

Oberseminar WS05/06

Static Detection of Safe Loop Bounds


Christoph Cullmann

Betreuer: Florian Martin

Why Loop Bounds?

- Today:
 - many time-critical programs on embedded systems
- We need WCET calculations for them
- Loop bounds are needed to calculate the WCET

Current Situation

- There exists already tools for WCET calculation
 - Example: AbsInt's aiT :)
 - aiT already includes a loop analysis
- 

How does it work?

- Works on transformed loops
- Cooperates with value analysis iteratively
- To find loop bounds, use:
 - pattern matching
 - slicing
 - dominator/post-dominator analysis

Why multiple Iterations?

- Programs tend to include nested loops
 - Inner loops will depend on outer loops in many cases
 - Each iteration deals with loops on same nesting depth
 - Restart of the value analysis with the computed loop bounds after each round
-
-

Why come up with new stuff?

- Current pattern matching needs:
 - manual adoption for each loop type
 - manual adoption for each compiler, even each optimization mode
- It's hard to integrate more complex loops
e.g. loops with multiple internal alterations
of loop counter

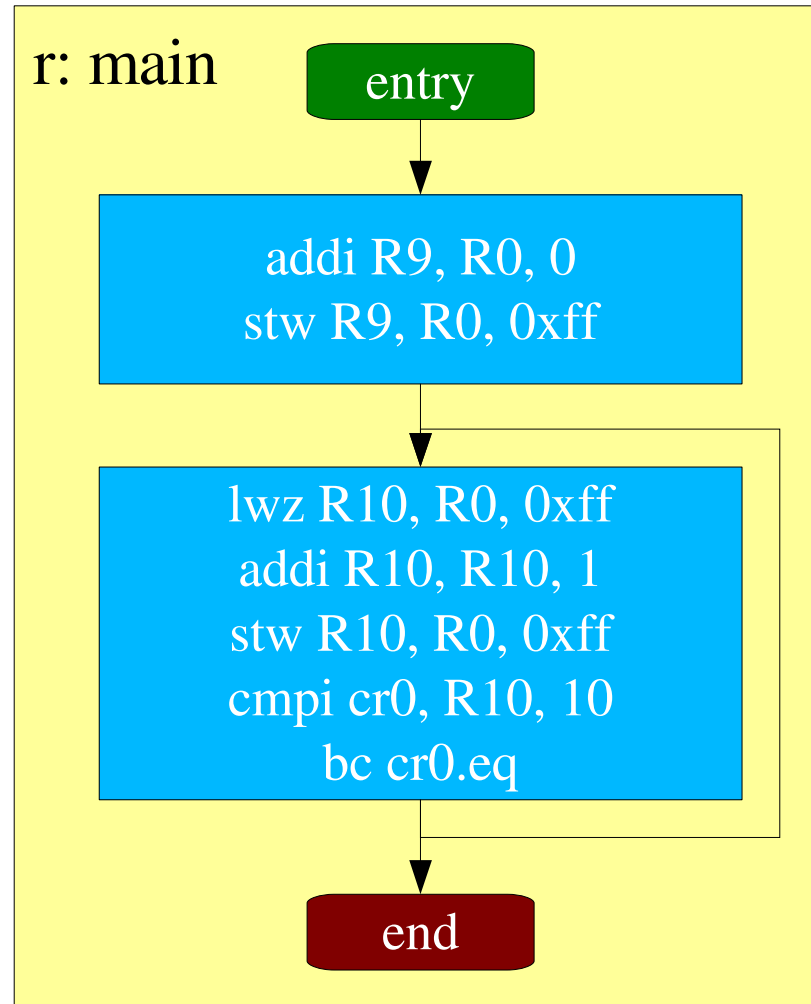
What will stay the same?

- Works on transformed loops
 - Uses dominator analysis & slicing for finding and classifying exits
 - Cooperates with existing value analysis
 - The iteration scheme stays the same
-
-

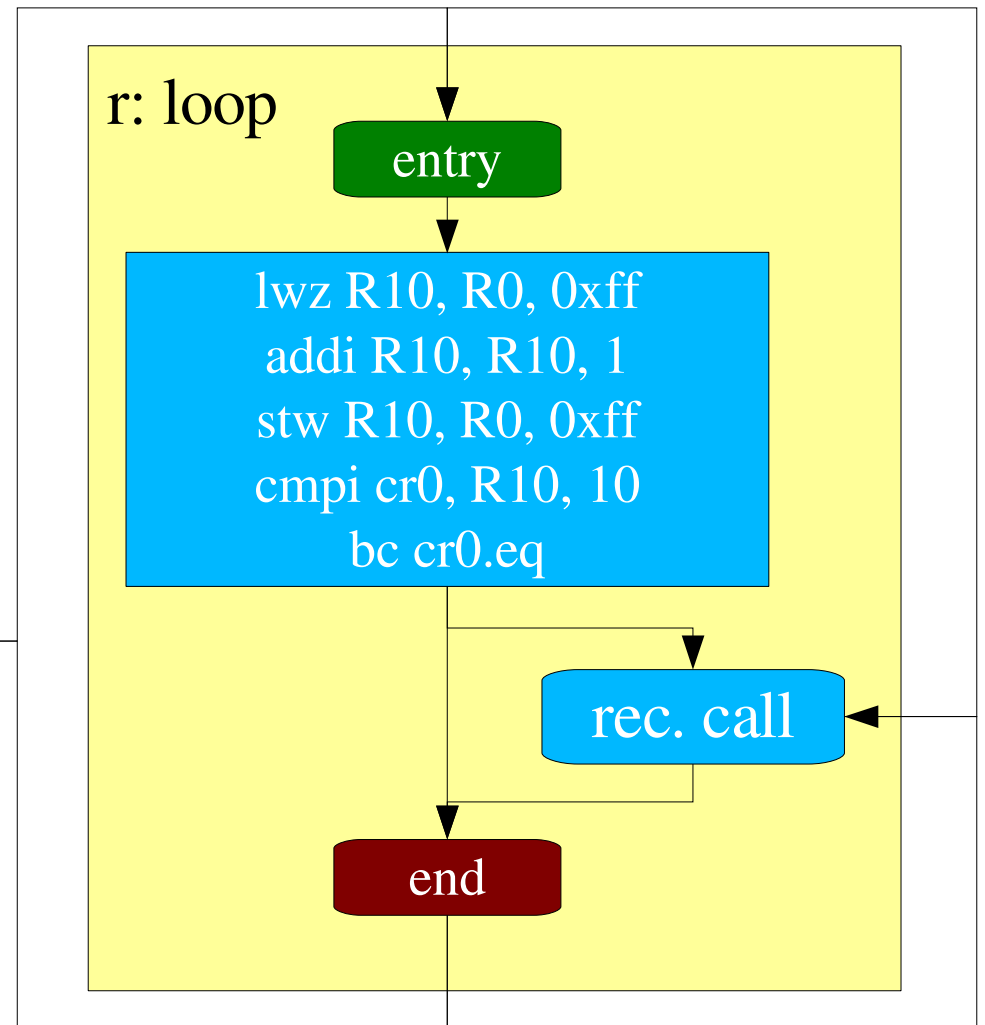
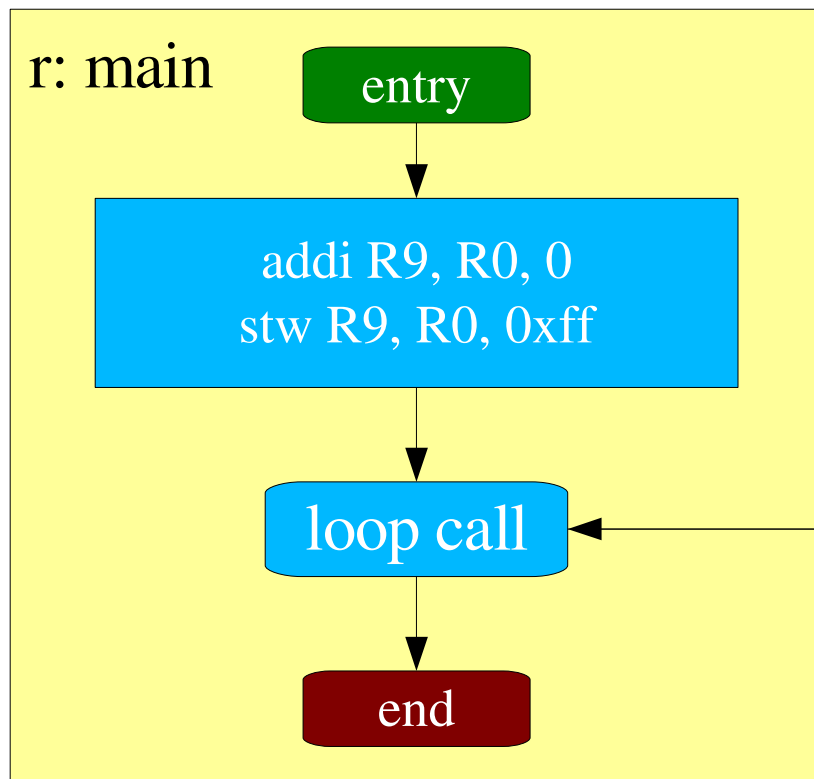
Where is the innovation hidden?

- The main difference:
 - it uses a new data-flow analysis to generate equations for the loop counters
- This leads to:
 - being more generic
 - being more flexible
 - but has problems with special cases :)

Running Example



After Loop Transformation



Overview: One Analysis Iteration

- Sorting of loops by nesting depth
- Classification of loops based on value analysis
- Collection of possible loop counters & start values
- Building of equations via data-flow analysis
- Inspecting all exits and calculating min/max iterations using found start values, equations and exit informations

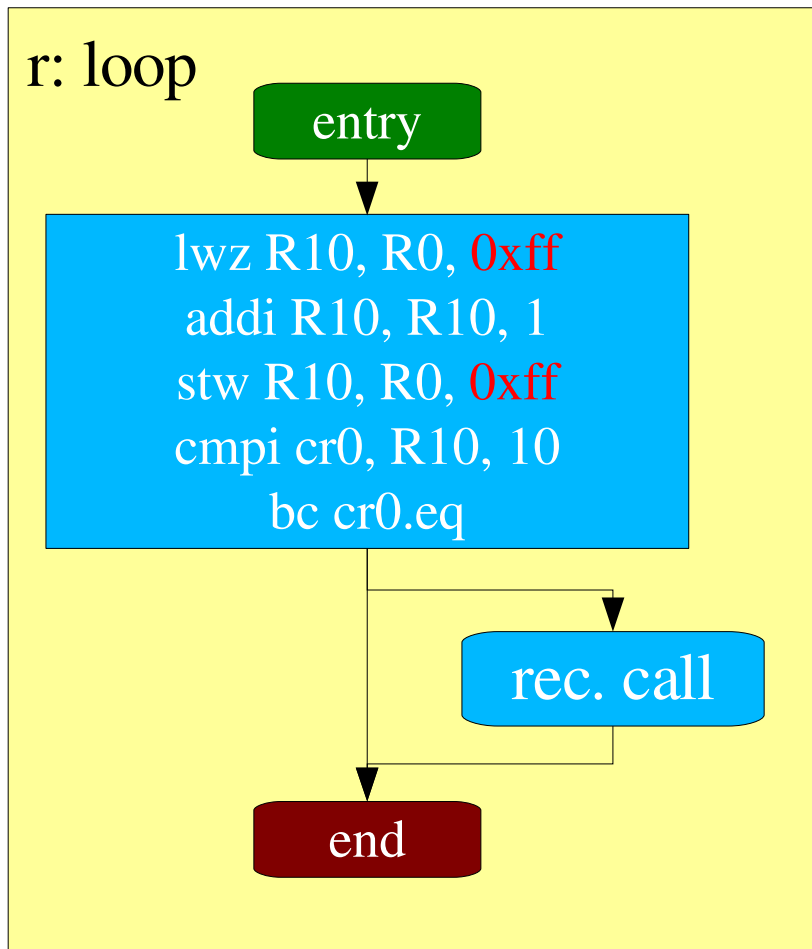
Loop Classification

- Classification in this types:
 - never reached loops
 - loops for which the recursive call is never reached
 - loops for which return node for the loop call is never reached
 - other loops: these ones will be analyzed more in detail
-
-

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
-
-

Results for example



- Loop routine accesses only memory cell 0xff:4 and register R10
- start value for 0xff:4 known, thanks to value analysis, not for R10
- Possible loop counters:
 - 0xff:4, with start [0,0]

Overview: Data Flow Analysis

- Covers all loops in the current nesting depth
- 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

Properties of the 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, memory
 - address: register number or memory address
 - it's width in bytes
 - A value is a quadruple of:
 - source variable (none indicates constant)
 - additive constant
 - minimal width in bytes we operate on
 - signedness: are we sign extended or not?
-
-

Applied to Example?

- Initial equations for our example on first call of the loop in main would consist only of:

 ((memory, 0xff, 4),
 [((memory, 0xff, 4), [0,0], 4, not extended)])
- In words:
 memory cell 0xff:4 = memory cell 0xff:4
 on loop start

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

Why two phases?

- First phase easy to implement for all instructions in a generic way, using only value analysis
- This leads already to conservative generation of equations, **we are on the safe side!**
- Second phase only needed for a few instructions:
 - constant addition
 - load/store
 - masking/sign extension

■ ...

Example: addi R10, R10, 1

- incoming flow: { R10 = <0xff:4> }
- First phase: kill equation for R10
- Second phase:
 - search for equation of R10 in incoming flow
 - alteration of the values found in this equation (add 1 to each additive constant)
- outgoing flow: { R10 = <0xff:4> + [1, 1] }

Transfer: for a recursive loop call

- We want to find invariants for one iteration
- Consequence:
 - we don't want to propagate the flow from the last iteration to the next one
- Solution:
 - we discard the flow-input and add again the initial equations of the loop to the call edge

Transfer: for loop return

- We don't want (and need) to propagate flow outside of loops in the current nesting
- Solution: discard input, propagate empty set of equations

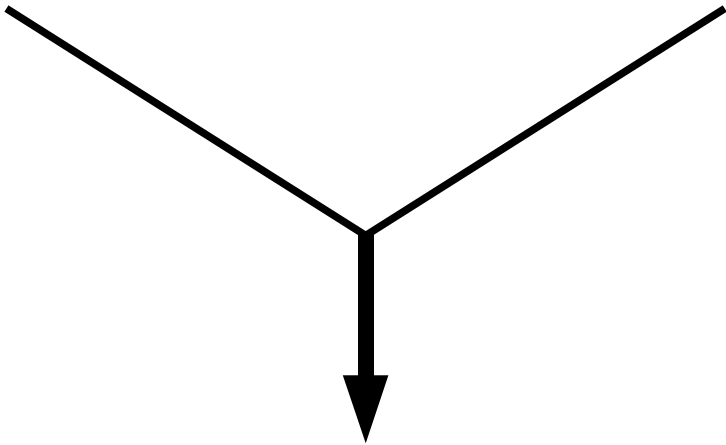
Combination Function

- We aim for safe results, therefore result will only contain:
 - equations for destination variable for which in both given sets already equations existed
 - merged value lists

A Combining Example

{ R10 = <0xff:4> + [1,1],
<0xff:4> = <0xff:4> }

{ R10 = <0xff:4>,
<0xff:4> = <0xff:4>,
R13 = [1,1] }



{ R10 = <0xff:4> + [0,1], <0xff:4> = <0xff:4> }

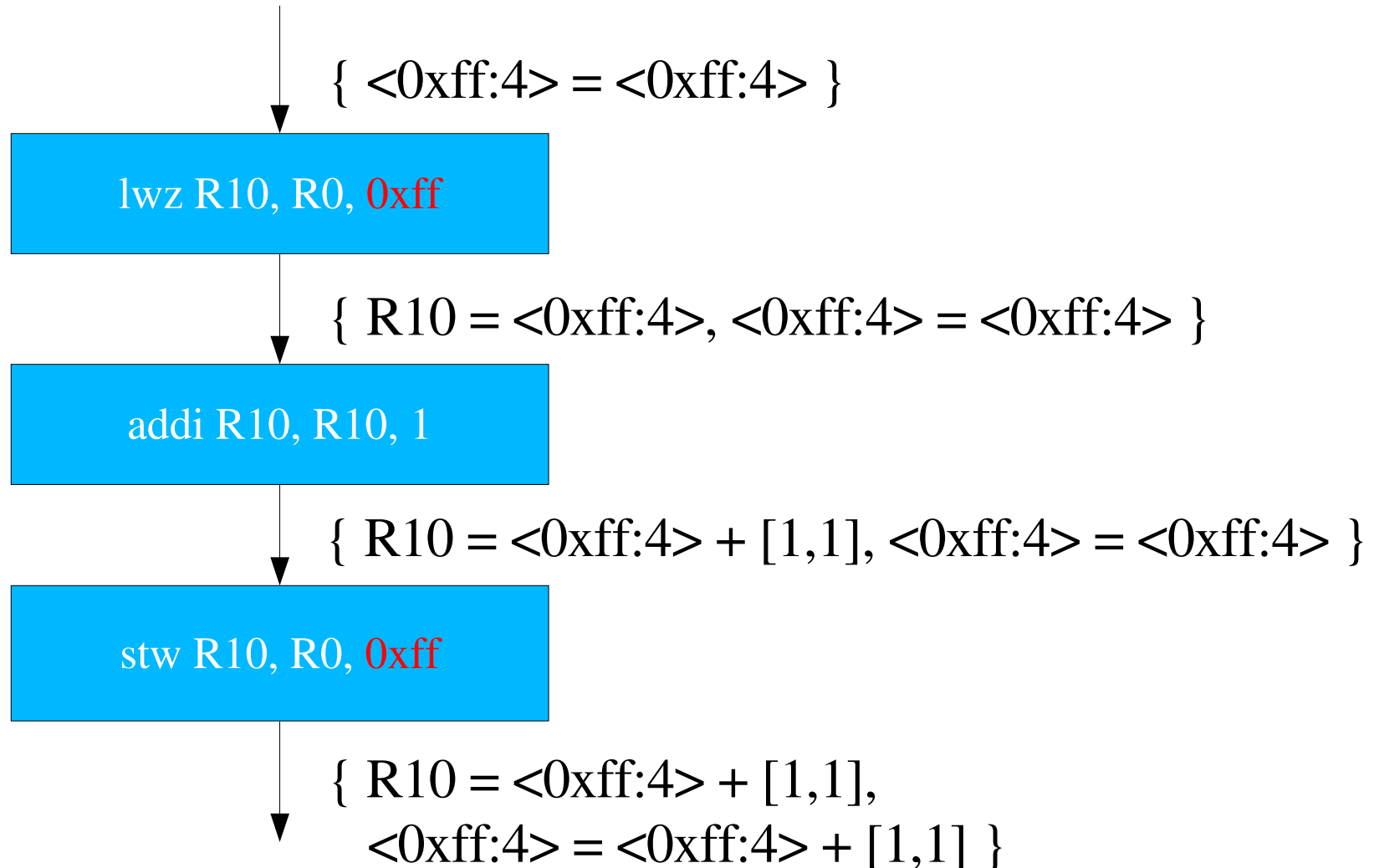
Widening

- Good news:
 - Transfer function monotone
 - Only finite count of destination variables possible
 - Problem:
 - the constants in the values can change per iteration
 - only bound by range of integer
 - Solution:
 - Kill all equations which changed since last iteration, but existed there already
-
-

Widening Example

- Previous iteration:
 $\{ R10 = \langle 0xff:4 \rangle + [0,1], R12 = R12 + [0,1] \}$
- Current iteration:
 $\{ R10 = \langle 0xff:4 \rangle + [0,2], R12 = R12 + [0,1], R13 = [1,1] \}$
- Widening Result:
 $\{ R12 = R12 + [0,1], R13 = [1,1] \}$

Applied to Part of Example



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
 - Two cases implemented for the PPC platform:
 - slice backwards to cmp/cmpi/cmpl/cmpli
 - “zero overhead loop” branch
(decrement counter & branch on zero)
 - Identify variables/constants used in the compare
 - Identify the condition: =, <, >, ...
-
-

Back to example

- Only one exit in the loop, always used:

```
....  
cmpi cr0, R10, 10  
bc cr0.eq
```

- slice backwards to cmpi
- Used variables: R10
- Used constants: 10
- Condition: =

Now: Use the equations

- Get the equations for the variables used in the compare
 - Check if the variable value depends only of one variable + additive constant
 - Check if this variable is a loop counter
(use the equations for this variable which reach the recursive loop call)
-
-

Applied to Example

- variable used in compare: R10
- equations for R10 at the compare:
$$R10 = \langle 0xff:4 \rangle + [1,1]$$
- base variable of R10: $\langle 0xff:4 \rangle$
- equations for $\langle 0xff:4 \rangle$ at recursive call:
$$\langle 0xff:4 \rangle = \langle 0xff:4 \rangle + [1,1]$$

Derive the loop count

- Use the equations we have now
- We get:
 - increment per iteration
 - increment before the compare
 - possible assignment of constant to counter
- If we have equations not matching this pattern, discard the exit, no computation possible in the current implementation


Once more: Example

- Remember:
 - before compare: $R10 = \langle 0xff:4 \rangle + [1,1]$
 - per iteration: $\langle 0xff:4 \rangle = \langle 0xff:4 \rangle + [1,1]$
- increment per iteration:
 - $[1, 1]$
- increment before compare:
 - $[1,1]$

What's still needed?

- Get the start value of the loop counter:
Ask the value analysis!
- Use the knowledge of the width of the variables
Do we work in 8, 16 or 32 bit range?
- Use the knowledge of the signedness
Does the compare type match our variable type?

Our loved Example

- Start value:
[0,0]
 - Range: 32 bit
 - Signedness: signed
- 


Solve the problem

- We have now:
 - start value: $[0, 0]$
 - end value: $[10, 10]$
 - increment per iteration: $[1, 1]$
 - increment before compare: $[1, 1]$
 - Solution: possible iterations: $[10, 10]$
-
-

Combine the bounds

- For each loop: calculate bound for each exit this way
- Combine the results to conservative bounds for the whole loop
- In our example, that would result in:
bounds for “loop”: [10, 10]

What we do now?

- Write out all loop bounds for all contexts
 - Let value analysis read them in, to refine their results
 - Start the next iteration of our analysis, if needed
 - Otherwise: We are done :)
- 

That's it!

- We use now a data flow analysis to get loop invariants
 - It is more generic, no fixed patterns
 - It provides a portable solution
 - First practical tests show that it's speed is usable
-
-

Future work

- Possible Extensions:
 - applying the method to other platforms:
ARM, ...
 - Refining the implementation for:
 - more complex exit checks
 - dealing with loops with multiple first calls
 - allow constant multiplication in the equations
 - use a more powerful equation solver (omega, ...)
-
-

***Thanks & Bon Apetite
Questions?***

