Abstract
Computing Worst Case Execution Time (WCET) is crucial in the area of Real-Time Embedded Systems. In order to ease this computation, we developed a tool, named oRange, able to provide loops upper bounds of C programs. These bounds are obtained by static analysis while combining flow analysis and abstract interpretation. oRange covers binary operators (+, −, ∗) as loop increment, nested loops, non-recursive function calls and simple loop conditions (==, ! =, <, <=, >, >=, &&). We also provide a comparison to the recent work of Ermedahl et al., based on the Mälardalen benchmark suite.

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 [13]. 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 document is a summary of [17], which is 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. Section 4 compares our results to the ones presented in [10]. Section 5 gives our conclusions.

2 Related Work
Several researches have been done about loop bounds. They have different criteria:

- the source language to which the method is applied (C, RTL, Fortran, . . . ),
- the management of unstructured statements (exit, goto, . . . ),
• the type of loop conditions ($<$, $<=$, $>$, $>$=, $!=$, &&, ||),
• the type of increment used (+k, −k, $\times k$, $/k$, 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.

To get a wider description of related work, please refer to [17].

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]. It integrates function calls (except recursive ones). It deals with C programs which are correct by hypothesis, and without interruptions.

Most of these restrictions are relaxed using Calipso [6], a code simplifier. Our analysis considers loop constant increments (+, −, $\times$, $/$) 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.

3 oRange

oRange performs a static analysis on C programs and provides two expressions for each loop (nested or un-nested) in this program. These expressions total and max are defined as follows: total represents the total number of times the body of a loop will be executed, max represents the maximum number of times the body of a loop will be executed among the times this loop is executed. For instance, in the following program:

```c
int boucle (int n){
    
    for (i=0;i<n;i++) ... //loop named Bi
    
    }
```
```c
void main...
    boucle (5);
    boucle (10);
}
totalBi = 15 and maxBi = 10.
```

3.1 Method

oRange is a tool based on Ammarguellat method of *Recognizing Recurrence Relations*. It uses abstract interpretation in order to describe recurrence relations between variables and a symbolic form (expression) of its value. Our own method is developed in three steps.

3.1.1 Identification/Normalisation of loops

First, we use Ammarguellat method to identify increments and increment variables of each loop, then we use these informations to build a normal form for all loops. This step is context insensitive. See [16] for full description of loop normalisation.

3.1.2 Total and Max expressions construction

Second, we build abstract stores describing total and max expressions for each loop. These abstract stores are built context dependently. The construction is done in depth in order to obtain total and max expressions for all loops, including nested ones. We modify the method of recognizing recurrence relations by introducing fixed-point operations.

For each loop from top to bottom, we build its total and max expressions by replacing their abstract store obtained by recognizing recurrence relations from bottom to top.

Note that when a total expression can not be built, if we have the corresponding max expression and the just upper loop total expression, oRange will perform an approximation (overestimation) for the total expression being built.

3.1.3 Total and Max expression computation

The third step performs a computation of the formulae obtained at the previous step and a propagation of the context from top to bottom.

Because of the nature of some expressions (sum of floor values), there are some overestimations.
4 Results

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 a 2-GHz Core 2 Duo Processor running Linux.

<table>
<thead>
<tr>
<th></th>
<th>P</th>
<th>BLT</th>
<th>Ratio</th>
<th>Exact</th>
<th>T (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Our analysis</td>
<td>33</td>
<td>138</td>
<td>84%</td>
<td>total 121\max 114</td>
<td>45.26</td>
</tr>
<tr>
<td>Ermedahl</td>
<td>28</td>
<td>104</td>
<td>63%</td>
<td>84</td>
<td>499.25</td>
</tr>
<tr>
<td>Total</td>
<td>35</td>
<td>164</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

P is the number of programs being analysed. BLT is the total of bounded loops. Exact is the number of expression oRange obtains without any approximation/overestimation.

oRange finds 50% of additional loops bounded in duff, fft1, lcdnum, ns, qurt. Ermedahl et al. obtain better results on fac, fir, ndes, ud because either oRange does not provide results (recursivity) or oRange over-estimates (multiple increment, break, if statements...). On the other hand, oRange analyses 5 additional programs: compress, lms, minver, sqrt, st

5 Conclusions

We have presented a static loop bound analysis based on flow analysis and abstract interpretation. It proceeds 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 are currently trying to generalize the if instruction evaluation. Today, we consider loops with only a single condition containing $v_i$ expression and $<,\leq,>,\geq,=,!=,$ 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 possible values for the increment variable.

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.
References


