Static Detection of Parametric Loop Bounds on C Code

Olha Honcharova, Christoph Cullmann and Florian Martin

AbsInt Angewandte Informatik GmbH, Science Park 1, 66123 Saarbrücken, Germany, (honcharova, cullmann, florian)@absint.com

Abstract: Calculation of upper loop bounds is a key component for static derivation of precise WCET estimates. This paper describes a novel data-flow based analysis to calculate conservative loop bounds on the source level, where the delivered bounds are parametric with respect to variables and expressions constant within the loop. The new method is compared with two loop-bounding tools and demonstrates more precise bounds.

Keywords: parametric loop bound analysis, static analysis, source code analysis

1. INTRODUCTION

Loop bounds computed on source code can be useful in a variety of cases, e.g. in parametric WCET analyses like the one in Hübner’s (2006), for early WCET estimation (TimingExplorer1), for automatic parallelization, vectorization and for replacement of manual loop annotations.

Applied together with object code based analysers for loop bounds, analyses operating on source code can provide more flexibility and precision, as the source code provides much information that is lost in the machine code. To apply the results obtained on the source code to the binary code, it is necessary to transform them by a WCET aware compiler.

We are going to apply in this analysis a parametric approach that gives us a gain in efficiency by allowing to analyze a program as a whole at once, in a context-independent way. The idea is to express the upper loop bound as a formula depending on parameters – variables and expressions being constant within the loop body. The symbolic formula is constructed once for each loop. Afterwards, from the formula instantiated with values, which are acquired externally (from value analysis of a WCET calculation framework), a concrete loop bound can be computed with low computational effort.

This paper presents a parametric data-flow based loop bound analysis on C code and compares its implementation, OLoop, to the loop bounding tool of the SWEET WCET tool by Ermethal et al. (2007), and to the oRange loop bounding tool by Cassé et al. (2008).

The aim of this analysis is to obtain bounds for each loop that are conservative, but as precise as possible. To keep the analysis fast, scalable and terminating, we confined it to “counter-type” loops with integer loop counters and with counter modifications by fixed increments, decrements or assignments only. As exit conditions only simple comparisons are allowed. In practice it is no essential restriction, as this type of loop is the most often used in embedded systems.

The rest of the document is organized as follows. Section 2 provides a brief explanation of the analysis, shows a running example and then describes the method and the framework of the analysis. Section 3 gives an overview over existing researches on loop bound analysis. Evaluation of the work is given in Section 4. Finally, Section 5 concludes the paper.

2. ANALYSIS

2.1 Overview

A loop bound is a maximal iteration number of a loop; in this work it is calculated after looking over all available loop exits as a minimal value among their iteration numbers. An iteration number for a specific loop exit is counted as number of iterations until a guaranteed exit via this exit. This calculation rule is applied if none of the iteration counts is infinite, and for the latter case the loop is unbounded. Taking the minimal iteration number from all exits is sound, as we only consider exits which are checked in every loop iteration.

Such an exit-point dependent bound relies on its conditional exit, where the condition is further called an exit condition. A variable, on which an exit condition relies, is called a loop counter.

To compute an exit-point specific loop bound, we need to know all updates of the loop counter within the loop and to check whether these updates are constant in the loop body. To be sure that variables, on which the counter updates rely, are constant within the loop, it is necessary to track their updates. It is also important to assure that these variables are not changed via pointer accesses.

The implementation of our method is called OLoop. It is based on a collection of data-flow analyses (see the book of Nielsen et al. (1999) for more information on data-flow analysis), using the framework of the Static Analysis Tool Integration Engine (SATIntE) by Schordan (2007), the

1 Created by AbsInt Angewandte Informatik GmbH
LLNL-ROSE compiler, and the Program Analyzer Generator (PAG) by Martin (1999), with an additional C++
module combining the collected data. The framework is
depicted in Figure 1: the analysis specification is given to
SATInE, which in its turn uses PAG and LLNL-ROSE for
generation of the analysis.

![Diagram of analysis framework]

**Fig 1. Tool's framework**

Our implementation involves the following parts:
- **loop detection analysis**
  for finding loops and recursion in the source code,
- **loop-based dominator analysis and exit conditions analysis**
  to discover exit conditions of a loop,
- **pointer analysis**
  to detect variables of which an address was taken and
  which thus may be changed via pointers,
- **used variables analysis**
  to detect variables being used in a loop,
- **changes analysis**
  to discover all changes of variables in a loop,
- **equations construction phase**
  integrating the data of the previous analyses into
  formulas for loop bounds.

### 2.2 Example

To demonstrate how the analysis works, be the following
program snippet given as input:

```c
do
  {  
    for (x = 0; x < b; x--) // inner loop
        x = 2;

    y = 10;
    if (y >= b)
        break;

    y = 5;
  } while (1); // outer loop
```

In the following we describe the analysis stages and their
application to this example. The run of the analysis on
this example is depicted in Figure 2.

#### 2.3 Loop Detection

The first stage of this loop bound analysis is the detection
of loops. The analysis recognizes while, do..while and for
loops on the control-flow graph of the program, generated
from the source code. Recursions are detected and all the
loops inside of recursive functions are not considered —
quite behavior demonstrates the orange loop bounding
tool.

The loop detection analysis is implemented as a simple
dominator analysis (see Nielson et al. (1999) for infor-
manation on dominator analysis), extended with a function
call list for recursion detection and a mapping to store detected
loops. In the latter mapping **loop headers**, which are the
nodes containing syntactical loop constructs, are mapped
to the sets of the corresponding loop body nodes.

**Example**  Loop detection recognizes two loops from the
snippet and assigns them unique identifiers 1 and 2. It also
creates a mapping for querying a set of nodes belonging to
a specific loop for the later analyses.

#### 2.4 Exit Conditions

The aim of this analysis is to detect and to parse loop exit
conditions. As only exits controlled by a single condition
are considered, a helper **loop-based dominator** analysis is
used. It identifies for each loop the conditions leading
to an exit and being visited in each iteration as a list
of nodes, which have a direct successor outside the loop
and dominate the header by means of the **loop-based
domination**.

Loop-based domination is a usual dominator analysis, run
inside the loop, ignoring all incoming and outgoing edges
of the loop.

Next, as we are here interested on conditions rather than
other statements, all nodes dominating the loop header
but not being condition checks, are removed. Further, the
exit conditions analysis collects and stores the conditions
as a mapping of loop headers to sets of mappings, each
of which constitutes in its turn a mapping of a node with
condition to a parsed condition.

An exit condition might be situated either in a loop header
or in a loop body. Supported exit condition types are the non-compound expressions with comparison operators
(\(=, \neq, >, \geq, <, \leq\)) and booleans (or, more precisely for
\(C\), integers compared to zero).

For simplicity, we concentrate on the case that a counter
variable is on the left and an expression is on the right.

Each collected condition forms a triple of a loop counter (a
variable on the left in the condition check), compare sign
and an expression (constant expression on the right in the
condition check, containing no assignments, increments,
etc.). It is possible to extend the analysis with some more
condition types.

**Example**  For the inner loop 1 the analysis finds the exit
condition \(x \geq b\), represented as a triple \((x, \geq b)\).
2. Pointer Analysis

1. Loop Detection
   inner loop: 1
   outer loop: 2

4. Exit Conditions
   1: (x ≥ b), (y ≥ b)
   2: (false), (y ≥ b)

3. Used Variables
   1: x, b
   2: x, b, y

5. Changes Analysis
   full iteration updates:
   1: x → x₀ + 1
   2: x → ∞, y → y₀ + 15
   partial updates
   2: x → ∞, y → y₀ + 10

6. Equations Construction

bound for inner loop 1 (for):

\[ x₀ \geq b ? \]
\[ \text{yes:} \]
\[ 0 \text{ loop not entered} \]
\[ \text{no:} \]
\[ 1 > 0 ? \]
\[ \text{yes:} \]
\[ \frac{b - x₀}{1} \text{ parametric bound} \]
\[ \infty \text{ condition will never hold} \]

bound for outer loop 2 (do..while):

\[ y₀ + 10 \geq b ? \]
\[ \text{yes:} \]
\[ 1 \text{ condition holds in first iteration} \]
\[ \text{no:} \]
\[ 15 > 0 ? \]
\[ \text{yes:} \]
\[ y₀ + 10 + 15 \geq b \]
\[ \text{yes:} \]
\[ 2 \text{ condition holds in second iteration} \]
\[ \text{no:} \]
\[ \frac{b - y₀ - 10}{15} + 1 \]
\[ \infty \text{ condition will never hold} \]

Fig 2. Run of the analysis.

The formulae can be statically simplified, but this makes not much sense on this stage, as this should be performed later anyway: after substitution of parameters with their known values.

For the outer loop 2 two conditional exits are detected: (false, false) for the condition in the loop header and (y ≥ b).

2.5 Pointer Analysis

This analysis is aimed to solve the problems involved with pointer existence: whenever an address of a variable is picked, it is possible to change that variable at some point through a pointer access. For now, this analysis is a small but safe replacement for a more thorough pointer analysis like Stattelmann (2008). The current version eliminates the possibility of relying on variables which could be changed arbitrarily by collecting all variables whose address is taken. In the future, it will be easy to switch to a more advanced pointer or alias analysis.

Example For this example no pointers were found.
2.6 Used Variables

Used variables detection is another data-flow analysis, necessary to identify loop counters and to determine their changes within a loop. Data is collected to a mapping; each loop is associated with a set of variables used in it.

This analysis requires to have unique identifiers for all variables, which is not guaranteed, if names of variables are used as identifiers. For those reasons, variable renaming is used the same way as Stattelmann (2008) did by means of the LLNL-ROSE compiler. After variable renaming, the analysis operates on unique identifiers which are then used by the analysis instead of variables.

Example  Used variables of the inner loop are \( x \) and \( b \), the ones of the outer loop are \( x \), \( b \) and \( y \).

2.7 Changes Analysis

For each potential loop counter detected in the previous phase, the changes analysis calculates for each loop exit point a set of expressions, called updates, that indicate how the counter is modified from the entry of the loop routine to this point in one iteration. All variables are initialized at the loop begin with a special value: identity.

The data-flow analysis looks for all operators, which possibly change variables, and parses the interesting updates, storing them as increments, decrements, assignments (=, +, -, ++, --), if the update is linear and plain.

Non-linear updates, indirect assignments in condition checks, compound updates (assignments inside of other assignments) of variables are not considered because of complexity. Nonetheless, the analysis stays safe by assigning to all variables updated in that way an infinite update, and excluding them from future considerations as a possible loop counter.

A data-flow analysis is specified by a set of rules on data flow, namely the transfer function (for transforming data depending on an instruction) and the combine function (how data is merged when two flows are meeting into one). Examples for understanding the transfer functions of the changes analysis can be found in Figures 3 and 4.

Fig. 3. Transfer function of changes analysis, part 1.

For loop 1 the changes analysis finds updates to variable \( x \) compared to its initial value at the loop header, \( x_0 \). Alternatively possible updates are collected into sets. Mappings are labeled by the numbers of the loops to which they refer.

The work of the combine function is illustrated by Figure 5: two different updates for a variable are merged into a set of both updates, unless one of the updates is infinite. For the latter case an infinite update is released.

Fig. 4. Transfer function of changes analysis, part 2.

1: \( x \rightarrow x_0 + 1 \),
2: \( y \rightarrow (y_0 + 2, 9) \)
3: \( y \rightarrow (y_0 + 2, 9) \)

Fig. 5. Combine function of the changes analysis.

Additionally, a data-flow analysis might have a widening function for speeding up the analysis. In the changes analysis, widening overcomes the problem of updates in nested loops: it prevents an endless computation of updates for variables that are used in the current loop but are modified in one of the inner loops. See Figure 6 for an example: different values are merged into an undefined update, unless one of the updates being merged is an identity, e.g. \( y \rightarrow y_0 \).

Fig. 6. Widening function in the changes analysis.

At present, a loop counter may have only a constant update, but in the future it is possible to use updates collected by the changes analysis to allow in some cases simple non-constant updates.

Example  The inner loop has one exit and two updates of variable \( x \) that can be easily combined into one. The change of the variable till the exit point in one iteration is:

1: \( x \rightarrow x_0 + 1 \)

The outer loop has two exit points and contains updates for two variables \( x \) and \( y \). For the exit condition in the loop header the collected single-iteration update is called a full-iteration update and is:

2: \( x \rightarrow \infty, y \rightarrow y_0 + 15 \)
Variable $x$ was assigned an undefined update, as it is changed by the inner loop and is thus too hard to calculate.

The update for the other exit condition that is in the loop body, is called a partial update:

$$2: \ x \rightarrow \infty, \ y \rightarrow y_0 + 10$$

2.8 Equations Construction

The loop bound construction module is implemented as a C++ class that processes the information and outputs text formulas for bounds.

On this stage the collected data is combined into parametric loop bounds after a number of checks:

- are there any exit conditions of unsupported types,
- do exit conditions contain variables with possible modification via pointers,
- is the right-hand side expression of an exit condition changed in the loop,
- is the left-hand side expression of an exit condition not a single variable,
- do updates of the loop counter rely on variables not constant within the loop,
- can the counter be not updated at all,
- does the exit condition in the loop header hold right away.

For all these special cases, a special bound value is assigned: $\infty$, 0 or 1. Otherwise, the algorithm proceeds.

For every exit point an individual bound is calculated, and the minimum of them becomes the loop bound.

Bounds are constructed on the base of a pair of general equations, derived from the corresponding exit conditions and taking into account the placement of the exit (header of post-test or pre-test loop, or outside the header).

For example, the following general equations are obtained from exit condition $counter > c$:

$$counter_{init} + n * incr_{full} > c$$

if the exit is in the loop header, and

$$counter_{init} + (n - 1) * incr_{full} + incr_{part} > c$$

if the exit is in the loop body.

Here

- $counter_{init}$ is the initial value of the loop counter,
- $c$ is some constant expression,
- $n$ is the number of loop iterations, no matter partial and full,
- $incr_{full}$ is the loop counter increment after a full iteration,
- $incr_{part}$ is respectively an increment of the counter up to the given loop exit.

The resulting loop bound is a text formula with conditional (condition ? formula_if_true : formula_if_false) and arithmetic operations easy to calculate by any calculator, given all necessary values of parameters.

Example: For the inner loop, the equation looks like

$$x_0 + n * 1 \geq b,$$

where $x_0$ is the initial value of the loop counter, $n$ is the loop bound, 1 is the full update of the loop counter $x$.

For the outer loop, one exit condition gives no equation, and the other yields

$$y_0 + (n - 1) * 15 + 10 \geq b,$$

where 15 is the full update of the loop counter, and 10 is a partial one.

These equations solved with respect to $n$ yield the bounds with respect to specific exits. See Figure 2 for commented parametric bounds for this example.

3. RELATED WORK

Several groups have worked on the problem of determining loop bounds at the source level.

Healy et al. (2000) 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 loop exits, conditional loop exits with simple comparison operators and either positive or negative constant increments. Their analysis is 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). Analysis data is fed into an external symbolic algebra system that solves the equation systems for loop bounds.

Kirner (2006) bases his work on the work of Healy et al. (2000) in order to adapt the analysis to C programs, but without nested loops. He introduces source code annotations for flat loops so that, if no bound can be automatically found, the analysis can fall back to user annotations.

The TuBound tool by Prantl et al. (2008a) is a portable tool for high-level WCET analysis of C++ programs. It includes a term-based loop bounder TeBo, entirely written in Prolog. The loop bounding algorithm exploits several features of Prolog: to calculate loop bounds, a symbolic equation is constructed, which is then solved and reduced by a set of term-rewriting rules. For solving the equations constraint logic programming over finite domains is used. Loop bounds and constraints are derived for iteration-variable based loops. TeBo also handles nested loops, in particular the ones with triangular iteration space. Multiple exits are supported, although multiple increments are not, as well as multiple different loop counters for different loop exits, see Prantl et al. (2008b). Unfortunately, we found no detached from the TuBound tool evaluation of TeBo, and therefore did not compare it to OLoop.

In the recent work of Ermedahl et al. (2007), a component of the SWEET WCET analysis tool is used for loop bound computation. This component is based on abstract execution, a form of symbolic execution, which is itself based on abstract interpretation. Their analysis uses intervals, where an interval represents all possible values of variables. Their interval analysis is combined with an interpretation of the loop counter increment called congruence analysis in order to take into account more counter increment possibilities. Exit conditions might be compound.
The method of the oRange loop bounding tool by Cassé et al. (2008) combines loop bound expression building as Healy et al. (2000) to abstract interpretation as Ermelhahl et al. (2007) by extending the approach of Ammaguellat and Harrison (1990). They have used a parser from C, based on OCAML, transforming the initial C program in order to remove unstructured statements like goto, break, continue or irregular switch. Multiple updates of a loop counter are not supported. The update in a loop must be a constant within the loop increment. The method considers nested loops and conditional exits with comparison operators. In some cases, this method also deals with && in loop condition.

Beside the above mentioned analyses working on the source level, there are loop analyses available for the object code level, too.

For example, the WCET analyzer aIT includes such a loop analysis introduced in Cullmann and Martin (2007). This analysis is similar to OLoop as it is based on data-flow analysis, too, and implemented in the PAG-framework. But in contrast to OLoop, it does not calculate parametric bounds and is combined with a value analysis to calculate concrete bounds. As it works on the object code level, the analysis must cover the semantics of the underlying hardware for each target and the notion of variables like in C is unknown, and the analysis must work directly on the hardware registers and memory addresses.

OLoop is context-independent and parametric unlike SWEET and oRange. It is driven by abstract interpretation as oRange and SWEET. It does not perform any code transformations like oRange. Unlike OLoop, oRange and SWEET take into account some compound conditions. But OLoop, unlike oRange and also in many cases SWEET, supports multiple updates of a loop counter. For more detailed information on OLoop please refer to the master’s thesis of Honcharova (2009).

4. EVALUATION

For our experiments we use a selection of benchmarks that were collected by the Real-Time Research Center at the University of Mälardalen – the Mälardalen WCET benchmark suite, which also was earlier used for evaluation of the oRange and the SWEET tools. The data-flow analyses for this evaluation were built with SATIRE 0.8.2.

Table 1 summarizes the results of the evaluation and compares OLoop to the two aforementioned tools. Values subscribed with E like $E_E$ represent the statistics given by Ermelhahl et al. (2007), values subscribed with $O$ were provided by Cassé et al. (2008).

Reasons for overestimation in sample programs are all specified in the table, see Overestimation column.

For a number of programs ($adpcm, fir, fft, lucent, sqrt$ and $insert100$) the analysis demonstrated better results than the other tools.

On the other hand, for some examples worse results were obtained:

- for $duff$, as its syntax was not recognized by SATIRE. It contains loop and switch constructs that are irregularly included into each other;
- for $insertsort$, where compound exit conditions occur and where a condition’s left side is changed within the loop;
- for $compress$, where compound exit conditions, complex counter updates are present as well as other complex calculations;
- for $lma$, where some non-integer counters exist.

As can be seen, the analysis demonstrates a result of 81% safely bounded loops and 80% precisely bounded loops. Here a formula is called precise if, instantiated with precise values of parameters from an external value analysis, it becomes an actual exact loop bound.

The precise loop bounding percentage assumes precise values of parameters acquired externally, and would decrease, if some of these values are not precise enough. However, most of the loops analyzed in the benchmark are simple counter loops (with an integer counter, a simple exit condition, single arithmetic update of variable and single exit), for which precise values of parameters are easy to obtain. 135 loops of 169, or 80%, are of that kind and hence it is expected that precision of the analysis on real values of parameters would not differ much from the estimated one.

Table 2. Timing of loop bound analysis

<table>
<thead>
<tr>
<th>Program</th>
<th>Loc</th>
<th>Loops</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop3.c</td>
<td>82</td>
<td>150</td>
<td>1.5</td>
</tr>
<tr>
<td>nsichneu.c</td>
<td>4253</td>
<td>1</td>
<td>18</td>
</tr>
<tr>
<td>statemate.c</td>
<td>1276</td>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>other</td>
<td>64-579</td>
<td>0.18</td>
<td>&lt; 1</td>
</tr>
</tbody>
</table>

Table 2 gives an idea on the timing of the analysis. Most programs were analyzed in less than a second, while a couple of others needed more time to run. loop3 is a short program with lots of loops; statemate is an automatically generated code of more than 1000 lines; nsichneu simulates an extended Petri net and contains a single loop with body of 4000 lines of code. The reason of the long execution time for these programs lies on the large volumes of data flowing along the control-flow graph of some of the auxiliary flow analyses.

5. CONCLUSION

The presented parametric loop bound analysis is based on the concept of data-flow analysis, which is driven by abstract interpretation, and was built with help of SATIRE, LLNL-ROSE compiler and PAG.

Loop bound detection proceeds by gathering necessary information (set of loops and their nodes, set of used variables and their linear updates, loop exits and their conditions) in auxiliary flow analyses and by processing all the gathered data in a module written in C++. The latter module performs reasoning on the loop types, exit condition types, types of updates and outputs to some extent simplified parametric formulas, relying on variables constant within the loop.

The analysis is restricted to loop counters of integer type. As there is no architecture independent semantics of C on
Table 1. Results of OLoop, compared to SWEET and oRange

<table>
<thead>
<tr>
<th>Program</th>
<th>Loops</th>
<th>CL</th>
<th>Overestimation</th>
<th>B</th>
<th>E</th>
<th>%E</th>
<th>%E_E</th>
<th>%E_O</th>
</tr>
</thead>
<tbody>
<tr>
<td>adpcm.c</td>
<td>18 16</td>
<td>[2]:1, [1]:1</td>
<td>17 17</td>
<td>94%</td>
<td></td>
<td>30%</td>
<td>92%</td>
<td></td>
</tr>
<tr>
<td>bs.c</td>
<td>1 0</td>
<td>[1]:1, [4]:1</td>
<td>0 0</td>
<td>0%</td>
<td></td>
<td>0%</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>cnt.c</td>
<td>4 4</td>
<td>[1]:1, [2]:1</td>
<td>4 4</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>cover.c</td>
<td>3 3</td>
<td>[1]:1, [2]:1</td>
<td>3 3</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>ctc.c</td>
<td>3 3</td>
<td>[1]:1, [2]:1</td>
<td>3 3</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>dsft.c</td>
<td>2 1</td>
<td>syntax not supported by SATIRE</td>
<td>0%</td>
<td></td>
<td>50%</td>
<td>100%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>edn.c</td>
<td>12 12</td>
<td></td>
<td>12 12</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>expr.c</td>
<td>3 3</td>
<td></td>
<td>3 3</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>fac.c</td>
<td>1 1</td>
<td>recursive functions not supported</td>
<td>0%</td>
<td></td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>fdct.c</td>
<td>2 2</td>
<td></td>
<td>2 2</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>ff.c</td>
<td>10 7</td>
<td>[1]:1, [4]:1, [8]:3</td>
<td>7 7</td>
<td>94%</td>
<td>10%</td>
<td>94%</td>
<td>10%</td>
<td></td>
</tr>
<tr>
<td>friball.c</td>
<td>1 0</td>
<td>[2]:1</td>
<td>0 0</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>fr.c</td>
<td>2 2</td>
<td></td>
<td>2 2</td>
<td>100%</td>
<td>50%</td>
<td>100%</td>
<td>50%</td>
<td></td>
</tr>
<tr>
<td>insert.c</td>
<td>2 1</td>
<td>[1]:1, [2]:1</td>
<td>1 1</td>
<td>100%</td>
<td>50%</td>
<td>100%</td>
<td>50%</td>
<td></td>
</tr>
<tr>
<td>jcomplex.c</td>
<td>2 0</td>
<td>[1]:2</td>
<td>0 0</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>jactint.c</td>
<td>3 3</td>
<td></td>
<td>3 3</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>lcdnum.c</td>
<td>1 1</td>
<td></td>
<td>1 1</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>lastrng.c</td>
<td>11 11</td>
<td>[2]:1</td>
<td>11 11</td>
<td>91%</td>
<td>91%</td>
<td>91%</td>
<td>91%</td>
<td></td>
</tr>
<tr>
<td>matmul.c</td>
<td>5 5</td>
<td></td>
<td>5 5</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>ndec.c</td>
<td>12 12</td>
<td></td>
<td>12 12</td>
<td>100%</td>
<td>91%</td>
<td>100%</td>
<td>91%</td>
<td></td>
</tr>
<tr>
<td>ns.c</td>
<td>4 4</td>
<td></td>
<td>4 4</td>
<td>100%</td>
<td>25%</td>
<td>100%</td>
<td>25%</td>
<td></td>
</tr>
<tr>
<td>nsectc.c</td>
<td>1 1</td>
<td></td>
<td>1 1</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td></td>
</tr>
<tr>
<td>qsort-exa.c</td>
<td>6 0</td>
<td>[2]:5, [4]:1, [7]:1</td>
<td>2 2</td>
<td>94%</td>
<td>5%</td>
<td>5%</td>
<td>5%</td>
<td></td>
</tr>
<tr>
<td>qsort.c</td>
<td>1 1</td>
<td>[1]:1</td>
<td>1 1</td>
<td>100%</td>
<td>33%</td>
<td>100%</td>
<td>33%</td>
<td></td>
</tr>
<tr>
<td>queue.c</td>
<td>4 4</td>
<td>[2]:2, [7]:4</td>
<td>0 0</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>state.c</td>
<td>0 0</td>
<td></td>
<td>0 0</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>ud.c</td>
<td>11 11</td>
<td></td>
<td>11 11</td>
<td>100%</td>
<td>91%</td>
<td>100%</td>
<td>91%</td>
<td></td>
</tr>
<tr>
<td>bsort.c</td>
<td>3 3</td>
<td></td>
<td>3 3</td>
<td>100%</td>
<td>60%</td>
<td>60%</td>
<td>60%</td>
<td></td>
</tr>
<tr>
<td>compress.c</td>
<td>7 2</td>
<td>[2]:2, [4]:1, [5]:1, [7]:1</td>
<td>2 2</td>
<td>94%</td>
<td>45-64%</td>
<td>45-64%</td>
<td>45-64%</td>
<td></td>
</tr>
<tr>
<td>fms.c</td>
<td>6 6</td>
<td>[6]:4</td>
<td>6 6</td>
<td>90%</td>
<td>90%</td>
<td>90%</td>
<td>90%</td>
<td></td>
</tr>
<tr>
<td>minor.c</td>
<td>17 15</td>
<td>[4]:2</td>
<td>16 15</td>
<td>68%</td>
<td>68%</td>
<td>68%</td>
<td>68%</td>
<td></td>
</tr>
<tr>
<td>recursion.c</td>
<td>0</td>
<td>recursive functions not supported</td>
<td>0%</td>
<td></td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>sofze.c</td>
<td>1 1</td>
<td></td>
<td>1 1</td>
<td>100%</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td></td>
</tr>
<tr>
<td>sl.c</td>
<td>4 4</td>
<td></td>
<td>4 4</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td>100%</td>
<td></td>
</tr>
</tbody>
</table>

| Total   | 169 135 | 137 133 | 79% | 51%  | 70.75% |

<table>
<thead>
<tr>
<th>%E</th>
<th>%E_E</th>
<th>%E_O</th>
</tr>
</thead>
<tbody>
<tr>
<td>60%</td>
<td>45-64%</td>
<td>45-64%</td>
</tr>
<tr>
<td>81%</td>
<td>63%</td>
<td>64%</td>
</tr>
</tbody>
</table>

Loops - number of loops in the program
CL - number of counter loops
Overest. - reasons for overestimation and number of loops affected:
1 - condition's right-hand side expression is changed in the loop
2 - compound/complex exit conditions (function calls, array accesses, etc)
3 - some important variable might be changed via pointer
4 - not supported updates of counter (shifts, non-constant, etc)
5 - nested assignments or updates in conditions
6 - non-integer type of variables
7 - too complex calculations (e.g., loop counter changed in nested loop)
B - number of loops bound by OLoop
E - number of loops precisely bound by OLoop
%E - percentage of loops precisely bound by OLoop
%E_E - percentage of loops precisely bound by SWEET
%E_O - percentage of loops precisely bound by oRange
%B - percentage of loops bound by OLoop
%B_E - percentage of loops bound by SWEET
%B_O - percentage of loops bound by oRange

overflows in signed integers, the current analysis does not consider overflows. The analysis is context-free: each loop is analyzed only once, as it appears in the source code. Loops in recursive functions are not considered.
For the loop bound calculation loop’s conditional exits are considered (at loop headers and the ones performed via break, return); supported condition types are the ones with a single comparison operator or a boolean.

The analysis has been evaluated on the Mälardalen WCET benchmark suite. The evaluation showed that 81% of loops were assigned a safe bound. 79% of loops were exactly bounded, which is better than results of the SWEET and the oRange bounding tools.

The framework used for implementation has currently some limitations in the C parser. As SATHEX develops, they hopefully will be eliminated. To achieve better efficiency and timing of OLoop, especially on larger code, there could be several optimizations held for the internal data structures. Furthermore, the analysis would profit from adding support of the goto loop exit, and it might also be equipped with detection of loop bounds for simple goto loops.

REFERENCES


