Hardware-Software Codesign

9. Worst Case Execution Time Analysis

Lothar Thiele
System Design

- Specification
- System Synthesis
  - Estimation
- SW-Compilation
  - Intellectual Prop. Code
  - Machine Code
- Instruction Set
- HW-Synthesis
  - Intellectual Prop. Block
  - Net lists
Performance Estimation Methods – Illustration

- Worst-case
- Best-case
- Real system
- Measurement
- Simulation
- Probabilistic estimation
- Worst case (formal) analysis

\[ \text{e.g. delay} \]

→ chapter 6
→ chapter 9-10
Contents

- Introduction
  - problem statement, tool architecture
- Program Path Analysis
- Value Analysis
- Caches
  - must, may analysis
- Pipelines
  - Abstract pipeline models
  - Integrated analyses

The slides are based on lectures of Reinhard Wilhelm.
Industrial Needs

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

Sideairbag in car,
Reaction in <10 mSec

Wing vibration of airplane,
sensing every 5 mSec
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)
Measurement – Industry's “best practice”

Works if either
• worst-case input can be determined, or
• exhaustive measurements are performed

Otherwise:
Determine upper bound from execution times of instructions

Unsafe: Execution Time Measurement
(Most of) Industry’s Best Practice

**Measurements**: determine execution times directly by observing the execution or a simulation on a set of inputs.
- Does *not guarantee* an upper bound to all executions in general.
- *Exhaustive execution* in general *not possible*! Too large space of (input domain) × (set of initial execution states).

**Compute upper bounds** along the *structure* of the program:
- Programs are hierarchically structured.
- Statements are nested inside statements.
- So, try to compute the upper bound for a statement from the upper bounds of its constituents -> does this work?
Sequence of Statements

$A \equiv A_1; A_2$

Constituents of $A$: $A_1$ and $A_2$

Upper bound for $A$ is the sum of the upper bounds for $A_1$ and $A_2$

$\text{ub}(A) = \text{ub}(A_1) + \text{ub}(A_2)$
Conditional Statement

A ≡ if B then A1 else A2

Constituents of A:
1. condition B
2. statements A1 and A2

ub(A) = ub(B) + max(ub(A1), ub(A2))
Loops

\[ A \equiv \text{for } i \leftarrow 1 \text{ to } 100 \text{ do } \]

\[ A1 \]

\[ i \leftarrow 1 \]

\[ i \leq 100 \]

\[ \text{if } i \leq 100, \text{ then } \text{yes, else } \text{no} \]

\[ \text{ub}(A) = \]

\[ \text{ub}(i \leftarrow 1) + 100 \times (\text{ub}(i \leq 100) + \text{ub}(A1)) + \text{ub}(i \leq 100) \]
Where to start?

**Assignment**

\[ x \leftarrow a + b \]

\[ u_b(x \leftarrow a + b) = \text{cycles}(\text{load } a) + \text{cycles}(\text{load } b) + \text{cycles}(\text{add}) + \text{cycles}(\text{store } x) \]

 Assumes constant execution times for instructions

\[ \text{store } x \]

Not applicable to modern processors!

<table>
<thead>
<tr>
<th>move</th>
<th>cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Modern Hardware Features

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

- These features make *WCET 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

\[ x = a + b; \]

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

PPC 755

Execution Time (Clock Cycles)

Best Case

Worst Case

Clock Cycles
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
Overall Approach: Modularization

Micro-architecture Analysis:
- Uses Abstract Interpretation
- Excludes as many Timing Accidents as possible
- Determines WCET for basic blocks (in contexts)

Worst-case Path Determination
- Maps control flow graph to an integer linear program
- Determines upper bound and associated path
Overall Structure

Executable program

Control-Flow-Graph
to improve WCET bounds for loops

CFG Builder

Loop Unfolding

Static Analyses
- Value Analyzer
- Cache/Pipeline Analyzer

Static Analyses

Micro-architecture Analysis

Path Analysis
- ILP-Generator
- LP-Solver
- Evaluation

Worst-case Path Determination

Timing Information

Loop-Bounds

WCET-Visualization
Contents

- Introduction
  - problem statement, tool architecture

- *Program Path Analysis*

- Value Analysis

- Caches
  - must, may analysis

- Pipelines
  - Abstract pipeline models
  - Integrated analyses
Control Flow Graph (CFG)

what_is_this {
    read (a,b);
    done = FALSE;
    repeat {
        if (a>b)
            a = a-b;
        elseif (b>a)
            b = b-a;
        else done = TRUE;
    } until done;
    write (a);
}
Program Path Analysis

- **Program Path Analysis**
  - which sequence of instructions is executed in the worst-case (longest runtime)?
  - *problem*: the number of possible program paths grows exponentially with the program length

- **Model**
  - we know the upper bounds (number of cycles) for each basic block from static analysis
  - number of loop iterations must be bounded

- **Concept**
  - transform structure of CFG into a set of (integer) linear equations.
  - solution of the Integer Linear Program (ILP) yields bound on the WCET.
**Basic Block**

**Definition:** A basic block is a sequence of instructions where the control flow enters at the beginning and exits at the end, without stopping in-between or branching (except at the end).

```
t1 := c - d
 t2 := e * t1
 t3 := b * t1
 t4 := t2 + t3
if t4 < 10 goto L
```
Basic Blocks

- Determine basic blocks of a program:
  1. Determine the first instructions of blocks:
     - the first instruction
     - targets of un/conditional jumps
     - instructions that follow un/conditional jumps
  2. Determine the basic blocks:
     - there is a basic block for each block beginning
     - the basic block consists of the block beginning and runs
       until the next block beginning (exclusive) or until the
       program ends
Control Flow Graph with Basic Blocks

"Degenerated" control flow graph (CFG)
- the nodes are the basic blocks

```
i := 0
t2 := 0
L  t2 := t2 + i
i := i + 1
if i < 10 goto L
x := t2
```
/* k >= 0 */
s = k;
WHILE (k < 10) {
    IF (ok)
        j++;
    ELSE {
        j = 0;
        ok = true;
    }
k ++;
}
r = j;
Calculation of the WCET

**Definition:** A program consists of $N$ basic blocks, where each basic block $B_i$ has a worst-case execution time $c_i$ and is executed for exactly $x_i$ times. Then, the WCET is given by

$$WCET = \sum_{i=1}^{N} c_i \cdot x_i$$

- the $c_i$ values are determined using the static analysis.
- how to determine $x_i$?
  - structural constraints given by the program structure
  - additional constraints provided by the programmer (bounds for loop counters, etc.; based on knowledge of the program context)
Structural Constraints

Flow equations:

\[
\begin{align*}
d1 &= d2 = x_1 \\
d2 + d8 &= d3 + d9 = x_2 \\
d3 &= d4 + d5 = x_3 \\
d4 &= d6 = x_4 \\
d5 &= d7 = x_5 \\
d6 + d7 &= d8 = x_6 \\
d9 &= d10 = x_7
\end{align*}
\]
Additional Constraints

\[ s = k; \]

\[ \text{WHILE } (k < 10) \]

\[ \text{if } (ok) \]

\[ j++; \]

\[ j = 0; \]

\[ ok = \text{true}; \]

\[ k++; \]

\[ r = j; \]

Loop is executed for at most 10 times:

\[ x_3 \leq 10 \cdot x_1 \]

B5 is executed for at most one time:

\[ x_5 \leq 1 \cdot x_1 \]
ILP with structural and additional constraints:

\[
WCET = \max \left\{ \sum_{i=1}^{N} c_i \cdot x_i \mid d_1 = 1 \land \sum_{j \in \text{in}(B_i)} d_j = \sum_{k \in \text{out}(B_i)} d_k = x_i, i = 1 \ldots N \right\}
\]

program is executed once

additional constraints
Contents

- Introduction
  - problem statement, tool architecture
- Program Path Analysis
- **Value Analysis**
- Caches
  - must, may analysis
- Pipelines
  - Abstract pipeline models
  - Integrated analyses
Overall Structure

Executable program

CFG Builder

Loop Unfolding

Control-Flow-Graph to improve WCET bounds for loops

Static Analyses

Value Analyzer

Cache/Pipeline Analyzer

Micro-architecture Analysis

Path Analysis

ILP-Generator

LP-Solver

Evaluation

Timing Information

WCET-Visualization

Micro-architecture Analysis

Worst-case Path Determination

Loop-Bounds
Abstract Interpretation (AI)

- **Semantics-based method** for static program analysis

- **Basic idea of AI**: Perform the program's computations using value descriptions or *abstract values* in place of the concrete values, start with a description of all possible inputs.

- AI supports *correctness* proofs.
Abstract Interpretation – the Ingredients

- **abstract domain** – related to concrete domain by abstraction and concretization functions, e.g. \( L \rightarrow \text{Intervals} \), where \( \text{Intervals} = LB \times UB, LB = UB = \text{Int} \cup \{-\infty, \infty\} \) instead of \( L \rightarrow \text{Int} \)

- **abstract transfer functions** for each statement type – abstract versions of their semantics, e.g. \( + : \text{Intervals} \times \text{Intervals} \rightarrow \text{Intervals} \) where \( [a,b] + [c,d] = [a+c, b+d] \) with + extended to \(-\infty, \infty\)

- **a join function** combining abstract values from different control-flow paths, e.g. \( \cup : \text{Interval} \times \text{Interval} \rightarrow \text{Interval} \) where \( [a,b] \cup [c,d] = [\min(a,c), \max(b,d)] \)
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).
Value Analysis

- Intervals are computed along the CFG edges
- At joins, intervals are "unioned"

\[ D1: [-4,4], A0: [0x1000, 0x1000] \]

\[
\text{move } #4, D0
\]

\[ D0: [4,4], D1: [-4,4], A0: [0x1000, 0x1000] \]

\[
\text{add } D1, D0
\]

\[ D0: [0,8], D1: [-4,4], A0: [0x1000, 0x1000] \]

\[
\text{move } (A0,D0), D1
\]

Which address is accessed here?

\[ \text{access } [0x1000, 0x1008] \]
Contents

- Introduction
  - problem statement, tool architecture
- Program Path Analysis
- Value Analysis
- Caches
  - must, may analysis
- Pipelines
  - Abstract pipeline models
  - Integrated analyses
**Caches: Fast Memory on Chip**

- **Caches are used**, because
  - Fast main memory is too expensive
  - The speed gap between CPU and memory is too large and increasing

- 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
Caches

- **Processor**
  - Access takes ~ 1 cycle
  - **Cache**
    - Access takes ~ 100 cycles
  - **Memory**
    - Fast, small, expensive
    - (relatively) slow, large, cheap

- **Bus**

---

- Fast, small, expensive
- (relatively) slow, large, cheap
Caches: How they work

- CPU wants to \textit{read/write at memory address} \(a\), sends a request for \(a\) to the bus.

- \textbf{Cases}: 
  - Block \(m\) containing \(a\) is in the cache (hit): request for \(a\) is served in the next cycle.
  - Block \(m\) is not in the cache (miss): \(m\) is transferred from main memory to the cache, \(m\) may replace some block in the cache, request for \(a\) is served as soon as possible while the transfer still continues.

- Several \textit{replacement strategies}: LRU, PLRU, FIFO,... determine which line to replace.
4-Way Set Associative Cache
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>access</th>
<th>age 0</th>
<th>age 1</th>
<th>age 2</th>
<th>age 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>m₀</td>
<td>m₁</td>
<td>m₂</td>
<td>m₃</td>
<td></td>
</tr>
<tr>
<td>m₄ (miss)</td>
<td>m₄</td>
<td>m₀</td>
<td>m₁</td>
<td>m₂</td>
</tr>
<tr>
<td>m₁ (hit)</td>
<td>m₁</td>
<td>m₄</td>
<td>m₀</td>
<td>m₂</td>
</tr>
<tr>
<td>m₅ (miss)</td>
<td>m₅</td>
<td>m₁</td>
<td>m₄</td>
<td>m₀</td>
</tr>
</tbody>
</table>
Cache Analysis

How to statically precompute cache contents:

- **Must Analysis:**
  For each program point (and calling context), find out which blocks are in the cache. Determines safe information about cache hits. Each predicted cache hit reduces WCET.

- **May Analysis:**
  For each program point (and calling context), find out which blocks may be in the cache. Complement says what is not in the cache.

  Determines safe information about cache misses. Each predicted cache miss increases BCET.
Contribution to WCET

Information about cache contents sharpens timings.

- ref to s
  - $t_{hit}$
  - $t_{miss}$

- if s is in must-cache:
  - $t_{WCET} = t_{hit}$
  - otherwise: $t_{WCET} = t_{miss}$

- if s is in may-cache:
  - $t_{BCET} = t_{hit}$
  - otherwise: $t_{BCET} = t_{miss}$
Abstract Domain: Must Cache

Abstraction

\[
\begin{array}{c}
\{s\} \\
\{z,x\} \\
\{s\}
\end{array}
\]
Abstract Domain: Must Cache

Concretization

\[ z, x \in \{ s \in \{ \emptyset, \{z,x\}, \{s\} \} \} \]
Cache with LRU: Transfer for must

concrete
[ access s ]

abstract
[ access s ]

"young"
[ access s ]
Cache Analysis: Join (must)

Join (must)

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

```
{ c }
{   }
{ a }
{ d }
```

“intersection + maximal age”

```
{   }
{   }
{ a, c }
{ d }
```

**Interpretation:**
memory block a is definitively in the (concrete) cache => always hit
Abstract Domain: May Cache

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

\[ \{t\} \]

\[ \{\} \]

\[ \{a\} \]
Abstract Domain: May Cache

Concretization

\[ m \in \{z, s, x\} \]
\[ n, o \in \{z, s, x, t\} \]
\[ p \in \{z, s, x, t, a\} \]
Cache with LRU: Transfer for may

**concrete**

<table>
<thead>
<tr>
<th>x</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>x</td>
</tr>
<tr>
<td>z</td>
<td>t</td>
</tr>
<tr>
<td>y</td>
<td>z</td>
</tr>
</tbody>
</table>

**abstract**

\[
\begin{align*}
\{ x,t \} & \quad \{ s \} \\
\{ y,s \} & \quad \{ x, t \} \\
\{ z \} & \quad \{ y, z \} \\
\{ \} & \quad \{ \} 
\end{align*}
\]
Cache Analysis: Join (may)

Join (may)

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

“union + minimal age”

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

**Interpretation:**
all blocks may be in the cache; none is definitely not in the cache.
Contribution to WCET

Information about cache contents sharpens timings.

\[
\begin{align*}
\text{if } s \text{ is in must-cache:} & \quad t_{WCET} = t_{hit} \\
\text{otherwise} & \quad t_{WCET} = t_{miss} \\
\text{if } s \text{ is in may-cache:} & \quad t_{BCET} = t_{hit} \\
\text{otherwise} & \quad t_{BCET} = t_{miss}
\end{align*}
\]
Contribution to WCET

- Information about cache contents sharpens timings.

```plaintext
while . . . do [max n]

  .
  .
  ref to s
  .
  .

  od

within loop

  n * t_{miss}
  n * t_{hit}
  t_{miss} + (n - 1) * t_{hit}
  t_{hit} + (n - 1) * t_{miss}
  ...
```
Contexts

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

- First Iteration loads the cache:
  - Intersection loses most of the information.

- Distinguish as many contexts as useful:
  - 1 unrolling for caches
  - 1 unrolling for branch prediction (pipeline)
Contents

- Introduction
  - problem statement, tool architecture
- Program Path Analysis
- Value Analysis
- Caches
  - must, may analysis
- **Pipelines**
  - Abstract pipeline models
  - Integrated analyses
Comparison of Architectures

- **Single Cycle**
  - T1
  - LW
  - SW

- **Multiple Cycle**
  - T1 T2 T3 T4 T5
  - LW
  - SW
  - IF RF EX MEM WB

- **Pipelining**
  - T1 T2 T3 T4 T5
  - LW
  - SW
  - IF RF EX MEM WB

Einzyklenverarb.  
Mehrzyklenverarb.  
Pipelineverarb.
Hardware Features: Pipelines

- **Fetch**
- **Decode**
- **Execute**
- **WB**

Ideal Case: 1 Instruction per Cycle
Datapath of a Pipeline Architecture
Hardware Features: 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

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
Control Hazard

Program execution order (in instructions)

40 beq $1, $3, 28
44 and $12, $2, $5
48 or $13, $6, $2
52 add $14, $2, $2
72 lw $4, 50($7)
Data Hazard

Program execution order (in instructions)

- sub $2, $1, $3
- and $12, $2, $5
- or $13, $6, $2
- add $14, $2, $2
- sw $15, 100($2)
### Cache analysis
Prediction of cache hits on instruction or operand fetch or store

```
lwz r4, 20(r1)
```

**Hit**

### Dependence analysis
Analysis of data/control hazards

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

**Operand ready**

### Resource reservation tables
Analysis of resource hazards

![Resource reservation tables diagram]
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

- **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 for a Basic Block

- function `exec (b : basic block, s : abstract pipeline state)`
  - `t`: trace
    - interprets instruction stream of `b` (annotated with cache information) starting in state `s` producing trace `t`
    - `length(t)` gives number of cycles

- What is different?
  - Abstract states may lack information, e.g. about cache contents.
  - Assume local worst cases is safe (in the case of no timing anomalies)
  - Traces may be longer (but never shorter).
**What is different?**

- **Question:** What is the starting state for successor basic block? In particular, if there are several predecessor blocks in case of a join?

- **Alternative solutions:**
  - Proceed with sets of states, i.e. several “simulations”.
  - Combine states by assuming that the local worst case is safe.
Summary of Steps

- **Value analysis**

- **Cache analysis** using statically computed effective addresses and loop bounds

- **Pipeline analysis**
  - assume cache hits where predicted,
  - assume cache misses where predicted or not excluded,
  - only the “worst” result states of an instruction need to be considered as input states for successor instructions.
aiT-Tool

- **Input:** an executable program, starting points, loop iteration counts, call targets of indirect function calls, and a description of bus and memory speeds

- **Output:** computes **Worst-Case Execution Time** bounds of tasks