# ALF – A Language for WCET Flow Analysis

Jan Gustafsson, Andreas Ermedahl, Björn Lisper, Christer Sandberg,
and Linus Källberg
School of Innovation, Design and Engineering, Mälardalen University
Box 883, S-721 23 Västerås, Sweden
{jan.gustafsson,andreas.ermedahl,bjorn.lisper,christer.sandberg,
linus.kallberg}@mdh.se

2009-06-30

# Flow Analysis

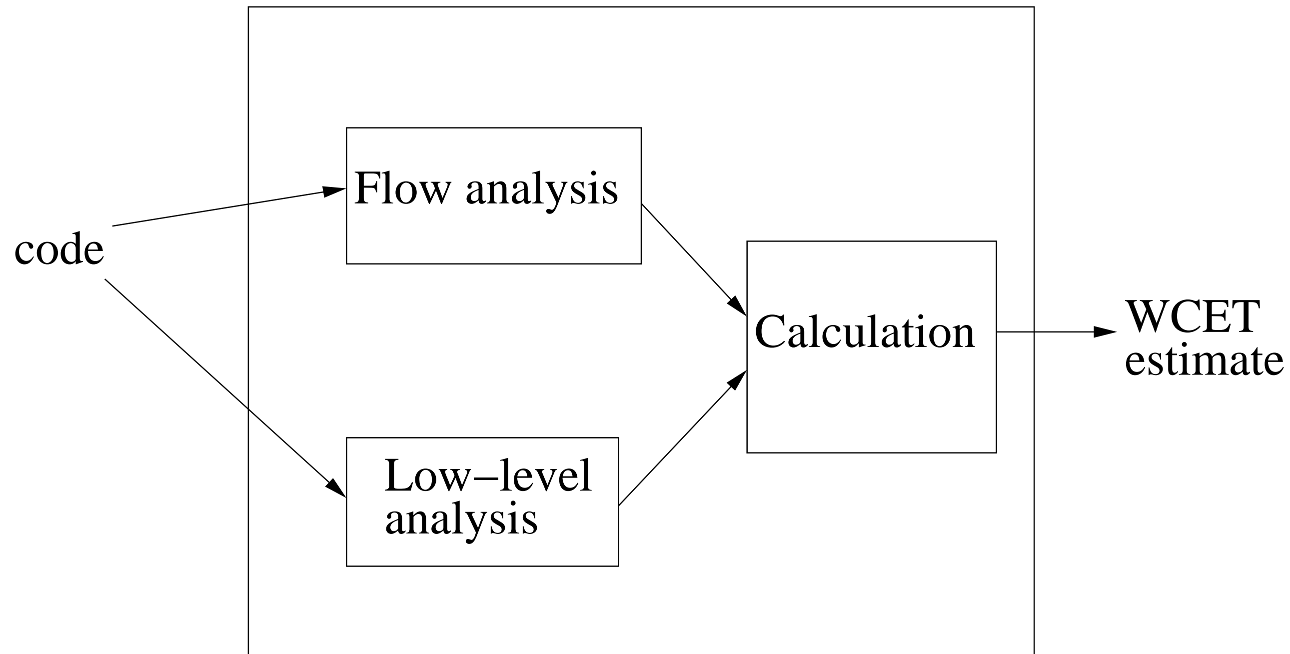
Find constraints on program flow:

- Max # of loop iterations
- Infeasible paths
- Etc.

Important part of WCET analysis

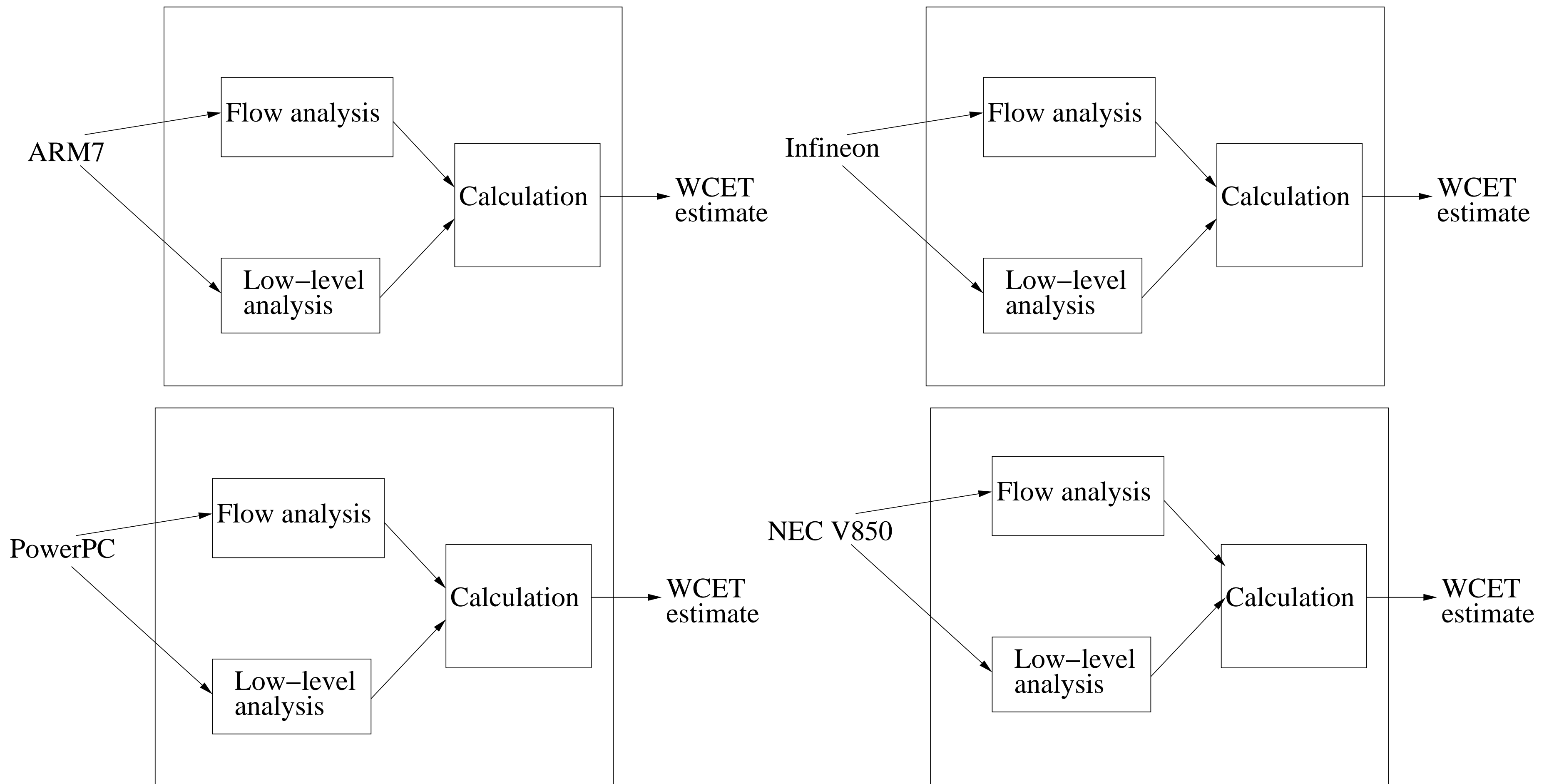
Difficult (precise analysis undecidable)

# Canonical Structure of WCET Analysis Tool

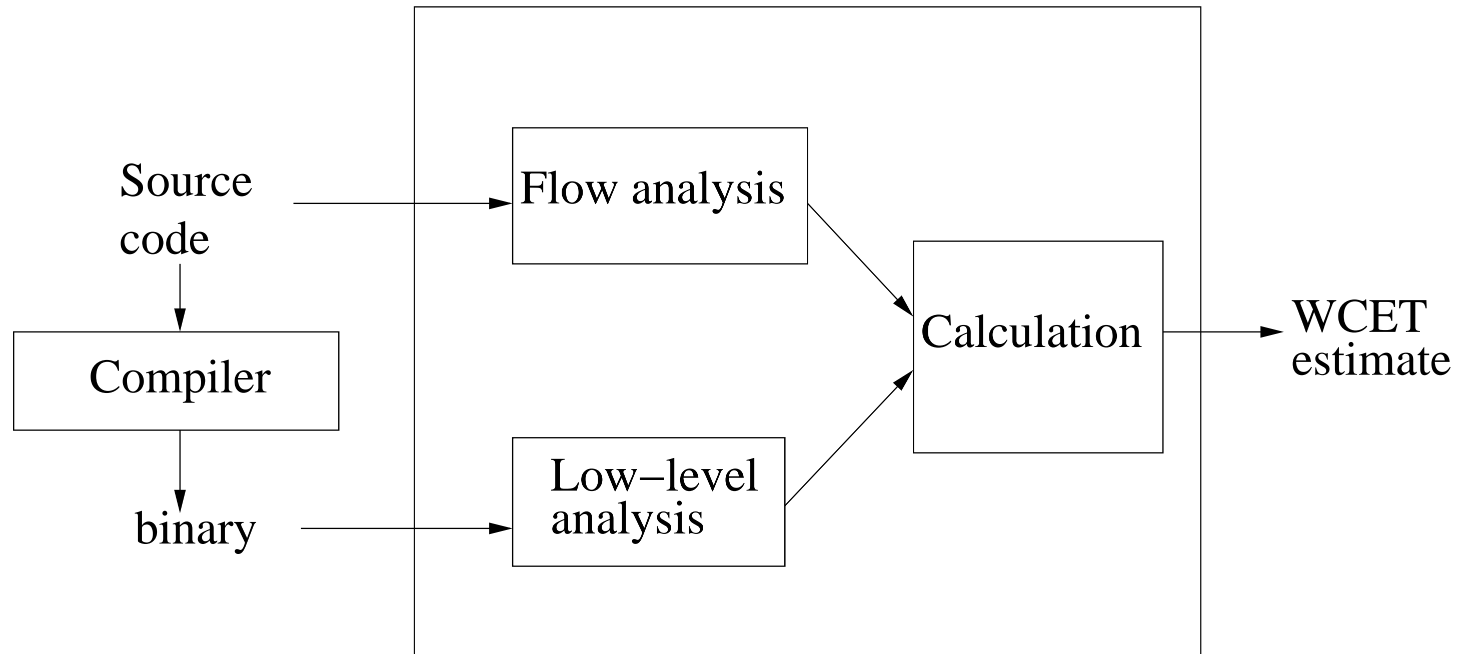



The binary is analyzed

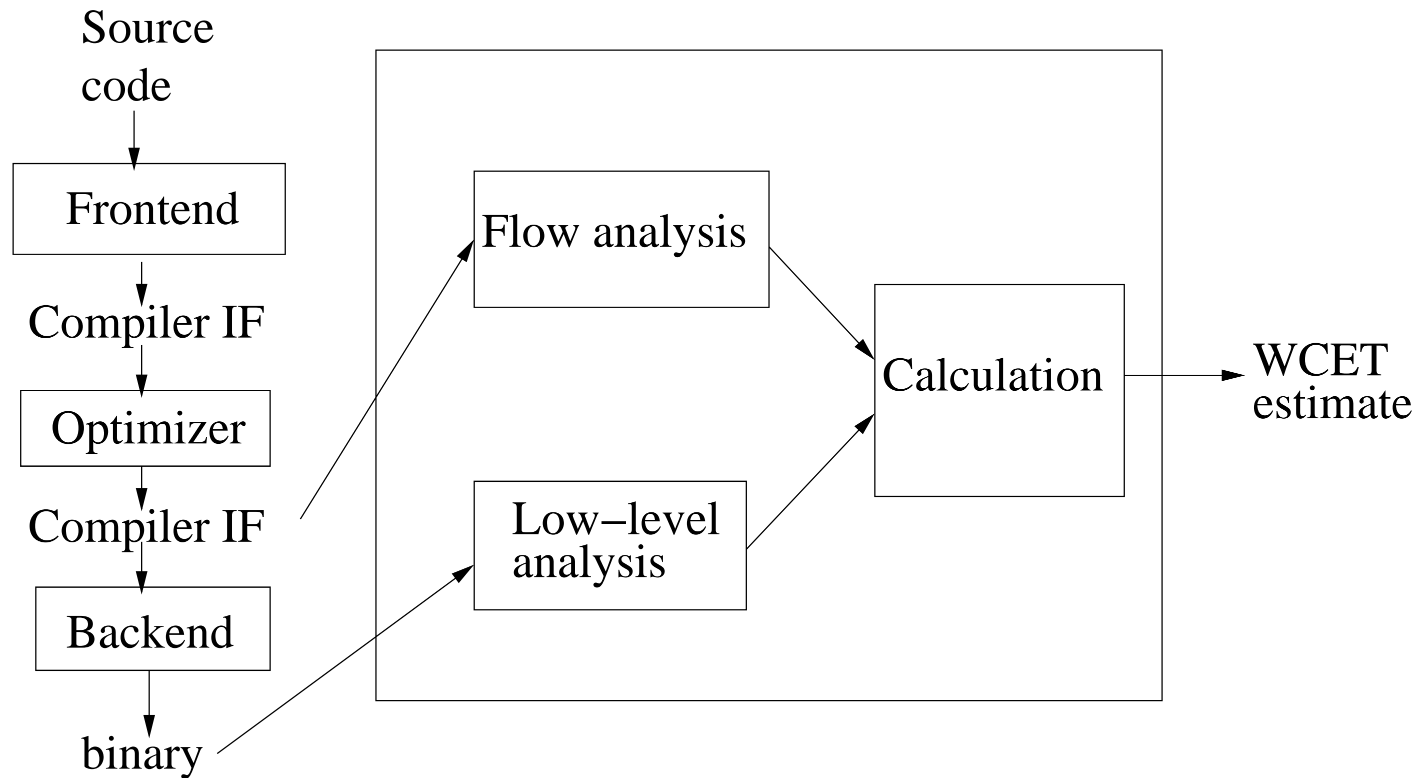
# There are Many Different Binary Formats



# Source Code Flow Analysis



# Intermediate Code Flow Analysis



# The Problem

Many different code formats to analyze!

Can be expensive to implement a new flow analysis for each format

Flow analysis techniques are usually quite format-independent

Can we avoid a proliferation of implementations?

# A Possible Solution: ALF

**ALF** $=$ **A**RTIST2 **L**anguage for WCET **F**low Analysis

An intermediate level code format

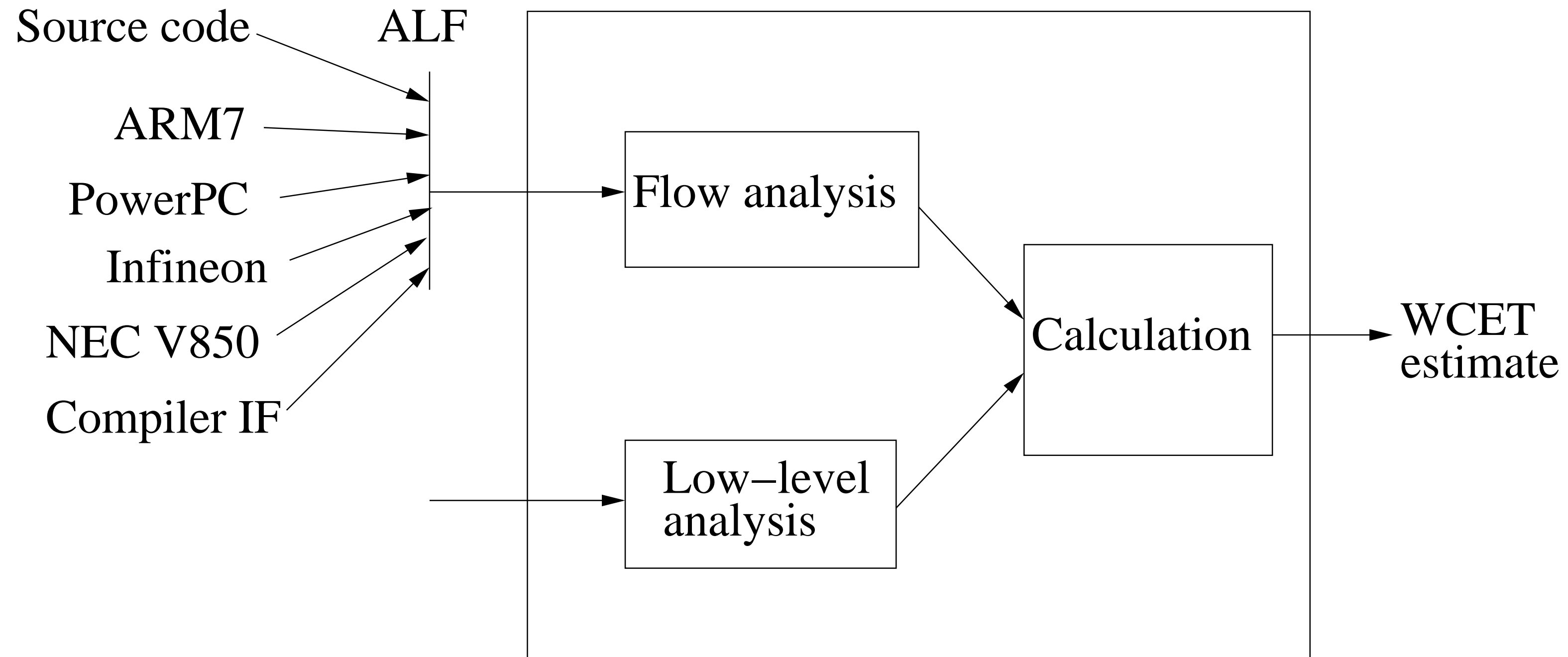
Designed for ease of program analysis rather than code generation

Idea: rather than building a new flow analysis component for each format to analyze,

- build a single component analyzing ALF
- create translators from the different formats into ALF

Provides a challenge for the design of ALF – must allow faithful translations from many different formats

# The ALF Solution



# ALF General Characteristics

Imperative language (stateful, executing statements change contents of memory)

Differs between data and code memory (only data memory can be written)

Memory model resembles model for unlinked code – symbolic base pointers

Both high- and low-level constructs for control flow (function call/return, but also dynamic jumps)

# A Note on ALF Syntax

Fully textual format, LISP/Erlang-like:

```
{ 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 } }
```

A common question: “why not XML?”

# True Answer

Because I don’t like XML very much!

# A Better Answer

It would be easy to define “XML-ALF” if desired, and translate between ALF and XML-ALF:

```
{ name ... }

<name> ... </name>
```

But we see other issues as much more important than the style of syntax, so this has low priority

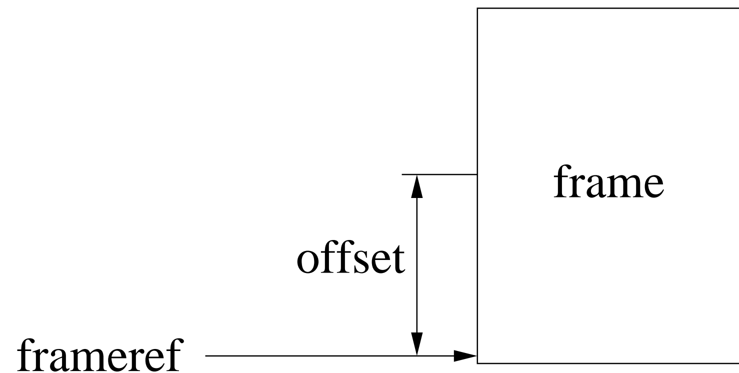
# ALF's Data Model

ALF's data memory consists of *frames*

Can be both statically and dynamically allocated

A symbolic *frameref* points to the start of a frame

An ALF data address is basically a pair (frameref, offset)



# Code Addresses

ALF's code addresses are called *labels*

A label is similar to a data address: a pair (lref, offset), where lref is a symbolic *label reference* and offset is numeric

ALF statements may have labels (but need not)

Can be used to model both fully symbolic labels (set offset to $0$ always), or addresses in memory (use offset as numeric address)

Both lref and offset can be dynamic: allows to model dynamic jumps

# ALF Values

Five kinds:

- numerical values: signed/unsigned integers, floats, etc.,
- framerefs,
- label references (lrefs),
- data addresses, and
- code addresses

Each value has a *size* (# of bits)

There are unbounded size integers

Finite size values are *storable*

Storable values can be *symbolic* or *bitstring* values

# ALF Operators

A number of different kinds of operators:

- on *data of limited size* (arithmetic/logic etc)
- on *data of unbounded size* (“mathematical” operations)
- on *bitstrings* (e.g., concatenation)
- a *conditional*
- operators to read from memory, and allocate memory

All operators (but one) are side-effect-free

# ALF Statements

The most important ones:

- `store` (concurrent assignment)
- `switch` (multiway jump with possibly dynamic targets)
- function `call` and `return`

# A Simple Example

C code:

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

Corresponding ALF code:

```
{ 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 } }
```

# Tool and Translators

A version of SWEET analyzing ALF is underway (current version can analyze some simple examples)

Work is done in the ALL-TIMES FP7 project

Translators (at different stages of completion):

- TU Vienna: SATIrE IF $\rightarrow$ ALF, will enable source level flow analysis
- AbsInt: aiT CRL2/PowerPC $\rightarrow$ ALF (binary level flow analysis)
- AbsInt: aiT CRL2/NEC V850 $\rightarrow$ ALF (ditto)
- IAR Systems: C compiler IF $\rightarrow$ ALF (source/IF level flow analysis)

# Conclusions and Further Research

A step towards an open framework for flow analysis

Allows the development of generic analyses, and comparisons of different analyses

May be used for other things than flow analysis, like different generic tools for analyzing/manipulating binaries

To do in the future:

More translators & implementations

XML syntax, for the so inclined

Development of accompanying formats for expressing flow analysis results, value constraints, and other similar information