# Static Loop Bound Analysis of C Programs Based on Flow Analysis and Abstract Interpretation

Marianne de Michiel, Armelle Bonenfant, Hugues Cassé, Pascal Sainrat
Université de Toulouse, Institut de Recherche en Informatique de Toulouse (IRIT)
[michiel, bonenfant, casse, sainrat]@irit.fr
Hipeac European Network of Excellence

## Abstract

*One of the important steps in processing the worst case execution time (WCET) of a program is to determine the loops upper bounds. Such bounds are crucial when verifying real-time systems. In this paper, we propose a static loop bound analysis which associates flow analysis and abstract interpretation. It considers binary operators (+, -, *, \) for the loop increment, nested loops, non-recursive function calls, simple loop conditions (==, !=,<, <=, >, >=, &&) and loop upper bound values (instead of intervals). We present the result of our analysis on the Mälardalen benchmark suite and compare them to the recent work of Ermedahl et al.*

*Keywords: Static Analysis, Loop bounds, Abstract interpretation, Comparison.*

## 1. Introduction

The WCET analysis is necessary when verifying real-time properties. Today, no automatic method for loop bounds analysis can give an exact answer for all loops. Some WCET case studies show that it is important to develop analysis to calculate such information as automatically as possible to reduce the need for manual annotations [12]. Feasible paths through the program have to be studied in order to extract some flow information which is used to statically bind the number of times loops are iterated.

This article presents an approach to calculate upper bounds of loops using flow analysis and abstract interpretation. It is organized as follows: section 2 presents related works. Section 3 presents our method and explains how abstract interpretation proposed by [5] has been adapted and extended to build expressions which represent loop bounds. An illustrating example is given in section 4. Section 5 presents our loop bound analysis results on Mälardalen WCET Benchmark suite [4] and compares them to the results presented in [10]. Section 6 gives our conclusions and presents how to extend our tool to solve more complex loop bounds.

## 2. Related Work

Several researches have been done about loop bounds.

- the source language to which the method is applied (C, RTL, Fortran...),
- the management of interruptions (exit, goto...),
- the type of loop conditions (<, <=, >, >=, = , !=, &&, ||),
- the type of increment used (+k, -k, ×k , /k, multiple increments),
- the management of context or not (without, context function calls are not considered, and only constant bounds are considered),
- the management of nested loops.

Healy et al. [14], Florida State University, show how to analyze RTL program representations. They construct expressions of loops using summations for some dependent loop nests and they evaluate them using an algebraic simplifier. They consider exits of loop and only "+" and "−" constant increment but neither "∗", nor "/". They support <, <=, >, >=, = and ! = conditional operators, but neither logical `and`, nor logical `or`. Their analysis is also restricted to bound expressions which use integer constants or loop variables of nested loops (initial and limit values have to be constant because they do not consider the context).

Kirner bases his work [15] on [14] in order to adapt the analysis to C programs but without nested loops. He introduces, into the source code, annotations on flat loops so that, if no bound can be automatically found, the analysis can revert to user annotations.

Bound-T [1], an industrial tool, proposes a loop bound analysis which estimates ranges and increments of loop counters. It is based on the Omega project which handles Fortran nested loops (for loops) but procedure calls and arbitrary control flow are not processed.

aiT [2], another industrial tool, uses interval analyses on abstract interpretation and data flow analysis but, like previous quoted works, only considers loops counters using $+$ or $-$ operators.

In the recent work of Ermedahl et al. [10], a component of the SWEET WCET analysis tool is used for loop bound computation. This component is based on abstract execution (AE), a form of symbolic execution [14, 17], which itself is based on abstract interpretation. Their analysis uses intervals, where an interval represents all possible values of variables (loop, increment, input, ...). For instance, $i = [1..20]$ represents all concrete states where $1 <= i <= 20$ (the variable may be an input). The user can give a variation interval if it cannot be automatically computed. They combine their interval analysis with an interpretation of the loop counter increment called congruence analysis in order to take into account more counter increment possibilities.

The abstract interpretation [7] is used to consider each variable assigned in a loop. Ammarguellat et al. [5] extend it to consider any assigned variable in spite of function call.

Our method combines loop bound expression building as in Healy [15, 14] to abstract interpretation as in [10] by extending the Ammarguellat approach [5].

## 3. Method

In order to determine loop bounds, we develop a method of static analysis based on the execution flow analysis of C programs and on abstract interpretation. It adapts and extends the approach explained in [5]. It integrates function calls (except recursive ones). It deals with C programs which are correct by hypothesis, and without interruptions.

Most of these restrictions would be relaxed using Calipso [6], a code simplifier. Our analysis considers loop constant increments (+, -, ×, /) which are not modified by the loop but possibly by the program, considers also nested loops, and the following loop condition type: ==, !=,<, <=, >, >=. In some case our method deals with the && loop condition.

Our method consists in three steps:

In step 1, we construct an annotated context tree of the program. A context tree sums up the loop structure of the program in a tree whose nodes are either loops, or function calls. Each function call is completely expanded in the tree to take into account the full context of a loop. The annotations of this context tree are used in the next step to evaluate the bounds for each loop call.

The goal of this step is to automatically calculate data flow information for each loop and each function call. It considers each of them according to the Ammarguellat method i.e. each function independently of the other ones. In fact, we transform the initial loop into an equivalent one (*normal form*) in term of number of iterations and functional effects such as the abstract interpretation becomes easier and, if it is possible, we associate to it an expression of its number of iterations independently of its call context. The expression can represent the exact number of iterations or an expression bounding it. As an example, `for(i=0;i<N;i++)` is iterated `N` times, but we may only state that `for(i=0;(i<N && bool);i++)` is iterated at most `N` times, where `bool` is a boolean expression.

In step 2, we go through the tree in a depth first traversal and we try to associate to each loop of the tree an expression representing its bound depending of its context (function calls and including loops). In fact, for a loop $i$, we construct two expressions for the current context: one which represents the total number of iterations for the whole program execution, called $total_i$, and the other one which represents the maximum number of iterations for each startup, called $max_i$. We apply an extension of the Ammarguellat method described in [5] to evaluate *abstract stores*. An abstract store is a map from an identifier to an expression, or from an identifier to a subscript expression and an expression.

In step 3, for each loop call, we compute the value of the two previous expressions (including the context: for example, if the loop is located in a function, we evaluate the maximum for each call of the function). The maximum for a loop is then the maximum between the loop maxima for all "call contexts".

## 4. Example

In this section, we present a complete example of two nested loops.

```
for ( i=2;  i<8;  i+=2)        // L1
    for ( j=i ; j<8; j+=3);    // L2
```

**Step 1:** Table 1 shows the results of the step 1 analysis.

**Step 2:**

- $L_1$ has no nested loop then

  $e_1 = total_1 = max_1 = \lfloor 5/2 + 1 \rfloor$

- $L_2$ is nested in $L_1$:

  The local context of $L_1$ body without $L_2$ is called

  $\sigma = \{i \leftarrow i + 2\, v_0\}$.

| $Loop_i$ | $v_i$ | $init_i$ | $e_i$ | $c_i$ | $b_i$ |
|---|---|---|---|---|---|
| $L_1$ | $v_0$ | `i=2` | $\lfloor(8-2-1)/2+1\rfloor$ | $i<8$ | `for(j=i;j<8;j+=3) i+=2;` |
| $L_2$ | $v_1$ | `j=i` | $\lfloor(8-i-1)/3+1\rfloor$ | $j<8$ | `j=j+3;` |

**Table 1. Example**

As $e_2$ depends on a variable assigned in $\sigma$ then we apply $EvalStore$ on $e_2$ according $\sigma$ context:

$e_2$=$\lfloor(8-i-1)/3+1\rfloor\{i \leftarrow 2+2v_0\}$ becomes

$e_2$'=$\lfloor(8-(2+2v_0)-1)/3+1\rfloor$

Then we build $total_2$ and $max_2$ expressions:

$$total_2 = \sum_{v_0=0..\lfloor 5/2+1\rfloor-1} \lfloor(8-(2+2v_0)-1)/3+1\rfloor$$

$$max_2 = \max_{v_0=0..\lfloor 5/2+1\rfloor-1} \lfloor(8-(2+2v_0)-1)/3+1\rfloor$$

**Step 3:**

$$total_1 = max_1 = \lfloor 5/2+1\rfloor = 3$$

$$max_2 = \max_{v_0=0..3-1} \lfloor(8-(2+2v_0)-1)/3+1\rfloor$$

As $\lfloor(8-(2+2v_0)-1)/3+1\rfloor$ decreases, changing $v_0$ by 0 gives $max_2$:

$$max_2 = \lfloor 5/3+1\rfloor = 2$$

$$total_2 = \sum_{v_0=0..2} \lfloor(8-(2+2v_0)-1)/3+1\rfloor = 5$$

Although $total_2$ can not be easily evaluated, we estimate an upper bound:

$$total_2 \leq \lfloor \sum_{v_0=0..2} ((5-2v_0)/3+1)\rfloor = 6$$

If the upper bound would have been greater than the product between $max_2$ and $total_1$, our result would have been $2 \times 3 = 6$.

## 5. Measurements and evaluations

We have used Calipso [6] to transform the initial C program in order to remove unstructured statements like *goto*, *break* or *continue*[1] and we have implemented our approach in OCAML using the C parser FrontC [3]. In this part, we give loop bounds and computation times obtained with our method and compare them to the results of [10] on the Mälardalen WCET Benchmark suite [4]. While Ermedahl times were obtained with a 3 GHz PC running Linux, our times have been measured on a Core 2 Duo Processor 2GHz running Linux.

Table 2 shows the results of our loop bound analysis and includes the comparison with [10] obtained with the SWEET tool. Values with a subscripted $E$ like $B_E$ or $Total_E$ represent the statistics given in [10]. L is the number of loops counted according to the context, B the number of loops successfully bounded by our analysis and %B its percentage relative to L.

Rmax and Rtotal gives information about the cause of a loop bound overestimation for maximum and total bounds. The three numbers give the number of overestimated cases caused by, respectively, (1) conditionally executed loops, (2) overestimation in the evaluation of step 3 and (3) too complex loop conditions.

Emax and Etotal is the number of exactly bounded loops and %E its percentage on L. Time is the computation time for each benchmark and is expressed in seconds.

In average, we can resolve about 84% of loops contained in the benchmark suite versus 63% for [10]. In the same way, although our analysis is not parallel, our computation times (about 46s) are faster than Ermedhal approach (about 500s). Yet, there are some benchmarks where our results are worse. `fac` contains recursive functions and cannot be computed, `fir` because of multiple increments, `ndes` gives overestimated bounds because some loops are contained in `if` statements, `ud` contains a `break` tightening the loop bound but that cannot be exploited.

Table 3 shows results on programs of the benchmark suite not presented in [10].

## 6. Conclusion and Further Work

We have presented a static loop bound analysis based on flow analysis and abstract interpretation. It proceed by building a context tree of the program, by evaluating symbolic expressions of the loop bounds and then by resolving these expressions according the running context of the loop. Our first results improve previous works we are aware of [10, 8].

We do not use interval analysis as in [10, 1] because we

[1] Calipso, based on the OCAML parser FrontC, removes from C programs unstructured instructions like `goto, break`, continue or irregular `switch` with a minimized overhead on the execution time.

| Program | L | $B_E$ | B | %B | Rmax, Rtotal | Emax, Etotal | %E | Time |
|---|---|---|---|---|---|---|---|---|
| adpcm | 27 | 18 | 26 | 96% | $1,1,0$<br>$8,1,0$ | 25<br>18 | 92% | 9.233 |
| bs | 1 | 0 | 0 | 0% | | 0 | 0% | 0.08 |
| cnt | 4 | 4 | 4 | 100% | | 4 | 100% | 0.012 |
| cover | 3 | 3 | 3 | 100% | | 3 | 100% | 0.028 |
| crc | 6 | 6 | 6 | 100% | | 6 | 100% | 0.128 |
| duff | 2 | 1 | 2 | 100% | | 2 | 100% | 0.012 |
| edn | 12 | 12 | 12 | 100% | | 12 | 100% | 0.468 |
| expint | 3 | 3 | 3 | 100% | | 3 | 100% | 0.012 |
| fac | 1 | 1 | recursive functions are not supported | | | | | |
| fdct | 2 | 2 | 2 | 100% | | 2 | 100% | 0.032 |
| fft1 | 30 | 7 | 23 | 70% | 0,0,10 | 13 | 37% | 1.116 |
| fibcall | 1 | 1 | 1 | 100% | | 1 | 100% | 0.008 |
| fir | 2 | 2 | 1 | 50% | | 1 | 50% | 0.208 |
| insertsort | 2 | 1 | 2 | 100% | | 2 | 100% | 0.012 |
| jcomplex | 2 | 0 | 0 | 0% | | 0 | 0% | 0.004 |
| jfdctint | 3 | 3 | 3 | 100% | | 3 | 100% | 0.024 |
| lcdnum | 1 | 1 | 1 | 100% | | 1 | 100% | 0.008 |
| ludcmp | 11 | 6 | 11 | 100% | 0,2,0 | 9 | 81% | 0.284 |
| matmult | 7 | 7 | 7 | 100% | | 7 | 100% | 0.012 |
| ndes | 12 | 12 | 12 | 100% | 1,0,0 | 11 | 91% | 0.416 |
| ns | 4 | 1 | 4 | 100% | | 4 | 100% | 0.02 |
| nsichneu | 1 | 1 | 1 | 100% | | 1 | 100% | 32.066 |
| prime | 2 | 0 | 0 | 0% | | 0 | 0% | 0.008 |
| qsort-exam | 6 | 0 | 0 | 0% | | 0 | 0% | 0.052 |
| qurt | 3 | 1 | 3 | 100% | 1,0,0 | 2 | 100% | 0.028 |
| select | 4 | 0 | 0 | 0% | | 0 | 0% | 0.08 |
| statemate | 1 | 0 | 0 | 0% | | 0 | 0% | 0.944 |
| ud | 11 | 11 | 11 | 100% | 0,0,2 | 9 | 81% | 0.040 |
| $Total$ | 164 | | 138 | 84% | 1,3,12<br>8,3,12 | 121<br>114 | 74%<br>70% | 45.26 |
| $Total_E$ | 164 | 104 | - | 63% | | 84 | 51% | 499.25 |

**Table 2. Results of loop bound analysis**

| Program | L | Rmax, Rtotal | Bmax, Btotal | %B | Emax, Etotal | %E | Time |
|---|---|---|---|---|---|---|---|
| bsort100 | 3 | 0,0,1 | 3 | 100% | 3<br>2 | 100%<br>66% | 0.008 |
| compress | 11 | 0,0,1 | 8<br>6 | 73%<br>55% | 7<br>5 | 64%<br>45 % | 0.296 |
| lms | 12 | 0,1,0 | 9 | 75% | 8 | 66% | 0.284 |
| minver | 17 | 0,0,1 | 15 | 88% | 15<br>14 | 88%<br>82% | 0.076 |
| recursion | | | recursive functions are not supported | | | | |
| sqrt | 1 | | 1 | 100% | 1 | 0% | 0.008 |
| st | 7 | | 7 | 100% | 7 | 100% | 0.08 |
| Total | 215 | 1,4,15<br>8,4,15 | 179 | 83% | 162<br>151 | 75%<br>70% | 46.02 |

**Table 3. Results of loop bound analysis for additional programs**

replace loop bounds by expressions containing summations or maximum functions computed using all values of the loop variable. Yet, considering an interval may solve some problems: for instance, in case of arrays or input values coming from external functions, in particular from sensors in nested programs where values are not known but only an interval of inputs values is known. If a loop bound depends on such an input, it may be interesting to consider intervals in our approach. However, to maintain our short execution time, we have to limit interval consideration on very specific context. We can notice that when we cannot evaluate the bound of loop for which $e_i$ has been built, we produce $max$ and sometimes $total$ expressions that the user can use to evaluate manually as annotations.

Some kinds of loops as found in the sinus function are very frequently used in WCET Benchmark, and it is possible to express the three loop bounds of this function according to several input intervals if we recognize this function. It would require introducing patterns of loops for function libraries. Patterns are used for very simple loops by [8]. But, unlike this kind of pattern, our patterns as represented in our loop bound analysis would be more abstract and therefore applicable to more loop configurations. [8] also evaluates lower bounds of loops, whereas our tool does not evaluate them. However, when $max$ is evaluated, $min$ can be evaluated so it may be easy to extend our approach even if the lower bound cannot be evaluated in some cases. This information may be useful for example when we study the WCET with caches, for instance to know if the loop has been executed at least once.

It would be interesting to obtain results of aiT [2] on the Mälardalen WCET Benchmark suite in order to complete our comparison. Unfortunately, such results are not published to our knowledge.

We are currently trying to generalize the `if` instruction evaluation. Today, we consider loops with only a single condition containing $v_i$ expression and $<, <=, >, >=$ ,$==$,$! =$ operators. This could be extended to take into account `or` conditional expressions in loop conditions.

We also consider only one induction variable, monotonically increasing or decreasing variables. We will study an extension of the first step to increase the number of loops considered but it may increase the last step difficulties.

It may also be useful to construct expressions which could be evaluated directly by a mathematical solver. Another further work would be to examine multiple increments by changing the abstract store representation to take into account multiple values for variables.

## References


[1] Bound-t tool homepage, http://www.tidorum.fi/bound-t/, 2005.

[2] ait tool, http://www.absint.com. 2007.

[3] Frontc homepage, http://www.irit.fr/FrontC, 2007.

[4] Wcet project homepage, http://www.mrtc.mdh.se/projects/wcet/, 2007.

[5] Z. Ammarguellat and I. W. L. Harrison. Automatic recognition of induction variables and recurrence relations by abstract interpretation. SIGPLAN Not., 25(6):283–295, 1990.

[6] H. Cassé, L. Féraud, C. Rochange, and P. Sainrat. Une approche pour réduire la complexité du flot de contrôle dans les programmes C. Technique et Science Informatiques, 21(7):1009–1032, 2002.

[7] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 238–252, Los Angeles, California, 1977.

[8] C. Cullmann and F. Martin. Data-flow based detection of loop bounds. In 7th International Workshop on Worst-Case Execution Time Analysis, (WCET'2007), Pisa, Italy, July 2007.

[9] Andreas Ermedahl, A Modular Tool Architecture for Worst-Case Execution Time Analysis, PhD Thesis of Uppsala University, June 2003.

[10] A. Ermedahl, C. Sandberg, J. Gustafsson, S. Bygde, and B. Lisper. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In 7th International Workshop on Worst-Case Execution Time Analysis, (WCET'2007), Pisa, Italy, July 2007.

[11] C. Ferdinand, R. Heckmann, H. Theiling, and R. Wilhelm. Convenient user annotations for a wcet tool. In International Workshop on Worst-Case Execution Time Analysis, (WCET'2003), pages 17–20, 2003.

[12] J. Gustafsson and A. Ermedahl. Experiences from applying wcet analysis in industrial settings. In ISORC '07: Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing, pages 382–392, 2007.

[13] J. Gustafsson, B. Lisper, C. Sandberg, and L. Sjöberg. A prototype tool for flow analysis of c programs. In International Workshop on Worst-Case Execution Time Analysis, (WCET'2002), Vienna, June 2002.

[14] C. Healy, M. Sjödin, V. Rustagi, D. Whalley, and R. V. Engelen. Supporting timing analysis by automatic bounding of loop iterations. Real-Time Syst., 18(2-3):129–156, 2000.

[15] M. Kirner. Automatic loop bound analysis of programs written in C. Master's thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, 2006.

[16] M. de Michiel, A. Bonenfant, P. Sainrat and H. Cassé. Loop normalisation to evaluate maximum number of iteration of loop in WCET context, IRIT/RR–2008-3–EN, http://www.irit.fr, 2008.

[17] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper, Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In Proc. 27th IEEE Real-Time Systems Symposium (RTSS'06), Dec. 2006.