Data-Flow Based Detection of Loop Bounds

Christoph Cullmann and Florian Martin
AbsInt Angewandte Informatik GmbH
Science Park 1, D-66123 Saarbrücken, Germany
{cullmann,florian}@absint.com, http://www.absint.com

Abstract

*To calculate the WCET of a program, safe upper bounds on the number of loop iterations for all loops in the program are needed. As the manual annotation of all loops with such bounds is difficult and time consuming, the WCET analyzer **aiT** originally developed by Saarland University and AbsInt GmbH uses static analysis to determine the needed bounds as far as possible.*

*This paper describes a novel data-flow based analysis for **aiT** to calculate the needed loop bounds on the assembler level. The new method is compared with a pattern based loop analysis already in use by this tool.*

1. Introduction

To calculate the WCET for a program, safe upper bounds for the iterations of all included loops must be known. To get a precise WCET estimation, lower bounds should be known, too.

As programs tend to contain many loops with bounds depending on the call sites of the surrounding routine, relying on user annotations for loop bounds would cause too much work for the user. Beside that, there is also the inherent danger that user-annotated bounds could contain errors, as they need to be kept up to date while the application code is changing. Therefore **aiT** aims at deriving safe loop bounds automatically by using a static analysis.

Until now, a pattern-based approach for loop bound detection is used. This method needs adjustments for all supported compilers and in some cases even different optimization levels. While experience has shown that this works well for many simple loops, no bounds are detectable for more complex loops with multiple modifications of the loop counter inside one iteration.

To overcome these restrictions, we introduced a new method for loop bound detection that uses an interprocedural data-flow analysis to derive loop invariants from the semantics of the instructions. This new analysis does not depend on the used compiler or optimization level but only on the semantics of the instruction set for the target machine. It is able to handle loops with multiple exits and multiple modifications of the loop counter per iteration including modifications in procedures called from the loop. Additional, it detects and handles overflows of size limited datatypes.

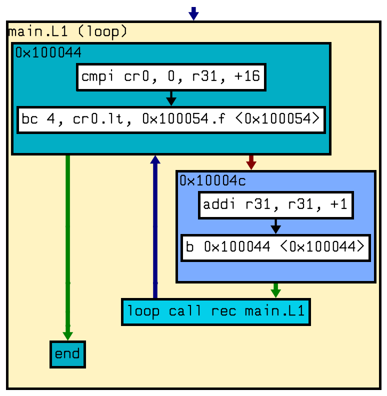
In this section, we describe the techniques behind the old and new loop analyses, compare their results, and provide insight on how the new analysis will be used in **aiT**. First we start in Section 2 with introducing the common basis of both analyses. In Section 3 two small examples for loops are shown that will be used later as running examples to illustrate the application of both analyses. Section 4 will cover the pattern-based approach. Then we introduce the new data-flow based approach in Section 5 and compare both analyses in Section 6. Finally we show how the new analysis is integrated into the WCET Analyzer **aiT** in Section 7.

2. Common Basis for Both Analyses

As both loop analyses have been developed to be used as part of the WCET Analyzer **aiT**, they are using the **aiT** framework presented in [3]. In particular, they operate on a control flow graph which is reconstructed from the machine executable (see [10]) and in which all loops have been transformed to tail-recursive routines by a loop transformation (described in [7]). The next section will show two example loop routines, which are used in the subsequent description of both analyses. A loop iteration equals one execution of the loop routine.

While the **aiT** framwork and the presented loop analyses work on the compiled executables, there are other approaches that work on the level of the programming language. For example in [6] and [5] a framework is described that works on the C sources of a program to calculate a WCET and the therefore needed loop bounds.

To avoid code duplication, the analyses use the existing value analyzer of the framework to query the addresses of



```
cmpi cr0, 0, r31, +16
bc 4, cr0.lt, 0x100054.f <0x100054>
addi r31, r31, +1
b 0x100044 <0x100044>
```

```
while (r31 < 16) // 0x100044
{
  r31 = r31 + 1; // 0x10004c
}
```

Figure 1. A loop with one loop test and single increment

memory accesses and to obtain knowledge about the contents of accessed registers and memory cells. As the value analysis produces integer intervals as approximations for addresses and memory contents, both loop analyses use intervals for their calculations, too. Beside this, the loop analyses query the value analysis for infeasible control-flow edges, i.e. edges that are not taken in any run of the program. This information is used in both analyses to exclude unreachable loops from loop bound detection. For more details about the value analysis please refer to [9]. The value analysis information allows separate analysis of the loops for each calling context and monitoring the loop counter even if it is a global variable, a function parameter or modified over a pointer, which is important as shown in [8].

The analyses take into account that programs often contain nested loops for which the iteration bounds of the inner loops depend on the iteration bounds of outer loops. Therefore both analyses sort the loops by their nesting depth and analyze them from the outside to the inside. After handling one nesting depth, value analysis is restarted with the new derived loop bounds as input to get more precise information while looking for the bounds of the inner loops.

As value analysis gets more precise if it also knows the lower bound of a loop, both analyses output not only the safe upper bounds needed to calculate any WCET, but intervals that are guaranteed to contain all possibilities for the number of loop iterations.

3. Running Examples

To illustrate the working of the two loop analyses, two simple loops found in programs for the *PowerPC* architecture are chosen as examples. Figures 1 and 2 show the corresponding loop routines.

Both loops use machine register 31 as their loop counter. We assume for the upcoming calculations and analyses that this register contains the value zero before the first loop iteration.

The loop in Figure 1 is a simple loop incrementing its loop counter in each iteration by exactly one. The loop is first entered with counter value 0, then with value 1, etc. until it reaches 16. When it is entered with counter value 16, the test `r31 < 16` fails for the first time so that there are no further loop iterations. Therefore, there are exactly 17 loop iterations. The loop analysis should thus return the interval $[17, 17]$ (the most precise answer) or any larger interval containing 17 (correct, but imprecise).

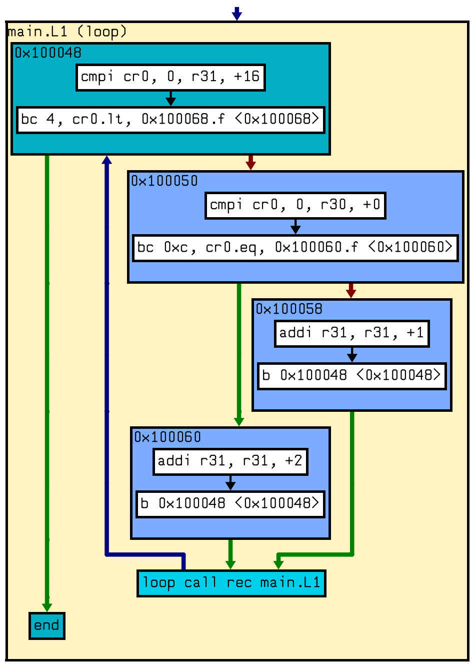
The loop in Figure 2 is similar, but a counter increment of one or two is possible, as the control flow forks into two branches inside the loop routine. The safe upper bound is still 17 as in the first example, but the lower bound is now only 9. The result of the loop analysis should thus be $[9, 17]$ or any larger interval.

4. The Pattern-Based Approach

The current loop analysis in **aiT** uses patterns to detect the loop bounds for common loop variants. These patterns are handcrafted for the supported compilers and their different optimization levels. Some intraprocedural analyses are used to handle the matching, like intraprocedural slicing and dominator/postdominator analysis.

A typical loop pattern to detect loops generated by C compilers from `for`-loops consists of the following conditions:

- The loop is only left by one conditional branch;
- the same compare of a register with a constant sets the condition for this branch in each iteration;
- the register that is compared is incremented by a constant value at the same instruction in each iteration;
- the start value of the register is known by the value analysis.



```
while (r31 < 16) // 0x100048
{
  if (r30 == 0) // 0x100050
  {
    r31 = r31 + 1; // 0x100058
  }
  else
  {
    r31 = r31 + 2; // 0x100060
  }
}
```

Figure 2. A loop with one loop test and two different increments

To match even such a simple pattern, multiple internal subanalyses must be performed. For this example pattern, the following steps would be needed:

- Check for a conditional branch instruction that dominates and postdominates the recursive call of the loop routine;
- slice backwards from the branch inside the loop routine to find the compare instruction modifying the condition flag evaluated by the branch instruction;
- test whether it is a compare of a register with a constant;
- slice backwards from the compare instruction to find all instructions modifying the registers/memory cells used in the compare instruction;
- test whether only one instruction is found in the last step and whether it is a constant addition/subtraction;
- test whether this one instruction dominates and postdominates the compare instruction;
- query the value analysis for the start value of the used register;
- calculate the bounds by using the now known start/end value and increment.

If we apply this pattern to our example loop of Figure 1, we get a match, as this loop is left only by a conditional branch after the compare of the loop counter with some constant and the loop counter is incremented in each round by one. The resulting bound would be $[17, 17]$, which is in this case the optimal solution.

The slightly more complex loop of Figure 2 is not matched by this pattern, as the loop counter is not incremented in each iteration by the same instruction, but by two different `addi` instructions in two different control-flow branches. Therefore no loop bound can be determined and thus no WCET is obtained.

Given how many steps are already needed for this simple pattern and that all this needs to be done by handwritten code, it is clear that bigger patterns to handle more complex loops, like the one shown above, are time consuming to implement correctly and to maintain. This illustrates the need for a new kind of loop analysis, which will be presented in the next section.

5. Improved Loop Analysis Based on Data-Flow Analysis

To enhance the loop bound detection for more complex loops and to avoid the dependencies on compiler versions and optimization levels, a new loop bound analysis based on data-flow analysis was designed. The following provides a brief introduction to this new method. More information can be found in [2].

A run of the new analysis consists of the following phases:

1. Classification of all loops;
2. Detection of possible loop counters;
3. Data-flow analysis to derive the invariants;
4. Analysis of the loop tests to calculate the loop bounds.

5.1. Loop classification

In the first phase, loops are classified using information obtained from value analysis. Loops that can never be reached are excluded from further analysis and get the safe bound $[0,0]$ as the corresponding loop routines are never called. For the remaining loops, the algorithm checks whether value analysis already knows after how many recursive calls their loop routine cannot be called again. If this number is known, it can be taken as a safe upper bound for the loop, even if the further stages fail to produce results.

5.2. Search for possible loop counters

For the loops that still need to be analyzed, a simple intraprocedural analysis is run to search all registers and memory cells accessed inside the loop routine. Then it is checked whether value analysis knows their start value, i.e. their value before the first call of the loop routine. The registers and memory cells with known start value are considered as potential loop counters. They are further examined by a data-flow analysis to derive loop invariants (see below). Loops without any detected loop counter must remain unbounded.

Our first example loop (Figure 1) only accesses register 31. For our second example (Figure 2), the intraprocedural analysis would find registers 30 and 31. Assuming that value analysis only knows the start value of register 31, this register would be the only potential loop counter in both loops.

5.3. Invariant analysis

This data-flow analysis is the core of the improved loop analysis. For each potential loop counter detected in the previous phase, it calculates for each program point of the loop routine a set of expressions, called invariants, that indicate how the counter is modified from the entry of the loop routine to this point in each iteration.

The analysis uses a special language for the expressions, *IVALA*. Variables in *IVALA* expressions describe registers or memory cells, including information about the register number or memory address and the data size in bytes. The loop counter in our examples would be expressed in *IVALA* as $(register, 31, 4)$, as it is register 31, which is 4 byte wide.

The language allows to express assignment between variables, assignment of a constant integer interval to a variable, and modification of a variable by adding a constant integer interval. This seems to be very restrictive, as other modifications like non-constant addition or any kind of multiplication are not supported, but the evaluation in the next section will show that it is sufficient to detect most loop bounds in a program, as the most common loops are counting loops. Besides, this restriction serves to keep the complexities of invariant analysis and of the subsequent bound calculation within reasonable bounds.

For the loop routine of our first example shown in Figure 1 the analysis would e.g. calculate the following expression set for the ingoing edge of the recursive call of the loop routine:

$$\{(register, 31, 4) = (register, 31, 4)^\circ + [1, 1]\}$$

where $(register, 31, 4)^\circ$ is a placeholder for the value of $(register, 31, 4)$ at the beginning of the loop iteration. The expression indicates that register 31 is incremented by exactly one in each iteration. For the example in Figure 2 the analysis would calculate:

$$\{(register, 31, 4) = (register, 31, 4)^\circ + [1, 2]\}$$

This provides the information that the register is incremented by one or two.

5.4. Evaluation of the loop tests and bound calculation

In this phase, for each loop all existing loop tests will be evaluated. A loop test is a basic block with a conditional branch leaving the loop routine. For each test a bound will be calculated. All these bounds are then combined to one bound for the whole loop. The following steps are needed to calculate the bound for a loop test:

- The branch type is determined;
- the compare instruction evaluating the condition used by the branch is searched;
- the variables used in the compare instruction are detected;
- the flow-analysis results are used to get expressions for the found variables;
- an equation system is built and solved to get the concrete loop bound.

A detailed description of this process can be found in [2].

For our first example (Figure 1), this process would look as follows:

- Inspection of the branch in basic block `0x100044` yields that the loop is left on greater-equal.
- A search for the corresponding compare instruction finds the first instruction in the block.
- As variable $(register, 31, 4)$ and the constant integer 16 are used, the exit expression is $(register, 31, 4) \geq 16$.
- The flow-analysis will yield that $(register, 31, 4)$ is incremented by one in each iteration.
- The solver will compute the concrete bound $[17, 17]$, which is the optimal solution.

The handling of the second example is analogous, except that the flow-analysis delivers an increment of $[1, 2]$ and therefore the solver would calculate the bound $[9, 17]$.

Both examples show comparisons with integer constants as loop test but comparisons of two variables are supported, too, as long as the value analysis is able to detect a constant interval for one of them.

6. Practical Evaluation

While the new analysis is more generic by design, we still need to demonstrate that it is applicable to real-world programs. Therefore an extensive evaluation with both code from a compiler benchmarks suite and with real software from the embedded-system world was performed in [2].

test	optimal	old analysis	new analysis
do_char_001	$[1, \infty]$	$[1, \infty]$	$[1, \infty]$
do_char_008	$[16]$	$[16]$	$[16]$
do_char_009	$[16]$	$[16]$	$[1, \infty]$
do_char_010	$[1, 16]$	$[1, 16]$	$[1, \infty]$
for_char_001	$[17]$	$[1, \infty]$	$\mathbf{[17]}$
for_char_017	$[17]$	$[17]$	$[17]$
for_char_049	$[1]$	$[1, 17]$	$[1, 17]$
for_char_058	$[17]$	$[1, \infty]$	$\mathbf{[17]}$
for_char_061	$[9]$	$[1, \infty]$	$\mathbf{[9]}$
for_char_062	$[17]$	$[1, \infty]$	$\mathbf{[17]}$
for_int_001	$[17]$	$[1, \infty]$	$\mathbf{[17]}$
for_int_017	$[17]$	$[17]$	$[17]$
for_int_049	$[1]$	$[1, 17]$	$[1, 17]$
for_int_058	$[17]$	$[1, \infty]$	$\mathbf{[17]}$
for_int_061	$[9]$	$[1, \infty]$	$\mathbf{[9]}$
for_int_062	$[17]$	$[1, \infty]$	$\mathbf{[17]}$

Table 1. Single loop synthetic tests, *DiabData*

The results show that the new analysis method works for most loops equally well or better than the pattern-based method. Only in some corner cases, the old analysis takes the lead, as it has special patterns for them.

The runtime costs of both analyses are comparable: the new analysis is slower than the pattern-based approach only by a constant factor of at most three for some tests. Table 2 shows measured runtimes of both analyses for four different tasks out of industrial real-time software for a *PowerPC MPC755*. The runtimes were measured on a 3.2 GHz Pentium 4 with 2 GB RAM running *Linux*.

test	old analysis	new analysis
mpc755_1	43.54	63.75
mpc755_2	3.82	9.25
mpc755_3	0.53	0.77
mpc755_4	0.47	0.69

Table 2. Runtimes of analyses in seconds

To show that the new analysis is compiler-independent, Tables 1 and 3 present the results of both analyses for code generated by the *DiabData* ([11]) and *GNU* C compiler ([4]), respectively. While both analyses work reasonably well for the DiabData compiler, only the data-flow based analysis works for the GNU C compiler without adjustments. To obtain comparable results, the pattern-based analysis would require additional effort to develop loop patterns adapted to the code generated by the GNU C compiler.

test	optimal	old analysis	new analysis
do_char_001	$[1, \infty]$	$[1, \infty]$	$[1, \infty]$
do_char_008	$[16]$	$[1, \infty]$	$\mathbf{[16]}$
do_char_009	$[16]$	$[1, \infty]$	$\mathbf{[16]}$
do_char_010	$[1, 16]$	$[1, \infty]$	$\mathbf{[1, 16]}$
for_char_001	$[17]$	$[1, \infty]$	$\mathbf{[17]}$
for_char_017	$[17]$	$[1, \infty]$	$\mathbf{[17]}$
for_char_049	$[1]$	$[1, \infty]$	$\mathbf{[1, 17]}$
for_char_058	$[17]$	$[1, \infty]$	$\mathbf{[17]}$
for_char_061	$[9]$	$[1, \infty]$	$\mathbf{[9]}$
for_char_062	$[17]$	$[1, \infty]$	$\mathbf{[17]}$
for_int_001	$[17]$	$[1, \infty]$	$\mathbf{[17]}$
for_int_017	$[17]$	$[1, \infty]$	$\mathbf{[17]}$
for_int_049	$[1]$	$[1, \infty]$	$\mathbf{[1, 17]}$
for_int_058	$[17]$	$[1, \infty]$	$\mathbf{[17]}$
for_int_061	$[9]$	$[1, \infty]$	$\mathbf{[9]}$
for_int_062	$[17]$	$[1, \infty]$	$\mathbf{[17]}$

Table 3. Single loop synthetic tests, *GNU*

7. Summary and Outlook

As the evaluation has shown, both analyses have some benefits in their own areas. While the pattern-based analysis can keep the lead for special cornercases where handcrafted patterns can play out their strength, the data-flow based analysis works best for typical loops occurring in standard programs. This flexibility of the new analysis is reached by it's expressions, which are powerful enough to handle loops with multiple exits, multiple/conditional changes of the loop counter and overflows of the used datatypes.

As **aiT** is aimed to provide the best loop bound detection possible, both analyses will be used in combination. First the fast pattern-based analysis is applied, and only for the loops it is not able to handle, the more generic new analysis is run. This avoids any slow down for the analysis of programs for which the old analysis already detected all bounds, and enables the calculation of the WCET for programs with more complex loops.

This combined strategy is already in use for the *PowerPC* and *M32* architectures, with plans to extend it to the *VAMP* architecture (described in [1]) in the near future.

References

[1] S. Beyer. *Putting it all together - Formal Verification of the VAMP*. PhD thesis, Saarland University, Saarbrücken, 2005.

[2] C. Cullmann. Statische Berechnung sicherer Schleifengrenzen auf Maschinencode. Diploma Thesis, Universität d. Saarlandes, 2006.

[3] C. Ferdinand, F. Martin, C. Cullmann, M. Schlickling, I. Stein, S. Thesing, and R. Heckmann. New Developments in WCET Analysis. In T. Reps, M. Sagiv, and J. Bauer, editors, *Program Analysis and Compilation, Theory and Practice: Essays dedicated to Reinhard Wilhelm*, volume 4444 of *LNCS*, pages 12–52. Springer Verlag, 2007.

[4] GNU Project. *GCC Version 3.3*, 2006.

[5] J. Gustafsson, B. Lisper, C. Sandberg, and N. Bermudo. A tool for automatic flow analysis of c-programs for wcet calculation. In B. Werner, editor, *In Eight IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 106 – 112, Guadalajara, Mexico, January 2003. IEEE.

[6] J. Gustafsson, B. Lisper, C. Sandberg, and L. Sjöberg. A prototype tool for flow analysis of c programs. In G. Bernat, editor, *WCET 2002 Workshop*, Vienna, June 2002.

[7] F. Martin, M. Alt, R. Wilhelm, and C. Ferdinand. Analysis of Loops. In *Proceedings of the International Conference on Compiler Construction (CC'98)*. Springer-Verlag, 1998.

[8] C. Sandberg. Inspection of industrial code for syntactical loop analysis. In *WCET 2004 Workshop*, Catania, July 2004.

[9] M. Sicks. Adreßbestimmung zur Vorhersage des Verhaltens von Daten-Caches. Diploma Thesis, Universität d. Saarlandes, 1997.

[10] H. Theiling. Extracting Safe and Precise Control Flow from Binaries. In *Proceedings of the 7th Conference on Real-Time Computing Systems and Applications*, Cheju-do, South Korea, December 2000.

[11] Windriver. *DiabData C Compiler Version 4.4*, 2006.