Path-Sensitive Resource Analysis Compliant with Assertions

Duc-Hiep CHU and Joxan JAFFAR

National University of Singapore (NUS)

1 Oct 2013

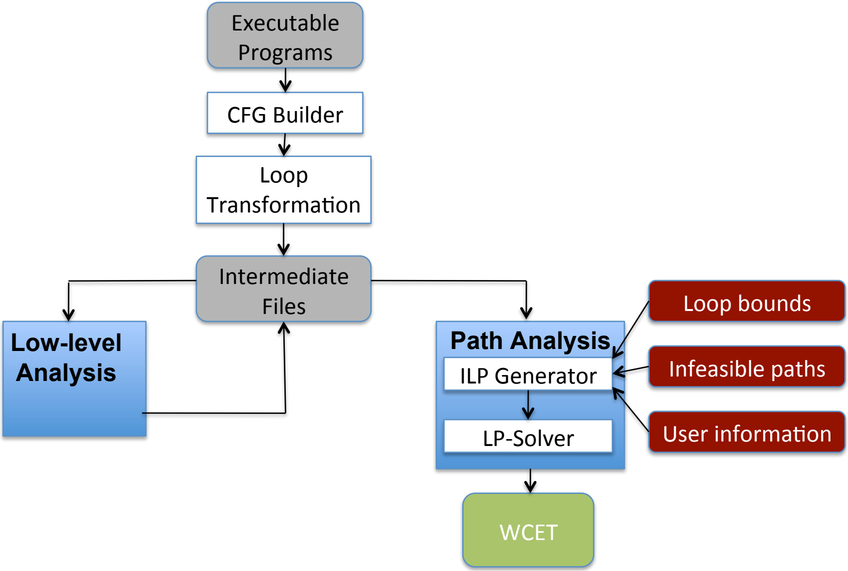
OUTLINE

1. Problem definition
2. We need both path-sensitivity and assertions
3. Path-sensitivity and assertions don't mix
4. Our solution

Analysis of Worst-Case Resource Usage

- Important for designing real-time and embedded systems
  - *cumulative* resource (e.g., timing)
  - *non-cumulative* resource (e.g., memory high-water mark)
- *Extremely hard* due to the requirement of high precision
- Redeeming factors:
  - Loops/recursions are statically bounded
  - The users/certifiers are willing to help
- We restrict the presentation to WCET (or timing) analysis
  - Results are extensible to non-cumulative resource

Architecture of A Traditional WCET Analyzer



Executable Programs

CFG Builder

Loop Transformation

Intermediate Files

Low-level Analysis

Path Analysis

ILP Generator

LP-Solver

Loop bounds

Infeasible paths

User information

WCET

Implicit Path Enumeration Technique (IPET)

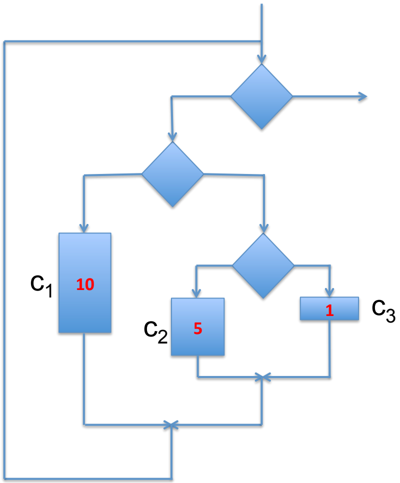
- Introduced by Li and Malik [1995]
- Employs Integer Linear Programming (ILP)
- Simple, elegant, fast, but *path-insensitive*
- Supports user information

Example: IPET

```
c1 = 0, c2 = 0, c3 = 0;
i = 0, t = 0;
while (i < 9) {
    if (*) {B1:  c1++; t += 10; }
    else {
        if (i == 1) {B2:  c2++; t += 5; }
        else {B3:  c3++; t += 1; }
    }
    i++;
    assert(c1 <= 4);
}
```

$$\text{maximize}(10 \cdot c_1 + 5 \cdot c_2 + 1 \cdot c_3) \text{ wrt. } c_1 + c_2 + c_3 \leq 9 \wedge c_1 \leq 4 \wedge c_2 \leq 1$$

Example: IPET



$$\text{maximize}(10 \cdot c_1 + 5 \cdot c_2 + 1 \cdot c_3) \text{ wrt. } c_1 + c_2 + c_3 \leq 9 \wedge c_1 \leq 4 \wedge c_2 \leq 1$$

Manual Annotations in IPET

- Annotating loop bounds (e.g., $c_1 + c_2 + c_3 \leq 9$)
  - Is *mandatory* to produce a bound
  - Precision depends on the precision of given loop bounds
  - Automation: some simple form of loop bound analysis (However, precision can be affected due to *complicated* loops)
- Annotating infeasible paths (e.g., $c_2 \leq 1$)
  - Fundamentally hard due to the exponential number of infeasible paths
  - Automation: usually ad-hoc (e.g., detecting simple conflict patterns)
- Annotating other user information (e.g., $c_1 \leq 4$)
  - Information that is too hard to automatically extract from the code
  - Additional information the users know, but not in the code
  - Via the use of what we shall call assertions

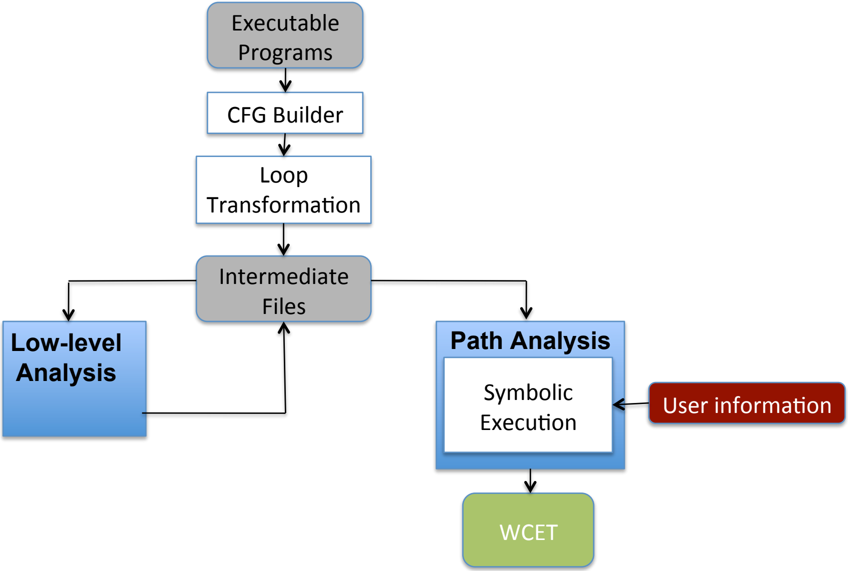
Manual Annotations in IPET

- Annotating loop bounds (e.g., $c_1 + c_2 + c_3 \leq 9$)
  - Is *mandatory* to produce a bound
  - Precision depends on the precision of given loop bounds
  - Automation: some simple form of loop bound analysis (However, precision can be affected due to *complicated* loops)
- Annotating infeasible paths (e.g., $c_2 \leq 1$)
  - Fundamentally hard due to the exponential number of infeasible paths
  - Automation: usually ad-hoc (e.g., detecting simple conflict patterns)
- Annotating other user information (e.g., $c_1 \leq 4$)
  - Information that is too hard to automatically extract from the code
  - Additional information the users know, but not in the code
  - Via the use of what we shall call assertions

Manual Annotations in IPET

- Annotating loop bounds (e.g., $c_1 + c_2 + c_3 \leq 9$)
  - Is *mandatory* to produce a bound
  - Precision depends on the precision of given loop bounds
  - Automation: some simple form of loop bound analysis (However, precision can be affected due to *complicated* loops)
- Annotating infeasible paths (e.g., $c_2 \leq 1$)
  - Fundamentally hard due to the exponential number of infeasible paths
  - Automation: usually ad-hoc (e.g., detecting simple conflict patterns)
- Annotating other user information (e.g., $c_1 \leq 4$)
  - Information that is too hard to automatically extract from the code
  - Additional information the users know, but not in the code
  - Via the use of what we shall call assertions

Our Proposed Framework



Executable Programs

CFG Builder

Loop Transformation

Intermediate Files

Low-level Analysis

Path Analysis

Symbolic Execution

User information

WCET

The Need for Assertions

- The analysis precision could highly depend on the inputs and the programmer knows about the input set (i.e., the environment where the program is run)
- Making use of such user information can be crucial

```
c = c1 = 0;
t = 0;
for (i = 0; i < 100; i++) {
    c++;
    if (A[i] != 0) {
        c1++;
        t += 1000;
    } else { t += 1; }
}
assert(c1 <= c / 10);
```

The Need for Local Assertions

- Consider `bubblesort`, input `a[]` contains element in $[min, max]$
- User information: there are M elements equal to max
- Local assertion (counter `c` is reset) is easier to derive
- IPET does not support local assertions

```
c = 0; t = 0;
for (i = N-1; i >= 1; i--) {
    c = 0;
    for (j = 0; j <= i-1; j++)
        if (a[j] > a[j+1]) {
            c++;
            t += 100; tp = a[j];
            a[j] = a[j+1]; a[j+1] = tp;
        } else { t += 1; }
        assert(c <= N-M);
    }}
```

The Need for Path-sensitivity

- Path-sensitivity is necessary for precision too (i.e., assertions only will not be sufficient)

```
c = c1 = c2 = 0;
t = i = 0;
while (i < 10) {
    c++;
    if (i mod 3 == 0) {
        c1++; i *= i; t += 30;
    } else { c2++; t += 1; }
    i++;
    assert(???);
}
```

Path-sensitivity and Assertions Together

- User needs to provide less information (e.g., $c_1 \leq 4$)
- The rest the system can automatically figure out (e.g., $c_1 + c_2 + c_3 \leq 9$ and $c_2 \leq 1$)

```
c_1 = c_2 = c_3 = 0;
i = 0, t = 0;
while (i < 9) {
    if (*) {B1:  c_1++; t += 10; }
    else {
        if (i == 1) {B2:  c_2++; t += 5; }
        else {B3:  c_3++; t += 1; }
    }
    i++;
    assert(c_1 <= 4);
}
```

How Do We Achieve Path-sensitivity?

- We can afford path-sensitivity, but up to loops only (Chu and Jaffar [2011])
    - We perform symbolic execution where loops are unrolled
    - Scalability is achieved by (1) performing abstraction after each loop iteration (i.e., contexts are merged); (2) summarizing with interpolation for reuse
    - Note that (1) is inevitable for any unrolling technique

Loop Unrolling and Assertions Don’t Mix

```
c = 0;
i = 0, t = 0;
while (i < 9) {
    if (*) {B1: c++; t += 10; }
    else {
        if (i == 1) {B2: t += 5; }
        else {B3: t += 1; }
    }
    i++;
    assert(c <= 4);
}
```

- Attempt 1: Perform context merge at the end of each loop iteration
    - Information about c is lost
    - The provided assertion will never be fired
    - Worst-case bound: 90 (block **B1** is executed 9 times)

Loop Unrolling and Assertions Don't Mix

```
c = 0;
i = 0, t = 0;
while (i < 9) {
     if (*) {B1:  c++; t += 10; }
     else {
          if (i == 1) {B2:  t += 5; }
          else {B3:  t += 1; }
     }
     i++;
     assert(c <= 4);
}
```

- Attempt 2: Try under-approximation by keeping the context of c from the worst-case path
  - Worst-case bound: $10 + 10 + 10 + 10 + 1 + 1 + 1 + 1 + 1 = 45$
  - This bound is unsound
  - Counter-example:
    - Replace "`if (*)`" with "`if prime(i)`"
    - The timing: $1 + 5 + 10 + 10 + 1 + 10 + 1 + 10 + 1 = 49$
  - Reason: when the assertion starts to kick in, block **B2** is no longer available for execution (due to greedy treatment)

Loop Unrolling and Assertions Don't Mix

```
c = 0;
i = 0, t = 0;
while (i < 9) {
     if (*) {B1:  c++; t += 10; }
     else {
          if (i == 1) {B2:  t += 5; }
          else {B3:  t += 1; }
     }
     i++;
     assert(c <= 4);
}
```

- Attempt 2: Try under-approximation by keeping the context of c from the worst-case path
  - Worst-case bound: $10 + 10 + 10 + 10 + 1 + 1 + 1 + 1 + 1 = 45$
  - This bound is unsound
  - Counter-example:
    - Replace "`if (*)`" with "`if prime(i)`"
    - The timing: $1 + 5 + 10 + 10 + 1 + 10 + 1 + 10 + 1 = 49$
  - Reason: when the assertion starts to kick in, block **B2** is no longer available for execution (due to greedy treatment)

Loop Unrolling and Assertions Don't Mix

```
c = 0;
i = 0, t = 0;
while (i < 9) {
     if (*) {B1:  c++; t += 10; }
     else {
          if (i == 1) {B2:  t += 5; }
          else {B3:  t += 1; }
     }
     i++;
     assert(c <= 4);
}
```

- Attempt 2: Try under-approximation by keeping the context of c from the worst-case path
  - Worst-case bound: $10 + 10 + 10 + 10 + 1 + 1 + 1 + 1 + 1 = 45$
  - This bound is unsound
  - Counter-example:
    - Replace "`if (*)`" with "`if prime(i)`"
    - The timing: $1 + 5 + 10 + 10 + 1 + 10 + 1 + 10 + 1 = 49$
  - Reason: when the assertion starts to kick in, block **B2** is no longer available for execution (due to greedy treatment)

Loop Unrolling and Assertions Don't Mix

- Fundamentally, "Being compliant with assertions" requires the analysis to be fully path-sensitive wrt. assertion variables
- This interferes with greedy treatment of loops (merge & summarize)

Loop Unrolling and Assertions Don't Mix

- Fundamentally, "Being compliant with assertions" requires the analysis to be fully path-sensitive wrt. assertion variables
- This interferes with greedy treatment of loops (merge & summarize)

Solution: A Two-Phase Algorithm (for each loop)

- Phase 1:
  - Perform loop unrolling with iteration abstraction and interpolation
  - Eliminate two kinds of paths:
    - Infeasible paths (detected from path-sensitivity)
    - Dominated paths. (1) We track frequency variables which will be used later in some assertion. (2) For paths which modify the tracked variables *in the same way*, we keep the one whose resource usage *dominates* the rest
- Phase 2:
  - Disregard all paths *violating* the assertions
  - Employ a dynamic programming approach with interpolation for DAG

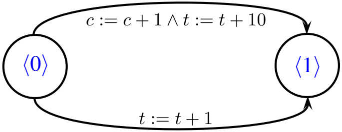
Solution: A Two-Phase Algorithm (for each loop)

- Phase 1:
  - Perform loop unrolling with iteration abstraction and interpolation
  - Eliminate two kinds of paths:
    - Infeasible paths (detected from path-sensitivity)
    - Dominated paths. (1) We track frequency variables which will be used later in some assertion. (2) For paths which modify the tracked variables *in the same way*, we keep the one whose resource usage *dominates* the rest
- Phase 2:
  - Disregard all paths *violating* the assertions
  - Employ a dynamic programming approach with interpolation for DAG

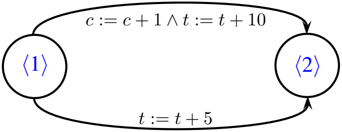
Phase 1: Removal of Infeasible Paths

```
c = 0; i = 0, t = 0;
while (i < 9) {
    if (*) {B1:  c++; t += 10; }
    else { if (i == 1) {B2:  t += 5; } else {B3:  t += 1; }}
    i++;
    assert(c <= 4);
}
```

First iteration: remove the path executing **B2**



Second iteration: remove the path executing **B3**

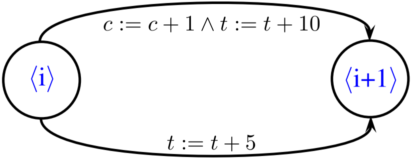


Other iterations, i.e., $i = 2..8$: reuse the analysis of the first iteration

Phase 1: Removal of Dominated Paths

```
c = 0, i = 0, t = 0;
while (i < 9) {
    if (*) {B1:  c++; t += 10; }
    else {
        if (*) {B2:  t += 5; }
        else {B3:  t += 1; }
    }
    i++;
    assert(c <= 4);
}
```

- Notice the change from `if (i == 1)` to `if (*)`
- All iterations, i.e., $i = 0..8$ (remove the path executing **B3**):



Phase 2: An Instance of the RCSP

- Phase 2 finds the longest path in the DAG produced by Phase 1, now taking into account the provided assertion(s)
- In this example, the number of contexts for counter c is linear, a simple dynamic programming algorithm would suffice
- In general, when loops are nested and the number of interested counters is more than 1, it is an instance of the Resource Constrained Shortest Path (RCSP) problem
- RCSP can be addressed efficiently, also by using interpolation technique (Jaffar *et al.* [2008])

Experiments

Benchmark	LOC	Path-Sensitive (Symbolic execution w. loop unrolling)				Path-Insensitive (IPET)	
		w.o. Assertions		w. Assertions		w.o. As	w. As
		Bound	T(s)	Bound	T(s)		
sparse_array	$<$ 100	110404	1.50	10404	3.48	110404	10404
bubblesort100	$<$ 100	515398	5.52	**49798**	11.45	1019902	1019902
watermark	$<$ 100	1010	1.74	**20**	5.45	*	*
conflict100	$<$ 100	1504	3.47	**759**	9.22	1504	1129
insertsort100	$<$ 100	515794	4.91	**30802**	7.78	1020804	1020804
crc	128	1404	7.73	1084	8.61	1404	1084
expint	157	15709	4.40	**859**	4.56	-	-
matmult100	163	3080505	4.55	131705	5.54	3080505	131705
fir	276	1129	2.35	**793**	2.39	-	-
fft64	219	7933	5.52	**1733**	6.04	-	-
tcas	400	159	3.84	**81**	3.9	172	94
statemate	1276	2103	9.65	**1103**	9.73	2271	1271
nsichneu_small	2334	483	9.43	**383**	9.51	2559	2459

Conclusion

- Precision of path analysis comes from two sources:
    - Path-sensitivity via symbolic simulation
    - User assertions to limit possible execution traces
- Symbolic simulation while compliant with assertions is not trivial
- We resolve the scalability issue by a two phase algorithm, of which the key is to make use of interpolation concept for reuse

D. H. Chu and J. Jaffar. Symbolic simulation on complicated loops for wcet path analysis. In *EMSOFT*, 2011.

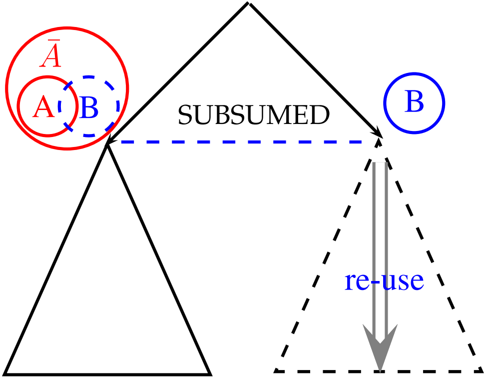
J. Jaffar, A. E. Santosa, and R. Voicu. Efficient memoization for dynamic programming with ad-hoc constraints. In *AAAI*, 2008.

Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *DAC*, 1995.

Questions & Answers

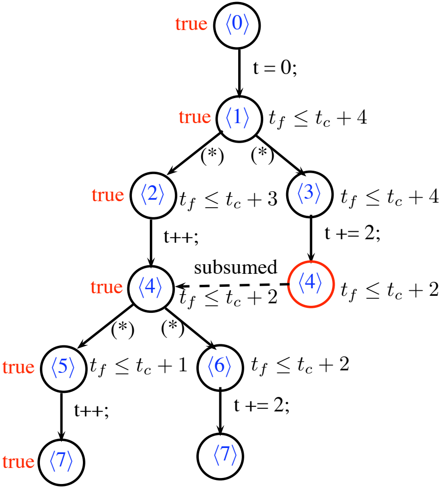
Interpolation for Reuse

- A and B share the same program point
- A does not subsume B
- Generalize the context of A to $\bar{A}$, aka an interpolant, while preserving the infeasible paths
- B is subsumed by $\bar{A}$
- The summarized analysis of A can be safely reused in B

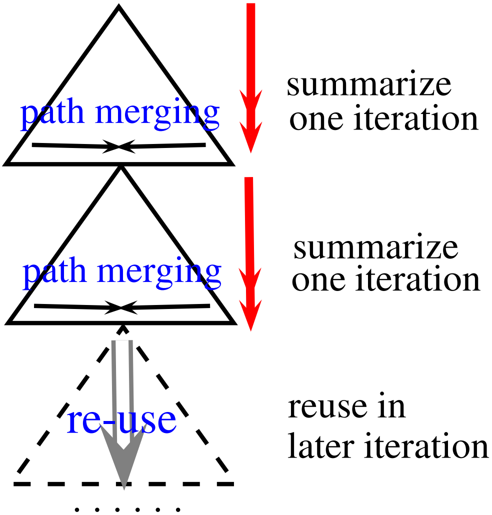


Example: Interpolation for Reuse

```
⟨0⟩ t = 0;
⟨1⟩ if (*)
⟨2⟩     t++;
    else
⟨3⟩     t += 2;
⟨4⟩ if (*)
⟨5⟩     t++;
    else
⟨6⟩     t += 2;
⟨7⟩
```



Interpolation for Reuse (with loop)



Phase 2: An Instance of the RCSP