Formal Verification of Loop Bound Estimation for WCET Analysis

VSTTE'13

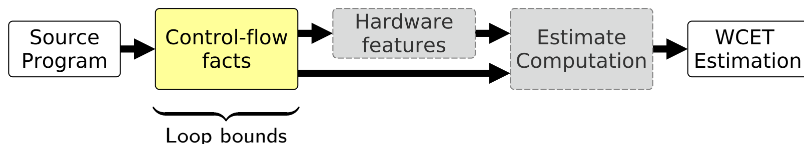
Sandrine Blazy[1], David Pichardie[2] and André Maroneze[1]

[1]IRISA - University of Rennes 1
[2]Harvard University/INRIA

19/05/2013

WCET Context

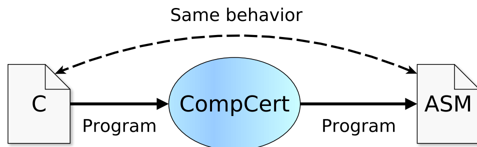
- Real-time systems: timing constraints
- Worst-Case Execution Time (WCET): undecidable
  - Measurements (tests): underestimation
  - Static analysis: (safe) overestimation



- Context assumptions
  - No diverging executions
  - No recursion

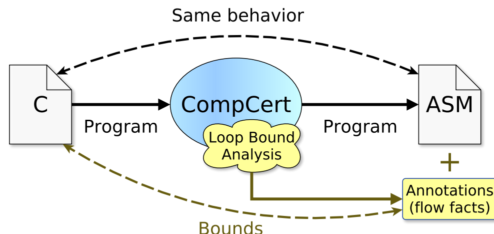
Providing guarantees for WCET estimation

Objective: extend CompCert to produce sound flow annotations, specifically loop bounds



Providing guarantees for WCET estimation

Objective: extend CompCert to produce sound flow annotations, specifically loop bounds



Contribution: formalization of a loop bound analysis

- Provides bounds for WCET tools at the assembly level
- Integrated in CompCert
  - Provides guarantees about loop bounds in the compiled code

    Main theorem: *start-to-end correctness*

Compiling a C program (P_C) into an assembly program (P_{ASM}) ensures that bounds for P_{ASM} are valid bounds for P_C.

  - Extraction into executable code

- Based on a state-of-the-art WCET tool: SWEET[1]

[1] SWEdish Execution Time tool

SWEET-inspired method for bounding loops

- Based on the following observation:
  In a *terminating* and *deterministic* loop, at each iteration, the set of variable states $(x_1, x_2, \dots)$ is unique.

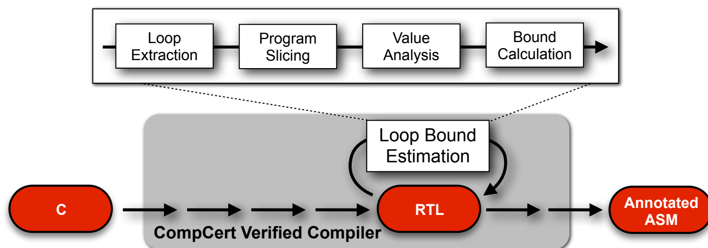
  Example:

```
          i = 0;
    Loop: while (i <= N) {
            j = 0;
            while (j <= M) {
    Inner:    ...
              j++;
            }
            ...
            i++;
          }
```

- Program history at `Inner`, restricted to variables (i, j):
  $(0,0) \rightarrow (0,1) \rightarrow (0,2) \rightarrow \cdots \rightarrow (1,0) \rightarrow (1,1) \rightarrow \cdots \rightarrow (N,M)$
- Longest history trace $\Rightarrow$ upper bound for number of iterations
  $UB = (N+1) \times (M+1)$ iterations
- With several variables $x, y, z, w \dots$, this naive solution does not scale

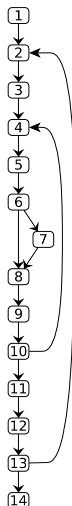
Formalization challenges

- Loop structure
  - Reconstruction from the CFG
- Precision
  - Program slicing, *variable slicing*
- Global bounds for nested loops
  - Combination of bounds from outer loops



Loop bound method on an example

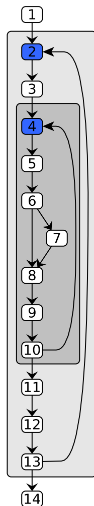
Program to be analyzed




```
n = 5, i = 0, w = 0.0
do {
    j = 0
    do {
        a[i][j] = i+1+j+1
        if (i == j)
            a[i][j] *= 5.0
        w += a[i][j]
        j++
    } while (j <= n)
    b[i] = w
    i++
} while (i <= n)
return
```

Loop bound method on an example

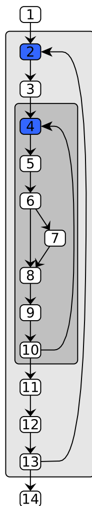
Loop extraction




```
n = 5, i = 0, w = 0.0
do {
    j = 0
    do {
        a[i][j] = i+1+j+1
        if (i == j)
            a[i][j] *= 5.0
        w += a[i][j]
        j++
    } while (j <= n)
    b[i] = w
    i++
} while (i <= n)
return
```

Loop bound method on an example

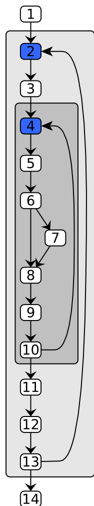
Value analysis




```
n = 5, i = 0, w = 0.0
  n:[5,5] i:[0,0] j:??    w:?? a:??
do {
  n:[5,5] i:[0,5] j:??    w:?? a:??
    j = 0
  n:[5,5] i:[0,5] j:[0,0] w:?? a:??
    do {
  n:[5,5] i:[0,5] j:[0,5] w:?? a:??
        a[i][j] = i+1+j+1
  n:[5,5] i:[0,5] j:[0,5] w:?? a:??
        if (i == j)
  n:[5,5] i:[0,5] j:[0,5] w:?? a:??
            a[i][j] *= 5.0
  n:[5,5] i:[0,5] j:[0,5] w:?? a:??
        w += a[i][j]
  n:[5,5] i:[0,5] j:[0,5] w:?? a:??
        j++
  n:[5,5] i:[0,5] j:[1,6] w:?? a:??
    } while (j <= n)
  n:[5,5] i:[0,5] j:[6,6] w:?? a:??
    b[i] = w
  n:[5,5] i:[0,5] j:[6,6] w:?? a:??
    i++
  n:[5,5] i:[1,6] j:[6,6] w:?? a:??
} while (i <= n)
  n:[5,5] i:[5,5] j:[6,6] w:?? a:??
return
```

Loop bound method on an example

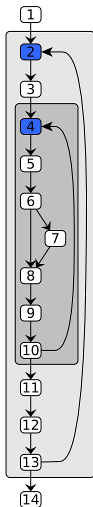
$|dom(n)| \times |dom(i)| \times |\mathbf{dom(j)}| \times |\mathbf{dom(w)}| \times |\mathbf{dom(a)}| = +\infty$ (useless)




```
n = 5, i = 0, w = 0.0
  n:[5,5] i:[0,0] j:??    w:?? a:??
do {
  n:[5,5] i:[0,5] j:??    w:?? a:??
    j = 0
  n:[5,5] i:[0,5] j:[0,0] w:?? a:??
    do {
  n:[5,5] i:[0,5] j:[0,5] w:?? a:??
        a[i][j] = i+1+j+1
  n:[5,5] i:[0,5] j:[0,5] w:?? a:??
        if (i == j)
  n:[5,5] i:[0,5] j:[0,5] w:?? a:??
            a[i][j] *= 5.0
  n:[5,5] i:[0,5] j:[0,5] w:?? a:??
        w += a[i][j]
  n:[5,5] i:[0,5] j:[0,5] w:?? a:??
        j++
  n:[5,5] i:[0,5] j:[1,6] w:?? a:??
    } while (j <= n)
  n:[5,5] i:[0,5] j:[6,6] w:?? a:??
    b[i] = w
  n:[5,5] i:[0,5] j:[6,6] w:?? a:??
    i++
  n:[5,5] i:[1,6] j:[6,6] w:?? a:??
} while (i <= n)
  n:[5,5] i:[5,5] j:[6,6] w:?? a:??
return
```

Loop bound method on an example

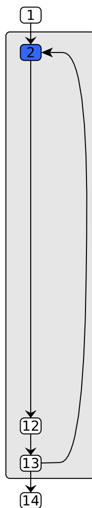
Better solution: simplify program before value analysis




```
n = 5, i = 0, w = 0.0
do {
    j = 0
    do {
        a[i][j] = i+1+j+1
        if (i == j)
            a[i][j] *= 5.0
        w += a[i][j]
        j++
    } while (j <= n)
    b[i] = w
    i++
} while (i <= n)
return
```

Loop bound method on an example

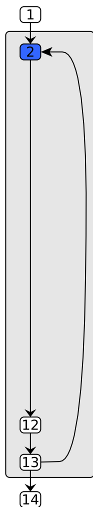
Using program slicing: slicing w.r.t. loop header 2




```
n = 5, i = 0//, w = 0.0
do { // slicing criterion
    // j = 0
    goto L1 // do {
        // a[i][j] = i+1+j+1
        // if (i == j)
        //   a[i][j] *= 5.0
        // w += a[i][j]
        // j++
    // } while (j <= n)
    // b[i] = w
    L1: i++
} while (i <= n)
return
```

Loop bound method on an example

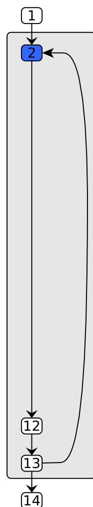
Value analysis on the sliced program




```
n = 5, i = 0//, w = 0.0

do { // slicing criterion
  n:[5,5] i:[0,5]
    // j = 0
  n:[5,5] i:[0,5]
    goto L1 // do {

        // a[i][j] = i+1+j+1

        // if (i == j)

        //   a[i][j] *= 5.0

        // w += a[i][j]

        // j++

    // } while (j <= n)

    // b[i] = w
  n:[5,5] i:[0,5]
    L1: i++
  n:[5,5] i:[1,6]
} while (i <= n)
  n:[5,5] i:[6,6]
return
```

Loop bound method on an example

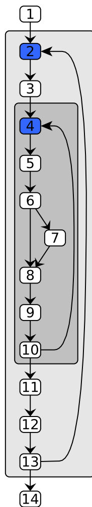
n:[5,5] (size 1) $\times$ i:[0, 5] (size 6) $\Rightarrow$ 6 iterations for the outer loop




```
n = 5, i = 0//, w = 0.0

do { // slicing criterion
  n:[5,5] i:[0,5]
    // j = 0
  n:[5,5] i:[0,5]
    goto L1 // do {

        // a[i][j] = i+1+j+1

        // if (i == j)

        //   a[i][j] *= 5.0

        // w += a[i][j]

        // j++

    // } while (j <= n)

    // b[i] = w
  n:[5,5] i:[0,5]
    L1: i++
  n:[5,5] i:[1,6]
} while (i <= n)
  n:[5,5] i:[6,6]
return
```

Loop bound method on an example

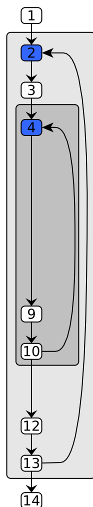
Bounding the inner loop (header 4)




```
n = 5, i = 0, w = 0.0
do {
    j = 0
    do {
        a[i][j] = i+1+j+1
        if (i == j)
            a[i][j] *= 5.0
        w += a[i][j]
        j++
    } while (j <= n)
    b[i] = w
    i++
} while (i <= n)
return
```

Loop bound method on an example

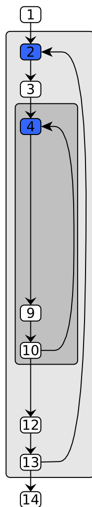
Slicing on the inner loop (header 4)




```
n = 5, i = 0//, w = 0.0
do {
    j = 0
    do {
        // a[i][j] = i+1+j+1
        goto L2 // if (i == j)
          // a[i][j] *= 5.0
        L2: // w += a[i][j]
        j++
    } while (j <= n)
    // b[i] = w
    i++
} while (i <= n)
return
```

Loop bound method on an example

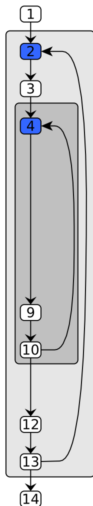
Value analysis on the sliced loop




```
n = 5, i = 0//, w = 0.0

do {

    j = 0

    do {
  n:[5,5] i:[0,5] j:[0,5]
        // a[i][j] = i+1+j+1
  n:[5,5] i:[0,5] j:[0,5]
        goto L2 // if (i == j)

          // a[i][j] *= 5.0
  n:[5,5] i:[0,5] j:[0,5]
        L2: // w += a[i][j]
  n:[5,5] i:[0,5] j:[0,5]
        j++
  n:[5,5] i:[0,5] j:[1,6]
    } while (j <= n)

    // b[i] = w

    i++

} while (i <= n)

return
```

Loop bound method on an example

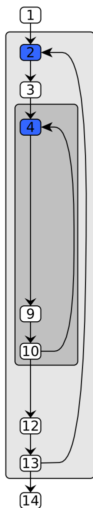
i:[0,5] (size 6) $\times$ j:[0, 5] (size 6) $\Rightarrow$ 36 iterations for the inner loop




```
n = 5, i = 0//, w = 0.0

do {

    j = 0

    do {
  n:[5,5] i:[0,5] j:[0,5]
        // a[i][j] = i+1+j+1
  n:[5,5] i:[0,5] j:[0,5]
        goto L2 // if (i == j)

          // a[i][j] *= 5.0
  n:[5,5] i:[0,5] j:[0,5]
        L2: // w += a[i][j]
  n:[5,5] i:[0,5] j:[0,5]
        j++
  n:[5,5] i:[0,5] j:[1,6]
    } while (j <= n)

    // b[i] = w

    i++

} while (i <= n)

return
```

Loop bound method on an example

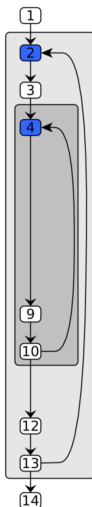
Extra precision: compute *interesting variables* in the loop: *use* $\cap$ *def* $\cap$ *live*




```
n = 5, i = 0//, w = 0.0

do {

    j = 0

    do {
  n:[5,5] i:[0,5] j:[0,5]
        // a[i][j] = i+1+j+1
  n:[5,5] i:[0,5] j:[0,5]
        goto L2 // if (i == j)

          // a[i][j] *= 5.0
  n:[5,5] i:[0,5] j:[0,5]
        L2: // w += a[i][j]
  n:[5,5] i:[0,5] j:[0,5]
        j++
  n:[5,5] i:[0,5] j:[1,6]
    } while (j <= n)

    // b[i] = w

    i++

} while (i <= n)

return
```

Loop bound method on an example

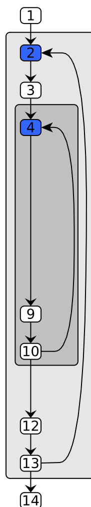
$$IVars(loop\ 4) = \{j\}$$




```
n = 5, i = 0//, w = 0.0

do {

    j = 0

    do {
                    j:[0,5]
        // a[i][j] = i+1+j+1
                    j:[0,5]
        goto L2 // if (i == j)

          // a[i][j] *= 5.0
                    j:[0,5]
        L2: // w += a[i][j]
                    j:[0,5]
        j++
                    j:[1,6]
    } while (j <= n)

    // b[i] = w

    i++

} while (i <= n)

return
```

Loop bound method on an example

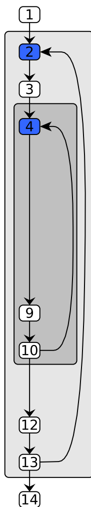
Interesting variable domains at loop entry: $j : [0, 5] \Rightarrow 6$ iterations




```
n = 5, i = 0//, w = 0.0

do {

    j = 0

    do {
                    j:[0,5]
        // a[i][j] = i+1+j+1
                    j:[0,5]
        goto L2 // if (i == j)

          // a[i][j] *= 5.0
                    j:[0,5]
        L2: // w += a[i][j]
                    j:[0,5]
        j++
                    j:[1,6]
    } while (j <= n)

    // b[i] = w

    i++

} while (i <= n)

return
```

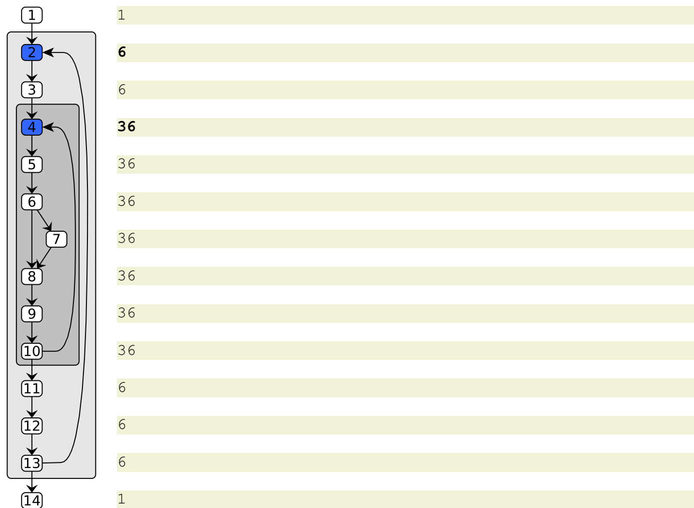
Loop bound method on an example

Product of the nested bounds: $6 \times 6 = 36$ iterations for the inner loop




```
n = 5, i = 0//, w = 0.0

do {

    j = 0

    do {
                    j:[0,5]
        // a[i][j] = i+1+j+1
                    j:[0,5]
        goto L2 // if (i == j)

          // a[i][j] *= 5.0
                    j:[0,5]
        L2: // w += a[i][j]
                    j:[0,5]
        j++
                    j:[1,6]
    } while (j <= n)

    // b[i] = w

    i++

} while (i <= n)

return
```

Loop bound method on an example



Outline

1. WCET Context and Example
2. Method formalization
   - Semantics and correctness theorems
   - Loop nestings
   - Program Slicing
   - Value Analysis
   - Bound Computation
3. Results and Conclusion

Instrumented semantics

Equivalent to the RTL semantics if no recursion, adapted to the WCET

- No function calls
- Execution counters incremented at each step

$$\sigma = < l, E, c_{glob}, c_{loc} >$$

- l: program node
- E: environment (map $Var \to Val$)
- c_{glob} and c_{loc} : global and local execution counters
  - local counters: reset at loop exit
- $P \Downarrow c_{glob}$: P terminates with global counters c_{glob}

Theorem (Bound correctness)

Let P be a program s.t. $P \Downarrow c_{glob}$ and l a node in P.
Then $c_{glob}(l) \leq \texttt{bound}(P)(l)$.

Start-to-end correctness

Bounds obtained at the RTL level are valid at the assembly level.

Theorem (Start-to-end correctness)

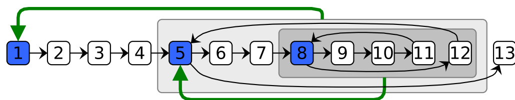
Let $P_C \xrightarrow{comp} (P_{ASM}, bounds)$ *be the compilation of a C source program free of runtime errors into an assembly program and a set of bounds.*
For any finite execution of P_{ASM}*, producing trace* `tr`*, we have*

$$\#\mathtt{tr}_{\downarrow\mathtt{a}} \leq bounds[\mathtt{a}]$$

where $\#\mathtt{tr}_{\downarrow\mathtt{a}}$ *denotes the number of occurrences of the event associated to* `a` *in trace* `tr`*.*

Proof obtained via CompCert's preservation of annotations.

Loops and loop nestings



- Loops can be described as triples (**header**, **parent**, **nodes**)
  - header: loop entry node
  - parent: immediate ancestor in the loop hierarchy
  - nodes: all nodes contained in the loop
- Formalized as a Coq record containing *functions*...

```
Record nesting := {
loop : vertex → loop     header : loop → vertex     parent : loop → loop ...
```

- ... and also *properties*, used in the proofs:

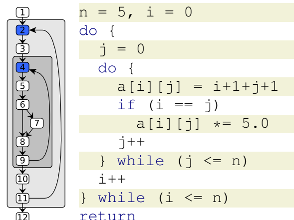
`enter_via_header`: *every path that enters a loop contains the loop header*

`cycle_at_not_header`: *every cycle at* `n` *contains* `header(loop(n))`   ...   }

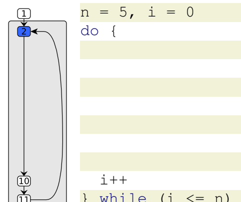
Program Slicing

- Program simplification, w.r.t. a given statement (*slicing criterion*)
  - *Elimination* of statements not affecting the criterion
  - Preserves program behavior (up to the slicing criterion)

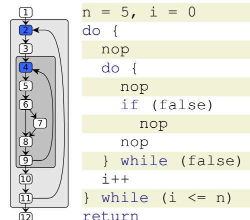
Original program



Program slice (criterion: node **2**)



CFG-preserving program slice



- Correctness: *bounds obtained in the program slice are valid bounds for the original program* (i.e., there is no underestimation)

Main Theorem of Program Slicing

Theorem (Correctness of Program Slicing)

Let P be a program and l a node in P. Let $P' = \texttt{slice}(P, l)$ be the slice of P w.r.t the criterion l. If M is a safe bound for all local counters reachable at l in P':

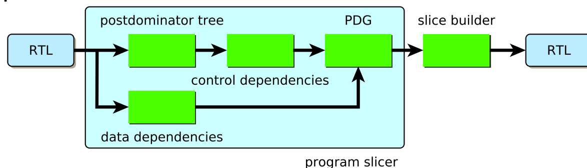
$$\forall \sigma \in \texttt{reach}(P'),\ \sigma.c_{loc}(l) \leq M$$

then M is also a safe bound for all local counters reachable at l in P:

$$\forall \sigma \in \texttt{reach}(P),\ \sigma.c_{loc}(l) \leq M$$

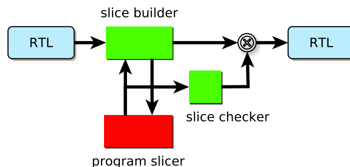
Proving Program Slicing Correctness

- Direct proof



  - Costly; efficient algorithms often hard to prove (e.g. graph-based)

- *A posteriori* validation



  - Checker is easy to prove and efficient

Value Analysis

- Abstract interpretation over RTL
- Domain: intervals of machine integers

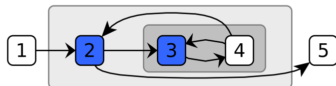
Theorem (Value Analysis Is Correct)

Let P be a program. Let $res = \mathtt{value}(P)$ be the result of the value analysis and σ_n a reachable state.
For each program node l, we have:
$\forall v \in \mathtt{dom}(\sigma_n.E), \sigma_n.E(v) \in res(l)(v)$.

- Value analysis not detailed here (see SAS'13 for more details)

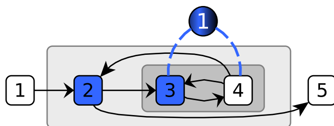
Computing Bounds

- Bounds: product of variable domain sizes and outer loops
- Proof intuitively simple, but complex to prove
  - Inductive reasoning about paths and reachable states
- Decomposed in three lemmas, definining relations between:
  1. counters for nodes in the same loop
  2. counters for nodes in nested loops
  3. counters and variable domain sizes



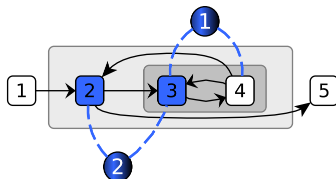
Computing Bounds

- Bounds: product of variable domain sizes and outer loops
- Proof intuitively simple, but complex to prove
  - Inductive reasoning about paths and reachable states
- Decomposed in three lemmas, definining relations between:
  1. counters for nodes in the same loop
  2. counters for nodes in nested loops
  3. counters and variable domain sizes



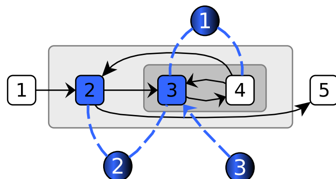
Computing Bounds

- Bounds: product of variable domain sizes and outer loops
- Proof intuitively simple, but complex to prove
  - Inductive reasoning about paths and reachable states
- Decomposed in three lemmas, defining relations between:
  1. counters for nodes in the same loop
  2. counters for nodes in nested loops
  3. counters and variable domain sizes



Computing Bounds

- Bounds: product of variable domain sizes and outer loops
- Proof intuitively simple, but complex to prove
  - Inductive reasoning about paths and reachable states
- Decomposed in three lemmas, definining relations between:
  1. counters for nodes in the same loop
  2. counters for nodes in nested loops
  3. counters and variable domain sizes



Computing Bounds – Lemma 1

Lemma (Header node counter bounds loop counters)

For all reachable states $\sigma \in \texttt{reach}(P)$ *and program nodes* l *in* P,

$$\sigma.c_{glob}(l) \ \leq \ \sigma.c_{glob}(\texttt{header}(\texttt{loop}(l)))$$

Proof: obtained via the definition of loops (single entry point)

Computing Bounds – Lemma 2

Lemma (Relation between global and local counters)

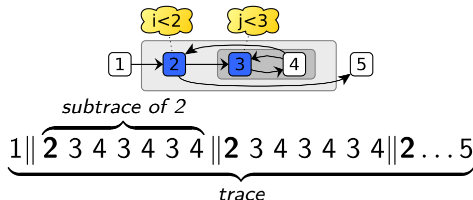
Let h be a loop header and p_h the header of its parent loop, i.e. $p_h = \texttt{header}(\texttt{parent}(\texttt{loop}(h)))$. *Let M bound the local counter of h:*

$$\forall \sigma \in \texttt{reach}(P), \sigma.c_{loc}(h) \;\leq\; M$$

Therefore, $\forall \sigma \in \texttt{reach}(P), \sigma.c_{glob}(h) \;\leq\; M \times \sigma.c_{glob}(p_h)$.

Proof: induction on execution traces

Bounds: max occurrences of h per subtrace $\times$ number of subtraces of p_h



Computing Bounds – Lemma 3

Lemma (Local counters are bounded)

For all reachable states $\sigma \in \mathtt{reach}(P)$, *we have*

$$\sigma.c_{loc}(h) \;\leq \prod_{x \,\in\, \mathtt{live}(h) \,\cap\, \mathtt{use}(\mathtt{loop}(h)) \,\cap\, \mathtt{def}(\mathtt{loop}(h))} |\mathtt{value}(P')(h)(x)|$$

- Proof: *pigeonhole principle* applied on a finite trace
- Precision: slicing + liveness + interesting variables + value analysis

Outline

1. WCET Context and Example
2. Method formalization
   - Semantics and correctness theorems
   - Loop nestings
   - Program Slicing
   - Value Analysis
   - Bound Computation
3. Results and Conclusion

Results (Mälardalen WCET benchmarks)

Program	#L	Our tool #LE	Our tool %LE	SWEET #LE	SWEET %LE	Our tool #GB	Our tool %GB	SWEET #GB	SWEET %GB
1 adpcm	27	13	48%	22	81%	16	59%	18	67%
2 cnt	4	4	100%	4	100%	4	100%	4	100%
3 crc	6	4	67%	6	100%	6	100%	6	100%
4 edn	12	9	75%	11	92%	12	100%	12	100%
5 expint	2	2	100%	2	100%	2	100%	2	100%
6 fdct	2	2	100%	2	100%	2	100%	2	100%
7 fft1	29	3	10%	6	21%	7	24%	7	24%
8 fibcall	1	1	100%	1	100%	1	100%	1	100%
9 fir	2	1	50%	1	50%	1	50%	2	100%
10 insertsort	2	1	50%	1	50%	1	50%	1	50%
11 jfdctint	3	3	100%	3	100%	3	100%	3	100%
12 ludcmp	11	6	55%	6	55%	6	55%	6	55%
13 matmult	7	7	100%	7	100%	7	100%	7	100%
14 ndes	12	12	100%	12	100%	12	100%	12	100%
15 ns	4	4	100%	4	100%	4	100%	4	100%
16 qurt	3	2	67%	3	100%	3	100%	3	100%
17 ud	11	10	91%	10	91%	10	91%	11	100%
Geometric mean			69%		79%		79%		83%

Conclusion and further work

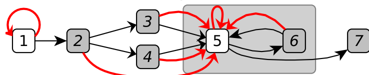
- Practical tools often involve combining different methods
  - Proving $A + B$ implies 3 proofs: A, B, and $A \underline{+} B$; correctly composing analyses is a delicate art
  - Balance between proof and validation is necessary
- Executable, proven tool, from C to assembly
- Obtains useful results
  - Comparable to SWEET's
  - Precision can be improved through value analysis
- Reusable components: loops for RTL, program slicing
- Further development
  - Integration with an ILP solver: tighter global bounds

Backup

Proving Slice Correctness

Proof via *weak simulation* (of finite executions) between original and sliced programs.

- Lemma: *Weak simulation preserves counters for all nodes in the slice*
- Simulation based on *next observables* and *relevant variables*
  - Next observable *NObs*: `vertex` $\rightarrow$ `option vertex`
    Closest successor that belongs to the slice



  - Relevant variables *RV*: `vertex` $\rightarrow$ *Var**
    Variables which may *influence* nodes in the slice
    $\Rightarrow$ Irrelevant variables may be sliced away
- *NObs* and *RV* sets computed by efficient, untrusted OCaml code and validated *a posteriori* in Coq
  - Fixpoint on mutually recursive sets *RV*, *NObs* and *slice*