Critical Embedded Real-Time Systems

Systèmes Temps Réel Embarqués Critiques

STREC - WCET - Introduction

Florian Brandner
Télécom ParisTech

Outline

Sub-Module Outline

1. **Static Program Analysis**
   - Program Representation
   - Program Semantics
   - Data-Flow Analysis
2. Worst-Case Execution Time Analysis
3. Static cache analysis (single task)

Program Representation

Reason About Program Behavior

Goals:

- We would like to reason about the behavior of a program
- We would like to make definitive statements about a program

  Examples:

  - The code that is actually executed by the program
  - Global data/memory cells accessed by the program
  - Size of the stack used by the program
  - ...

Questions:

- What does a program actually do?
- What is the semantics of the program?
- How can a program be represented (in order to reason about it)?

TELECOM ParisTech

Example: A Simple Program

C Source Code

```
int count_str(char *x) {
  int c = 0;

  if (!x)
    return -1;

  while(*x) {
    if (*x != ' ')
      c++;

    x++;
  }

  return c;
}
```

MIPS Assembly

```
count_str:
        beqz    a0,38 exit
        nop
continue:
        lb      a1,0(a0)
        nop
        beqz    a1,30 loop-end
        move    v0,zero
loop-start:
        addiu   a0,a0,1
        xori    v1,a1,0x20
        lb      a1,0(a0)
        sltu    v1,zero,v1
        bnez    a1,18 loop-start
        addu    v0,v0,v1
loop-end:
        jr      ra
        nop
exit:
        jr      ra
        li      v0,-1
```

Compiler

From C source to assembly: (somewhat simplified)

- Textual representation of the program (C source code)
  $\Longrightarrow$ The compiler parses of the source code
- Data structure representing code (Abstract Syntax Tree)
  $\Longrightarrow$ The compiler translates the program to machine code
- Machine code representation (Control-Flow Graph)
  $\Longrightarrow$ The compiler generates the final executable

Compiler

From C source to assembly: (somewhat simplified)

- Textual representation of the program (C source code)
  $\Longrightarrow$ The compiler parses of the source code
- Data structure representing code (Abstract Syntax Tree)
  $\Longrightarrow$ The compiler translates the program to machine code
- Machine code representation (Control-Flow Graph)
  $\Longrightarrow$ The compiler generates the final executable

What is a **control-flow graph** (CFG)?

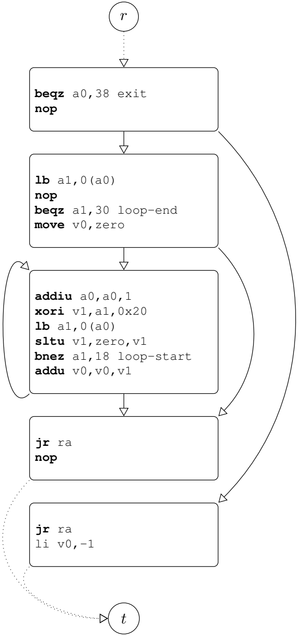
Control-Flow Graph

Data structure to represent code:

- Represented as a form of graph
- Graph nodes:
  - Individual instructions or
  - Sequences of instructions called **basic block**
- Graph edges:
  - Link from a graph node (instruction) to another
  - Instructions that might execute after executing an instruction
    (Basic blocks that might execute after executing a basic block)
- This allows to represent all possible executions of a program from start to end

TELECOM ParisTech

Example: Control-Flow Graph



Program Semantics

Control-flow graphs are merely a program representation:

- A CFG only indicates which instructions may succeed/proceed other instructions (or basic blocks)
- A CFG does not say anything about program semantics (What the program does?)
- The semantics depends on the instructions within the CFG

Program Semantics

Control-flow graphs are merely a program representation:

- A CFG only indicates which instructions may succeed/proceed other instructions (or basic blocks)
- A CFG does not say anything about program semantics (What the program does?)
- The semantics depends on the instructions within the CFG

We need something in addition to reason about programs . . .

Data-Flow Analysis

aka. Abstract Interpretation

Data-Flow Analysis

One technique to *reason* about programs:

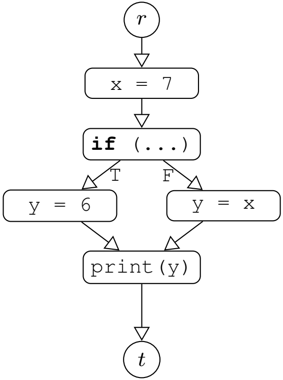
- This is often called **statically analysis**
- Model the flow of information through a program
- Based on a generic *framework*
  - Abstractions (aka. Domain)
  - Transformation functions (Domain $\rightarrow$ Domain)
  - Meet/join operator (Domain $\times$ Domain $\rightarrow$ Domain)
- Given an instance of a framework
  - Build and solve data-flow equations
  - Obtain over- or under-approximation of program behavior

Example: Constant Propagation

Determine whether a variable always has a constant value:

```
x = 7;
if (...)
  y = 6;
else
  y = x;
print(y);
```

(a) Program source

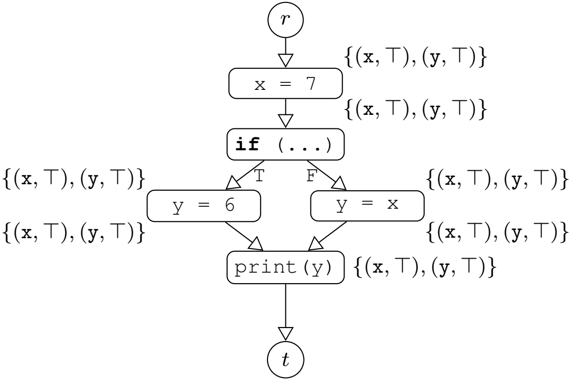


(b) Machine-level control-flow graph

Example: Constant Propagation

Associate each instruction with information on variable values:

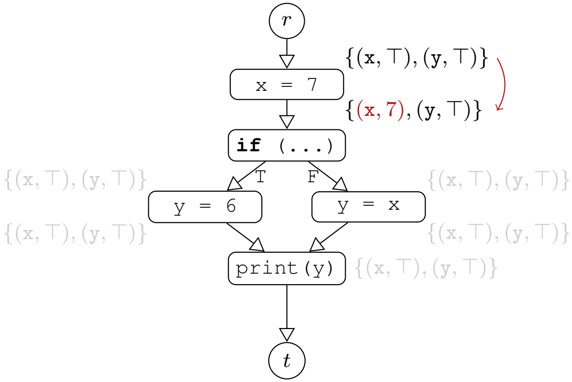
- Take information before instruction (Domain)
- Transform (check for constants)
- Propagate result to successors (forward analysis)



Example: Constant Propagation

Associate each instruction with information on variable values:

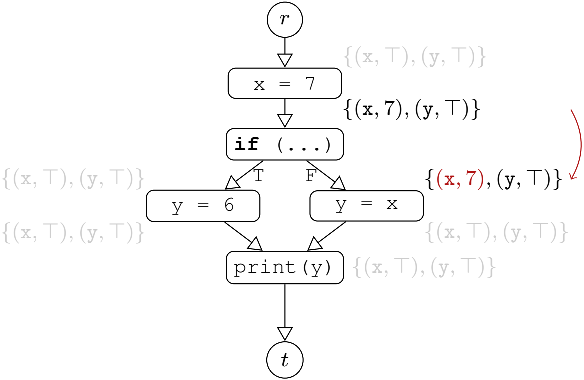
- Take information before instruction (Domain)
- Transform (check for constants)
- Propagate result to successors (forward analysis)



Example: Constant Propagation

Associate each instruction with information on variable values:

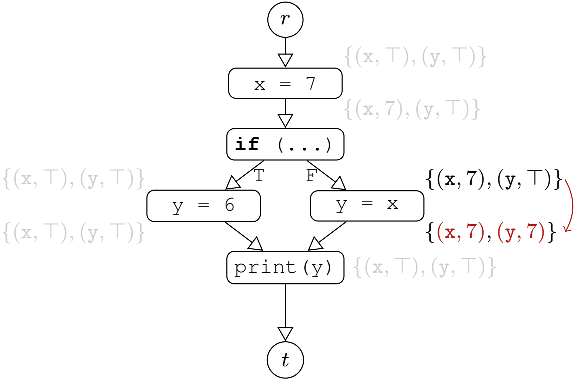
- Take information before instruction (Domain)
- Transform (check for constants)
- Propagate result to successors (forward analysis)



Example: Constant Propagation

Associate each instruction with information on variable values:

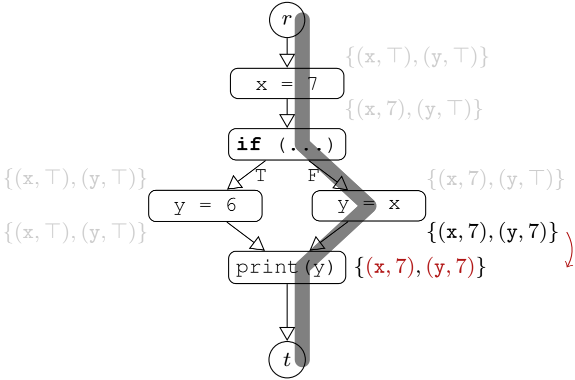
- Take information before instruction (Domain)
- Transform (check for constants)
- Propagate result to successors (forward analysis)



Example: Constant Propagation

Associate each instruction with information on variable values:

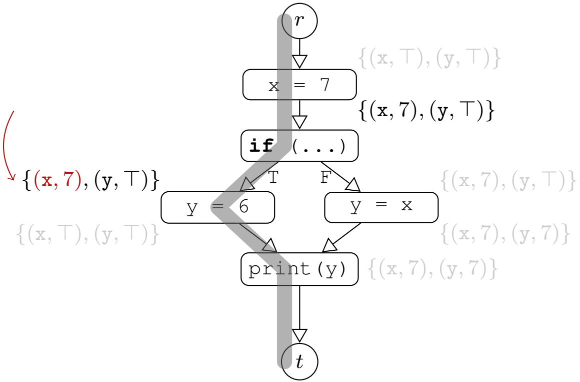
- Take information before instruction (Domain)
- Transform (check for constants)
- Propagate result to successors (forward analysis)



Example: Constant Propagation

Associate each instruction with information on variable values:

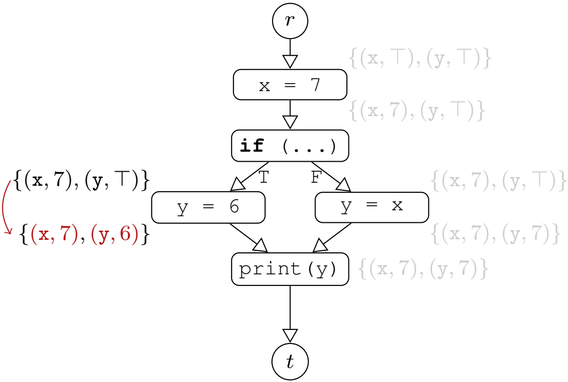
- Take information before instruction (Domain)
- Transform (check for constants)
- Propagate result to successors (forward analysis)



Example: Constant Propagation

Associate each instruction with information on variable values:

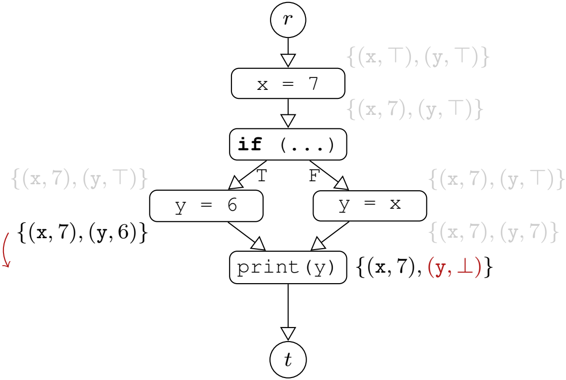
- Take information before instruction (Domain)
- Transform (check for constants)
- Propagate result to successors (forward analysis)



Example: Constant Propagation

Associate each instruction with information on variable values:

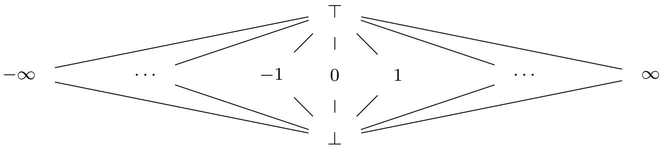
- Take information before instruction (Domain)
- Transform (check for constants)
- Propagate result to successors (forward analysis)



Abstract Domain

Represents information known about the program:

- Based on partial orders (lattices)
- Information is refined by descending the lattice
- Special elements:
    - $\top$ (Top):
      The top-most element in the lattice, representing that *no* information is yet available
    - $\perp$ (Bottom):
      The least element, representing contradicting information
- Example: constant propagation



Transfer Functions

Transform the information Domain $\rightarrow$ Domain

- Capture the effect of instructions on the analysis information
- Can be almost freely defined

- Example: constant propagation

$$t(i, I) = \begin{cases} I \setminus \{(v, \hat{d}) | (v, \hat{d}) \in I\} \cup \{(v, \hat{c})\} & \text{, if } i \text{ is } \mathtt{v} = \hat{c} \\ I \setminus \{(v, \hat{d}) | (v, \hat{d}) \in I\} \cup \{(v, \hat{c}) | (w, \hat{c}) \in I\} & \text{, if } i \text{ is } \mathtt{v} = \mathtt{w} \\ I \setminus \{(v, \hat{d}) | s(v, \hat{d}) \in I\} & \text{, if } i \text{ is } \mathtt{v} = \ldots \\ I & \text{, otherwise.} \end{cases}$$

Meet/Join Operation

Combine information at control-flow joins:

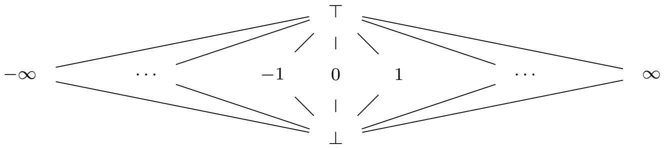
- Find least upper/greatest lower bound of two values
- Need to satisfy certain properties
  - Monotonicity ensures termination
  - Distributivity ensures optimal solution using iterative solving

- Notation:
  - $a \sqcap b$ (meet operator):
    smallest common ancestor of a and b
  - $a \sqcup b$ (join operator):
    greatest common descendent of a and b

Example: Join of Constant Propagation

The lattice for constant propagation is shown below:

- $1 \sqcup 2 = \bot$:
  The variable is either 1 or 2 depending on the predecessor. After a join we know that it is not constant, i.e., $\bot$.
- $\top \sqcup 2 = 2$:
  The variable is 2 at one predecessor. No information is available for the other predecessor. After a join the variable could still be constant, i.e., 2.



Static Analysis Contexts

Two problems:

- The behavior of an instruction might depend on call nesting
  $\Longrightarrow$ Possibly resulting in different information
- An instruction might be executed several times
  $\Longrightarrow$ Possibly resulting in different information

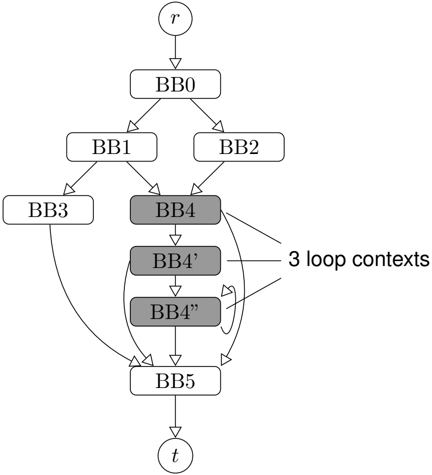
Static Analysis Contexts

Two problems:

- The behavior of an instruction might depend on call nesting
  $\Longrightarrow$ Possibly resulting in different information
- An instruction might be executed several times
  $\Longrightarrow$ Possibly resulting in different information

- Contexts:
  - Associate one or more contexts with each instruction
  - Allows to differentiate between diverging information

Example: Loop Contexts



- *Duplicate* basic blocks
- Each copy represents a set of loop iterations
  - BB4: Iteration 1
  - BB4': Iteration 2
  - BB4'': Iteration 3 – *n*
- Each copy might represent different information

Value Range Analysis

Value Range Analysis

Determine for each variable the range of possible values:

- Extension of constant propagation (from before)
- Find constant lower- and upper-bounds for each variable
- We will only consider a simplified analysis here

- What is done with it?
  - Needed for cache analysis (access addresses)
  - Used in loop bounds analysis (loop bounds)
  - Used to detect infeasible conditions (flow-facts)

Value Range Analysis in a Nutshell

Domain:

- Set of triples over all program variables
- Variable $\times \mathbb{N} \times \mathbb{N}$

Transfer functions:

- Perform arithmetic on value ranges (interval arithmetic)
- Example: Addition
  $[a, b] + [c, d] = [a + c, b + d]$

Join operator:

- $[a, b] \sqcup [c, d] = [\min(a, c), \max(b, d)]$

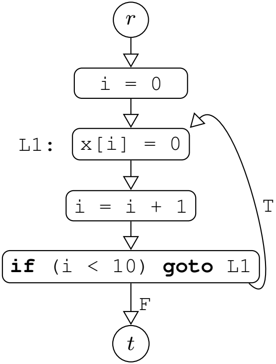
Group Exercise: Range Analysis

Determine the range of memory addresses accessed by `x[i]`:

- Assume that `x` is a global variable at address `0x100`
- Each element of `x` is 4 bytes large
- What are the initial states of the analysis?
- Which role plays the condition `if (i <10)`?

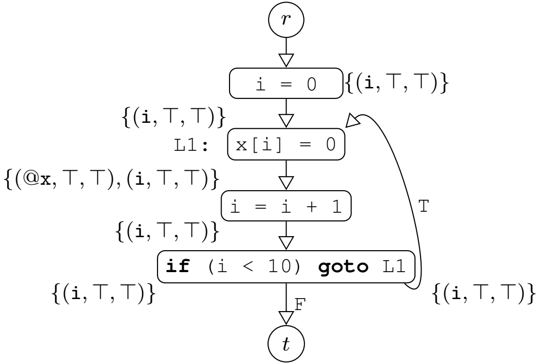
```
for(i = 0; i < 10; i++)
  x[i] = 0;
```

(a) Program source

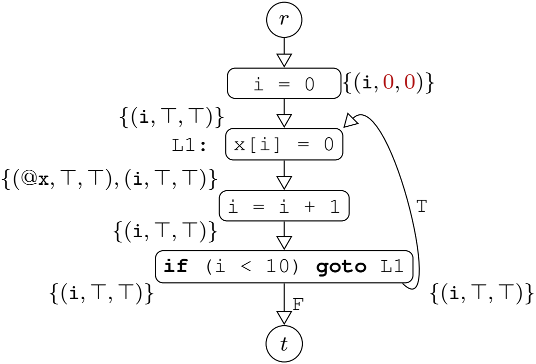


(b) Machine-level control-flow graph

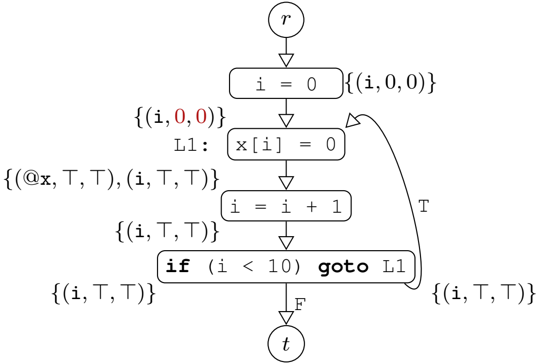
Example: Range Analysis



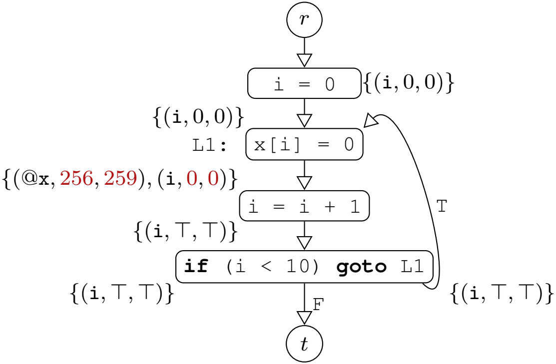
Example: Range Analysis



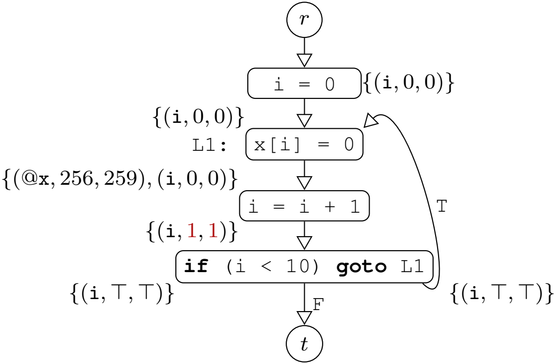
Example: Range Analysis



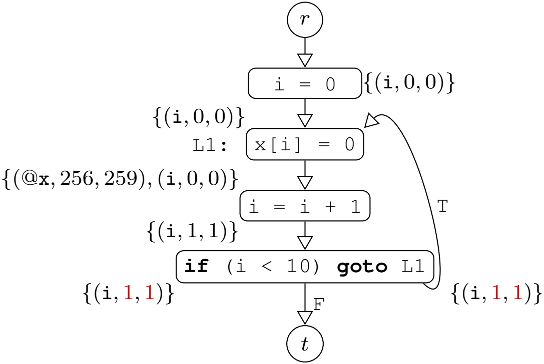
Example: Range Analysis



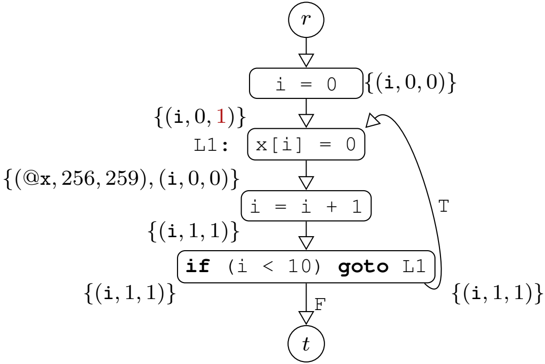
Example: Range Analysis



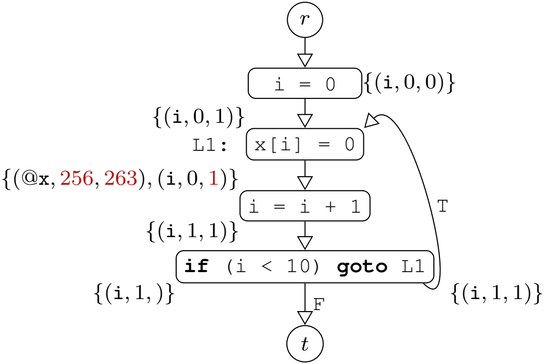
Example: Range Analysis



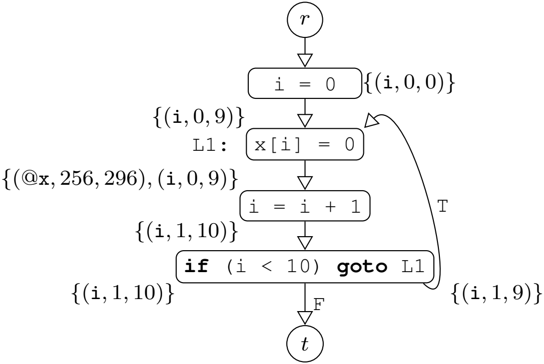
Example: Range Analysis



Example: Range Analysis



Example: Range Analysis



Outline

Sub-Module Outline

1. Static Program Analysis
2. **Worst-Case Execution Time Analysis**
   - Definitions
   - Static analysis vs. measurements
   - Implicit Path Enumeration
3. Static cache analysis (single task)

Worst-Case Execution Time

Worst-Case Execution Time

Real-time systems:

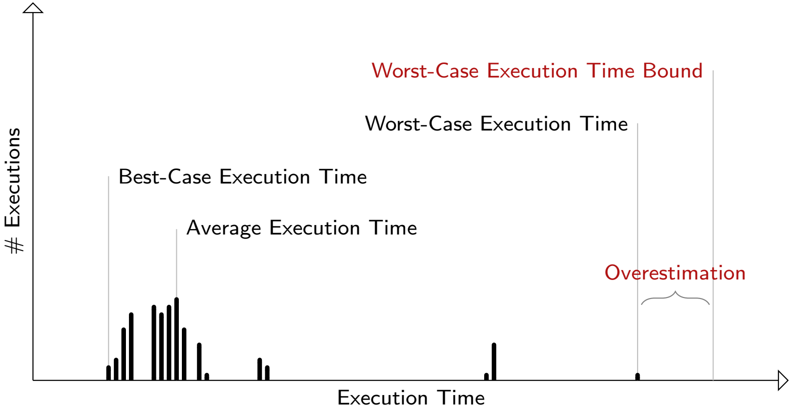
- So far in this course:
  - Scheduling of real-time tasks
  - Each task τ_i has a *Worst-Case Execution Time* C_i (WCET)
  - Each task τ_i has a deadlines (D_i)
  - Can we schedule the whole system?
- Next few sessions:
  - How can we define the WCET ?
  - How can we determine the WCET (C_i)?
  - How long does it take to finish a computation?

    $\Longrightarrow$ We need to analyze (*reason* about) the program!

TELECOM ParisTech

Worst-Case Execution Time (2)

Some definitions related to timing analysis:



Assume we could observe **all** possible inputs/executions.

Worst-Case Execution Time Bound

Actually, we search for a WCET bound

- Safety:
  A bound is safe when it is *larger* than any observable actual WCET
  $\Longrightarrow$ How can we ensure that the obtained bound is safe?

- Overestimation:
  Imprecision in the analysis lead to overestimation
  $\Longrightarrow$ How can we ensure that the bound is tight?

- From now on: WCET denotes the WCET bound

  WCET . . . WCET bound   actual WCET . . . WCET

Factors Impacting the WCET

Factors that may impact the WCET:

- The program source (algorithm)
- The program input (data)
- The compiler (generating machine-level code)
- The hardware platform
  - Processor pipeline
  - Computational units
  - Branch prediction
  - Caches
  - Buffers
  - Main memory
  - Bus arbitration
  - ...
- Other tasks in the system (preemption, competition)

WCET Challenges

What so difficult with that?

- What is the program doing?
    - Or: which instructions are executed?
    - Depends on algorithms/programing languages/ compilers/…
    - Often also dependent on program inputs

- What are the possible inputs?
    - Usually too many options to explore them all

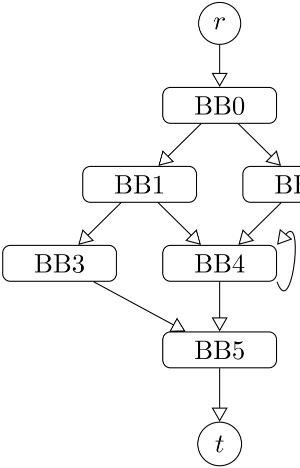
- How long do the instructions take?
    - Highly dependent on hardware design

WCET Analysis Approaches

Three main approaches:

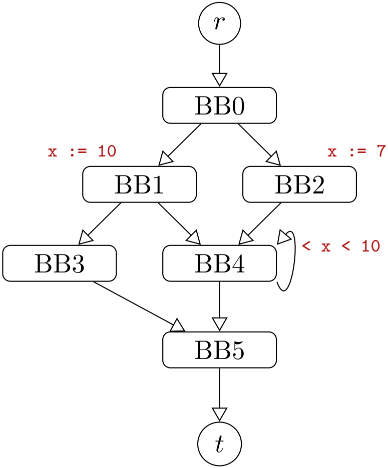
- Measurements: (no guarantee)
  - Simply run the program many times (testing)
  - Covering *all* classes of inputs
  - Covering *all* execution paths
  - Take maximum (multiplied by x)
- Probabilistic Analysis: (requires preconditions)
  - Take measurements (as above)
  - Fit a probabilistic distribution
  - Select WCET subject to a threshold using the distribution
- Static Program Analysis: (generally safe)
  - Analyze code by *abstractions*, e.g., data-flow analysis
  - Extract and annotate information from/to code
  - Safe WCET when abstractions are safe

Example: Static WCET Analysis



Three analysis phases:

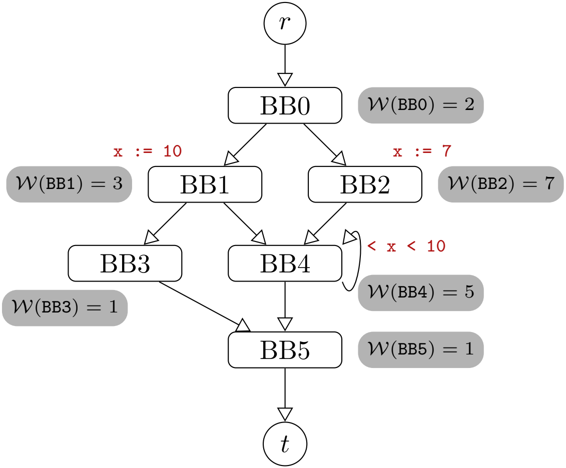
Example: Static WCET Analysis



Three analysis phases:

(1) Loop bounds & flow facts

Example: Static WCET Analysis

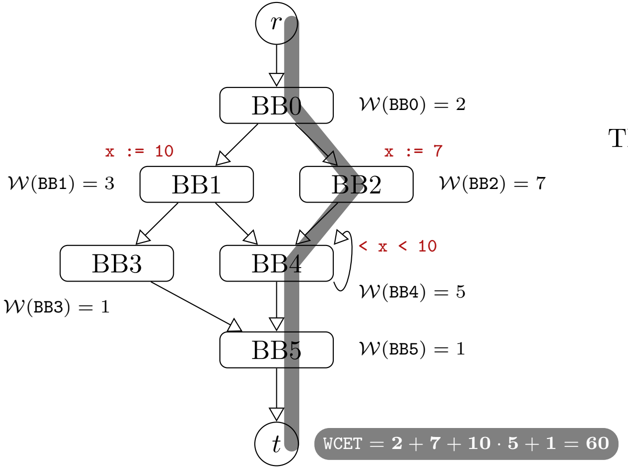


Three analysis phases:

(1) Loop bounds & flow facts

(2) Pipeline & caches

Example: Static WCET Analysis



Three analysis phases:

(1) Loop bounds & flow facts

(2) Pipeline & caches

(3) Longest path search (IPET)

What's next?

- Today:
  - Loop bounds and flow-facts analysis (Step 1)
  - Pipeline analysis (Step 2)
  - Implicit path enumeration (Step 3)
- Next session:
  - Analyzing data/instruction caches (Step 2)

Loop Bounds and Flow Facts

Flow Facts

Information on infeasible program executions:

- Loop bounds:
  The number of iterations of a loop can not exceed a given constant k.

- Recursion bounds:
  May refer to recursion depth (depth of call tree) or number of total recursive calls (number of nodes in the call tree).

- Mutual exclusion:
  Two branch conditions a and b are mutually exclusive, i.e., $a \Rightarrow \neg b$.

- Generic flow facts:
  Relate the execution frequencies of two program points to each other.

Simple Loop Bounds

Trivial analysis for counting loops:

- Easily recognizable patterns (covers most loops)
- Simply take results from range analysis

- Example:

```
for (int i = 0; i < n; i++) {
  ...
}
```

Complex Loop Bounds

Beyond the scope of this course:

- Two major sources of complexity:
  - Complex conditions
  - Nested loops where inner bounds depend on outer loops
- Great challenge for analysis (manual annotations)
  - Former case is equivalent to the halting problem (NP-hard)
  - The later case is well understood
  - Loops in real-time software are typically *well-behaved*

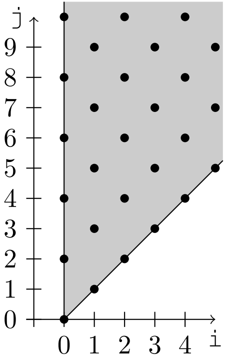
Example: Complex Loops Bounds

Construct linear equations describing *iteration space*

- Equations specify a (parametric) polytope
- Count the number of *integer points* within the polytope

```
for(int i = 0; i < n; i++)
{
  for(int j = i; j < 2*n; j+2)
  {
    ...
  }
}
```

(a) Program code



(b) Corresponding polytope

Pipeline Analysis

Pipeline Analysis

Compute potential states of the processor pipeline:

- Hardware utilization captured using *state machines*
- Abstract interpretation:
  - *Brute force* enumeration of all possible states
  - Sets of pipeline states (Domaine)
  - Compute all potential successor states (Transfer functions)
  - Take union of all states on joins (Meet)
  - Abstractions are difficult due to dynamic pipeline behavior

    $\Longrightarrow$ Interaction with caches, branch prediction, ...

    $\Longrightarrow$ Predictable processors have been proposed[1]

[1] http://patmos.compute.dtu.dk/

TELECOM ParisTech

Instruction Timing

How do we obtain the instruction timing?

- Consider all states involving a given instruction
  - From the first attempt to fetch the instruction . . .
  - To its completion in the pipeline
- Problem:
  - Execution of instructions may overlap
  - Same time instant is *counted several times*
- Solution:
  - Consider basic blocks (sequences of instructions) at once
  - Consider states *in the middle* of control-flow edges
  - Find longest sequence from incoming to outgoing edge
    (longest path search on an acyclic graph)

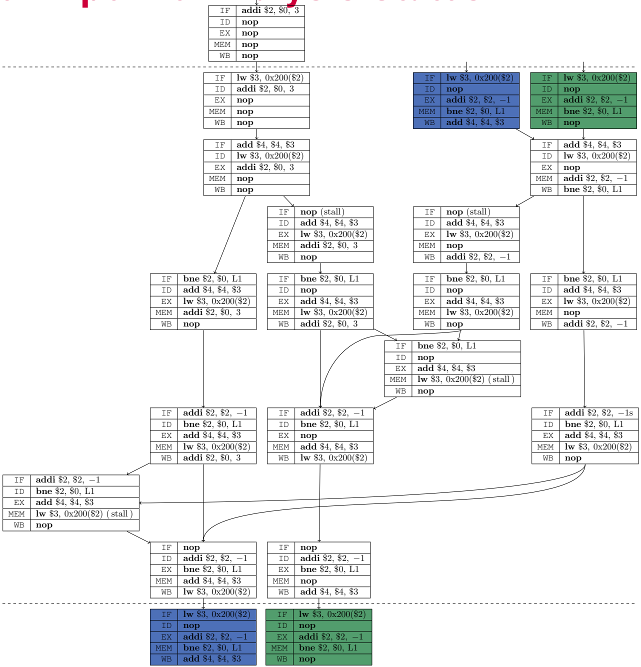
Example: Pipeline Analysis

Assume a pipelined MIPS processor

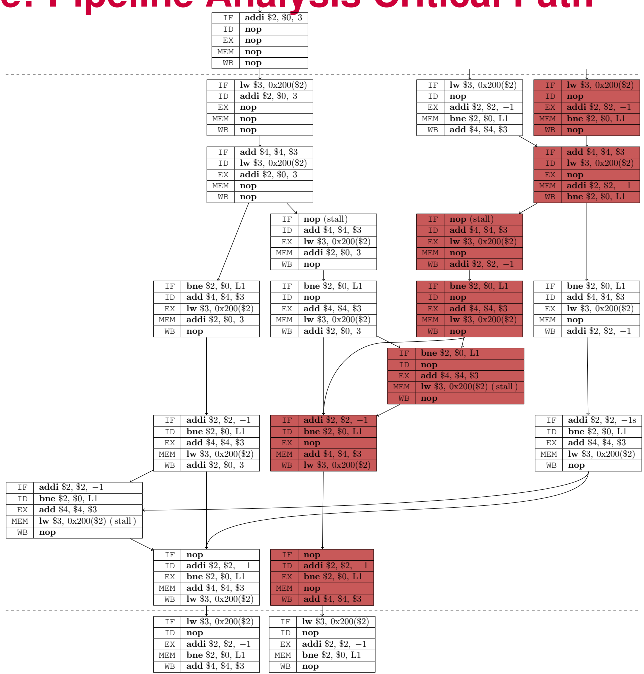
- With 5-stages (IF, ID, EX, MEM, WB)
- Branches execute in EX (2 branch delay slots)
- Instruction and data caches with 16 byte blocks
- IF/MEM are stalled on cache misses for a cycle
- We consider all possible cache states

```
0x14   addi $2, $0, 3
     L1:
0x18   lw $3, 0x200($2)
0x1C   add $4, $4, $3
0x20   bne $2, $0, L1
0x24   addi $2, $2, -1
0x2C   nop
```

Example: Pipeline Analysis States



Example: Pipeline Analysis Critical Path



Limitations

Which cases are covered by the analysis?

- Contiguous execution of the program
  - No interrupts (perturbation of pipeline state)
  - No preemption (requires interrupts)
  - No faults (electric glitches)
  - No operating system calls (often excluded from analysis)
  - No interference in multi-core architectures
- Software correctness
  - Analysis considers all cases right or wrong
  - But does not distinguish between them
  - That is somebody else's problem

Implicit Path Enumeration Technique
(aka. IPET)

Bounding the WCET

What have we got so far?

- Analysis of program semantics: (Step 1)
  - Range analysis of program variables
  - Analysis of loop bounds
  - Analysis of generic flow constraints
- Analysis of hardware behavior: (Step 2)
  - Analysis of pipeline states
  - Missing: Caches and branch predictors

Bounding the WCET

What is left to do?

- Actually bounding the WCET

- Problem statement:
  - Find longest execution from program start to its termination
    - Variants: find longest execution of a loop/function/...
  - Equivalent to the **longest paths** in the control-flow graph
    - Nodes of the graph represent basic blocks
    - Edge weights represent basic block execution times (cf. pipeline analysis)

Longest Paths in Directed Acyclic Graphs

Apply dynamic programming to weighted DAG $G = (V, E, \mathcal{W})$:

1. Compute a *topological order*
2. Visit each node n according to the topological order
   Compute:

$$\text{dist}(n) = \max_{(m,n) \in E} \mathit{dist}(m) + \mathcal{W}(m, n)$$

Simple algorithm in linear time $O(|V| + |E|)$.

Limitations

Dynamic programming can not cope with:

- Cyclic graphs (loops)
- Flow facts (infeasible paths)

Realistic programs cannot be handled.

Implict Path Enumeration Technique (IPET)

Build linear equations modeling execution flow:

- Control-flow edges are represented by flow variables
- Flow variables indicate the number of times code executes
- Build a huge linear equation system
  - Solved using standard software (e.g., CPLEX, Gurobi, lpsolve)
  - Maximize execution flows according to edge weights

- Kirchhoff's law:
  The sum of the **flow entering** a control-flow node has to **match** the **flow leaving** the node.

IPET Base Equations

Given a weighted control-flow graph $G = (V, E, \mathcal{W})$ and a mapping of edges to flow variables $\mathcal{F}$:

- Flow for program entry r:

$$\sum_{(r,n)\in E} \mathcal{F}(r, n) = 1$$

- Flow for program exit t:

$$\sum_{(n,t)\in E} \mathcal{F}(n, t) = 1$$

- Flow equations of node $n \in V$:

$$\forall n \in V: \sum_{(k,n)\in E} \mathcal{F}(k, n) = \sum_{(n,m)\in E} \mathcal{F}(n, m)$$

- **Maximizing**:

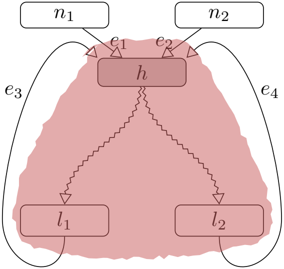
$$max. \sum_{(m,n)\in E} \mathcal{F}(m, n) \cdot \mathcal{W}(m, n)$$

Loop Bounds in IPET

Given a reducible loop L with bound $\hat{b}$ and loop header h:

$$\sum_{(n,h)\in E} \mathcal{F}(n,h) \leq \hat{b} \cdot \sum_{(n,h)\notin L} \mathcal{F}(n,h)$$

Example:

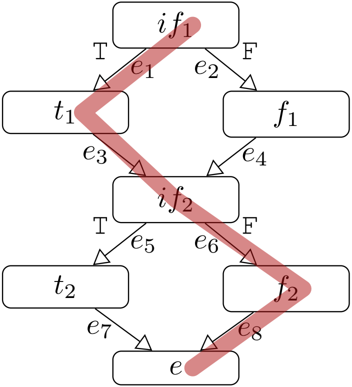


- Loop: $L = \{h, \ldots, l_1, l_2\}$ (red)
- Header: h (darker node)
- Pre-entries: $n_1, n_2 \notin L$
- Equations:

  $e_1 + e_2 + e_3 + e_4 \leq \hat{b} \cdot (e_1 + e2)$

Group Exercise: Infeasible Paths in IPET

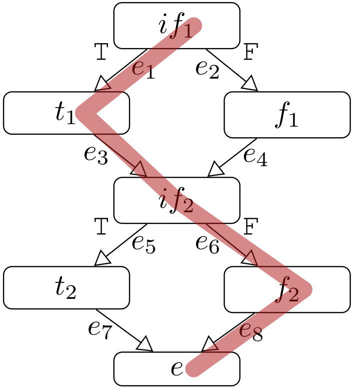
Determine the equations to exclude the highlighted path:



- Assume that the in-flow of *if*$_1$ might be larger than 1
- Hint:
  Think about the flows related to node *if*$_2$

Group Exercise: Infeasible Paths in IPET

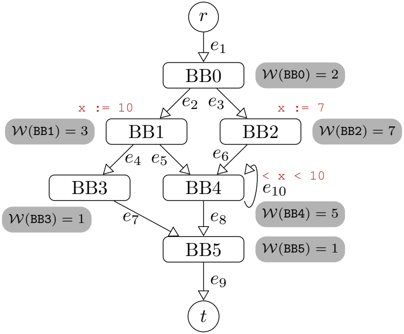
Determine the equations to exclude the highlighted path:



- Assume that the in-flow of if_1 might be larger than 1
- <u>Hint:</u>
  Think about the flows related to node if_2
- <u>Solution:</u>

$$e_6 \leq e_4$$

Example: IPET

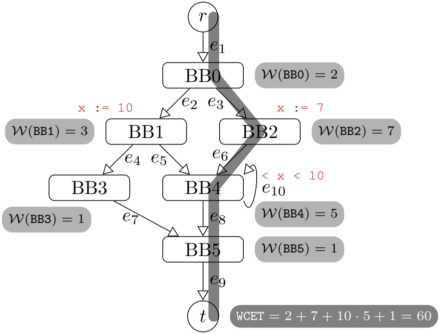


$$e_1 = 1$$
$$e_1 = e_2 + e_3$$
$$e_2 = e_4 + e_5$$
$$e_3 = e_6$$
$$e_4 = e_7$$
$$e_5 + e_6 + e_{10} = e_8 + e_{10}$$
$$e_7 + e_8 = e_9$$
$$e_9 = 1$$

$$e_5 + e_6 + e_{10} \leq 10 \cdot (e_5 + e_6)$$

Maximize : $2e_2 + 2e_3 + 3e_4 + 3e_5 +$
$7e_6 + e_7 + 5e_8 + e_9 + 5e_{10}$

Example: IPET (2)



$$1 = 1$$
$$1 = 0 + 1$$
$$0 = 0 + 0$$
$$1 = 1$$
$$0 = 0$$
$$0 + 1 + 9 = 1 + 9$$
$$0 + 1 = 1$$
$$1 = 1$$

$$0 + 1 + 9 \leq 10 \cdot (0 + 1)$$

Maximize : $2 \cdot 0 + 2 \cdot 1 + 3 \cdot 0 + 3 \cdot 0 +$
$7 \cdot 1 + 0 + 5 \cdot 1 + 1 + 5 \cdot 9$

Summary

- Worst-case execution time
  - Bounds vs. actual WCET
  - Overestimation
- Obtaining WCET estimations
  - Static program analysis (guaranteed safe)
  - Measurements (safety not guaranteed)
  - Probabilistic analysis (some prerequisites)
- Static WCET analysis
  - Based on data-flow analysis/abstract interpretation
  - Value range analysis (software behavior)
  - Pipeline analysis (hardware behavior)
  - Implicit path enumeration (compute WCET)