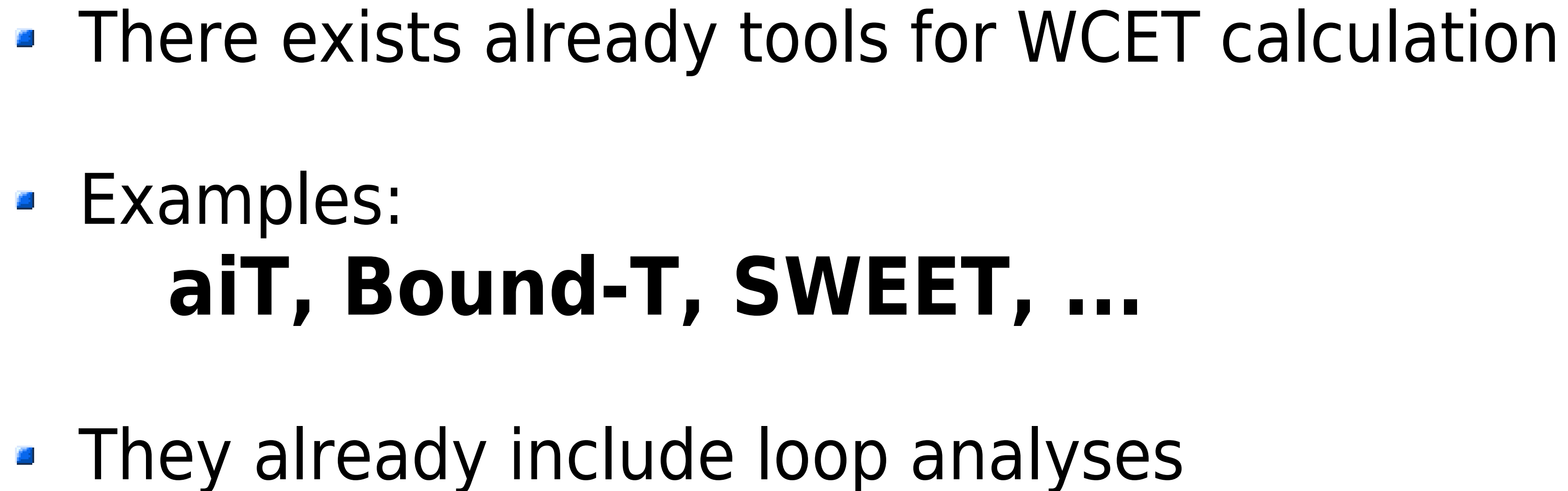
Data-Flow Based Detection of Loop Bounds

Christoph Cullmann   **Florian Martin**

Motivation

- Most time-critical programs on embedded systems contain loops
- We need WCET calculations for them
- Loop bounds are needed to calculate the WCET

Tools out there



aiT's loop analysis

- Works on machine code
- Loops are transformed to procedures
- Cooperates with value analysis iteratively, multiple iterations to cover nested loops
- To find loop bounds, use:
  - pattern matching
  - slicing
  - dominator/post-dominator analysis

Why come up with new stuff?

- Current pattern matching needs:
  - manual adaption for each loop type
  - manual adaption for each compiler, even each optimization mode

- It's difficult to integrate more complex loops
  e.g. loops with multiple internal alterations
  of loop counter

How to improve?

- Idea:
  - Keeping the proven iterative approach in combination with value analysis
  - replace the fixed patterns with a new data-flow analysis to generate equations for the loop counters
- This should lead to:
  - being more generic
  - being more flexible

Analysis Steps

- Collection of loop counters & start values
- Building of equations for this loop counters via data-flow analysis
- Inspecting all exits and calculating min/max iterations using found start values, equations and exit informations

Detection of Loop Counters

- Candidates for loop counters:
  - registers accessed in the loop routine
  - memory cells accessed in the loop routine

- Simple analysis on the control graph, using value analysis results for memory access

- Ask value analysis for start values, only keep registers/memory cells with known start values

The Data Flow Analysis

- Is started with equations for found possible loop counters at first call
- Will result in equations for possible loop counters annotated to all edges in the loop routine
- This equations:
  show alteration of counter in one loop iteration

The used Flow Problem

- Data flow value:
  set of equations

- Each equations contains:
  - destination variable
  - set of all possible values

- Important:
  - a variable on the right side stands symbolic for the start-value of this variable at the loop entry

Variables? Values?

- A variable is a triple of:
  - type: register or memory
  - register number or memory address
  - width in bytes
    (used for overflow checking)

- A value is a quadruple of:
  - source variable
  - additive constant
  - minimal width in bytes we operate on
  - signedness

Transfer function

- Forward problem
- Transfer function, different handling for:
  - normal blocks with instruction
  - recursive loop call of loops
  - returns from loops
  - others: identity
- Other interesting parts:
  Combination & Widening

Transfer: for Instructions

- Two phases:
  - remove all equations for variables which are modified by this instruction
  - add new equations you can derive out of the semantics of the instruction

Transfer: loop rec. call/return

- loop recursive call:
  - We want to find invariants for one iteration
  - propagate the initial equations of the loop to the call edge

- loop return:
  - We don't want to propagate flow outside of loops in the current nesting
  - propagate empty set of equations to the return edge

Combination Function

- We aim for safe results, therefor result will only contain:
  - equations for destination variable for which in both given sets already equations existed
  - right side of equations need to be merged, if not possible, equation must be removed

Widening

- Problem:
  - the constants in the values can change per iteration
  - only bound by range of integer

- Solution:
  - Remove all equations which changed since last iteration, but existed there already

Evaluation of all exits

- First: classification of exit:
  - always reached in a iteration?
  - only reached sometimes?

  this is done by a dominator analysis

- Only always reached exits can give information about maximal iteration count

- All exits can give information about minimal iteration count

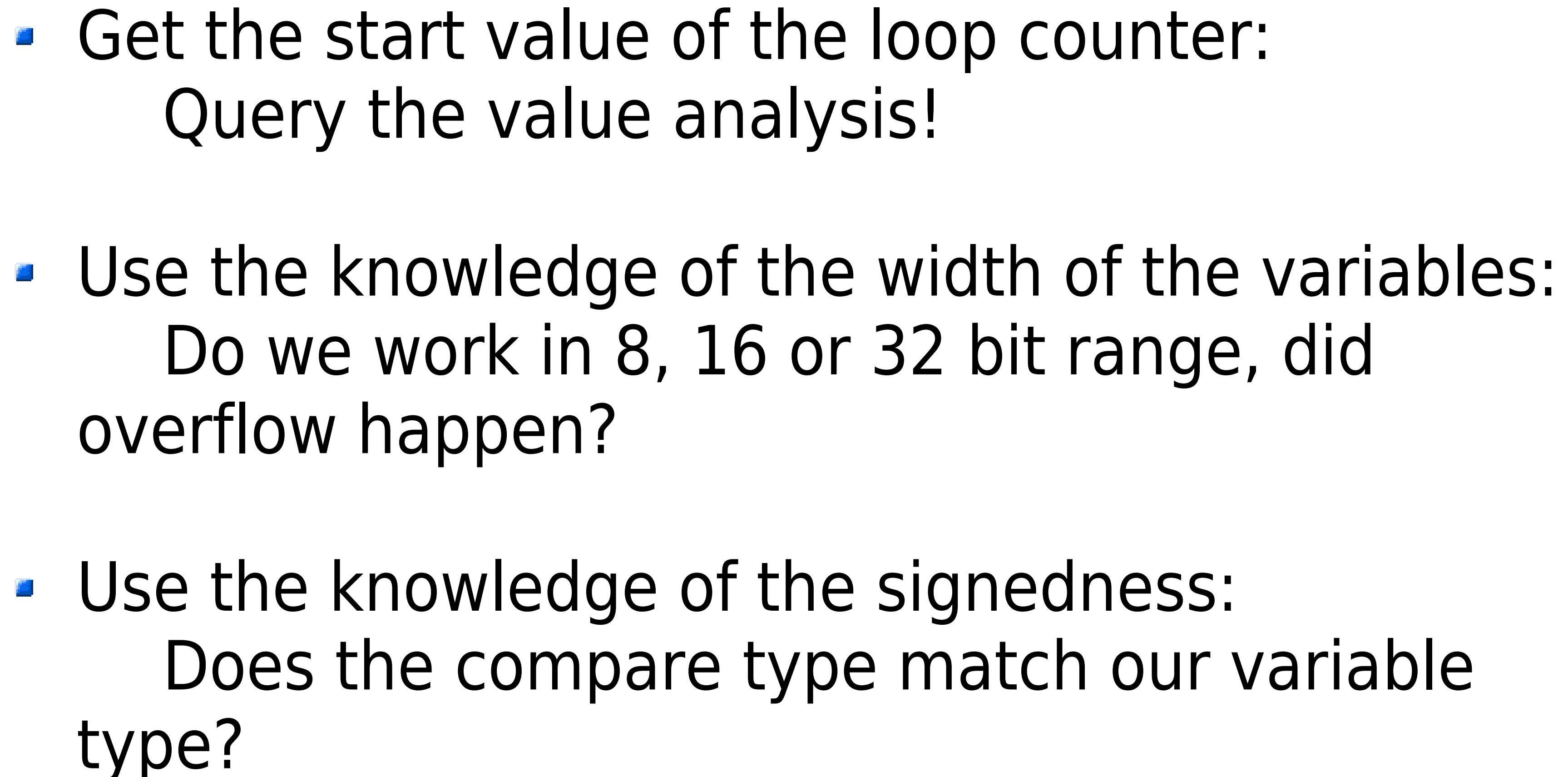
What to do for each exit?

- Exits are conditional branches
- Identify variables/constants used in the compare
- Identify the condition: =, <, >, ...
- Build a equation for the found condition and used variables

Derive the loop count

- We have now:
  - equations for loop counter
  - equation for loop exit
- We can derive:
  - increment per iteration
  - increment before the compare
  - possible assignment of constant to counter

What's still needed?



- Get the start value of the loop counter:
  Query the value analysis!

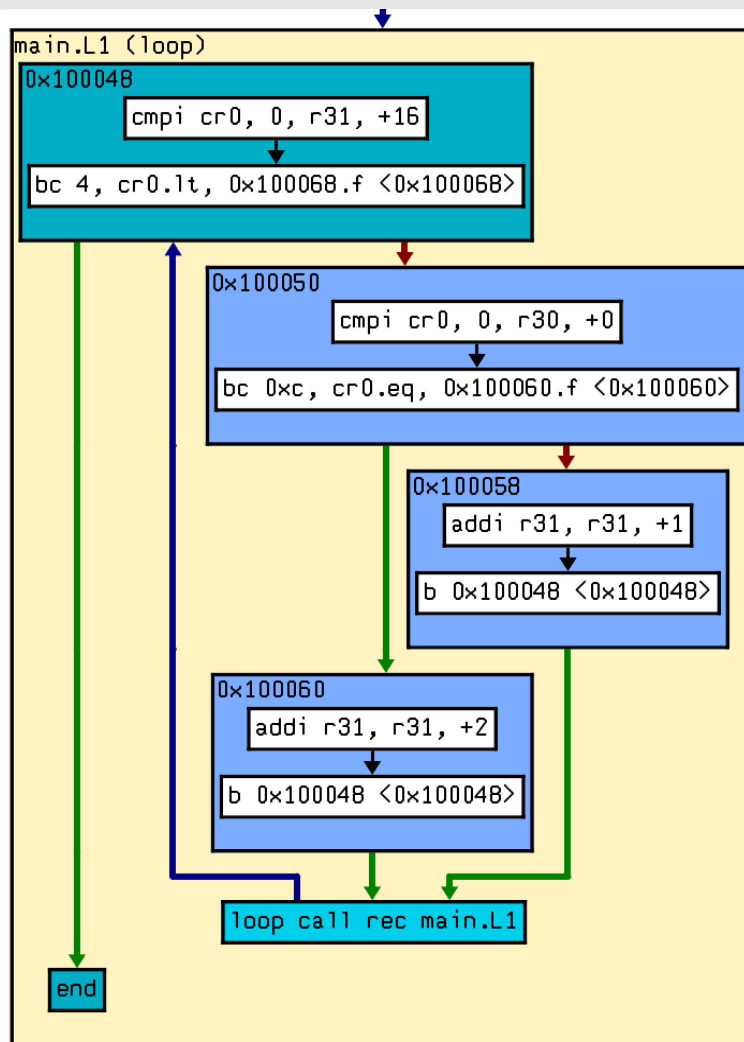
- Use the knowledge of the width of the variables:
  Do we work in 8, 16 or 32 bit range, did overflow happen?

- Use the knowledge of the signedness:
  Does the compare type match our variable type?

Combine the bounds

- For each loop: calculate bound for each exit this way
- Combine the results to conservative bounds for the whole loop

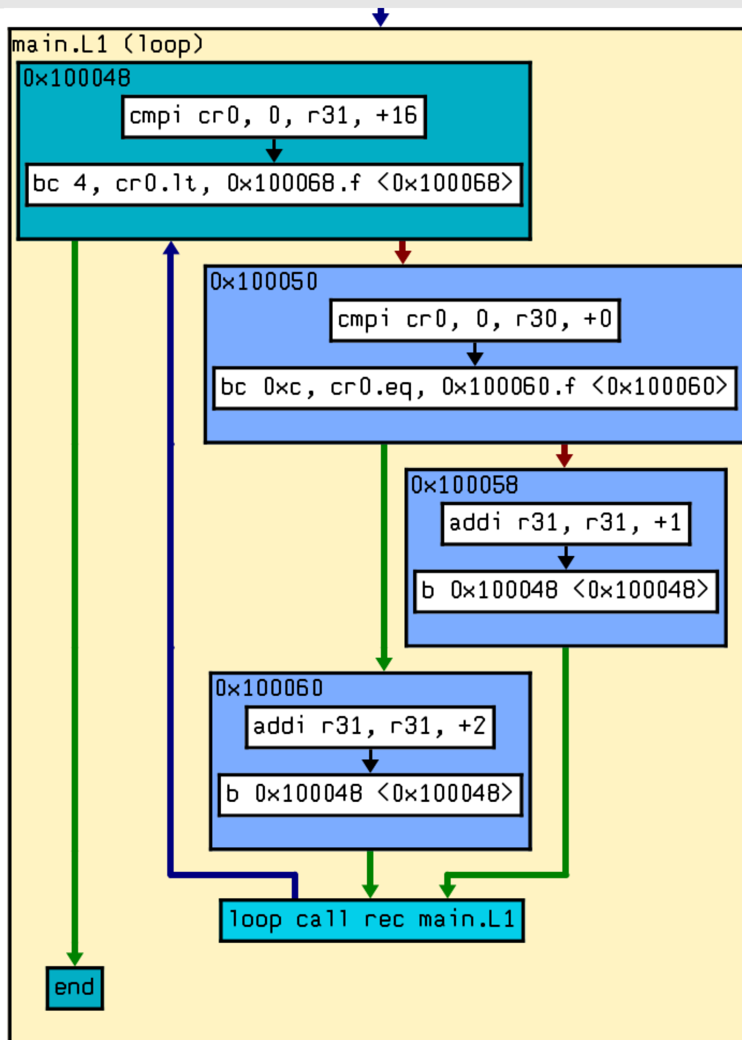
Simple Example



C code:

```
while (r31 < 16)
{
    if (r30 == 0)
    {
        r31 = r31 + 1;
    }
    else
    {
        r31 = r31 + 2;
    }
}
```

Equations for example



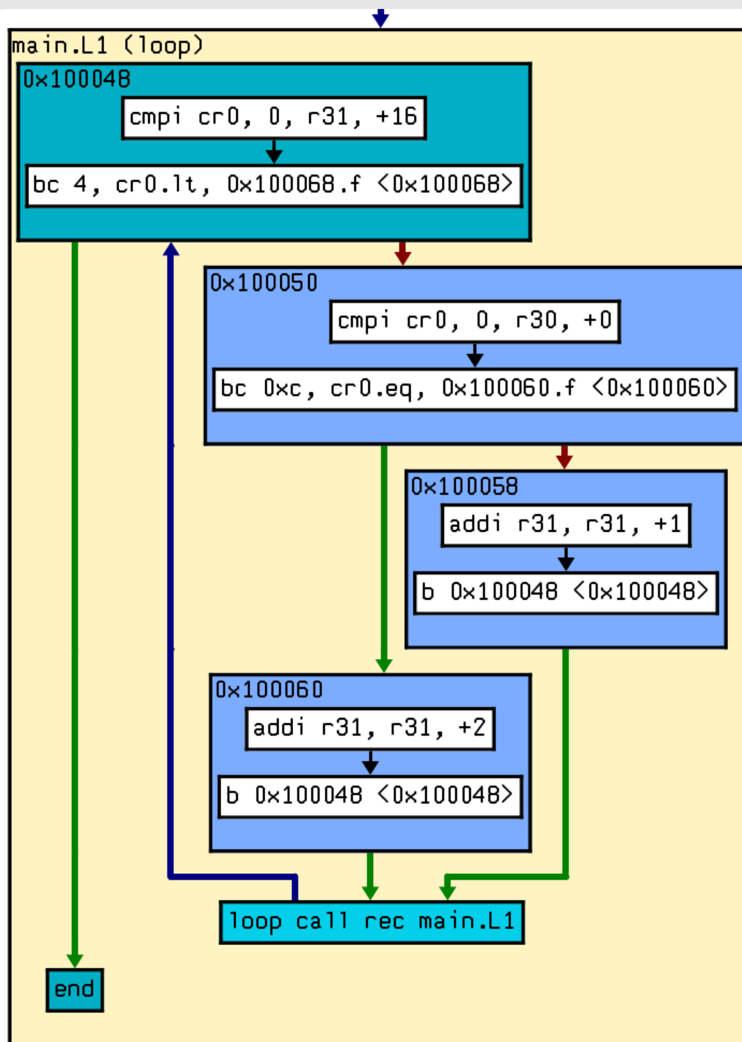
{ r31 = r31; }

{ r31 = r31; }

{ r31 = r31 + 1; }

{ r31 = r31 + 2; }

Bounds for example



Equation for one iteration:

$r31 = r31 + [1, 2]$

Start value (for example):

$r31 = 0$

Loop exit equation:

$r31 >= 16$

Solution:

min: 9

max: 17

That's it!

- We use now a data flow analysis to get loop invariants instead of only fixed patterns

- Practical evaluation shows:
  - It is less compiler dependent
  - It provides a portable solution
  - It's speed/complexity is usable

Future work

- Applying the method to more architectures
- Refining the implementation for more complex exit checks
- Allow constant multiplication in the equations
- Use a more powerful equation solver to allow more expressive equations

Thanks & Questions?