Working around Loops for Infeasible Path Detection in Binary Programs

J. Ruiz, H. Cassé, M. De Michiel

IRIT - University Toulouse III, France

September 17, 2017

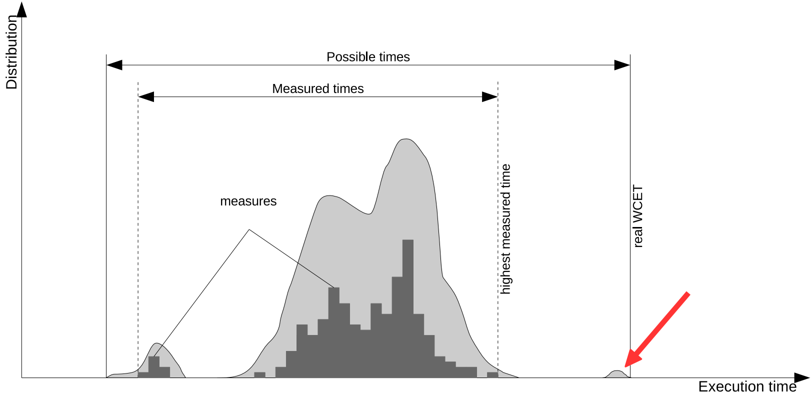
Outline

- Introduction: The infeasible path problem
- Program and machine representation
- Program analysis
- Finding infeasible paths
- Experiments and conclusions

Introduction: The infeasible path problem

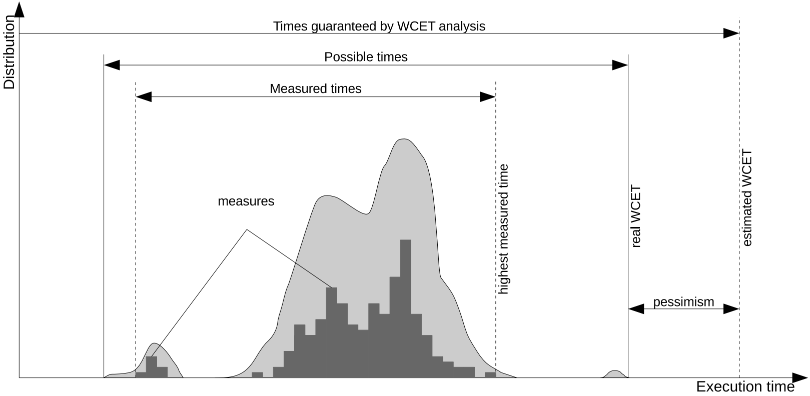
WCET: Worst Case Execution Time

- **WCET analysis** gives a safe upper bound of the execution time of a critical system



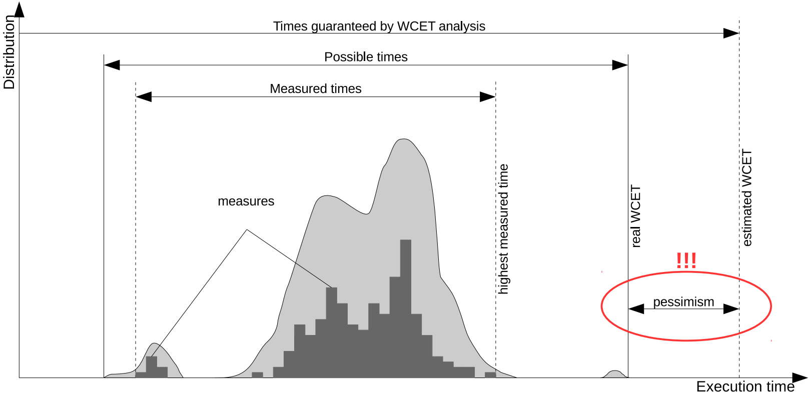
WCET: Worst Case Execution Time

- **WCET analysis** gives a safe upper bound of the execution time of a critical system



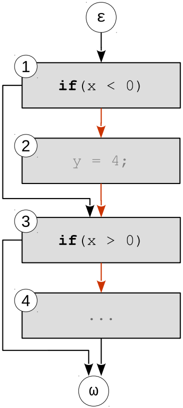
WCET: Worst Case Execution Time

- **WCET analysis** gives a safe upper bound of the execution time of a critical system



Infeasible paths

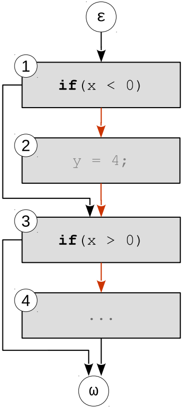
Infeasible paths: A major source of pessimism



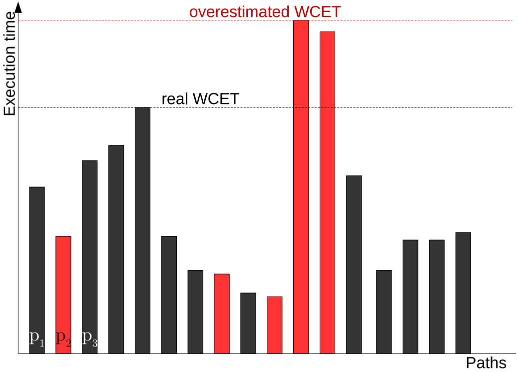
Infeasible paths

Infeasible paths: A major source of pessimism

- Path ① → ② → ③ → ④ is semantically impossible (= infeasible)
- But taken in account in WCET estimation!
- If path is expensive: worsen WCET precision (+ pessimism)



Infeasible paths



Solution: detect infeasible paths

Program and machine representation

Working on binary code

Analyzing binary code is *harder* :

- Low expressivity of machine instructions
- Loosely typed registers
- Obscure structure of the program
- Obscure structure of data in memory

Working on binary code

Analyzing binary code is *harder* :

- × Low expressivity of machine instructions
- × Loosely typed registers
- × Obscure structure of the program
- × Obscure structure of data in memory

but more *reliable* and *efficient*:

- ✓ Compiler independent, does not require compiler certification
- ✓ No transfer of properties from source to binaries
- ✓ Accounts for compiler optimizations

Working on binary code

Analyzing binary code is *harder* :

- × Low expressivity of machine instructions
- × Loosely typed registers
- × Obscure structure of the program
- × Obscure structure of data in memory

but more *reliable* and *efficient*:

- ✓ Compiler independent, does not require compiler certification
- ✓ No transfer of properties from source to binaries
- ✓ Accounts for compiler optimizations

although

- × Architecture-dependent?

Working on binary code

× Architecture-dependent? **No!**

Translate to and work on *semantic instructions*

- a RISC-like instruction set of abstract instructions

```
ADD r1, r3, #1
LDR r3, [r11, #-8]
REV r1, r0
```

```
ADD r1, r3, #1
  | seti t1, 1
  | add r1, r3, t1
LDR r3, [r11, #-8]
  | seti t2, -8
  | add t1, r11, t2
  | load r3, t1
REV r1, r0
  | scratch r1
```

Representing the state of the machine

A program state is a set of possible maps of each register and memory cell to 32-bit values:

$$\begin{array}{rcl|rcl|rcl}
r_0 & \mapsto & 0 & r_0 & \mapsto & 0 & r_0 & \mapsto & 0 \\
\mathbf{r_1} & \mapsto & \mathbf{1} & \mathbf{r_1} & \mapsto & \mathbf{2} & \mathbf{r_1} & \mapsto & \mathbf{3} \\
r_2 & \mapsto & 22 & r_2 & \mapsto & 22 & r_2 & \mapsto & 22 \\
 & \ldots & & & \ldots & & & \ldots & \\
r_{15} & \mapsto & -234 & r_{15} & \mapsto & -234 & r_{15} & \mapsto & -234 \\
 & \ldots & & & \ldots & & & \ldots & \\
[\mathrm{0x8000}] & \mapsto & -1 & [\mathrm{0x8000}] & \mapsto & -1 & [\mathrm{0x8000}] & \mapsto & -1 \\
[\mathrm{0x8004}] & \mapsto & 64 & [\mathrm{0x8004}] & \mapsto & 64 & [\mathrm{0x8004}] & \mapsto & 64 \\
 & \ldots & & & \ldots & & & \ldots &
\end{array}$$

Abstracting program states

We express abstract states in function of an initial program state

Registers	r_0	0
	r_1	$2 \times r_1^*$
	r_2	$r_2^* + r_0^*$
	$\dots$	
	r_{15}	$\top$
Memory	$\dots$	
	[0x8000]	[0x8000]*
	[0x8004]	64
	$\dots$	
Predicates	$r_1 < 10$	
	$r_2 = 2.r_1$	

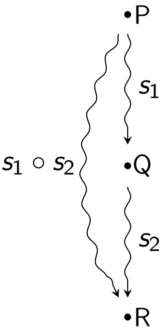
- $\top$ represents any value (safe approximation)
- v^* is the initial value of v, at the beginning of the analysis

Abstracting program states

We express abstract states in function of an initial program state

Abstract states represent the execution of a code segment

They can be composed by an operator $\circ$.

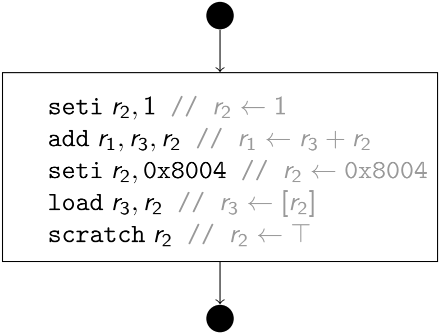


Program analysis

On nodes

CFG nodes are sequences of instructions $d \leftarrow f(a, b)$.
For each instruction, we update any state s such that $s(d) = f(s(a), s(b))$

Registers	r_0	r_0^*
	r_1	r_1^*
	r_2	r_2^*
	r_3	r_3^*
	...	
Memory	...	
	[0x8000]	$[0x8000]^*$
	[0x8004]	$[0x8004]^*$
	...	
Predicates		

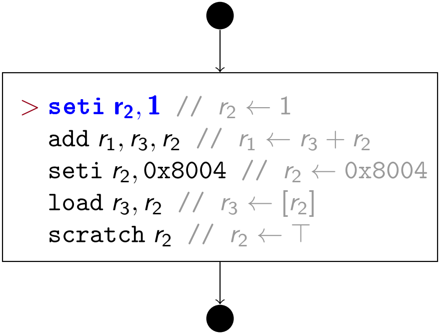


`seti` $r_2, 1$ // $r_2 \leftarrow 1$
`add` r_1, r_3, r_2 // $r_1 \leftarrow r_3 + r_2$
`seti` r_2, `0x8004` // $r_2 \leftarrow$ `0x8004`
`load` r_3, r_2 // $r_3 \leftarrow [r_2]$
`scratch` r_2 // $r_2 \leftarrow \top$

On nodes

CFG nodes are sequences of instructions $d \leftarrow f(a, b)$.
For each instruction, we update any state s such that $s(d) = f(s(a), s(b))$

Registers	r_0	r_0^*
	r_1	r_1^*
	r_2	**1**
	r_3	r_3^*
	...	
Memory	...	
	[0x8000]	$[0x8000]^*$
	[0x8004]	$[0x8004]^*$
	...	
Predicates		

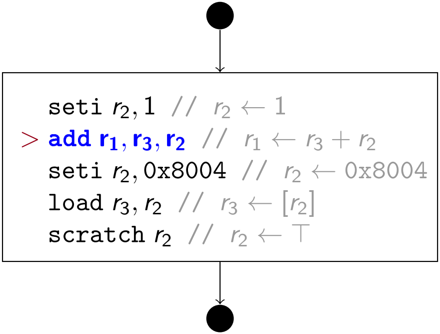


```
> seti r2, 1 // r2 ← 1
  add r1, r3, r2 // r1 ← r3 + r2
  seti r2, 0x8004 // r2 ← 0x8004
  load r3, r2 // r3 ← [r2]
  scratch r2 // r2 ← ⊤
```

On nodes

CFG nodes are sequences of instructions $d \leftarrow f(a, b)$.
For each instruction, we update any state s such that
$s(d) = f(s(a), s(b))$

Registers	r_0	r_0^*
	r_1	$\mathbf{r_3^* + 1}$
	r_2	1
	r_3	r_3^*
	...	
Memory	...	
	[0x8000]	[0x8000]*
	[0x8004]	[0x8004]*
	...	
Predicates		

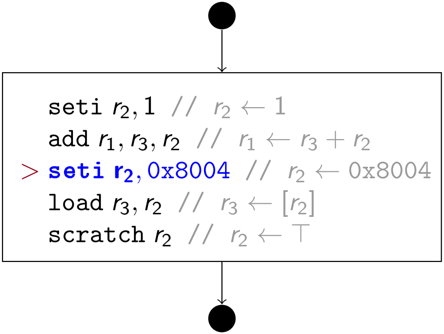


seti $r_2, 1$ // $r_2 \leftarrow 1$
> **add $\mathbf{r_1, r_3, r_2}$** // $r_1 \leftarrow r_3 + r_2$
seti r_2, 0x8004 // $r_2 \leftarrow$ 0x8004
load r_3, r_2 // $r_3 \leftarrow [r_2]$
scratch r_2 // $r_2 \leftarrow \top$

On nodes

CFG nodes are sequences of instructions $d \leftarrow f(a, b)$.
For each instruction, we update any state s such that $s(d) = f(s(a), s(b))$

Registers	r_0	r_0^*
	r_1	$r_3^* + 1$
	r_2	0x8004
	r_3	r_3^*
	. . .	
Memory	. . .	
	[0x8000]	$[0x8000]^*$
	[0x8004]	$[0x8004]^*$
	. . .	
Predicates		

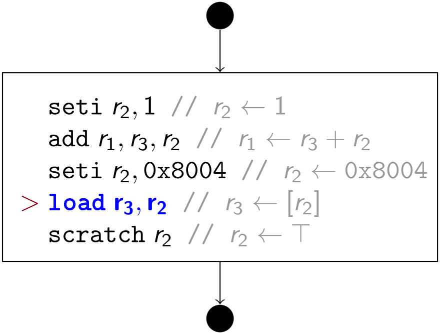


seti $r_2, 1$ // $r_2 \leftarrow 1$
add r_1, r_3, r_2 // $r_1 \leftarrow r_3 + r_2$
> **seti r_2**, 0x8004 // $r_2 \leftarrow$ 0x8004
load r_3, r_2 // $r_3 \leftarrow [r_2]$
scratch r_2 // $r_2 \leftarrow \top$

On nodes

CFG nodes are sequences of instructions $d \leftarrow f(a, b)$.
For each instruction, we update any state s such that $s(d) = f(s(a), s(b))$

Registers	r_0	r_0^*
	r_1	$r_3^* + 1$
	r_2	0x8004
	r_3	**[0x8004]***
	. . .	
Memory	. . .	
	[0x8000]	$[0x8000]^*$
	[0x8004]	$[0x8004]^*$
	. . .	
Predicates		

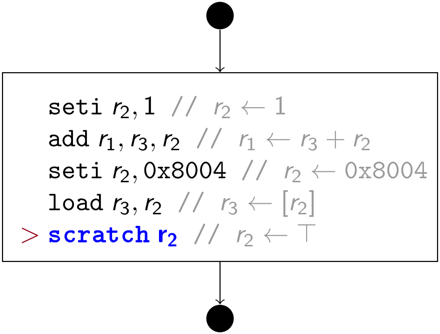


seti $r_2, 1$ // $r_2 \leftarrow 1$
add r_1, r_3, r_2 // $r_1 \leftarrow r_3 + r_2$
seti r_2, 0x8004 // $r_2 \leftarrow$ 0x8004
> **load r_3, r_2** // $r_3 \leftarrow [r_2]$
scratch r_2 // $r_2 \leftarrow \top$

On nodes

CFG nodes are sequences of instructions $d \leftarrow f(a, b)$.
For each instruction, we update any state s such that $s(d) = f(s(a), s(b))$

Registers	r_0	r_0^*
	r_1	$r_3^* + 1$
	r_2	$\top$
	r_3	$[0x8004]^*$
	…	
Memory	…	
	[0x8000]	$[0x8000]^*$
	[0x8004]	$[0x8004]^*$
	…	
Predicates		

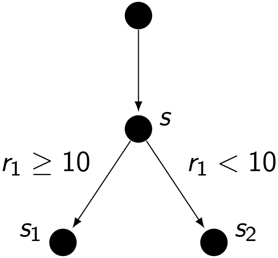


seti r_2, 1 // $r_2 \leftarrow 1$
add r_1, r_3, r_2 // $r_1 \leftarrow r_3 + r_2$
seti r_2, 0x8004 // $r_2 \leftarrow$ 0x8004
load r_3, r_2 // $r_3 \leftarrow [r_2]$
> **scratch r_2** // $r_2 \leftarrow \top$

On forks

Duplicate state s in two states, one for each path :

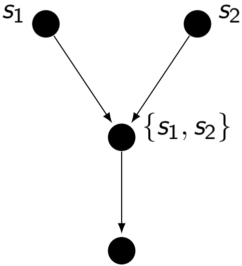
- s_1 is s with the added predicate $r_1 \geq 10$
- s_2 is s with the added predicate $r_1 < 10$



On joins

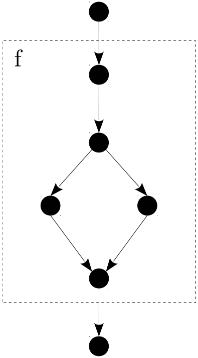
Keep the states from both paths

(sometimes shrink into one to deal with the complexity)



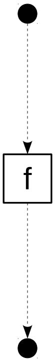
On function calls

- Analyze each function only once
- On call, compose with the state(s) issued from the called function
  $s \Longrightarrow s_f \circ s$



On function calls

- Analyze each function only once
- On call, compose with the state(s) issued from the called function
  $s \Longrightarrow s_f \circ s$



Loops

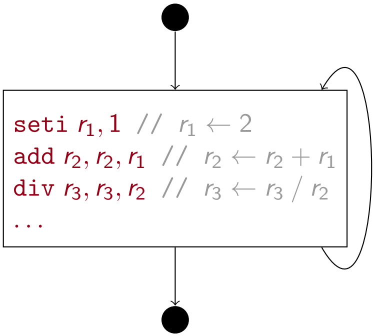
- Common point between loop bodies and functions:
  they are Single Entry Single Exit (SESE) regions
- **Reuse state composability** for each loop h:
  - *process* the body of h *once*, separately, resulting in s_h
  - it is a function, we can compute $(s_h)^n$, the effect of n iterations
  - we now know the state of the program *for any iteration n*

Loops

Example for a loop h:

s_h	r_0	r_0^*
	r_1	2
	r_2	$r_2^* + 2$
	r_3	r_3^* / r_2^*

$(s_h)^n$	r_0	r_0^*
	r_1	2
	r_2	$r_2^* + 2\mathbf{n}$
	r_3	$\top$



seti $r_1, 1$ // $r_1 \leftarrow 2$

add r_2, r_2, r_1 // $r_2 \leftarrow r_2 + r_1$

div r_3, r_3, r_2 // $r_3 \leftarrow r_3 / r_2$

...

Finding infeasible paths

SMT solving

Evaluate the satisfiability of a system?

- Straight forward with SMT (SAT Modulo Theory) solvers
- Two possible answers:
  - "SAT" ⇒ the path represented by the state is feasible
  - "UNSAT" ⇒ the path represented by the state is infeasible

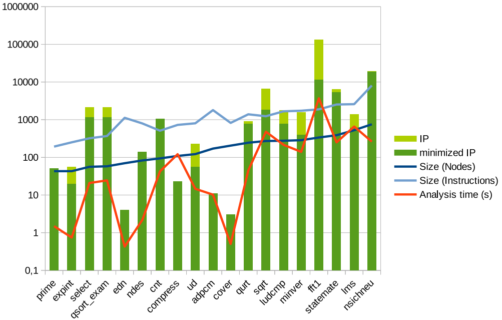
Tightening the WCET estimation

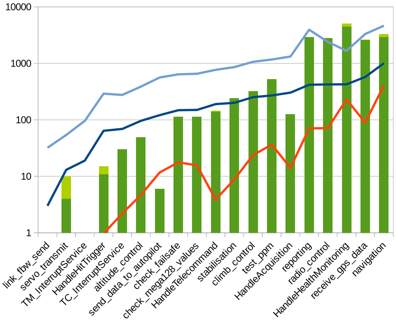
- WCET estimation computed by a system linear constraints

  "ILP system"
- inject infeasible paths as additional data flow constraints
- WCET estimation *may* be reduced (if the path of the WCET was infeasible)

Experiments and conclusions

Experimental results: Infeasible paths

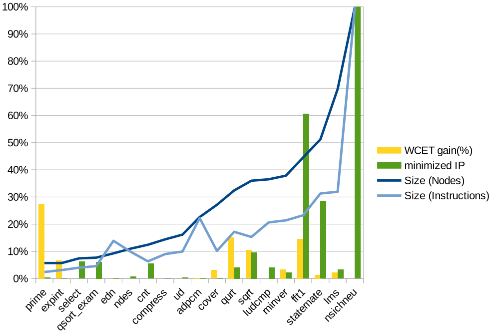




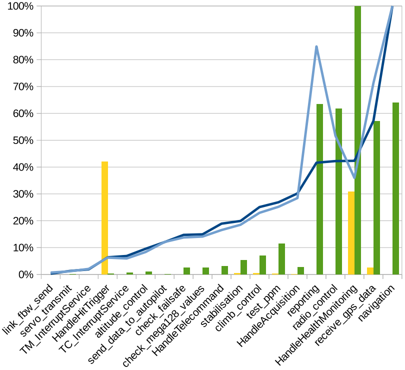
Mälardalen

Debie, PapaBench

Experimental results: WCET gain



Mälardalen



Debie, PapaBench

Conclusion

- Results are very variable and unpredictable
  - improvement is often negligible ($< 0.1\%$)
  - but sometimes important ($10 - 40\%$)
- Precise abstraction of the program is important
- Analysis scales reasonably
  - Limiting SMT calls is key