Topic 1 – Flow Analysis

Björn Lisper
School of Innovation, Design, and Engineering
Mälardalen University

bjorn.lisper@mdh.se

2014-06-28

Definition of WCET

Definition of WCET for program P:

$$WCET(P) = \max_{s,p} T(s,p)$$

where $T(s,p)$ is execution time for path p starting in state s

Max over all possible paths. There can be very (exponentially) many paths. How deal with this in WCET analysis?

Answer: make some controlled approximations to obtain a cost model where the order of execution of program fragments is irrelevant. This will allow the use of standard optimization techniques to bound the WCET

On the next slides we'll see how this is done . . .

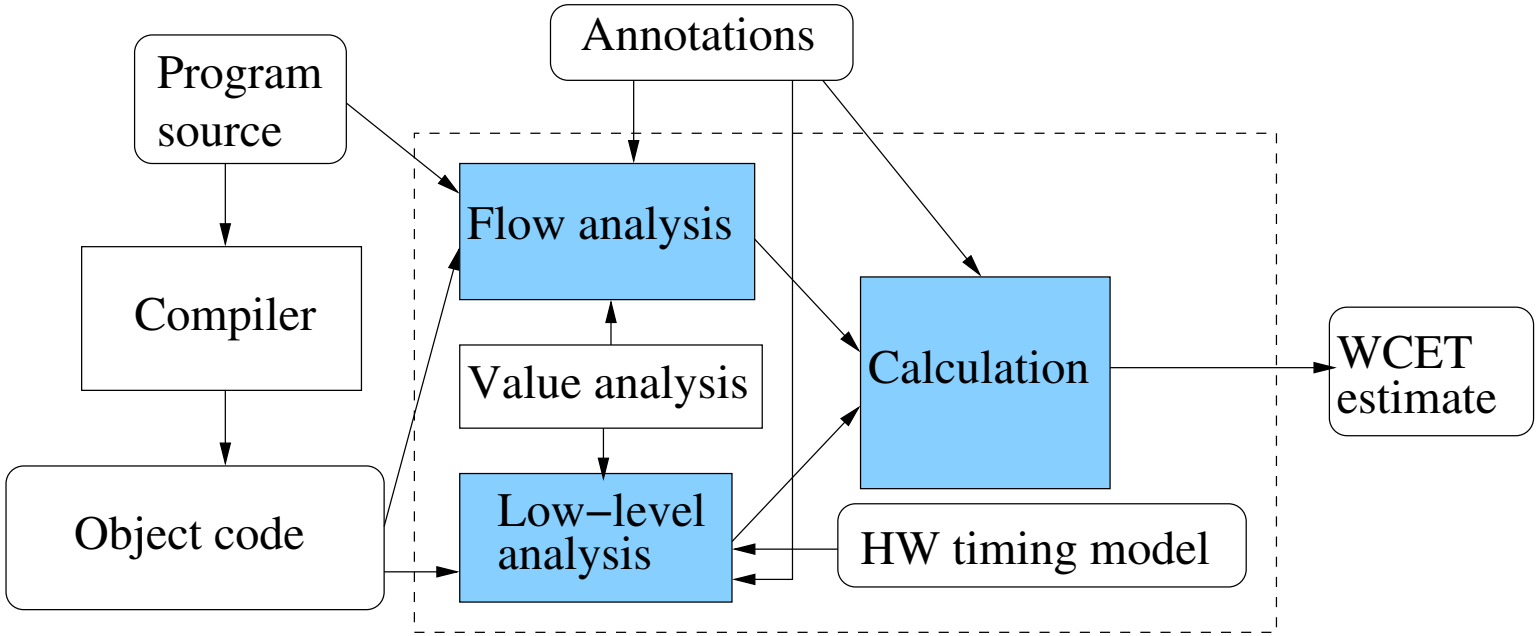
Standard Scheme for WCET Analysis

WCET analysis is typically broken down into the following steps:

- Constrain possible program flows (“high-level”, or “flow” analysis)
- Estimate hardware impact to bound WCET for program fragments (“low-level” analysis)
- Use information to produce a safe WCET estimate (calculation)

Controlled approximations, trade some precision to obtain a feasible analysis

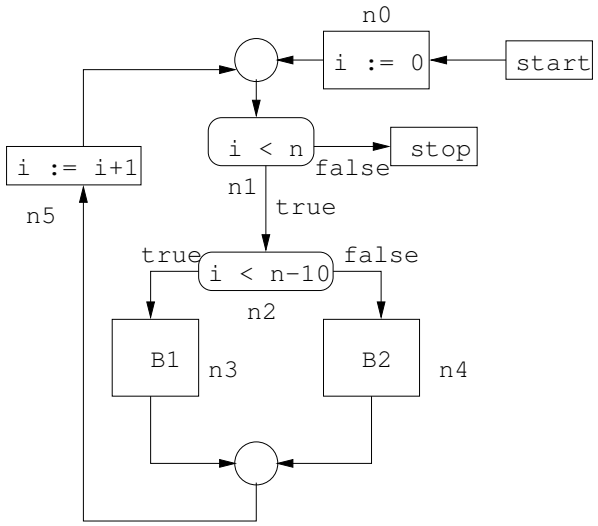
Structure of WCET Analysis



The Implicit Path Enumeration Technique (IPET)

A constraint-based approach

Based on control-flow graph (CFG) program representation:



For each node n_i (basic block) in the CFG, obtain an **upper bound** t_i for its **local WCET** by low-level analysis

Define *execution counter* x_i for each n_i:

- Initialised to 0
- Incremented by 1 each time n_i is executed
- A “virtual variable”, not present in the code

For any path p, $execution\_time(p) \leq \sum x_i t_i$ where the x_i’s are the final values of the execution counters for p

Thus, $\max \sum x_i t_i$ over all paths will provide a safe WCET bound

WCET Bounds Estimation in the IPET Model

Assume that we can express program flow constraints as *arithmetic constraints on the execution counters* x_i

WCET estimation (calculation) becomes a maximization problem:

$$\max \sum x_i t_i$$

subject to flow constraints

If flow constraints are **linear**, then this becomes an *Integer Linear Programming* (ILP) problem

Can be solved by standard methods, using some off-the-shelf tool!

Linear Flow Constraints ("Flow Facts")

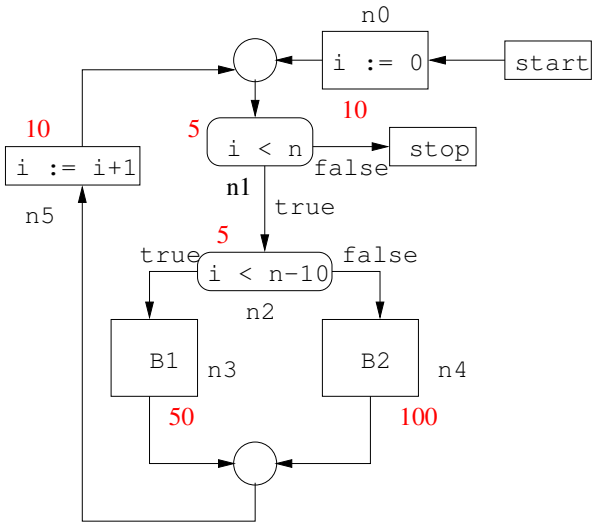
Many flow constraints can indeed be expressed as linear constraints on execution counters:

- $x_i \geq 0$ for all n_i (counters must be non-negative)
- **Structural flow constraints** (from CFG structure):
  - For each node $\neq$ start, stop, $\sum$ input counters $= \sum$ output counters
  - $x_i = 1$ if n_i is the start or stop node of the CFG
- **Semantic constraints**:
  - Loop iteration bounds (capacity bounds): $x_i \leq c$
  - Infeasible paths (mutual exclusivity): $x_i + x_j \leq c$
  - Relative bounds (often in nested loops): $x_i = c \cdot x_j$

Example

With structural constraints only (counters $x_0, \ldots, x_5$, assume execution times $t_0 = t_5 = 10$, $t_1 = t_2 = 5$, $t_3 = 50$, $t_4 = 100$):

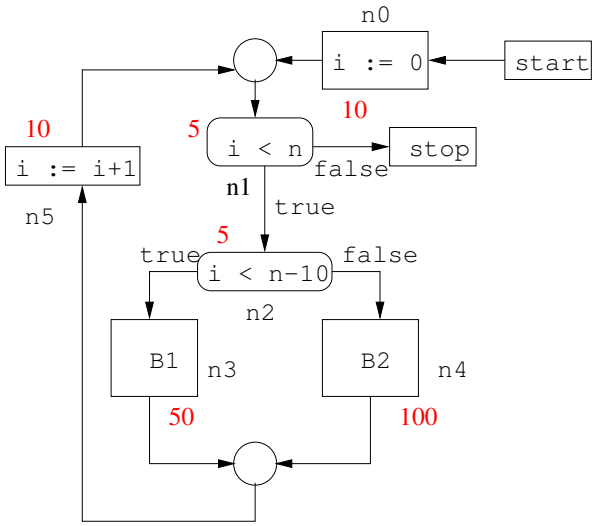
$$
\begin{aligned}
x_i &\geq 0, \quad i = 0, \ldots, 5 \\
x_0 &= 1 \\
x_1 &= x_0 + x_5 \\
x_2 &= x_1 - 1 \\
x_2 &= x_3 + x_4 \\
x_5 &= x_3 + x_4
\end{aligned}
$$



Unbounded problem, no solution

Say we know $n = 20$. Add loop bound constraint $x_1 \leq 21$:

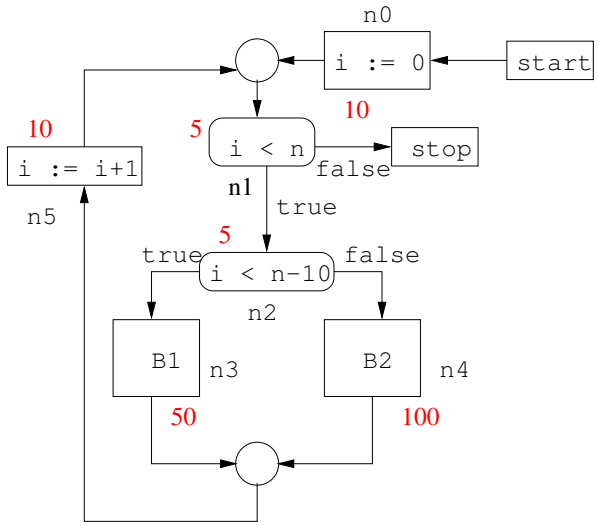
$$
\begin{aligned}
x_i &\geq 0, \quad i = 0, \ldots, 5 \\
x_0 &= 1 \\
x_1 &= x_0 + x_5 \\
x_2 &= x_1 - 1 \\
x_2 &= x_3 + x_4 \\
x_5 &= x_3 + x_4 \\
x_1 &\leq 21
\end{aligned}
$$



Solution $x_0 = 1$, $x_1 = 21$, $x_2 = 20$, $x_3 = 0$, $x_4 = 20$, $x_5 = 20$, $t = 2415$

Second conditional gives new constraints $x_3 \leq 10$, $x_4 \leq 10$:

$$
\begin{aligned}
x_i &\geq 0, \quad i = 0, \ldots, 5 \\
x_0 &= 1 \\
x_1 &= x_0 + x_5 \\
x_2 &= x_1 - 1 \\
x_2 &= x_3 + x_4 \\
x_5 &= x_3 + x_4 \\
x_1 &\leq 21 \\
x_3 &\leq 10 \\
x_4 &\leq 10
\end{aligned}
$$



New solution $x_0 = 1$, $x_1 = 21$, $x_2 = 20$, $x_3 = 10$, $x_4 = 10$, $x_5 = 20$, $t = 1915$

The example demonstrates an interesting fact

Upper loop iteration bounds are necessary and sufficient to **bound** the WCET

However, other flow constraints (infeasible paths etc) can often yield **tighter** bounds

We have seen an improvement of up to 30% for industrial code

Not negligible! WCET bounds must often be tight to be useful

Refinements of the IPET model

There are a number of refinements to the basic IPET model that aim to increase the precision:

- Pipeline overlap costs on CFG edges
- Context-dependent local WCET bounds for basic blocks
- Local execution contexts

Pipelines

Pipelined execution starts execution of new instructions before old ones are completed:

Can give overlap effects between execution of program parts:

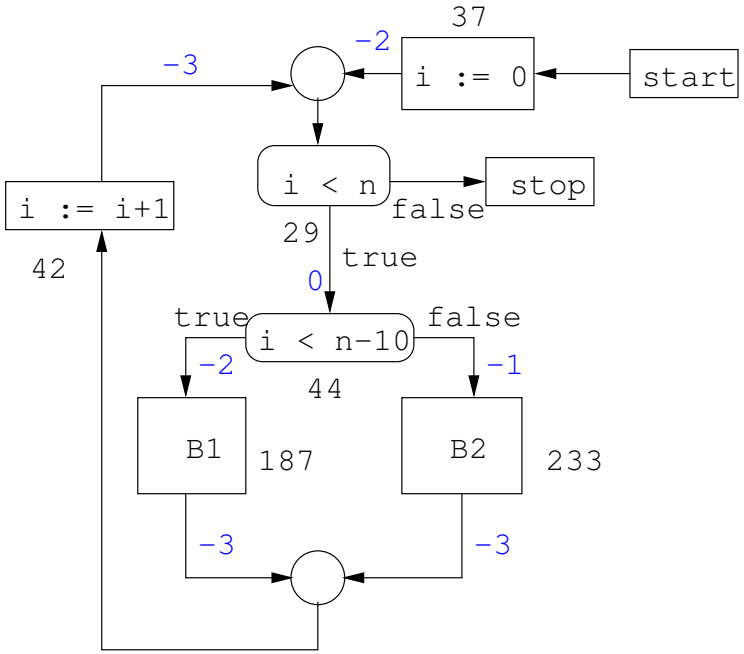
$$WCET(c_1; c_2) < WCET(c_1) + WCET(c_2)$$

Overlap beween two basic blocks can be modeled by a negative cost on their connecting edge

IPET execution counters must be added for these edges

$\Longrightarrow$ larger ILP problem to solve, but higher precision

Example: Pipeline Overlap Costs



Context-Dependent Low Level Analysis

A basic block can have different WCET's in different execution contexts

Typical case: for a loop body, the first iteration can be considered separately from the other ones

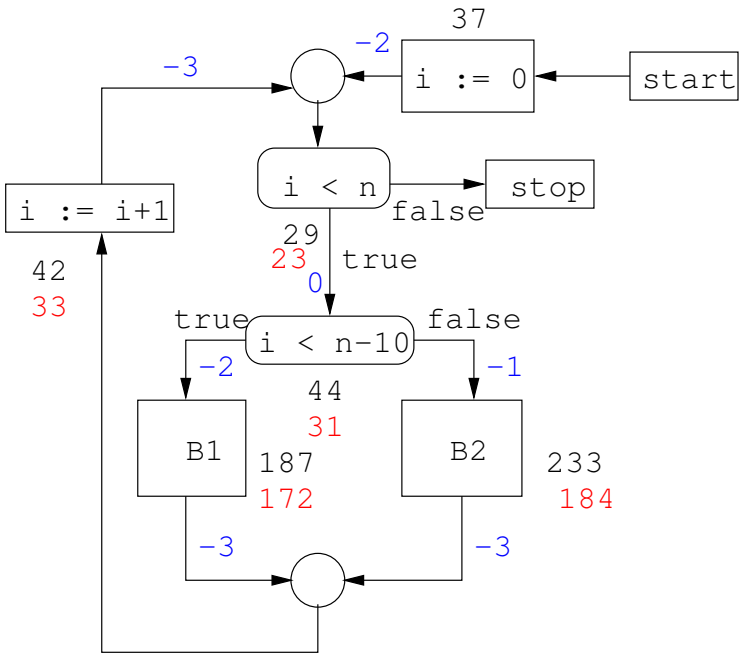
Chances are that the other iterations will run faster (cache reuse, branch predictor effects, . . .)

A *context-dependent* analysis, differing between the first iteration and the others, can take this into account

Each new context requires a new IPET execution counter

$\Longrightarrow$ larger ILP problem to solve, but higher precision

Example: Context-Dependent Local WCET's



Black: first loop iteration, red: following iterations

Local Execution Contexts

The basic IPET model uses *global* execution counters

Sometimes it is easier to find constraints on execution counters that are defined relative to a *local execution context*. Such counters are *reset at each entry into the context*

Example 1:

```
for i = 1 to 100 do
  for j = 1 to 50 do B;
```

Global execution counter $\#B$ for `B`: $\#B \leq 50 \cdot 100 = 5000$

Local execution counter $\#B_{local}$ relative to the inner loop: $\#B_{local} \leq 50$, can be found by analyzing the inner loop in isolation

Local counters must be converted into global before IPET calculation

Flow Analysis

Purpose: to automatically discover constraints on program flow

Can be done both on source- and object-code level

Flow analysis cannot be completely automated (halting problem).
User-assistance often needed

WCET analysis tools provide *annotation languages* to manually specify flow constraints

This is however laborious and error-prone (risk of underestimations)!

Goal: devise automatic methods that give good enough results for a large enough share of the time-critical software out there!

Classes of Flow Analysis Methods

Syntactically oriented methods:

Typically for loop iteration bounds. Matches loop structure to common patterns, identifying start, stop, step

Both on source level (common syntactic programming patterns), or binary level (typical loop patterns generated by some specific compiler)

Pros: simple to implement, fast

Cons: sensitive to syntax. Will fail in many cases

Classes of Flow Analysis Methods (II)

Semantically oriented methods:

Loop iteration bounds as well as other flow constraints

Analysis based on the semantics of the program rather than syntax

Methods often based on data flow analysis, or abstract interpretation (also model checking, symbolic execution). Typically some kind of fixed-point iteration is used

Pros: Less sensitive to syntax. Can be made more precise. More general classes of flow constraints. Possible to play around with tradeoff between precision and analysis speed

Cons: can be resource-consuming

Approach 1: Identify Loop Counter Parameters

Idea: find start value, step, value at exit for loop counters

Once known, an upper bound on the # of iterations can be calculated

Several approaches have been proposed, based on techniques like:

- data flow analysis
- solving Presburger formulas
- abstract interpretation

Pros: methods can be made simple and fast. Many loops have easily identifiable loop counters, and can be handled

Cons: restricted to loops with loop counters

Identifying Loop Counter Parameters: Example

A method based on abstract interpretation (value analysis). Originates from U. Saarland

Steps (for a given loop):

1. Identify loop counter
2. Bound the initial value of the loop counter
3. Find bounds for the loop counter increment (or decrement)
4. Analyze loop exits w.r.t. usage of the loop counter, deriving upper (or lower) bounds for the counter
5. Calculate upper bounds on # of iterations

Bounds on values are found by a value analysis (typically interval analysis), based on abstract interpretation

Example

```
/* i in interval [1..4] at entry */
s = 0;
n = 100;
while i < n do
  s = s + a[i];
  if complex_condition then i = i + 2 else i = i + 3;
```

Potential loop counters: `i`, `s` (updated in loop). However only `i` can affect the exit condition, thus identified as loop counter

Initial bound for `i`: `[1..4]` (obtained by value analysis)

Bound on increment of `i`: `[2..3]` (obtained by value analysis)

Upper bound on `i`: `[99..99]` (value analysis of exit condition)

Loop iteration bound: $(99 - 1)/2 = 49$

Approach 2: Abstract Execution

Originates from MDH (Gustafsson 2000), (Gustafsson et. al. 2006)

Find constraints on execution counters through a kind of symbolic execution of the program – “abstract execution” (AE)

AE executes the program with *abstract states*, representing sets of real (“concrete”) states

It can be seen as a a kind of value analysis (abstract interpretation)

Each abstract state transition corresponds to many possible concrete state transitions

We currently use *intervals* to represent sets of (numerical) values

Abstract states are then mappings from program variables to intervals

Concrete vs. Abstract States

Some concrete states for a program with (integer) variables $\mathtt{x}$ and $\mathtt{y}$:

$\mathtt{x}$	17
$\mathtt{y}$	3

$\mathtt{x}$	0
$\mathtt{y}$	-7

$\mathtt{x}$	4711
$\mathtt{y}$	1

$\mathtt{x}$	32767
$\mathtt{y}$	-32768

Some abstract states with intervals for a program with variables $\mathtt{x}$ and $\mathtt{y}$:

$\mathtt{x}$	$[0, 1]$
$\mathtt{y}$	$[-5, 5]$

$\mathtt{x}$	$[1, 1]$
$\mathtt{y}$	$[0, 32767]$

$\mathtt{x}$	$[-\infty, \infty]$
$\mathtt{y}$	$[5, \infty]$

$\mathtt{x}$	$\emptyset$
$\mathtt{y}$	$\emptyset$

Abstract Execution (II)

AE can be seen as a very context-sensitive value analysis (differing between all loop iterations)

Contrary to a conventional value analysis, it does not compute an abstract state for each program point: it's more similar to a real execution with a concrete state

Uses a scheme to successively collect constraints on execution counters during AE of loops

Abstract states can be *split* at conditions. This can result in many states

To counter this, abstract states can be *merged*. This can be used to control the number of abstract states. *Merge points* can be placed in the CFG. Merging can affect the precision, though

Abstract Execution (III)

Pros: can be precise. Can also compute more general constraints, like for infeasible paths. Can in principle use any abstract domain, not à priori restricted to loops with numerical loop counters.

Cons: can be time-consuming, potentially poor scalability. Risk of nontermination (can be controlled, though)

Example

```
i = INPUT; // i = [1..4]
#p = 0;
while (i < 10) {
    // point p
    #p = #p + 1;
    ...
    i=i+2;
}
// point q
```

(a) Code example

p	i at p	i at q
1	[1..4]	impossible
2	[3..6]	impossible
3	[5..8]	impossible
4	[7..9]	[10..10]
5	[9..9]	[10..11]
6	impossible	[10..11]

(b) Analysis

min.
#p: 3

max.
#p: 5

(c) Result

(`#p` is the execution counter for program point `p`)

Example, discussion

Note how we collected information about lower and upper bound of `#p` during the AE of the loop

This scheme can be extended to record a number of more complex flow constraints, including infeasible nodes, infeasible pairs of nodes, and more general infeasible paths

After the AE of the loop, three abstract states reside at `q` for analysis of the rest of the program

A merge point can be placed at the loop exit, allowing these states to merge into a single state

There's more information in our RTSS 2006 paper

Approach 3: The “Census” Method

Origin: MDH (Lisper 2003, Ermedahl et. al. 2007, Bygde et. al. 2008)

Also Dortmund (Lokuciejewski et. al. 2009)

Idea: for a given program point, the number of states provides an upper bound to the number of executions *provided the program terminates*

(Each iteration must encounter a new state: if not, the computation will loop indefinitely. Cf. Conway’s Game of Life)

The Census Method (II)

The method can be directly used to bound the # of loop iterations

Use the result of a value analysis in some point inside the loop. The abstract state will represent (at least) all concrete states. Thus, the # of concrete states represented by the abstract state will bound the # of iterations

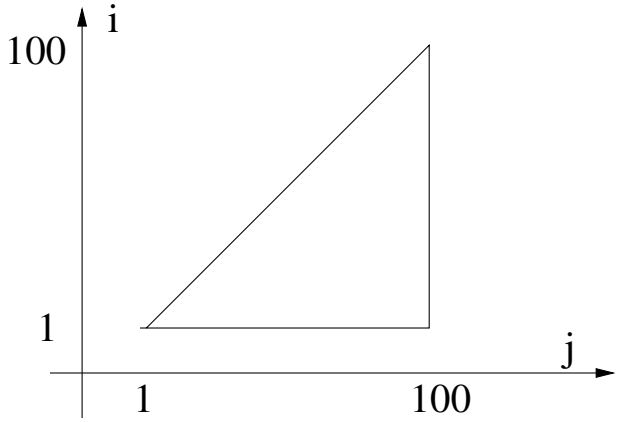
Yields a very general scheme for loop bounds analysis. The abstract domain can be varied to give different tradeoffs between precision and scalability. What is required is a method to count the # of represented concrete states

Pros: general, conceptually simple, tradeoff scalability/precision, can capture dependencies in nested loops, and give *parametric* bounds

Cons: risk of imprecision if used naïvely (we'll get back to this)

Example

```
for i = 1 to 100 do
  for j = i to 100 do
    loop-body
```



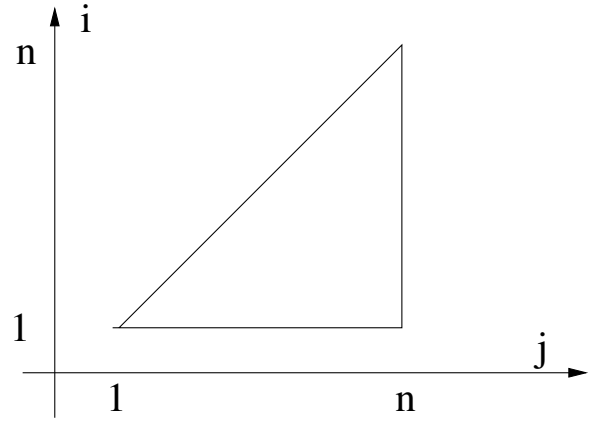

```
#loop-body <= 5050
```

In this example we use a *polyhedral abstract domain* (Halbwachs 1978) for the value analysis

This is a *relational* domain, can thus capture the triangular loop dependency

Parametric Loop Iteration Bounds

```
for i = 1 to n do
  for j = i to n do
    loop-body
```



`#loop-body <= n*(n+1)/2`

Use the polyhedral abstract domain with variables `i`, `j`, `n`

Symbolic methods for counting the # of integer points in polyhedra exist

Can be used to compute the parametric bound automatically

A Risk with the Census Method

Consider this example:

```
s = 0; i = 1
while i <= 100 do
  j = i
  while j <= 100 do
    s = s + x; j = j + 1
  i = i + 1
```

Basically same example as before, same iteration bound (5050)

However, the abstract state now also includes `s` and `x`

They cannot affect the loop execution, so they should not be included! Naïve inclusion will yield very large overestimations. (Each unbounded 32 bit variable adds factor 2^{32})

Solution: Slicing

Program slicing can be used to remove statements and variables that surely do not affect the loop conditions. The slicing uses the conditions as *slicing criterion*

A data flow analysis can track the dependencies

Example revisited (slice in red):

```
s = 0; i = 1
while i <= 100 do
  j = i
  while j <= 100 do
    s = s + x; j = j + 1
  i = i + 1
```

Only the slice needs to be analysed!

SWEET

Our prototype WCET analysis tool: SWEET = SWEdish Execution time Tool

Specialises in program flow analysis, can generate many types of flow constraints (so-called Flow Facts)

Implements Abstract Execution

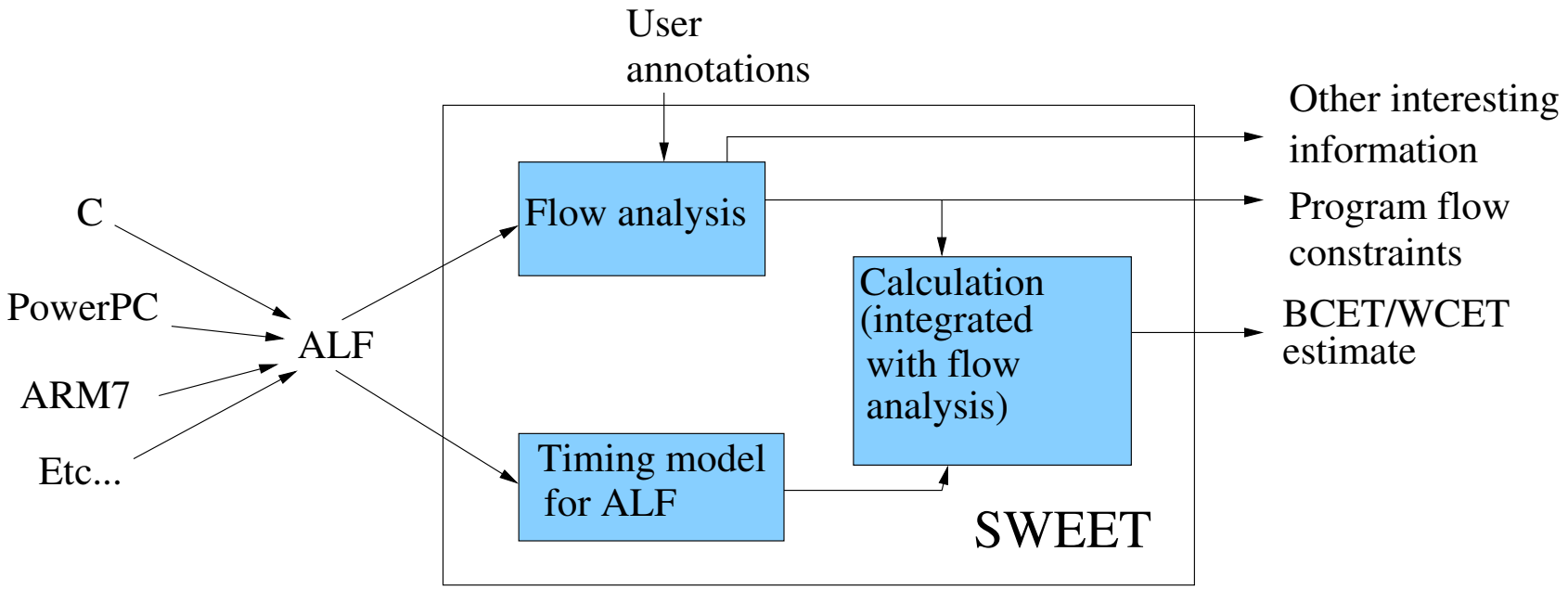
Current version “ALF-SWEET” analyses the ALF intermediate format

(An earlier version, NIC-SWEET, was integrated with the NIC C compiler)

Can analyse a variety of code formats by translation into ALF

Can compute BCET/WCET estimates through AE for simple ALF level timing models

Structure of SWEET



More SWEET Stuff

Extensive online documentation at

`http://www.mrtc.mdh.se/projects/wcet/sweet/`

Includes instructions for how to obtain the tool (it's open source)

It is installed on the virtual machine that we have distributed

You'll get some hands-on experience during the lab session

Examination

The examination assignment: **Extra exercise 3** in the lab instructions (find the “worst” inputs)

You are supposed to use SWEET in a smart way for this purpose (no exhaustive testing of the real code . . .)

The work will be done in groups

You should make a decent presentation of your solution in the morning the 2nd, clearly explaining your approach and your results