The Auspicious Couple: Symbolic Execution and WCET Analysis

Armin Biere, Jens Knoop, Laura Kovács, *Jakob Zwirchmayr*

TU Vienna, JKU Linz

July 9, 2013

WCET Analysis

WCET Analysis

- mandatory for safety-critical real-time systems

Computed WCET bounds

- must be safe
- shall be tight

Problem

- precise knowledge about the program

Symbolic Execution

Symbolic Execution

- use symbolic instead of concrete data

Control-flow split (branch)

- follow both paths
- assume respective condition

Problem: path explosion

- unbounded loops
- number of conditionals

Our Remedy

Combine symbolic execution and WCET analysis as a remedy

WCET analysis guides symbolic execution

- select only WCET relevant parts

Symbolic execution infers precise information

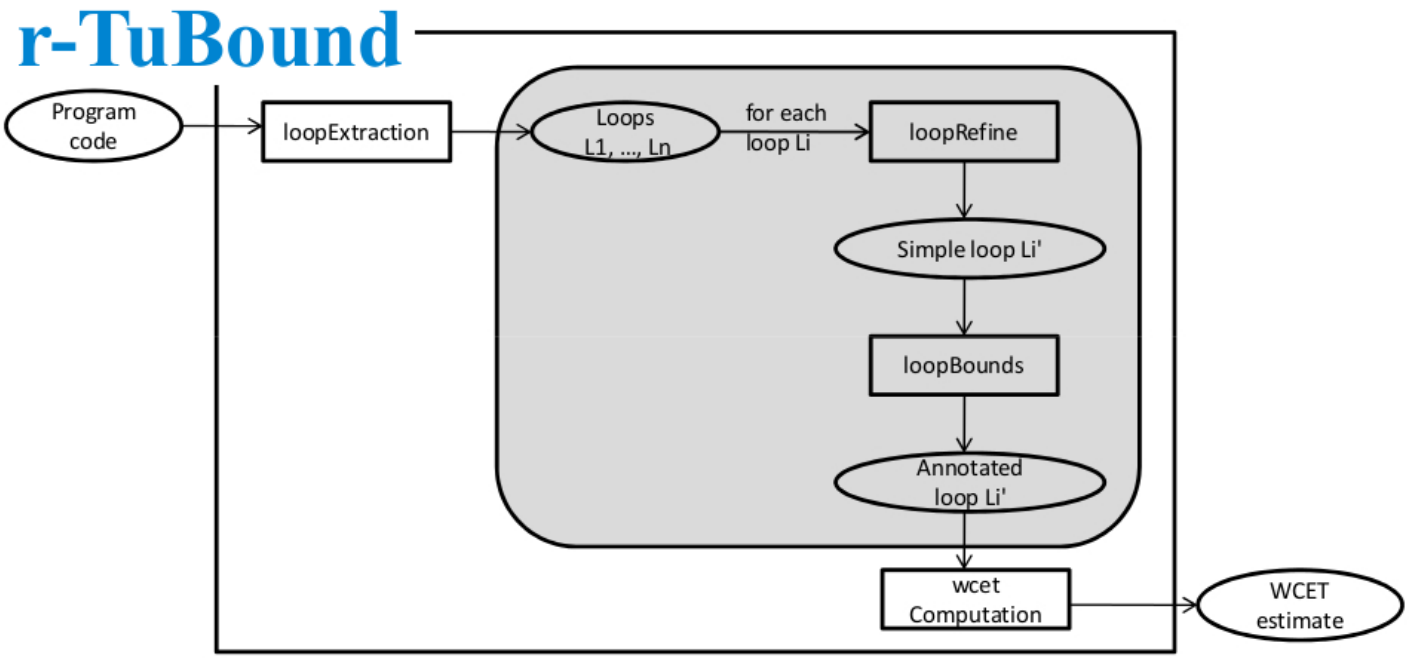
- for relevant parts

Partial vs full symbolic coverage

- full symbolic coverage often infeasible in practice

Partial coverage often good enough to improve the WCET estimate

r-TuBound


r-TuBound
Program code
loopExtraction
Loops L1, ..., Ln
for each loop Li
loopRefine
Simple loop Li'
loopBounds
Annotated loop Li'
wcet Computation
WCET estimate

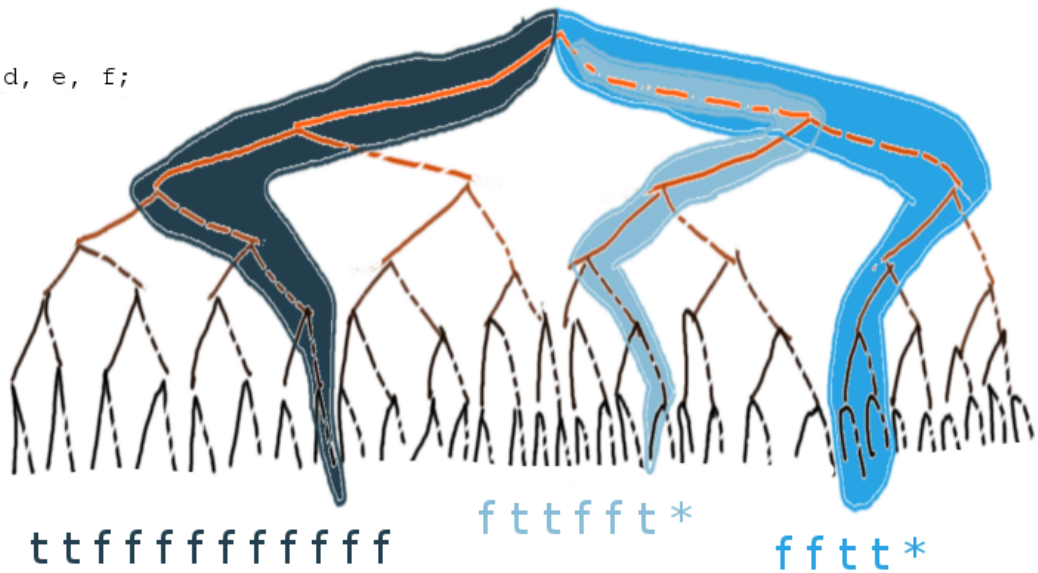
Symbolic Execution: SmacC

SmacC

- SMT representation of the program (BV, arrays)
- select paths via path-expressions

Exact analysis

```
int main ()
{
  int cnt = 0;
  int a, b, c, d, e, f;

  if (a)
    cnt++;

  if (b)
    cnt++;

  if (c)
    cnt++;

  if (d)
    cnt++;

  if (e)
    cnt++;

  if (f)
    cnt++;

  return cnt;
}
```



Full symbolic coverage requires execution of all paths!

Symbolic Execution in r-TuBound

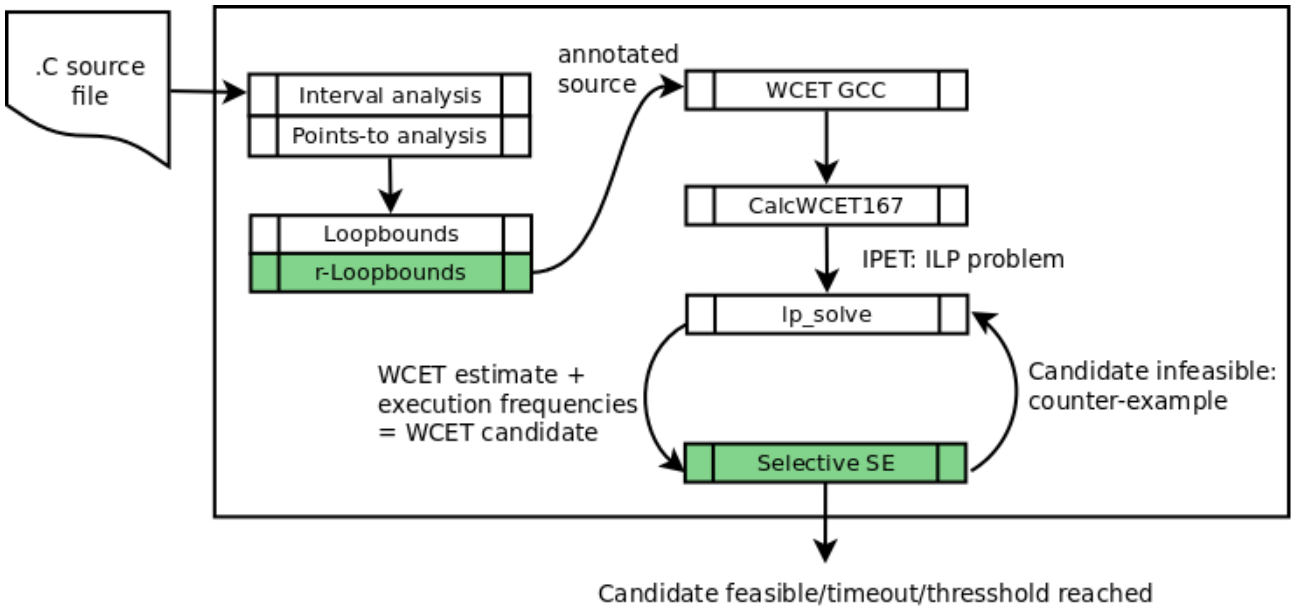
1) on selected program fragments
   - check properties on conditional updates to the loopcounter
   - if successful, loop bound computation safe

2) on single loops
   - only if all other techniques fail

3) on single paths
   - as post-process, after initial WCET anlysis
   - symbolically check feasibility of WCET path

= Selective Symbolic Execution

Architecture



Analyzing Program Fragments

Conditional update to loop counter `i` prevents bound calculation

```
int main (int flag) {
  int i;
  for (i = 0; i < 5; i++)
    if (i == 4 && flag) {
      i = 0;
      flag = 0;
    }
}
```

- verify that updates strictly increase(decrease) `i`
- can check arbitrary expressions (in bitvectors/array theory)

Success

- apply bound computation
- (combined minimal update)

Fails for example

Loop Bounds via Symbolic Execution

(r-)loopbounds fails to compute a loop bound

only then

- apply exhaustive symbolic execution of the loop

The loop + required decls + additional analysis information

- = reduced program
- example: program = reduced program

Symbolically execute reduced program

- with initial bound 0
- increase bound while loop cond is SAT in last iteration

Example: loop bound 9

Precise WCET Bounds

a.k.a WCET Squeezing

- post-proces for IPET based WCET analyzer
- allows to tighten WCET estimates
- ultimately prove WCET bounds precise

Is a combination of WCET analysis and symbolic execution

- overcomes problems inherent in both approaches!

Problems of the Approaches

Symbolic Execution deficiency: path explosion
(doesn't scale due to exponential number of paths)

IPET deficiency: considers little information about the program
(flow-facts)

Some Remedy

Combine IPET and Symbolic Execution for mutual benefit!

extract path from ILP result and symbolically execute it

Some Remedy

Combine IPET and Symbolic Execution for mutual benefit!

extract path from ILP result and symbolically execute it

Path explosion:

- less severe, initially examine only one path

Lack of information:

- rule out infeasible paths using precise symbolic execution
- by deriving new ILP constraints

Requires an initial WCET analysis

Squeezing in a Nutshell

in: ILP problem (from IPET), **out** wcet bound

1. solve ILP problem
2. extract "abstract" WCET path candidates from ILP
3. compute "concrete" path(s) encoded by abstract path
4. symbolically execute concrete path(s)
5. use result of execution to refine ILP problem or stop:
   - 5.1 path feasible: done (path is indeed WCET path)
   - 5.2 infeasible: refine ILP, goto 2

On termination:

- precise WCET bound (wrt the HW-model)
- optional: timeout, threshhold

Expected Results

Refined WC path is feasible:

- real WCET-path, overestimation due to hardware modelling
- precise bound

Refined WC path is infeasible + TO:

- some improvement after a few iteration
- estimate tightened

ILP WC path is feasible:

- no gain in precision
- precise bound

Example

Loop bound = 9

- analyze example with r-TuBound
- yields WCET estimate + ILP solution
  (computed from generated ILP problem)

ILP

- problem: constraints on execution frequencies
- solution: valid execution frequencies of blocks
- example: execution frequency of `then`-block = 9

The solution is INFEASIBLE

- no such concrete execution exists
- therefore, WCET bound is an over-estimation

Example contd.

ILP solution "executes" `then`-block 9 times

Path-expression: `tttttttttt` (for the `if`-cond)

- i.e. iteration 0 to 9 execute `then`-branch
- i.e. conditional evaluates to `true` 9 times

```
int main (int flag) {
  int i;
  for (i = 0; i < 5; i++)
    if (i == 4 && flag) {
      i = 0;
      flag = 0;
    }
}
```

- path-expression can be constructed from ILP
- specifies a symbolic execution of the program

Result of symbolic execution defines further steps

- feasible path: terminate, bound precise
- infeasible path: derive ILP constraint to exclude the path, recompute WCET bound

Example contd.

Squeezing the example:

- initially, 9 times `true`-block

Iteration 1:

- path infeasible, add constraint to exclude this path
- results in tighter solution

. . . Iteration 8 (terminating iteration):

- exact execution frequency inferred
- `true`-block once!

Constraints

Syntactic:

- comparable to a graph transformation
- introduces new ILP variables

Semantic:

- constraint over combined execution frequency of involved conditions
- requires more symbolic executions

Combination seems most effective

- peel a loop iteration (syntactic)
- constrain combined execution frequency up to peeled iteration

Discussion

Squeezing can be stopped at any time (time-limit)

Squeezing can be run until a specified improvement is observed

Relies on Partial symbolic coverage
full symbolic coverage if all but one paths are infeasible

Discussion

Squeezing can be stopped at any time (time-limit)

Squeezing can be run until a specified improvement is observed

Relies on Partial symbolic coverage
full symbolic coverage if all but one paths are infeasible

All 3 presented approaches

- are effective
- require only partial symbolic coverage

Further Applications

All 3 based on a similar symbolic execution infrastructure

- again, requiring only partial symbolic coverage

Further Applications

All 3 based on a similar symbolic execution infrastructure

- again, requiring only partial symbolic coverage

Precise execution frequencies for loops

- iff symbolice execution is applied to find a loop bound

$\Rightarrow$ cheap to track execution frequencies

Further Applications

All 3 based on a similar symbolic execution infrastructure

- again, requiring only partial symbolic coverage

Precise execution frequencies for loops

- iff symbolice execution is applied to find a loop bound

$\Rightarrow$ cheap to track execution frequencies

WCET path test-cases

- instantiate symbolic (input) variables (in SAT case)

$\Rightarrow$ use concrete values as test input

Further Applications

All 3 based on a similar symbolic execution infrastructure

- again, requiring only partial symbolic coverage

Precise execution frequencies for loops

- iff symbolice execution is applied to find a loop bound

$\Rightarrow$ cheap to track execution frequencies

WCET path test-cases

- instantiate symbolic (input) variables (in SAT case)

$\Rightarrow$ use concrete values as test input

Mode-sensitive WCET analysis

- modify program after WCET analysis

$\Rightarrow$ squeezing to automatically adapt IPET constraints accordingly

Conclusion

Symbolic execution is a powerful program analysis technique

- ... that "doesn't scale"
- and therefore shouldn't rely on full symbolic coverage

WCET analysis requires precise information about the program

- ... its result can serve as a selection mechanism
- to guide symbolic execution towards relevant program parts

The combination of symbolic execution is promising:

- WCET guidance is a remedy to the path explosion problem
- while symbolic execution helps inferring tighter bounds

$\Rightarrow$ 3 approaches successfully implemented in r-TuBound

$\Rightarrow$ 3 new approaches that can make use of the infrastructure

Thanks for your Attention!

Questions?

Examples - Binary Search

"Precise" analysis, e.g. array content

1) Mälardalen, `bs.c`, all data initialized
   - from theoretic WC path of `bs` to real WC execution path

2) Mälardalen, `bs.c` modified, some data initialized
   - from theoretic WC path of `bs` to a better WC path

3) Mälardalen, `bs.c` modified, no data initialized
   - theoretic WC path of `bs` is real WC path

Case 1 – Original Benchmark

```
int keys[15], vals[15];
int bs() {
  int x = 8, fvalue = -1, mid, up = 14, low = 0;
  keys = { ... }; // ALL initialized
  vals = { ... }; // ALL initialized
  while ( low <= up ) {
    mid = (low + up) >> 1;
    if ( keys[mid] == x ) {
      up = low - 1;
      fvalue = vals[mid];
    } else if ( keys[mid] > x )
      up = mid - 1;
    else low = mid + 1;
  }
  return fvalue;
}
```

Case 1 – Path Expression

by IPET: `L1L1L1L1L1L1L1T`

- `L/T` ... loop condition holds/does not hold
- `1,2,3` ... conditional blocks executed
- `?` ... one of the conditional blocks executed

infeasible!

feasible execution by SE: **L2L3L2L1T**

Case 2 – Modified Benchmark

```
...
int bs() {
    ...
  keys = { ... }; // SOME initialized
  vals = { ... }; // SOME initialized
  ...
}
```

PE by IPET: `L1L1L1L1L1L1L1T` infeasible!

feasible execution by SE: `L2L3L?L?L?T`, hence select

L2L3L1L1L1T

Case 3 – Modified Benchmark

```
...
int bs() {
    ...
  keys = { ... }; // ALL uninitialized
  vals = { ... }; // ALL uninitialized
  ...
}
```

PE by IPET: **L1L1L1L1L1L1L1L1T** feasible by SE!

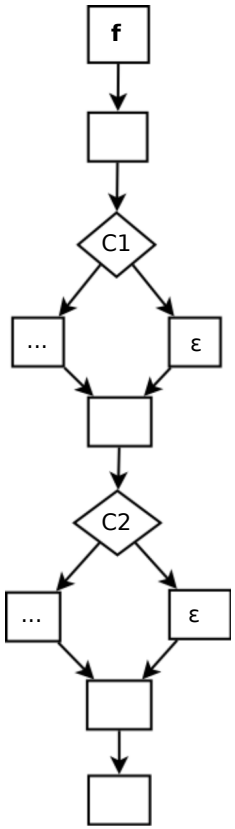
hence select **L1L1L1L1L1L1L1L1T**

Naive Refinement – Graph Level

Split decision nodes

- double part of the CFG to rule out one path
- might blow up ILP problem

Example 2

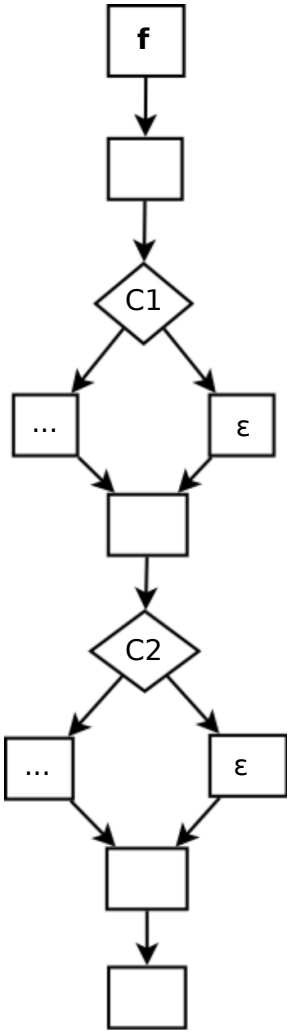



```
void f () {
  if(C1) {...}
  if(C2) {...}
}
```

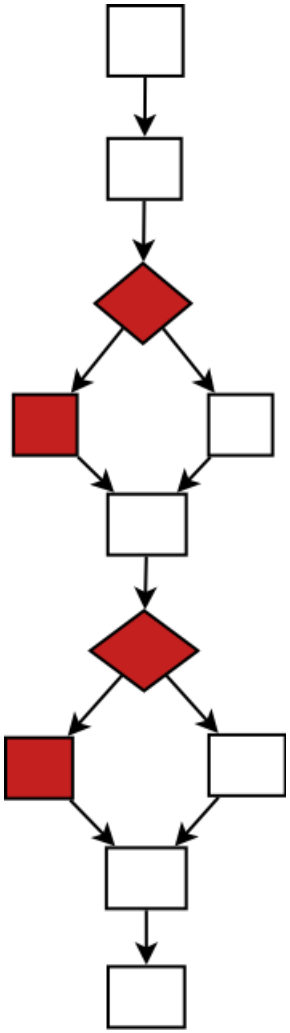
1
2
3
4

modes of operation: mutually exclusive execution parts

Graph Refinement I: Extract WC Path

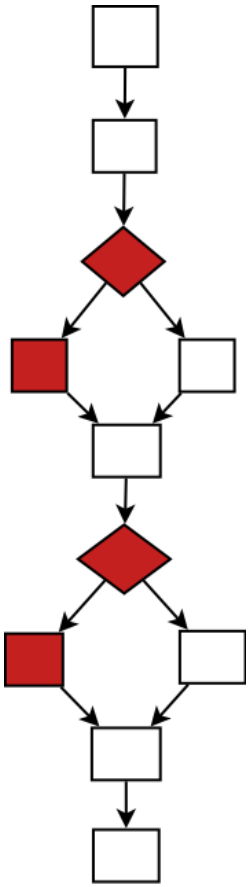


IPET ⇒
PE: 11

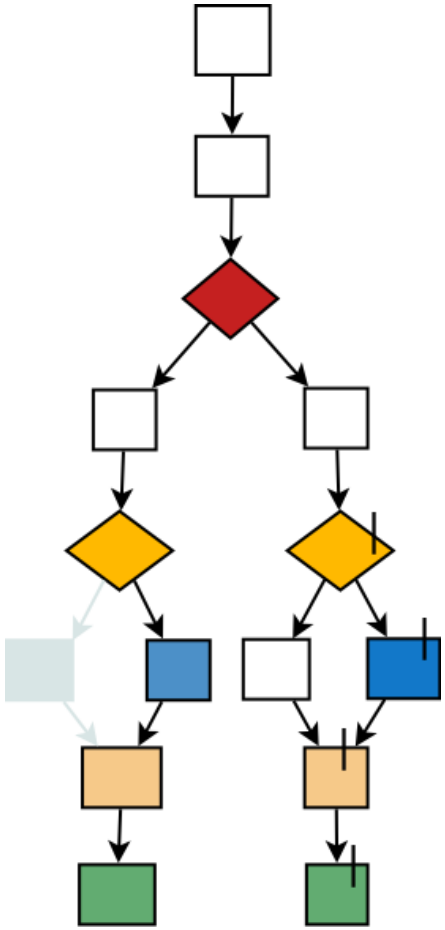


Construct **PE** from IPET solution

Graph Refinement II: Split WC Path



SE $\Rightarrow$ infeasible



Symbolically execute PE, **split** if execution is **infeasible**.

ILP Refinement: ILP Encoding

Step 1: as in graph approach (extract WC path)

Step 2: infeasible path $\Rightarrow$ ILP path constraint

$$
\begin{array}{l}
N = 1 \quad // \text{ entry} \\
b1 \leq N \\
c1 \leq b1 \\
t1 + f1 \leq c1 \\
b2 \leq c1 \\
c2 \leq b2 \\
t2 + f2 \leq c2 \\
b3 \leq c2 \\
X \leq b3 \\
X \leq 1 \quad // \text{ exit}
\end{array}
$$

restriction:

$$\mathbf{!(t1 \wedge t2)}$$

$$
\begin{array}{l}
N = 1 \\
b1 \leq N \\
c1 \leq b1 \\
t1 + f1 \leq c1 \\
b2 \leq c1 \\
c2 \leq b2 \\
t2 + f2 \leq c2 \\
b3 \leq c2 \\
X \leq b3 \\
X \leq 1 \\
\mathbf{t1 + t2 \leq 1}
\end{array}
$$

Step 3: add constraint and restart ILP solver.

$\Rightarrow$ better WCET estimate.

ILP Refinement: Loop Blocks

Alternative 1: peel loop, reduce frequency, add path constraint.

Alternative 2: path constraint including conditional block.

```
N = 1;
c1 = N;
t1 + f1 = c1;
loopHead = 1;
loopBody =
    loopHead *
    5;
loopBody = t2 +
    f2;
loopExit =
    loopHead;
X = loopExit;
```

restriction:
!(t1 ∧ 5t2)

```
N <= 1;
c1 <= N;
t1 + f1 <= c1;
loopHead <= 1;
loopBody <=
    loopHead *
    5;
loopBody <= t2 +
    f2;
loopExit <=
    loopHead;
t1 + t2 ≤ 5
X = loopExit;
```

Add constraint and restart ILP solver.

tb;dl

DdFR: Demand-driven Feasibility Refinement to automatically

- refine WCET estimates on-demand, by
- selective symbolic execution and
- automatic construction of restrictive ILP clauses

that constrain the ILP search for a new WCET candidate.

Encoding of Counterexamples

A counterexample is:

- infeasibility of a WC path

Control-flow graph level:

- naive, inefficient
- program transformation

ILP level:

- more efficient
- force solver to generate different solution

Further Benefits & Possibilities

Allows for and supports additional analyses

- before flow-fact computation (whole program)
- after WC path analysis (path)

Allows for combination with other approaches

- *criticality* – [Brandner/Hepp/Jordan, RTNS12]
- test-case generation – measurements on WC path
- concolic execution

Current State

WC Path extraction: automatic

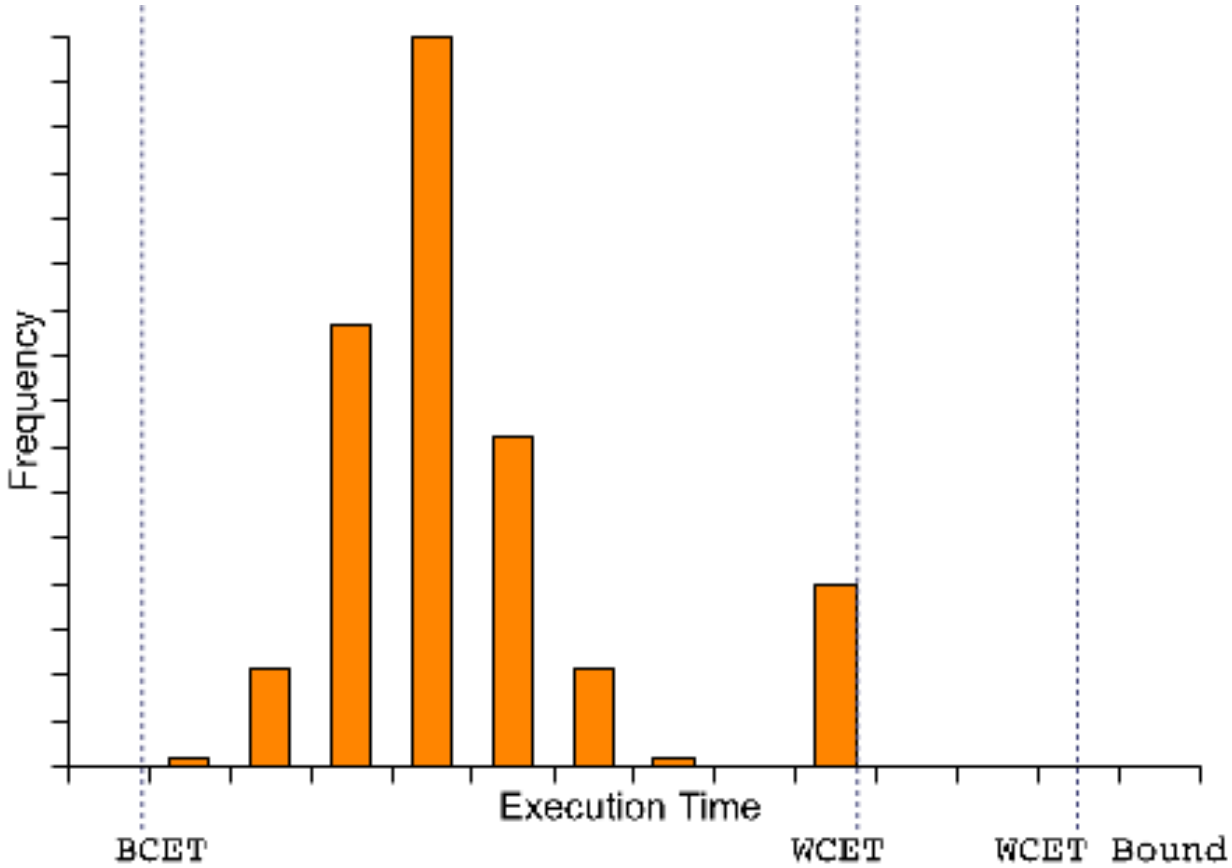
- modified `CalcWCET167`

Symbolic Execution: automatic

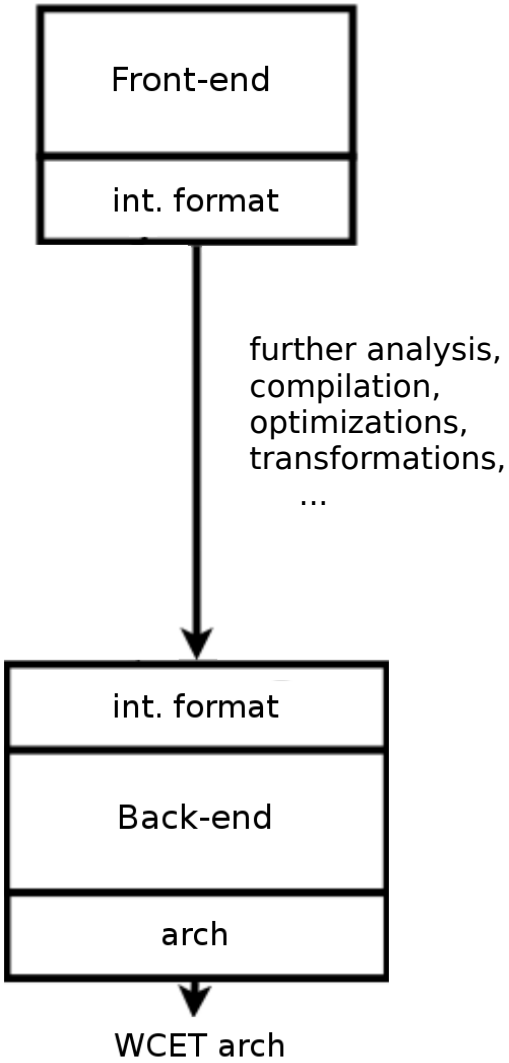
- preliminary, `SmacC` (C subset)
- PE translation needed, manual

Path generation / counterexample encoding: manual

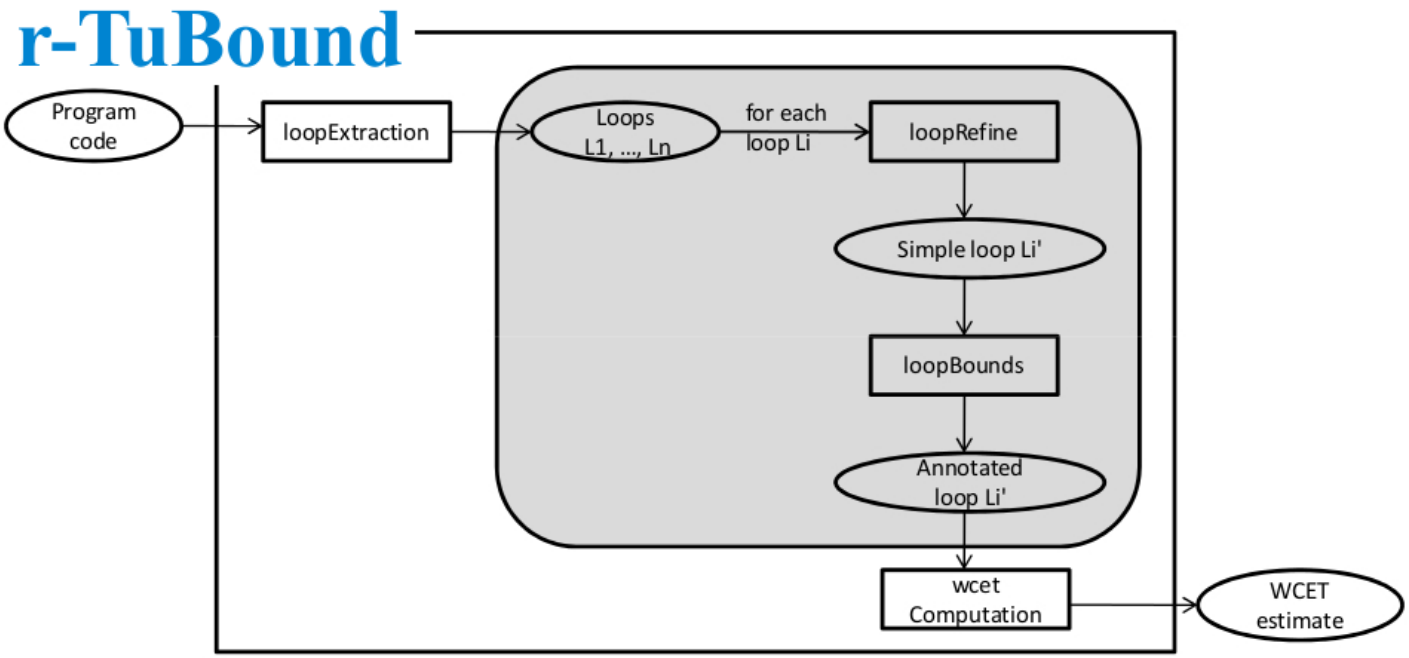
WCET Bound



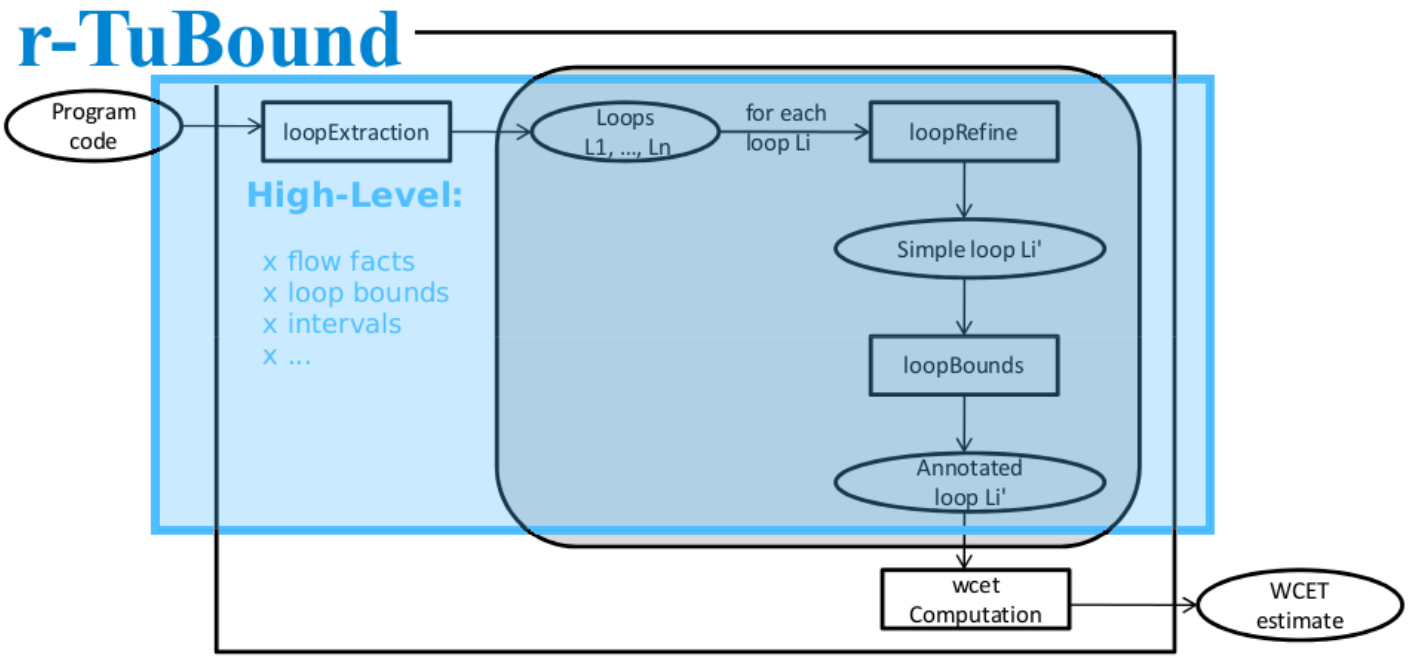
WCET Analysis Tool Chain



r-TuBound


r-TuBound
Program code
loopExtraction
Loops L1, ..., Ln
for each loop Li
loopRefine
Simple loop Li'
loopBounds
Annotated loop Li'
wcet Computation
WCET estimate

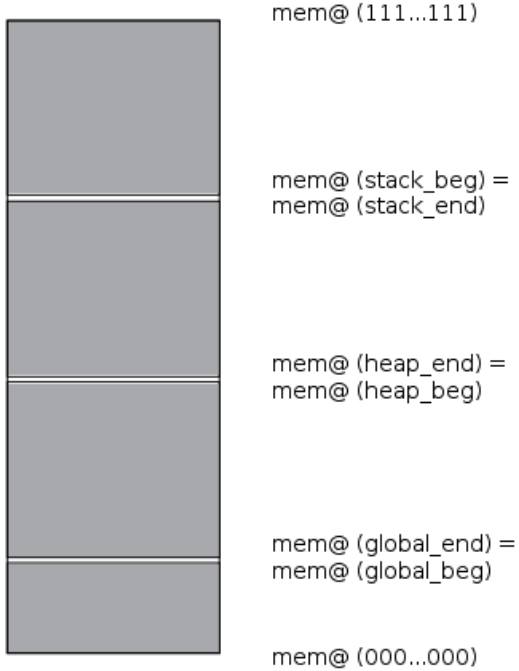
r-TuBound


r-TuBound
Program code
loopExtraction
Loops L1, ..., Ln
for each loop Li
loopRefine
High-Level:
x flow facts
x loop bounds
x intervals
x ...
Simple loop Li'
loopBounds
Annotated loop Li'
wcet Computation
WCET estimate

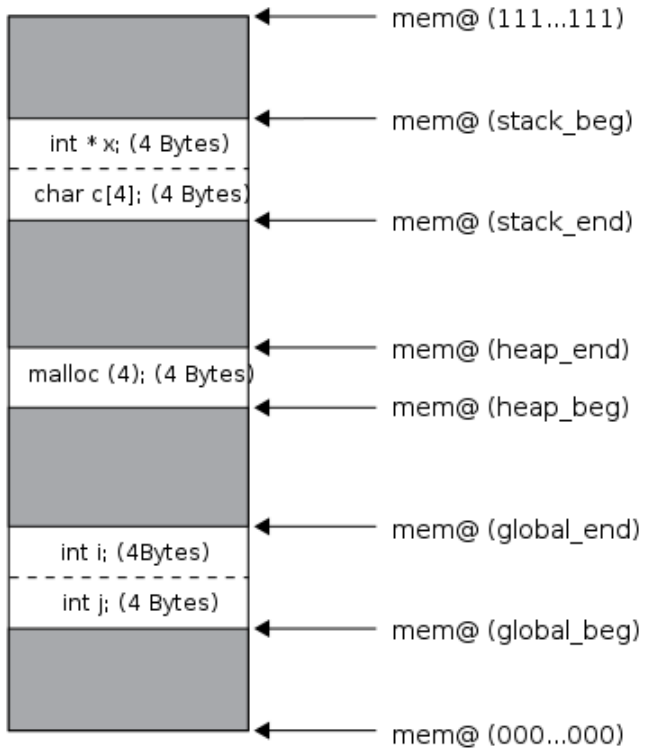
Memory Model

- BTOR array/variables: memory of program / address mem
- special variables define valid memory

no variables declared



variables `x`, `c`, `i`, `j`, `malloc(4)`



Assertion Check

```
void main () { int i; assert (i); }
```

BTOR variables:

$\{mem, global\_beg, global\_end, heap\_beg, heap\_end, stack\_beg, stack\_end, i\}$

Memory assumptions:

$$global\_beg \leq global\_end \wedge global\_end < heap\_beg \wedge$$
$$heap\_beg \leq heap\_end \wedge heap\_end < stack\_end \wedge$$
$$stack\_end \leq stack\_beg \wedge global\_beg = global\_end \wedge$$
$$heap\_beg = heap\_end \wedge i = stack\_beg - 4 \wedge stack\_end = stack\_beg - 4$$

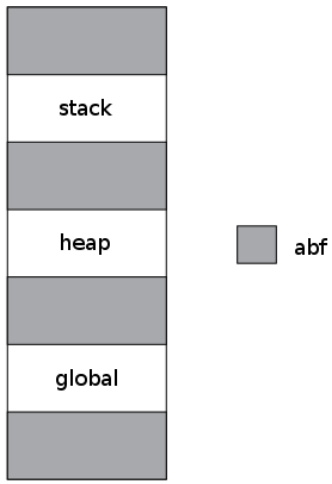
VC: $read(mem@i) == 0$

- *"is reading value 0 at memory position i possible"?*
- UNSAT if assertion holds (on path)

(Simple) Assignment Check

- validity of address (*addr*) a value is assigned to
- introduce *abf* outside any valid memory:

$$
\begin{aligned}
&abf > stack\_beg \ \vee \\
&abf > global\_end \ \wedge \\
&abf < heap\_beg \ \vee \\
&abf > heap\_end \ \wedge \\
&abf < stack\_end \ \vee \\
&abf < global\_beg
\end{aligned}
$$



- VC: $abf == addr$
  - *"can addr be equal to an address outside valid memory?"*
  - SAT if *addr* is invalid