Worst-Case Execution Time Analysis

Prof. Dr. Jian-Jia Chen

LS 12, TU Dortmund

13, May 2014
Most Essential Assumptions for Real-Time Systems

*Upper bound on the execution times:*

- Commonly, called the *Worst-Case Execution Time (WCET)*
What does Execution Time Depend on

- Input parameters
  - Algorithm parameters
  - Problem size
  - etc.

- Initial states and intermediate states of the system for executing
  - Cache configuration, replacement policies
  - Pipelines
  - Speculations
  - etc.

- Interferences from the environment
  - Scheduling
  - Interrupts
  - etc.
How to Derive the Worst-Case Execution Time (WCET)

- Most of industry’s best practice
  - Measure it: determine WCET directly by running or simulating a set of inputs.
    - There is no guarantee to give an upper bound of the WCET.
    - The derived WCET could be too optimistic.
  - Exhaustive execution: by considering the set of all the possible inputs
    - In general, not possible
    - The inputs have to cover all the possible initial states and intermediate states of the system, which is also usually not possible.

- Compute it
  - In general, not possible neither, as computing (tight) WCET for a program is *uncomputable* by Turing machines.
  - Based on some structures, it is possible and the derived solution is a safe upper bound of the WCET.
Why is It Uncomputable?

**Halting Problem**
Given the description of a Turing machine $m$ and its input $x$, the problem is to answer whether the machine halts on $x$.

**Theorem**
Halting Problem is undecidable (uncomputable). In other words, one cannot use an algorithm to decide whether another algorithm $m$ halts on a specific input.

**WCET is undecidable**
It is even undecidable if it terminates at all. Deriving the WCET is of course of undecidable.

Please refer to the textbook of Computational Complexity by Prof. Papadimitriou.
**Our objectives:**

- **Upper bound** of execution time as tightly as possible.
- All control-flow paths, by considering all possible inputs.
- All paths through the architecture, resulting from the potential initial and assumed intermediate architectural states.
Timing Analysis

By considering systems, in general, with

- finite architectural configurations, finite input domains, and bounded loops and recursion,

WCET is computable.
Timing Analysis

By considering systems, in general, with

- finite architectural configurations, finite input domains, and bounded loops and recursion,

WCET is computable.

But....... the search space is too large to explore it exhaustively!
Why is It Hard for Analyzing WCET?

Execution time $e(i)$ of machine instruction $i$

- In the good old time:
  $e(i)$ is a constant $c$, which could be found in the data sheet
- Nowadays, especially for high-performance processors:
  $e(i)$ also depends on the (architectural) execution state $s$. 

$$\min\{e(i, s) | s \in S\} \leq e(i) \leq \max\{e(i, s) | s \in S\},$$

where $S$ is the set of all states.

- Using $\max\{e(i, s) | s \in S\}$ is safe for WCET, but might be not tight since some states in $S$ might not be possible reached by some inputs.

- Execution history, resulting in a smaller set of reachable execution states, has to be enforced to improve the tightness of the analysis.
Variability of Execution 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

- **Architecture Analysis:**
  - Use Abstract Interpretation.
  - Exclude as many Timing Accidents as possible during analysis.
    - Certain timing accidents will never happen, e.g., at a certain program point, instruction fetch will never cause a cache miss.
    - The more accidents excluded, the lower (better) the upper bound.
  - Determine WCET for basic blocks, based on context information.

- **Worst-Case Path Determination:**
  - Map control flow graph to an Integer Linear Program (ILP).
  - Determine upper bound and associated path.

**High-Level Objectives: Upper Bound of WCET**

- It must be safe, i.e., not underestimate.
- It should be tight, i.e., not far away from real WCET.
- The analysis effort must be tolerable.
Overall Structure

Executable Binary
Program

Control-Flow-Graph (CFG)
Reconstruction

Loop Analysis and Unfolding

Loop Bounds

Static Analysis

Value Analyzer

Cache/Pipeline Analyzer

Path Analysis

ILP-Generator

ILP-Solver

Evaluation

Micro-Architecture Abstraction

Timing Information

WCET Visualization and Analysis

Micro-architecture Analysis

Worst-Case Path Analysis
Outline

Introduction

Program Path Analysis

Static Analysis
  Value Analysis
  Cache Analysis
  Pipeline Analysis
Control Flow Graph (CFG)

```python
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);
}
```
Basic Blocks

Definition: A basic block is a sequence of instructions where the control flow enters at the beginning and exits at the end, in which it is highly amenable to analysis.

\[
\begin{align*}
a[0] & := b[0] + c[0] \\
d & := a[0] \ast a[1] \\
e & := d/a[2] \\
\text{if } e < 10 \text{ goto } L
\end{align*}
\]
Basic Blocks

Definition: A basic block is a sequence of instructions where the control flow enters at the beginning and exits at the end, in which it is highly amenable to analysis.

Determining the basic blocks

- **Beginning:**
  - the first instruction
  - targets of un/conditional jumps
  - instructions that follow un/conditional jumps

- **Ending:**
  - the basic block consists of the block beginning and runs until the next block beginning (exclusive) or until the program ends

\[
a[0] := b[0] + c[0] \\
d := a[0] \times a[1] \\
e := d / a[2] \\
\text{if } e < 10 \text{ goto } L
\]
Program Path Analysis

- **Problem:** Which sequence of instructions is executed in the worst case (i.e., the longest execution time)?
- **Input:**
  - Timing information for each basic block, derived from static analysis (value/cache/pipeline analysis)
  - Loop bounds by specification
  - CFG derived from the executable binary program
- **Basic Concept:**
  - Transform structure of CFG into a set of (integer) linear equations
  - Solution of the Integer Linear Program (ILP) yields bound on the WCET.
# Program Path Analysis: Formal Definition

### Input

A CFG with $N$ basic blocks, in which each basic block $B_i$ has a worst-case execution time $c_i$, given by static analysis.

### Output

Suppose that each block $B_i$ is executed exactly $x_i$ times. What is the worst-case execution time

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

such that the values of $x_i$s satisfy the structural constraints in the CFG?

Note that additional constraints provided by the programmer (bounds for loop counters, etc.) can also be included.
Example for CFG Constraints

Flow equations: ($x_i$ is a variable)

- $d_1 = d_2 = x_1$
- $d_3 + d_9 = d_2 + d_8 = x_2$
- $d_4 + d_5 = d_3 = x_3$
- $d_6 + d_7 = d_8 = x_4$
- $d_4 = d_6 = x_5$
- $d_5 = d_7 = x_6$
The loop is executed for at most 20 times when $k$ is initialized with a non-negative number:

$$x_3 \leq 20x_1.$$ 

The basic block for $j = 0; ok = true;$ is executed for at most one time:

$$x_6 \leq x_1.$$
WCET: ILP Formulation

\[
WCET = \max \left\{ \sum_{i=1}^{N} c_i \cdot x_i \right\}
\]

\[d_1 = 1\]

and

\[
\sum_{j \in \text{in}(B_i)} d_j = \sum_{k \in \text{out}(B_j)} d_k = x_i, \ i = 1, \ldots, N
\]

and additional constraints
Outline

Introduction

Program Path Analysis

Static Analysis
  Value Analysis
  Cache Analysis
  Pipeline Analysis
Overall Structure

Executable Binary
Program

Control–Flow–Graph (CFG)
Reconstruction

Loop Analysis and Unfolding

Loop Bounds

Static Analysis

Value Analyzer

Cache/Pipeline Analyzer

Micro–Architecture Abstraction

Timing Information

Path Analysis

ILP–Generator

ILP–Solver

Evaluation

WCET Visualization
and Analysis

Micro–architecture Analysis

Worst–Case Path Analysis

Prof. Dr. Jian-Jia Chen (LS 12, TU Dortmund) 22 / 52
Outline

Introduction

Program Path Analysis

Static Analysis

Value Analysis

Cache Analysis

Pipeline Analysis
Value Analysis: Motivation and Method

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

- **Method**
  - Calculate intervals at all program points
  - By considering addresses, register contents, local and global variables.

**Abstract Interpretation**

Perform the program’s computation using value descriptions or abstract values in place of the concrete values.
Abstract Interpretation: Manipulation

- abstract domain - related to concrete domain by abstraction and concretization functions
  - Replace an integer/double operator by using intervals
  - e.g., $L = [3, 5]$ stands for $L$ is a value between 3 and 5
- abstract transfer functions for each statement type
  - e.g., operator $+$: $[3, 5] + [2, 6] = [5, 11]$
  - e.g., operator $-$: $[3, 5] - [2, 6] = [-3, 3]$
- a join function combining intervals from different paths
  - $[a, b]$ join $[c, d]$ becomes $[\min\{a, c\}, \max\{b, d\}]$
Value Analysis

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

**move #4, D0**

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

**add D1, D0**

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

**move (A0, D0), D1**

**access [0x1000, 0x1008]**

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

\[ D1: [-2, +2] \]
\[ D1: [-4, 0] \]
\[ D1: [-4, +2] \]
Outline

Introduction

Program Path Analysis

Static Analysis
  Value Analysis
  Cache Analysis
  Pipeline Analysis
Overall Structure

Executable Binary

Control-Flow-Graph (CFG) Reconstruction

Loop Analysis and Unfolding

Loop Bounds

Static Analysis

Value Analyzer

Cache/Pipeline Analyzer

Micro-Architecture Abstraction

Timing Information

Path Analysis

ILP-Generator

ILP-Solver

Evaluation

WCET Visualization and Analysis

Micro-architecture Analysis

Worst-Case Path Analysis
Caches: Fast Memory to Deal with the Memory Wall

- How they work:
  - dynamically
  - managed by replacement policy

- Why they work: *principle of locality*
  - spatial
  - temporal
Caches: Fast Memory to Deal with the Memory Wall

- How they work:
  - dynamically
  - managed by replacement policy

- Why they work: principle of locality
  - spatial
  - temporal

- Diagram:
  - CPU
  - Cache
  - Main Memory
  - Capacity:
    - CPU: 32 KB
    - Main Memory: 2 MB
  - Latency:
    - CPU: 3 cycles
    - Main Memory: 100 cycles

“hit” [ab]
Caches: Fast Memory to Deal with the Memory Wall

- How they work:
  - dynamically
  - managed by replacement policy

![Diagram of CPU, Cache, and Main Memory]

- Why they work: *principle of locality*
  - spatial
  - temporal
Caches: Fast Memory to Deal with the Memory Wall

• How they work:
  • dynamically
  • managed by replacement policy

CPU

“miss”
[ab]

Cache

Main Memory

Capacity: 32 KB
Latency: 3 cycles

2 MB
100 cycles

• Why they work: *principle of locality*
  • spatial
  • temporal
Caches: Fast Memory to Deal with the Memory Wall

- How they work:
  - dynamically
  - managed by replacement policy

![Diagram showing CPU, Cache, and Main Memory with latency and capacity details]

- Why they work: principle of locality
  - spatial
  - temporal
Caches: Fast Memory to Deal with the Memory Wall

- How they work:
  - dynamically
  - managed by replacement policy

- Why they work: *principle of locality*
  - spatial
  - temporal

![Diagram of CPU, Cache, and Main Memory with capacities and latencies.](image_url)
Caches: Fast Memory to Deal with the Memory Wall

- How they work:
  - dynamically
  - managed by replacement policy

- Why they work: *principle of locality*
  - spatial
  - temporal

![Diagram of CPU, Cache, and Main Memory with capacities and latencies](image)

- Capacity:
  - CPU: 32 KB
  - Main Memory: 2 MB

- Latency:
  - CPU: 3 cycles
  - Main Memory: 100 cycles

“miss” [ac] c₃!
Caches: Fast Memory to Deal with the Memory Wall

- How they work:
  - dynamically
  - managed by replacement policy

```
CPU → Cache → Main Memory
```

- Why they work: *principle of locality*
  - spatial
  - temporal

Capacity:
- 32 KB
- 2 MB

Latency:
- 3 cycles
- 100 cycles
Caches: Fast Memory to Deal with the Memory Wall

- How they work:
  - dynamically
  - managed by replacement policy

- Why they work: principle of locality
  - spatial
  - temporal

Diagram:
- CPU
- Cache
- Main Memory
- "hit" [ac]
- Capacity: 32 KB
- Latency: 3 cycles
- Capacity: 2 MB
- Latency: 100 cycles
Fully Associative Caches

Address:

Tag

Block offset

\( \log_2(8 \times b) \)

\( k' = \text{associativity} \)

MUX

Data

\( =? \)

Yes: Hit!

No: Miss!
Set-Associative Caches

Special cases:
- direct-mapped cache: only one line per cache set
- fully-associative cache: only one cache set
Replacement Policies

- Least-Recently-Used (LRU) used in Intel Pentium I and MIPS 24K/34K
- First-In First-Out (FIFO or Round-Robin) used in Motorola PowerPC 56x, Intel XScale, ARM9, ARM11
- Pseudo-LRU (PLRU) used in Intel Pentium II-IV and PowerPC 75x
- Most Recently Used (MRU) as described in literature

Each cache set is treated independently:
→ Set-associative caches are compositions of fully-associative caches.
Cache Analysis for LRU

Two types of cache analyses:

1. Local guarantees: classification of individual accesses
   - Must-Analysis $\rightarrow$ Underapproximates cache contents
   - May-Analysis $\rightarrow$ Overapproximates cache contents

2. Global guarantees: bounds on cache hits/misses
Challenges for Cache Analysis

1. Initial cache contents unknown.
2. Different paths lead to these points.
3. Cannot resolve address of \( z \).

Always a cache hit/always a miss?
Challenges for Cache Analysis

- Always a cache hit/always a miss?
  - 1. Initial cache contents unknown.
  - 2. Different paths lead to these points.
  - 3. Cannot resolve address of z.
Using Abstract Interpretation

Collecting Semantics =
set of states at each program point that any execution may encounter there

Two approximations:

Collecting Semantics  uncomputable
⊆ Cache Semantics  computable
⊆ γ(Absact Cache Sem.) efficiently com-putable
Using Abstract Interpretation

**Collecting Semantics** = set of states at each program point that any execution may encounter there.

Two approximations:

- Collecting Semantics uncomputable
- Cache Semantics computable
- $\subseteq$ Cache Semantics computable
- $\subseteq$ $\gamma$(Abstract Cache Sem.) efficiently computable
Using Abstract Interpretation

**Collecting Semantics** = set of states at each program point that any execution may encounter there

Two approximations:
- Collecting Semantics \( \subseteq \) Cache Semantics computable
- Collecting Semantics \( \subseteq \gamma(A) \) (Abstract Cache Sem.) efficiently computable
Using Abstract Interpretation

Collecting Semantics = set of states at each program point that any execution may encounter there

Two approximations:

- Collecting Semantics uncomputable
- Cache Semantics computable
- \( \subseteq \) \( \gamma \) (Abstract Cache Sem.) efficiently computable
Using Abstract Interpretation

Collecting Semantics = set of states at each program point that any execution may encounter there.

Two approximations:

- Collecting Semantics uncomputable
- Cache Semantics computable
- $\gamma$(Abstract Cache Sem.) efficiently computable
Least-Recently-Used (LRU): Concrete Behavior

“Cache Miss”:

```
  z
  y
  x
  t
```

LRU has notion of age

```
  s
  z
  y
  x
```

“Cache Hit”:

```
  z
  y
  s
  t
```

```
  s
  z
  y
  t
```
LRU: Must-Analysis: Abstract Domain

- Used to predict \textit{cache hits}.
- Maintains \textit{upper bounds on ages} of memory blocks.
- Upper bound \( \leq \) associativity \(\rightarrow\) memory block definitely cached.

\textbf{Example}

\(\ldots\) and its interpretation:

\textbf{Abstract state:}

\[
\begin{array}{c|c}
\text{age 0} & 
\begin{cases}
\{x\} \\
\{\} \\
\{s,t\} \\
\{\} \\
\end{cases} \\
\end{array}
\]

Describes the set of all concrete cache states in which \(x\), \(s\), and \(t\) occur,
- \(x\) with an age of 0,
- \(s\) and \(t\) with an age not older than 2.

\[
\gamma([\{x\}, \{\}, \{s, t\}, \{\}]) = \\
\{[x, s, t, a], [x, t, s, a], [x, s, t, b], \ldots\}
\]
Sound Update – Local Consistency

Concrete cache states \( \gamma \) \rightarrow \text{Abstract Update} \rightarrow \gamma \text{ (must′)}

Concrete cache states

Lifted Concrete Update

Prof. Dr. Jian-Jia Chen (LS 12, TU Dortmund)
LRU: Must-Analysis: Update

“Potential Cache Miss”:

“Definite Cache Hit”:

Why does $t$ not age in the second case?
LRU: Must-Analysis: Join

Need to combine information where control-flow merges.

Join should be conservative:

- $\gamma(A) \subseteq \gamma(A \sqcup B)$
- $\gamma(B) \subseteq \gamma(A \sqcup B)$

```
\begin{array}{c}
\{a\} \\
\{\} \\
\{c,f\} \\
\{d\}
\end{array}
\bigcup
\begin{array}{c}
\{c\} \\
\{e\} \\
\{a\} \\
\{d\}
\end{array}
\bigcup
\begin{array}{c}
\{\} \\
\{\} \\
\{a,c\} \\
\{d\}
\end{array}
```

“Intersection + Maximal Age”
LRU: Must-Analysis: Join

Need to combine information where control-flow merges.

Join should be conservative:

- $\gamma(A) \subseteq \gamma(A \sqcup B)$
- $\gamma(B) \subseteq \gamma(A \sqcup B)$

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

“Intersection + Maximal Age”
LRU: Must-Analysis: Join

Need to combine information where control-flow merges.

Join should be conservative:

\[
\begin{align*}
\gamma(A) & \subseteq \gamma(A \sqcup B) \\
\gamma(B) & \subseteq \gamma(A \sqcup B)
\end{align*}
\]

\[
\begin{array}{c}
\{a\} \\
\{\} \\
\{c,f\} \\
\{d\}
\end{array}
\sqcup
\begin{array}{c}
\{c\} \\
\{e\} \\
\{a\} \\
\{d\}
\end{array}
\sqcup
\begin{array}{c}
\{\} \\
\{\} \\
\{a,c\} \\
\{d\}
\end{array}
\]

“Intersection + Maximal Age”
LRU: Must-Analysis: Join

Need to combine information where control-flow merges.

Join should be conservative:

\( \gamma(A) \subseteq \gamma(A \sqcup B) \)
\( \gamma(B) \subseteq \gamma(A \sqcup B) \)

"Intersection + Maximal Age"
LRU: Must-Analysis: Join

Need to combine information where control-flow merges.

Join should be conservative:

- $\gamma(A) \subseteq \gamma(A \sqcup B)$
- $\gamma(B) \subseteq \gamma(A \sqcup B)$

```
\[
\begin{array}{c|c|c}
\{a\} & \{c\} & \{\}\ \\
\{\} & \{e\} & \{\} \\
\{c,f\} & \{a\} & \{a,c\} \\
\{d\} & \{d\} & \{d\}
\end{array}
\]
```

“Intersection + Maximal Age”
LRU: Must-Analysis: Join

Need to combine information where control-flow merges.

Join should be conservative:

• $\gamma(A) \subseteq \gamma(A \sqcup B)$
• $\gamma(B) \subseteq \gamma(A \sqcup B)$

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

“Intersection + Maximal Age”
LRU: Must-Analysis: Join

Need to combine information where control-flow merges.

Join should be conservative:
- $\gamma(A) \subseteq \gamma(A \sqcup B)$
- $\gamma(B) \subseteq \gamma(A \sqcup B)$

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>{a}</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>{c}</td>
<td></td>
</tr>
<tr>
<td>{c,f}</td>
<td>{e}</td>
<td></td>
</tr>
<tr>
<td>{d}</td>
<td>{a}</td>
<td>{a,c}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

“Intersection + Maximal Age”

How many memory blocks can be in the must-cache?
Example: Must-Analysis

entry \[\{\}, \{\}, \{\}, \{\}\]
Example: Must-Analysis

entry $[\{\}, \{\}, \{\}, \{\}]$

$\perp \sqcup [\{\}, \{\}, \{\}, \{\}] = [\{\}, \{\}, \{\}, \{\}]$

exit $\perp$
Example: Must-Analysis

\[
\begin{align*}
\text{entry} & \quad [\{\}, \{\}, \{\}, \{\}] \\
\downarrow & \\
A & \quad \bot \sqcup [\{\}, \{\}, \{\}, \{\}] = [\{\}, \{\}, \{\}, \{\}] \\
\downarrow & \\
B & \quad \{A\}, \{\}, \{\}, \{\} \\
\downarrow & \\
D & \quad \bot \\
\downarrow & \\
\text{exit} & \quad \bot
\end{align*}
\]
Example: Must-Analysis

entry  $[\{\}, \{\}, \{\}, \{\}]$

$\perp \sqcup [\{\}, \{\}, \{\}, \{\}] = [\{\}, \{\}, \{\}, \{\}]$

$[\{A\}, \{\}, \{\}, \{\}]$

$[\{A\}, \{\}, \{\}, \{\}]$

$[\{B\}, \{A\}, \{\}, \{\}] \sqcup [\{C\}, \{A\}, \{\}, \{\}] = [\{\}, \{A\}, \{\}, \{\}]$

exit  $\perp$

Prof. Dr. Jian-Jia Chen  (LS 12, TU Dortmund)
Example: Must-Analysis

entry \[\{\}, \{\}, \{\}, \{\}\]\n
\[\{D\}, \{\}, \{A\}, \{\}\} \sqcup \{\}, \{\}, \{\}, \{\}\] = \[\{\}, \{\}, \{\}, \{\}\]

\[\{A\}, \{\}, \{\}, \{\}\]  \[\{A\}, \{\}, \{\}, \{\}\]  \[\{A\}, \{\}, \{\}, \{\}\]  \[\{A\}, \{\}, \{\}, \{\}\]

No cache hits can be predicted!
Context-Sensitive Analysis/Virtual Loop-Unrolling

- **Problem:**
  - The first iteration of a loop will always result in cache misses.
  - Similarly for the first execution of a function.

- **Solution:**
  - Virtually Unroll Loops: Distinguish the first iteration from others
  - Distinguish function calls by calling context.

Virtually unrolling the loop once:

- Accesses to $A$ and $D$ are provably hits after the first iteration
- Accesses to $B$ and $C$ can still not be classified. Within each execution of the loop, they may only miss once.

$\rightarrow$ Persistence Analysis
LRU: May-Analysis: Abstract Domain

- Used to predict *cache misses*.
- Maintains *lower bounds on ages* of memory blocks.
- Lower bound $\geq$ associativity
  $\implies$ memory block definitely *not* cached.

### Abstract state:

<table>
<thead>
<tr>
<th></th>
<th>age 0</th>
<th></th>
<th>age 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>${x,y}$</td>
<td></td>
<td>${s,t}$</td>
<td></td>
</tr>
<tr>
<td>${}$</td>
<td></td>
<td>${}$</td>
<td></td>
</tr>
<tr>
<td>${u}$</td>
<td></td>
<td>${}$</td>
<td></td>
</tr>
</tbody>
</table>

and its interpretation:

Describes a set of all concrete cache states, where no memory blocks except $x$, $y$, $s$, $t$, and $u$ occur,

- $x$ and $y$ with an age of at least 0,
- $s$ and $t$ with an age of at least 2,
- $u$ with an age of at least 3.

$$\gamma([\{x,y\},\{}\{s,t\}\{u\}]) = \{[x,y,s,t],[y,x,s,t],[x,y,s,u],\ldots\}$$
LRU: May-Analysis: Update

“Definite Cache Miss”:

“Potential Cache Hit”:

Why does \( t \) age in the second case?
LRU: May-Analysis: Join

Need to combine information where control-flow merges.

Join should be conservative:

- $\gamma(A) \subseteq \gamma(A \sqcup B)$
- $\gamma(B) \subseteq \gamma(A \sqcup B)$

```
\begin{array}{c}
\{a\} \\
\{\} \\
\{c,f\} \\
\{d\}
\end{array}
\sqcup
\begin{array}{c}
\{c\} \\
\{e\} \\
\{a\} \\
\{d\}
\end{array}
=\begin{array}{c}
\{a,c\} \\
\{e\} \\
\{f\} \\
\{d\}
\end{array}
```

“Union + Minimal Age”
LRU: May-Analysis: Join

Need to combine information where control-flow merges.

Join should be conservative:

- \( \gamma(A) \subseteq \gamma(A \sqcup B) \)
- \( \gamma(B) \subseteq \gamma(A \sqcup B) \)

```
\[
\begin{array}{c}
\{a\} \\
\emptyset \\
\{c,f\} \\
\{d\}
\end{array}
\begin{array}{c}
\{c\} \\
\{e\} \\
\{a\} \\
\{d\}
\end{array}
\begin{array}{c}
\{a,c\} \\
\{e\} \\
\{f\} \\
\{d\}
\end{array}
\]
```

“Union + Minimal Age”
LRU: May-Analysis: Join

Need to combine information where control-flow merges.

Join should be conservative:

- \( \gamma(A) \subseteq \gamma(A \sqcup B) \)
- \( \gamma(B) \subseteq \gamma(A \sqcup B) \)

\[
\begin{array}{c}
\{a\} \\
\{\} \\
\{c,f\} \\
\{d\}
\end{array} 
\sqcup 
\begin{array}{c}
\{c\} \\
\{e\} \\
\{a\} \\
\{d\}
\end{array} =
\begin{array}{c}
\{a,c\} \\
\{e\} \\
\{f\} \\
\{d\}
\end{array}
\]

“Union + Minimal Age”
LRU: May-Analysis: Join

Need to combine information where control-flow merges.

Join should be conservative:

- \( \gamma(A) \subseteq \gamma(A \sqcup B) \)
- \( \gamma(B) \subseteq \gamma(A \sqcup B) \)

\[
\begin{array}{c}
\{a\} \\
\{\} \\
\{c,f\} \\
\{d\}
\end{array}
\sqcup
\begin{array}{c}
\{c\} \\
\{e\} \\
\{a\} \\
\{d\}
\end{array}
= 
\begin{array}{c}
\{a,c\} \\
\{e\} \\
\{f\} \\
\{d\}
\end{array}
\]

“Union + Minimal Age”
LRU: May-Analysis: Join

Need to combine information where control-flow merges.

Join should be conservative:

- $\gamma(A) \subseteq \gamma(A \sqcup B)$
- $\gamma(B) \subseteq \gamma(A \sqcup B)$

```
\begin{align*}
&\{a\} \\
&\{\} \\
&\{c,f\} \\
&\{d\}
\end{align*}
\sqcup
\begin{align*}
&\{c\} \\
&\{e\} \\
&\{a\} \\
&\{d\}
\end{align*}
= \\
\begin{align*}
&\{a,c\} \\
&\{e\} \\
&\{f\} \\
&\{d\}
\end{align*}
```

“Union + Minimal Age”
LRU: May-Analysis: Join

Need to combine information where control-flow merges.

Join should be conservative:

- $\gamma(A) \subseteq \gamma(A \sqcup B)$
- $\gamma(B) \subseteq \gamma(A \sqcup B)$

```
\[
\begin{array}{c}
\{a\} \\
\{} \\
\{c,f\} \\
\{d\}
\end{array}
\sqcup
\begin{array}{c}
\{c\} \\
\{e\} \\
\{a\} \\
\{d\}
\end{array}
= 
\begin{array}{c}
\{a,c\} \\
\{e\} \\
\{f\} \\
\{d\}
\end{array}
\]
```

“Union + Minimal Age”
Outline

Introduction

Program Path Analysis

Static Analysis

Value Analysis
Cache Analysis
Pipeline Analysis
Overall Structure

Executable Binary Program

Control-Flow-Graph (CFG) Reconstruction

Loop Analysis and Unfolding

Loop Bounds

Static Analysis

Value Analyzer

Cache/Pipeline Analyzer

Micro-Architecture Abstraction

Timing Information

Timing Information

Path Analysis

ILP-Generator

ILP-Solver

Evaluation

WCET Visualization and Analysis

Micro-architecture Analysis

Worst-Case Path Analysis

Prof. Dr. Jian-Jia Chen (LS 12, TU Dortmund)
Pipelines

An instruction execution consists of several sequential phases, e.g.,

- Fetch
- Decode
- Execute
- Write Back

<table>
<thead>
<tr>
<th></th>
<th>Inst 1</th>
<th>Inst 2</th>
<th>Inst 3</th>
<th>Inst 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decode</td>
<td>Fetch</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Execute</td>
<td>Decode</td>
<td>Fetch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write Back</td>
<td>Execute</td>
<td>Decode</td>
<td>Fetch</td>
<td></td>
</tr>
<tr>
<td>Write Back</td>
<td></td>
<td>Execute</td>
<td>Decode</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Write Back</td>
<td></td>
<td>Execute</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Write Back</td>
<td>Write Back</td>
</tr>
</tbody>
</table>
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.

- Data Hazards: Operands not yet available (Data Dependences)
  - By applying dependence analysis
- Control Hazards: Conditional branch
  - By applying dependence analysis
- Resource Hazards: Consecutive instructions use same resource
  - By applying analysis on the resource reservation tables
- Instruction-Cache Hazards: Instruction fetch causes cache miss
  - By applying static cache analysis
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
- 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
- Alternatives: use a set of states for an instruction instead of one starting state. This is useful when there are multiple predecessors.
Summary of WCET Analysis

- 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