only strategy
  - Pseudo-LRU (PowerPC 755 @ Airbus)
  - FIFO
  - worse under almost all aspects, but average-case performance!
Abstract Interpretation - the Ingredients

- Abstract domain - complete lattice \((A, v, t, u, >, ?)\)
- (monotone) abstract transfer functions for each statement/condition/instruction
- Information at program entry points
Contribution to WCET

while ... do [max $n$] ref to $s$

\[
\begin{align*}
\text{time} &= n \cdot t_{\text{miss}} \\
t_{\text{miss}} &= t_{\text{hit}} + (n - 1) \cdot t_{\text{miss}} \\
t_{\text{hit}} &= n \cdot (t_{\text{miss}} + t_{\text{hit}})
\end{align*}
\]

od

loop time
Contexts

Cache contents depends on the Context, i.e. calls and loops

First Iteration loads the cache =>
Intersection loses most of the information!

while cond do
  join (must)
Distinguish basic blocks by contexts

- Transform loops into tail recursive procedures
- Treat loops and procedures in the same way
- Use interprocedural analysis techniques, VIVU
  - virtual inlining of procedures
  - virtual unrolling of loops
- Distinguish as many contexts as useful
  - 1 unrolling for caches
  - 1 unrolling for branch prediction (pipeline)
Tool Architecture

Abstract Interpretations

Pipelines

Abstract Interpretation

Integer Linear Programming
Hardware Features: Pipelines

Ideal Case: 1 Instruction per Cycle
Pipelines

- Instruction execution is split into several stages
- Several instructions can be executed in parallel
- Some pipelines can begin more than one instruction per cycle: VLIW, Superscalar
- Some CPUs can execute instructions out-of-order
- Practical Problems: Hazards and cache misses
Pipeline Hazards:

- **Data Hazards**: Operands not yet available
  
  (Data Dependences)

- **Resource Hazards**: Consecutive instructions use same resource

- **Control Hazards**: Conditional branch

- **Instruction-Cache Hazards**: Instruction fetch causes cache miss
Static exclusion of hazards

Cache analysis: prediction of cache hits on instruction or operand fetch or store

lwz r4, 20(r1)  Hit

Dependence analysis: elimination of data hazards

add r4, r5, r6
lwz r7, 10(r1)
add r8, r4, r4  Operand ready

Resource reservation tables: elimination of resource hazards
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
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` \((b : \text{basic block}, s : \text{abstract pipeline state})\)

\(t : \text{trace}\)

interprets instruction stream of \(b\) (annotated with cache information) starting in state \(s\)

producing trace \(t\)

\(\text{length}(t)\) 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), hard to find one that
  • preserves information and
  • has a compact representation.
• So, collect sets of pipeline states.
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 = \(2^{\text{PS} \times \text{CS}}\)

\(\text{PS}\) = set of abstract pipeline states

\(\text{CS}\) = 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 Pipelines

• **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
Characteristics of Pipeline Analysis

• Abstract Domain of Pipeline Analysis
  - Power set domain
    • Elements: sets of states of a state machine
  - Join: set union

• Pipeline Analysis
  - Manipulate sets of states of a state machine
  - Store sets of states to detect fixpoint
  - Forward state traversal
  - Exhaustively explore non-deterministic choices
Abstract Pipeline Analysis vs Model Checking

• Pipeline Analysis is like state traversal in Model Checking
• Symbolic Representation: BDD
• Symbolic Pipeline Analysis: Topic of on-going dissertation
Nondeterminism

- In the reduced model, one state resulted in one new state after a one-cycle transition
- Now, one state can have several successor states
  - Transitions from set of states to set of states
Implementation

• Abstract model is implemented as a DFA
• Instructions are the nodes in the CFG
• Domain is powerset of set of abstract states
• Transfer functions at the edges in the CFG iterate cycle-wise updating each state in the current abstract value

\[
\max \{ \# \text{ iterations for all states} \} \text{ gives WCET}
\]

• From this, we can obtain WCET for basic blocks
Value Analysis

- **Motivation:**
  - Provide access information to data-cache/pipeline analysis
  - Detect infeasible paths
  - Derive loop bounds

- **Method:**
  - Calculate intervals at all program points, i.e. lower and upper bounds for the set of possible values occurring in the machine program (addresses, register contents, local and global variables) (Cousot/Cousot77)
Value Analysis II

D1: [-4,4], A[0x1000,0x1000]

- Intervals are computed along the CFG edges

- At joins, intervals are "unioned"

move.l #4,D0

D0[4,4], D1: [-4,4], A[0x1000,0x1000]

add.l D1,D0

D0[0,8], D1: [-4,4], A[0x1000,0x1000]

move.l (A0,D0),D1

Which address is accessed here?

move.l #4,D0

D1: [-4,4], A[0x1000,0x1000]

D0[4,4], D1: [-4,4], A[0x1000,0x1000]

add.l D1,D0

D0[0,8], D1: [-4,4], A[0x1000,0x1000]

move.l (A0,D0),D1

Which address is accessed here?
Interval Analysis in Timing Analysis

- Data-cache analysis needs effective addresses at analysis time to know where accesses go.
- Effective addresses are approximatively precomputed by an interval analysis for the values in registers, local variables
  - “Exact” intervals - singleton intervals,
  - “Good” intervals - addresses fit into less than 16 cache lines.
## Value Analysis (Airbus Benchmark)

<table>
<thead>
<tr>
<th>Task</th>
<th>Unreached</th>
<th>Exact</th>
<th>Good</th>
<th>Unknown</th>
<th>Time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8%</td>
<td>86%</td>
<td>4%</td>
<td>2%</td>
<td>47</td>
</tr>
<tr>
<td>2</td>
<td>8%</td>
<td>86%</td>
<td>4%</td>
<td>2%</td>
<td>17</td>
</tr>
<tr>
<td>3</td>
<td>7%</td>
<td>86%</td>
<td>4%</td>
<td>3%</td>
<td>22</td>
</tr>
<tr>
<td>4</td>
<td>13%</td>
<td>79%</td>
<td>5%</td>
<td>3%</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>6%</td>
<td>88%</td>
<td>4%</td>
<td>2%</td>
<td>36</td>
</tr>
<tr>
<td>6</td>
<td>9%</td>
<td>84%</td>
<td>5%</td>
<td>2%</td>
<td>16</td>
</tr>
<tr>
<td>7</td>
<td>9%</td>
<td>84%</td>
<td>5%</td>
<td>2%</td>
<td>26</td>
</tr>
<tr>
<td>8</td>
<td>10%</td>
<td>83%</td>
<td>4%</td>
<td>3%</td>
<td>14</td>
</tr>
<tr>
<td>9</td>
<td>6%</td>
<td>89%</td>
<td>3%</td>
<td>2%</td>
<td>34</td>
</tr>
<tr>
<td>10</td>
<td>10%</td>
<td>84%</td>
<td>4%</td>
<td>2%</td>
<td>17</td>
</tr>
<tr>
<td>11</td>
<td>7%</td>
<td>85%</td>
<td>5%</td>
<td>3%</td>
<td>22</td>
</tr>
<tr>
<td>12</td>
<td>10%</td>
<td>82%</td>
<td>5%</td>
<td>3%</td>
<td>14</td>
</tr>
</tbody>
</table>

1Ghz Athlon, Memory usage <= 20MB
Path Analysis
by Integer Linear Programming (ILP)

• Execution time of a program =

\[ \sum_{\text{Basic\_Block } b} \text{Execution\_Time}(b) \times \text{Execution\_Count}(b) \]

• 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
if a then
  b
elseif c then
  d
else
  e
endif
f

max: $4x_a + 10x_b + 3x_c + 2x_d + 6x_e + 5x_f$

where

$x_a = x_b + x_c$
$x_c = x_d + x_e$
$x_f = x_b + x_d + x_e$

Value of objective function: 19

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_a$</td>
<td>1</td>
</tr>
<tr>
<td>$x_b$</td>
<td>1</td>
</tr>
<tr>
<td>$x_c$</td>
<td>0</td>
</tr>
<tr>
<td>$x_d$</td>
<td>0</td>
</tr>
<tr>
<td>$x_e$</td>
<td>0</td>
</tr>
<tr>
<td>$x_f$</td>
<td>1</td>
</tr>
</tbody>
</table>
Timing Predictability

Experience has shown that the precision of results depend on system characteristics
- of the underlying hardware platform and
- of the software layers
- We will concentrate on the influence of the HW architecture on the predictability

What do we intuitively understand as Predictability?
Is it compatible with the goal of optimizing average-case performance?
What is a strategy to identify good compromises?
Predictability of Cache Replacement Policies
Uncertainty in Cache Analysis

1. Initial cache contents?
2. Need to combine information
3. Cannot resolve address of x...
4. Imprecise analysis domain/update functions

→ Need to recover information:
    Predictability = Speed of Recovery
Metrics of Predictability:

Two Variants:
M = Misses Only
HM evict & fill
Meaning of evict/fill - I

• **Evict**: *may*-information:
  - What is definitely not in the cache?
  - Safe information about Cache Misses

• **Fill**: *must*-information:
  - What is definitely in the cache?
  - Safe information about Cache Hits
Meaning of evict/fill - II

Metrics are independent of analyses:

→ evict/fill bound the precision of any static analysis!

→ Allows to analyze an analysis: Is it as precise as it gets w.r.t. the metrics?
Replacement Policies

- **LRU** - Least Recently Used
  Intel Pentium, MIPS 24K/34K
- **FIFO** - First-In First-Out (Round-robin)
  Intel XScale, ARM9, ARM11
- **PLRU** - Pseudo-LRU
  Intel Pentium II+III+IV, PowerPC 75x
- **MRU** - Most Recently Used
MRU - Most Recently Used

MRU-bit records whether line was recently used

```
[a, b, c, d] 0101
[e, b, c, d] 1101
[e, b, c, d] 0010
```

Problem: never stabilizes for 5 acc.
Tree maintains order:

Problem: accesses "rejuvenate" neighborhood
# Results: tight bounds

<table>
<thead>
<tr>
<th>Policy</th>
<th>$e_M(k)$</th>
<th>$f_M(k)$</th>
<th>$e_HM(k)$</th>
<th>$f_HM(k)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>LRU</td>
<td>$k$</td>
<td>$k$</td>
<td>$k$</td>
<td>$k$</td>
</tr>
<tr>
<td>FIFO</td>
<td>$k$</td>
<td>$k$</td>
<td>$2k - 1$</td>
<td>$3k - 1$</td>
</tr>
<tr>
<td>MRU</td>
<td>$2k - 2$</td>
<td>$\infty / 2k - 4^g$</td>
<td>$2k - 2$</td>
<td>$\infty / 3k - 4^g$</td>
</tr>
<tr>
<td>PLRU</td>
<td>$\left{ \frac{2k - \sqrt{2k}}{2k - \frac{3}{2} \sqrt{k}} \right}$</td>
<td>$2k - 1$</td>
<td>$\frac{k}{2} \log_2 k + 1$</td>
<td>$\frac{k}{2} \log_2 k + k - 1$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$k = 4$</th>
<th>$k = 8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Policy</td>
<td>$e_M$</td>
</tr>
<tr>
<td>LRU</td>
<td>4</td>
</tr>
<tr>
<td>FIFO</td>
<td>4</td>
</tr>
<tr>
<td>MRU</td>
<td>6</td>
</tr>
<tr>
<td>PLRU</td>
<td>5</td>
</tr>
</tbody>
</table>
## Results: tight bounds

<table>
<thead>
<tr>
<th>Policy</th>
<th>$e_M(k)$</th>
<th>$f_M(k)$</th>
<th>$e_{HM}(k)$</th>
<th>$f_{HM}(k)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>LRU</td>
<td>$k$</td>
<td>$k$</td>
<td>$k$</td>
<td>$k$</td>
</tr>
<tr>
<td>FIFO</td>
<td>$k$</td>
<td>$k$</td>
<td>$2k - 1$</td>
<td>$3k - 1$</td>
</tr>
<tr>
<td>MRU</td>
<td>$2k - 2$</td>
<td>$\infty/2k - 4^\delta$</td>
<td>$2k - 2$</td>
<td>$\infty/3k - 4^\delta$</td>
</tr>
<tr>
<td>PLRU</td>
<td>$\left{ \frac{2k - \sqrt{2k}}{2k - \frac{3}{2} \sqrt{k}} \right}$</td>
<td>$2k - 1$</td>
<td>$\frac{k}{2} \log_2 k + 1$</td>
<td>$\frac{k}{2} \log_2 k + k - 1$</td>
</tr>
</tbody>
</table>

$f(k) - e(k) \leq k$ in general
Results: instances for k=4,8

<table>
<thead>
<tr>
<th>Policy</th>
<th>$e_M$</th>
<th>$f_M$</th>
<th>$e_{HM}$</th>
<th>$f_{HM}$</th>
<th>$e_M$</th>
<th>$f_M$</th>
<th>$e_{HM}$</th>
<th>$f_{HM}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>LRU</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>FIFO</td>
<td>4</td>
<td>4</td>
<td>7</td>
<td>11</td>
<td>8</td>
<td>8</td>
<td>15</td>
<td>23</td>
</tr>
<tr>
<td>MRU</td>
<td>6</td>
<td>∞/4</td>
<td>6</td>
<td>∞/8</td>
<td>14</td>
<td>∞/12</td>
<td>14</td>
<td>∞/20</td>
</tr>
<tr>
<td>PLRU</td>
<td>5</td>
<td>7</td>
<td>5</td>
<td>7</td>
<td>12</td>
<td>15</td>
<td>13</td>
<td>19</td>
</tr>
</tbody>
</table>

Question: 8-way PLRU cache, 4 instructions per line  
Assume equal distribution of instructions over 256 sets:  
How long a straight-line code sequence is needed to obtain precise may-information?
Future Work I

OPT = theoretical strategy, optimal for performance
LRU = used in practice, optimal for predictability

• Predictability of OPT?
• Other optimal policies for predictability?
Future Work II

Beyond evict/fill:
• Evict/fill assume complete uncertainty
• What if there is only partial uncertainty?
• Other useful metrics?
LRU has Optimal Predictability, so why is it Seldom Used?

• LRU is more expensive than PLRU, Random, etc.
• But it can be made fast
  - Single-cycle operation is feasible [Ackland JSSC00]
  - Pipelined update can be designed with no stalls
• Gets worse with high-associativity caches
  - Feasibility demonstrated up to 16-ways
• There is room for finding lower-cost highly-predictable schemes with good performance
Classification of Pipelines

- **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
Recommendation for Pipelines

- Use *compositional pipelines*; often execution time is dominated by memory-access times, anyway.
- *Static branch prediction only*;
- *One level of speculation only*
Conclusion

• The timing-analysis problem for uninterrupted execution is solved - even for complex platforms and large programs.
• The determination of preemption costs is solved, but needs to be integrated into the tools.
• Feasibility, efficiency, and precision of timing analysis strongly depend on the execution platform.
Relevant Publications (from my group)

• C. Ferdinand et al.: Reliable and Precise WCET Determination of a Real-Life Processor, EMSOFT 2001
• M. Langenbach et al.: Pipeline Modeling for Timing Analysis, SAS 2002
• R. Wilhelm: AI + ILP is good for WCET, MC is not, nor ILP alone, VMCAI 2004
• R. Wilhelm: Determination of Execution-Time Bounds, CRC Handbook on Embedded Systems, 2005