SWEET – A Tool for WCET Flow Analysis

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

bjorn.lisper@mdh.se

2014-11-06
Outline

Short introduction to (static) WCET analysis

SWEET:

- Architecture
- Interfaces
- Abstract Execution
- Short demo
WCET

WCET = Worst-Case Execution Time

Definition: $WCET(P) =$ longest time to execute program $P$ uninterrupted on some given hardware

WCET analysis: to estimate or bound the WCET
Why WCET Analysis?

Many systems are real-time systems. These have timing constraints:

Absolute deadlines, maximum response times, etc.

Two classes:

- **Hard** real-time systems: timing constraints *must* be met
- **Soft** real-time systems: *desirable* that timing constraints are met
**WCET Estimation**

*Safe* WCET estimates are needed to safely decide if timing constraints are met:

Safe = will never underestimate real WCET

Static analysis can obtain safe WCET estimates

Important for hard real-time systems
What is Needed for Static WCET Analysis?

- A detailed timing model for the processor, to estimate the execution times of instructions
- Whatever constraints there may be on inputs, initial hardware state, etc.
- Constraints on the possible execution paths through the code (loop iteration bounds, infeasible path constraints, . . .)

Then (in principle) all feasible paths can be explored, and their costs (or upper bounds) decided

(For maximal precision, the linked binary should be analysed)
Making WCET Analysis Feasible

WCET is maximal execution time over all feasible paths. There can be very (exponentially) many paths. How deal with this?

**Answer:** make some controlled approximations to obtain a linear cost model. This will allow the use of standard optimization techniques to bound the WCET.
The Implicit Path Enumeration Technique (IPET)

Compute local WCETs $w_i$ for code fragments $i$ (typically basic blocks)

For each execution of a code fragment, replace its real execution time with its local WCET:

A relaxation. Some precision is sacrificed in order to get a problem that is less costly to solve
WCET Estimation as Integer Linear Programming

For each code fragment, introduce an *execution counter* counting its # of executions.

Then, for any path \( p \), \( \text{execution\_time}(p) \leq \sum_{i} x_i w_i \) where the \( x_i \)'s are the final values of the execution counters for \( p \).

Thus, \( \max \sum x_i w_i \) over all paths will provide a safe WCET bound.

Many program flow constraints can be expressed as *linear constraints* on the \( x_i \)'s.

An *integer linear programming problem*. Solution = WCET bound.
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, flow preservation):
  - 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): $x_i = c \cdot x_j$
Typical Structure of WCET Analysis

- Program source
- Compiler
- Object code
- Flow analysis
- Value analysis
- Low-level analysis
- Annotations
- HW timing model
- Calculation
- WCET estimate

SWEET – A Tool for WCET Flow Analysis
The analysis is typically done on CFG level:

```
numbers = local WCET bounds for basic blocks
```

\[ i := 0 \]
\[ i < n \]
\[ i < n-10 \]
\[ \text{stop} \]
\[ B1 \quad B2 \]
\[ i := i+1 \]
\[ \text{true} \]
\[ 37 \quad 42 \quad 29 \quad 44 \]
\[ 187 \quad 233 \]
Low-Level Analysis

Purpose: find local WCETs for code fragments (basic blocks)

Must (1) bound possible hardware states, (2) use this information to bound the execution time of instruction sequences

(1) is often done by abstract interpretation, (2) requires an accurate timing model for the instruction set
Flow Analysis

Purpose: derive safe and tight Flow Facts

Undecidable in general (would solve the Halting Problem)

Find methods that work well enough on interesting program classes

We’ll give an account for a method that is used in SWEET
SWEET

SWEET = SWEdish Execution time Tool

A program flow analysis tool, can generate many types of flow constraints (Flow Facts)

Analyses the “ALF” intermediate format

Can analyse different code formats by translation into ALF

Can compute BCET/WCET estimates for simple ALF level timing models
SWEET’s Flow Facts

Upper and lower loop iteration bounds

Various infeasible path constraints (single, pairwise, longer paths)

Can be context-sensitive (full call strings)

Given relative to execution contexts ("scopes"): main (global), function calls, loops

FOREACH Flow Fact: valid for each individual execution of the scope (execution counters reset at each entry)

FORALL Flow Fact: valid for all executions together of the scope (execution counters not reset)

(IPET Flow Facts are relative to the “main” scope)
Some Examples

: main : [] : bb4 <= 46 ;
A global loop bound: in scope main the basic block bb4 is executed at most 46 times in total

: main : [] : bb17 + bb19 <= 412 ;
An infeasible path constraint stating that bb17 and bb19 can be taken together at most 412 times in total

((main,BB1),baz) ((baz,BB7),bar) : bar : [] : bb9 <= 4
A bound on bb9, in scope bar, when bar is called from baz at BB7 which in turn is called from BB1 in main
Flow Fact Formats

SWEET can generate Flow Facts in different formats:

- It’s own native Flow Fact format (the most expressive format)
- The AIS annotation format for aiT
- The RapiTime annotation format

Allows SWEET to pass Flow Facts to the commercial WCET analysis tools aiT and RapiTime
Input Annotations

Can specify the ranges for variables in different program points

Typically used to constrain values of global variables, or function arguments, at entry

Can declare variables as volatile – for them, nothing more can be assumed than being in the specified range

Input annotations really specify the environment in which the program executes
Input Annotation Examples

PROG_ENTRY ASSIGN "a" TOP_INT || "b" TOP_INT ;
Program variables $a$ and $b$ can have any (int) value at program entry

PROG_ENTRY ASSIGN "a" INT 0 30 || "b" INT 0 100 ;
At program entry the value of $a$ is in the range $[0 \ldots 30]$ and value of $b$ is in the range $[0 \ldots 100]$

VOLATILE ASSIGN $k$ INT 66 68 ;
The variable $k$ is volatile, and will always have its value in the range $[66 \ldots 68]$
ALF

SWEET analyses the language ALF

ALF is an intermediate code format

Designed to represent both high- and low-level code faithfully (C as well as machine code)

Low-level constructs (load, store, arithmetic/logic operations, dynamic jumps) as well high-level constructs (function call/return etc.)

Memory consists of *frames* – memory areas with a name and size. Similar to unlinked code

Translators to ALF from C, PowerPC & NEC V850 binaries (experimental)
An Example

This C code:

```c
if(x > y) z = 42;
```

can be translated into the following ALF code:

```alf
{ switch
  { s_le 32 { load 32 { addr 32 { fref 32 x } } { dec_unsigned 32 0 } } }
   { load 32 { addr 32 { fref 32 y } } { dec_unsigned 32 0 } }

  { target { dec_unsigned 1 1 }
    { label 32 { lref 32 exit } { dec_unsigned 32 0 } }
  }

  { store { addr 32 { fref 32 z } } { dec_unsigned 32 0 }
    with { dec_signed 32 42 }
    { label 32 { lref 32 exit } { dec_unsigned 32 0 } }

```

SWEET – A Tool for WCET Flow Analysis
Abstract Execution

SWEET computes Flow Facts through a kind of symbolic execution – “abstract execution” (AE)

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

Can be seen as a value analysis (abstract interpretation)

Each abstract state transition corresponds to many possible concrete state transitions

SWEET currently uses intervals to represent sets of (numerical) values
Abstract Execution (II)

AE can be seen as a very context-sensitive value analysis (differing between 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. Merge points can be placed in the CFG. Merging can affect the precision, though
Merging Abstract States

Merge = take least upper bound, $\sqcup$

Common merge points: after join nodes, at loop exit edges
i = INPUT; // i = [1..4]
#p = 0;
while (i < 10) {
    // point p
    #p = #p + 1;
    ...
    i = i + 2;
} // point q

(a) Code example

<table>
<thead>
<tr>
<th>p</th>
<th>i at p</th>
<th>i at q</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[1..4]</td>
<td>impossible</td>
</tr>
<tr>
<td>2</td>
<td>[3..6]</td>
<td>impossible</td>
</tr>
<tr>
<td>3</td>
<td>[5..8]</td>
<td>impossible</td>
</tr>
<tr>
<td>4</td>
<td>[7..9]</td>
<td>[10..10]</td>
</tr>
<tr>
<td>5</td>
<td>[9..9]</td>
<td>[10..11]</td>
</tr>
<tr>
<td>6</td>
<td>impossible</td>
<td>[11..11]</td>
</tr>
</tbody>
</table>

(b) Analysis

min. #p: 3
max. #p: 5

(c) Result

(#p is the execution counter for program point p)
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 $q$.

If a merge point is placed at the loop exit, then these states are merged into a single state.
BCET/WCET Estimation with SWEET

SWEET has a simple mechanism to estimate BCET and WCET

Each basic block $i$ can be given a constant cost $c_i$

We add a variable $T$ that stands for time

$T$ is incremented with $c_i$ for each executed basic block $i$

AE now computes an interval for $T$

The final interval at end yields bounds for BCET and WCET

(Crude estimates, typically neither exact nor safe. But can still be useful to get a grip on what BCET and WCET is)
An Example

```
i := 0
start
i < n
i < n - 10
stop
false
B1 B2
falsetrue
i := i + 1
true
T := 0
37
T := T + 37
29
T := T + 29
44
T := T + 44
233
T := T + 233
187
T := T + 187
42
T := T + 42
```

SWEET – A Tool for WCET Flow Analysis
Other Analyses

SWEET can also make other analyses, like:

- A conventional value analysis
- data flow analysis (reaching definitions)
- program slicing

So it’s also becoming a general program analysis tool
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)
Conclusions

A tool for generating Flow Facts
Also: crude BCET/WCET estimation, value analysis, . . .
Can interface with WCET analysis tools that do precise low-level analysis
Things to improve (future work):
  • Better handling of failure to bound loops
  • More expressive input annotations
  • A larger assortment of abstract domains for value analysis
  • A lightweight relational domain for AE
  • More X-to-ALF translators
  • . . .