

# Loop Bound Analysis based on a Combination of Program Slicing, Abstract Interpretation, and Invariant Analysis

Andreas Ermedahl, Jan Gustafsson, Christer Sandberg, Stefan Bygde, and Björn Lisper

Mälardalen Real-Time Research Center (MRTC)  
Mälardalen University, Västerås, Sweden



## Paper motivation & content

- ★ **Motivation:** Safe upper loop bounds are a requirement to derive safe WCET estimates
  - ◆ Industrial case-studies show that giving these bounds manually can be a major hassle and a potential source of errors
  - ◆ Automatic loop-bound analyses preferable
- ★ **Content:** Automatic approach for deriving upper loop bounds based on a combination of standard program analysis techniques:
  - ◆ Program slicing
  - ◆ Abstract interpretation
  - ◆ Invariant analysis



## Key observation 1

- ★ **Terminating loops must reach a new program state for each new loop iteration**

- ◆ If the same state is reached more than once the loop will not terminate

```
int i=0;
while(i<100)
i++;
```

Terminating loop

The loop does not terminate since i is assigned the same value several times

```
int i=0;
while(i<=100) {
if(odd(i)) i++;
else i--;
```



## Key observation 2

- ★ **Not all variables and statements affect the outcome of the exit conditions of a loop**

```
int i,j,k=0;
k++;
while(i<100) {
i++;
j++;
```

Variables j and k do not affect the number of times the loop is iterated



## Basic idea

- ★ **Try to bind the number of reachable states for variables affecting the exit conditions of a loop**

- ◆ Given that the loop terminates, this number provides an upper loop bound

- ★ **Made in a three step approach...**

```
i=0;
while(i<100) {
// p
i++;
}
```

At program point p variable i can take the values 0,1,...,99

Thus, 100 possible program states within the loop = a safe upper loop bound



## 1. Program slicing

- ★ **Used to identify variables and statements that may affect the outcome of the exit conditions of a given loop**

- ◆ Remaining variables and statements are removed from the following analysis steps

```
// INPUT = [10..20]
1. int foo(int INPUT) {
2.
3.   int i = 1;
4.   while(i <= INPUT) { // p
5.
6.     i++;
7.   }
8.
9. }
```

The OUTPUT variable does not affect the outcome of the loop exit condition

Thus, statements 2, 5 and 8 can be removed from the loop bound calculation



## 2. Abstract interpretation

- Used to derive, for each program point, a safe approximation of the values held by the remaining variables

- Result used to limit the set of reachable states within the loop = an upper loop bound

```
// INPUT = [10..20]
1. int foo(int INPUT) {
2.
3.   int i = 1;
4.   while(i <= INPUT) { // p
5.
6.     i++;
7.   }
8.
9. }
```

An interval analysis would derive that  $i \in [1..20]$  and  $INPUT \in [10..20]$  at point p

A safe upper loop bound is therefore:  
 $size(i,p) * size(INPUT,p) =$   
 $size([1..20]) * size([10..20]) =$   
 $(20-1+1) * (20-10+1) =$   
 $20 * 11 = 220$



## 3. Invariant analysis

- Used to identify variables which can not be updated within the loop body

- These variables can be safely ignored in the loop bound calculation

```
// INPUT = [10..20]
1. int foo(int INPUT) {
2.
3.   int i = 1;
4.   while(i <= INPUT) { // p
5.
6.     i++;
7.   }
8.
9. }
```

The INPUT variable is invariant within the loop body

A safe upper loop bound is therefore:  
 $size(p,i) =$   
 $size([1..20]) =$   
 $(20-1+1) = 20$

Please note that the derived loop bound is input dependent



## Program slicing details

- Builds upon constructing a *program dependency graph* (PDG)

- Captures *data flow-* and *control flow-* dependencies between statements
- The complete program must be considered
- Takes the input of a pointer analysis

- We perform a *step-wise slicing* (to speed up the overall analysis):

- Slice upon *all* conditions in the program
- Make individual slices on the resulting program slice for each loop of interest



## Abstract interpretation details

- Our abstract domains can handle full ANSI-C

- Incl. pointers, arrays, structs, bit operations, overflow, ...

- We perform the analysis on the NIC intermediate code

- Widening* and *narrowing* used for termination

- Gives safe result but not always the smallest one

- We support the interval domain, the congruence domain, and their product

Interval domain:  
 $i \in [0..9]$  at p

```
1. int i = 0;
2. while(i < 10) {
3.   // p
4.   i += 2;
5. }
```

Product domain:  
 $i \in ([0..9] \cap 0 \pmod{2})$

Congruence domain:  
 $i \equiv 0 \pmod{2}$  at p

Resulting loop bounds:  
 • interval: 10  
 • congruence: inf  
 • product: 5



## Invariant analysis details

- Identifies all variables not updated within the loop body

- Including updates made in functions called from the loop body
- Including updates through pointers

- Additional *single-valued-uses analysis* used

- Uses the result of the abstract interpretation
- Identifies variables within the loop which always have a single value when used

```
1. while(i < 50) { // p
2.   temp = 1;
3.   i = i + temp;
4.   temp = 100;
5. }
```

Variable temp is not invariant in loop body

However, analysis gives that temp = 1 when used

Thus, temp can be ignored in the loop bound calculation



| Program    | Description   | #LC  | No. of loops |       |          | %E | Time (s) |       |
|------------|---|------|--------------|-------|----------|----|----------|-------|
|            |   |      | Bounded      | Exact | Analysis |    |          |       |
|            |   |      | #L           | #B    | #E       |    |          |       |
| adpcm      | Adaptive pulse code modulation algorithm.               | 879  | 27           | 18    | 67%      | 8  | 30%      | 48.6  |
| bs         | Binary search in an array of 15 integer elements.       | 114  | 1            | 0     | 0%       | 0  | 0%       | 0.81  |
| cat        | Counts non-negative numbers in a matrix.                | 207  | 4            | 4     | 100%     | 4  | 100%     | 0.24  |
| cover      | Program for testing many paths.                         | 640  | 3            | 3     | 100%     | 3  | 100%     | 0.32  |
| cre        | Cyclic redundancy check computation on 40 data bytes.   | 128  | 6            | 6     | 100%     | 6  | 100%     | 0.11  |
| duff       | Using "Duff's device" to copy 4i byte array.            | 86   | 2            | 1     | 50%      | 1  | 50%      | 0.04  |
| edm        | Finite Impulse Response (FIR) filter calculations.      | 285  | 12           | 12    | 100%     | 9  | 75%      | 0.71  |
| expint     | Series expansion computing an exponential integral.     | 157  | 3            | 3     | 100%     | 3  | 100%     | 0.04  |
| fac        | Recursive program to calculate factorials.              | 21   | 1            | 1     | 100%     | 1  | 100%     | 0.01  |
| fdct       | Fast Discrete Cosine Transform.                         | 239  | 2            | 2     | 100%     | 2  | 100%     | 0.05  |
| ffft       | Fast Fourier Transform using Cooley-Turkey algorithm.   | 219  | 30           | 7     | 23%      | 3  | 10%      | 5.30  |
| fibcall    | Iterative Fibonacci, used to calculate fib(30).         | 72   | 1            | 1     | 100%     | 1  | 100%     | 0.01  |
| fir        | Finite impulse response filter (signal processing).     | 276  | 2            | 2     | 100%     | 1  | 50%      | 0.28  |
| insert     | Insertion sort on a reversed array of size 10.          | 92   | 2            | 1     | 50%      | 1  | 50%      | 0.54  |
| complex    | Nested loop program.                                    | 64   | 2            | 0     | 0%       | 0  | 0%       | 0.04  |
| jpegint    | Discrete-cosine transformation on 8x8 pixel block.      | 375  | 3            | 3     | 100%     | 3  | 100%     | 0.06  |
| lcdm       | Read ten values, output half to LCD.                    | 64   | 1            | 1     | 100%     | 1  | 100%     | 0.01  |
| ludcmp     | LU decomposition algorithm.                             | 147  | 11           | 6     | 55%      | 5  | 45%      | 247.6 |
| matmult    | Matrix multiplication of two 20x20 matrices.            | 163  | 7            | 7     | 100%     | 7  | 100%     | 0.51  |
| ndiv       | Embedded code with many complex bit operations.         | 231  | 12           | 12    | 100%     | 12 | 100%     | 3.11  |
| ns         | Search in a multi-dimensional array.                    | 535  | 4            | 1     | 25%      | 1  | 25%      | 91.9  |
| nsichneu   | Simulates an extended Petri net.                        | 4253 | 1            | 1     | 100%     | 1  | 100%     | 1.13  |
| prime      | Search in a multi-dimensional array.                    | 535  | 2            | 0     | 0%       | 0  | 0%       | 0.05  |
| quort-ssan | Linear equations by LU decomposition.                   | 123  | 6            | 0     | 0%       | 0  | 0%       | 76.4  |
| quort      | Root computation of quadratic equations.                | 166  | 3            | 1     | 33%      | 1  | 33%      | 0.60  |
| select     | Selects the nth largest number in floating point array. | 114  | 4            | 0     | 0%       | 0  | 0%       | 19.6  |
| statements | Automatic generated code.                               | 1276 | 1            | 0     | 0%       | 0  | 0%       | 1.00  |
| ud         | Linear equations by LU decomposition.                   | 161  | 11           | 11    | 100%     | 10 | 91%      | 0.53  |
| Total      |   | -    | 164          | 104   | 63%      | 84 | 51%      | -     |

Total % bounded loops



## Summary of analysis result

- \* **63% of the loops get upper bounded**
- \* **51% of the loops are given an exact loop bound**
  - ◆ Usual successes: Simple loops with on one or two integer index variables
  - ◆ Usual failures: Complex loops, loops with floating point index variables, or loop with exit conditions using != or ==
- \* **Overall analysis time dominated by abstract interpretation**
  - ◆ Large abstract interpretation analysis time when slicing fails to remove many variables and statements
- \* **The product domain gives tighter loop bounds for 6 additional loops**

## Future work

- \* **Improved slicing on individual loops**
  - ◆ Will produce even smaller program slices
- \* **More powerful relational abstract domains in the abstract interpretation**
  - ◆ Represent constraints between values of variables
  - ◆ Will probably result in smaller states (= tighter loop bounds) but longer analysis times (slicing important!)
- \* **Method to guarantee loop termination**
- \* **Extension with infeasible path analysis**
  - ◆ Will be a combination of program slicing, abstract interpretation and abstract execution

# The End!

For more information:  
[www.mrtc.mdh.se/projects/wcet](http://www.mrtc.mdh.se/projects/wcet)