Inspection of industrial code for syntactical loop analysis

Christer Sandberg
Department of Computer Science and Engineering, Mälardalen University, Västerås, Sweden
christer.sandberg@mdh.se

Abstract

Flow analysis can be used in WCET analysis to, e.g., determine loop bounds and infeasible paths. Such information can be used by low level analysis and for actual WCET calculation. An efficient flow analysis method is syntactical analysis. This method identifies certain predefined syntactical constructs. It is not reasonable to believe that we will be able to use syntactical analysis to identify all conceivable constructs. Therefore we need to learn which to prioritize.

This paper suggests methods for inspection of industrial code to record properties of code that will give hints how to design a syntactical analysis. We describe the code properties to analyze, and present preliminary results for some industrial real-time code.

1 Introduction

We aim to develop methods for automatic estimation of WCET. *Automatic* in this context means that manual annotations of loops and infeasible paths should be avoided. The task for an automatic flow analyzer in this sense is to calculate "flow facts" such as loop bounds, infeasible paths etc. [4]. This can not in general be done without interaction with the user. If, e.g., a loop bound is dependent on input data these data has to be supplied by the user. The user may also need to supply which parts of the code that should be analyzed. If, e.g., infinite loops are part of the entire program the responsibility is on the user to exclude these.

There are several methods that can be used for flow analysis, [9, 5], one being abstract interpretation and another syntactical analysis.

Syntactical analysis will identify syntactical constructs where we can calculate the number of iterations by looking at the loop syntax (in particular there is no need to calculate the result of each iteration). The initial values of loop variables, loop increment and loop termination conditions can be translated to recurrence equations, which can be solved. The advantage of this method is that it is efficient, with a complexity that is linear to the size of the program. In best case we succeed in bounding all loops in the analyzed code. In case it finds loops which does not match any known pattern, the method can still be used as a "filter" to reduce the amount of work needed by other methods.

If we for example use syntactical analysis in combination with abstract interpretation, e.g., as described by Gustafsson, [7], the remaining loops may be possible to bound by the abstract interpretation. Specially, there is a possibility to syntactically find final values of all variables assigned in the loop body. In such cases the entire loop can be collapsed and replaced by a piece of sequential code, making life easier for other analysis methods. Even in cases when only a few occasional loops can be bounded by the syntactical analysis, this can still benefit in a combination with other methods since the tightest of the loop bounds found by the different methods can be chosen.

The syntactical method can be thought of as a data base of predefined patterns (syntactical constructs) and we can check for each loop if it matches any of the patterns in the database. Since there is a huge number of possibilities to write loops we can not expect the syntactical analysis to be able to recognize all of them. However, we want to fill this database with patterns that matches as many loops as possible when analyzing real-time code. Therefore we need to get an idea of what loop constructs that are common in real-time systems and that are fairly simple to identify without too much of computational efforts.

In cases when loop patterns are not suitable to be analyzed by the syntactical method some alternative methods need to be used. The results from this inspection might also be useful to find the requirements on such methods.

2 Related work

In comparison between specInt95 and some code for embedded systems, performed by Engblom [3], he concluded that using code from desktop applications as base for testing tools for embedded systems may be dangerous due to significant differences in programming style for these categories of programs.

An investigation of a large set of industrial code, carried out by Engblom [2] shows, e.g.,

- Recursive functions may be expected but are not common.
- Use of function pointers can be expected, but is in most programs rare or even absent.
- Deeply nested loops at a global level are quite common.
- Unstructured loops may occur.
- Multiple entries to the program may occur.
- Functions from other modules may be invoked.
- Multiple loop exits are quite common (almost 1 of 3 of those in the investigated programs).
- High decision nest complexity is not common.

In [11] we see that the use of a WCET tool in practice may lead to unexpected problems. Some of the problems encountered, related to loops, were:

- Sentinels in arrays are sometimes used as termination conditions in loops.
- Loop termination conditions may depend on input.
- Loop counters may be calculated using complex arithmetics.
- Complex arithmetics is sometimes used in loop termination conditions.

Another related investigation, [1], shows that for a certain operating system the program constructs were quite simple. No nested loops, unstructured code or recursion were found. Some function pointers were used.

Our conclusion is that a careful investigation of industrial code for real-time systems is needed to prepare a WCET analysis tool to handle code that exists in reality. For a syntactical analysis we specially need to get more knowledge about loop details.

3 The inspection

We will use an existing flow analyzing tool for automatic inspection, *SWEET* (SWEdish Execution Time tool, [8]). Most of the methods that needs to be implemented for this investigation are quite basic, and can probably be re-used when implementing the actual syntactical analysis, at least for the simple cases. This tool takes intermediate code, produced by a C compiler, as input.

Code needs to be inspected in a variety of views to get useful measurements. Some of the properties measured by Engblom can benefit the design of syntactical analysis. However we need to get a more detailed view, specifically about loops. The following inspection items will be explained in more detail in the subsequent sections: the overall program structure, the loop nesting and details about individual loops. Some of these items have been covered in previous work. They, however, need to be done also for the code used for this more detailed investigation, intended to aim the design of syntactical analysis.

3.1 Program structure

The program structure can be analyzed by examining the call graph. The depth of a call graph can give hints about how much calculation power that is needed for analyzing the code. Extending the call graph with information about the presence of loops will give more information. Comparing a call graph in DAG form with one in tree form will show to what extent functions are reused. Calculating loop bounds in functions called from different sites in a context sensitive manner will cost more in calculation efforts, but may give better (tighter) loop bounds. Recursive functions should be recorded separately. These may be hard to bound syntactically, but simple cases may be possible to handle in case an interprocedural analysis is performed.

3.2 Loop nesting

The nesting level of loops can be counted both locally (per function) and globally (per task or program). It is important to bound deeply nested loops, since they will have the highest influence on the final WCET. Also other analyzing methods (e.g., abstract interpretation) may suffer from high computational load when analyzing these.

3.3 Loop conditions

Reducible loops have a single entry point (we only consider reducible loops since we assume that irreducible loops are already replaced by their multiple reducible loops counterpart [12]).

There may be an arbitrary number of exits from loops. Those with no exits (infinite loops) may occur in real-time systems. We cannot find the loop bound of such a loop syntactically. However, although we can't analyze them we need to count them to conclude how big portions of a program that needs to be excluded from the analysis.

Loops that contain more than one exit branch have been shown to be analyzable syntactically [10], but are in general harder to calculate the bounds of, both in terms of computational efforts and implementation issues. Thus the number of loop exits is of interest as well as the number of targets of these exit branches.

For each termination condition there will be one or more variables involved (otherwise the loop is either equivalent to an infinite loop or with a non-looping construct). Each such variable needs to be carefully investigated. The following properties of the variable may be of interest:

- The initial value.
- The variable update in the loop.

Initial values

We are mostly interested in the initial value at the loop termination condition in which it is used (do loops may be handled a bit different, since the initial execution of the loop body might affect the initial value of some variables). The initial value can either be deduced from a constant, or depend on a variable that is updated in an outer loop or depend on an input value to the code.

Initialization from constants. If the initial value only depends on constants this will simplify the analysis, and makes it more likely that we can find a loop bound. The constant value can be found as an assignment from an expression containing only operands that are constants or that are other variables that recursively depend on constants. It is of interest to record the locations of the constants in terms of function nesting level. If all the constants are not present in the same function as were they are used, a more advanced syntactical analysis is needed (e.g., a global analysis).

Initialization from variables updated in an outer loop. There are certain kinds of nested loops where the loop bound of the inner loop can be calculated even if the number of iterations depend on an outer loop induction variable [8]. Therefore these class of initializations are of a certain interest. The following properties needs to be recorded:

- The loop nesting levels between the use and initialization. For example "triangular loops" can be recognized by a syntactical analysis, and it may be possible to find loop bounds in case there is a loop nesting level of one between the two loops.
- The properties of the source to the initialization are of interest. Gerlek et.al., [6], makes a classification of induction variables that may be useful as basis. The problem of finding the bound for a loop can be expected to vary based on these.
- Is the assignment of the initial value done from an expression containing more than one induction variables in outer loops? If so, it will be a harder case to handle.

Initialization from input values. The initial value can in some cases be deduced from some input value. Input to a real-time system may in general occur in various

ways. One would be that the code just use some global variable that appears uninitialized to the analyzer (e.g., the code reads from a labelled input port).

Different analyzable units (e.g., tasks) may need to communicate to each other, and this data will appear as inputs. The actual input data can in such case be deduced to some global entity, e.g., global variables. These variables may occur as initialized to the analyzer.

In case of an analysis local to functions, also the function arguments are input.

There is a special problem in identifying input values. The analyzer need to distinguish between those global variables that are input to the analyzed code and those that are rather constants or induction variables in the context of the use in a loop condition. This can normally not be done by looking at a part of the system code in isolation. In general it can be hard to perform an analysis on a complete system because of the interaction of an operating system.

In our inspection all definitions that depends on global variables will be recorded as inputs. When doing the syntactical analysis the user may supply information about which are input variables and the value of these. Optionally a certain pass just to identify intertask communication might be developed. Finding loop bounds if the initial values depend on input is much the same problem as finding it for constant initial values.

Updating of variables in the loop body

One or more of the variables in a loop termination condition must change their value in the loop, or the loop will never terminate using that condition. A simple form is a loop counter, i.e. a variable which value will contain the current iteration number while executing the loop. Also other forms of updates are of interest. If the value of the variable forms a series which we can identify, there is a good chance that we can calculate the loop bound. The following properties will be recorded:

- Is the update self-referencing, i.e. is the right hand side an expression that includes the target of the assignment (directly or indirectly) within the loop.
- The operator(s) applied in update(s). If only linear updates are involved a calculation of the loop bounds can be expected to be less complicated.
- The other operand(s). This may be constant (directly or indirectly) or an input dependent variable. In both cases it will probably be easier to find loop bounds than for induction variable dependent initialization values.
- Do the other operand alters its value in the loop? In case it does, it is probably harder to find a resulting loop bound (e.g., if an increment is altered in a conditional statement).
- Is the update statement conditional? If so, it is in general not possible to find the wanted series.

3.4 Branch conditions

Branch conditions are also of interest to the syntactical analysis. Infeasible paths can be found if the reaching definition for the involved variables are calculated in a context sensitive manner. The values of variables involved in conditions therefore needs to be recorded in the same manner as initial values of loop conditions.

3.5 Arrays

In case array elements are used in place for simple variables in loop termination conditions this imposes difficulties to the syntactical analysis. Such language elements can take many different syntactical forms. Below are listed some that will be recorded in the first round.

- *Initial value.* A variable in a loop termination condition is an array element (or its value depends on an array element), and this variable is loop invariant. For certain sub-cases we might be able to find loop bounds.
- *Update.* A variable in a loop termination condition is updated using an expression containing an array element (or a variable which value depends on an array element). The updated variable is an induction variable. The index variable might be an induction variable in an outer loop.
- *The "sentinel problem".* There is some loop termination condition that compares an array element with some other value. The array index is an induction variable in the current loop. This category can be tricky to handle. But since we know that strings as well as sometimes pointer arrays are often traversed this way, it is important to know the number of occurrences of this kind. Maybe we can calculate the loop bounds for some sub-cases.
- *Other uses of array values.*

3.6 Pointers

The use of pointers in loop termination conditions imposes problems for the syntactical analysis. Pointers that points to a distinct variable in a certain context might give us some hope. We should distinguish these uses of pointers from other.

4 Preliminary results

Code from three insdustrial systems (in total more than 80k lines of source code) has been inspected concerning the properties shown in the tables below. Inspection has been done in a context sensitive manner, meaning that each call site of a function has been counted rather than the function definition. The reason was that we wanted to be able to detect interprocedural dependencies.

The maximum call tree depth per root function (e.g., task) was 7. There were 392 loops with a single exit while there were 166 with 2, 58 with 3, and 12 with more that 3 exits. We found that 592 loops had a single target of the loop exit, which might still give them a hope to be syntactically bound, while 36 had multiple exits. No recursive function was found. Three infinite loops and three uses of function pointers were found in one of the systems.

In table 1 we can see that the global nesting depth is quite big. The difference between the local and global nesting means that functions containing loops are often called within loops. The rightmost column is a count of the nodes in a scope tree (a call tree where each function call and each loop instance has a node, as discussed in 3.1) where only the loop scopes are counted.

Table 2 shows the re-use of functions. Note that only functions containing loops has been considered.

The used method for performing the inspection in a context sensitive manner resulted in some of the root-functions (tasks) growing quite large. As a result of this,

Table 1: *The number of loops with a certain nesting level.*

Level	Global count	Local count	Scope count
0	68927	574	0
1	51262	53	384
2	8985	1	1771
3	250		9036
4	106		14919
5	32		28467
6			36138
7			26834
8			10893
9			1056
10			64

Table 2: *Re-use of functions.*

Nr of calls per function	Nr of functions
1	148
2	44
3	26
4	15
5-9	27
10-14	15
15-19	9
20-30	8
30-40	10
40-50	5
>50	74

the analysis could not be completed in some cases. The analysis of loop variables could only be carried out for 290 loops. Of these, 138 were found to be dependent on dereferenced pointers and was not further investigated.

In the remaining loops there were 175 variables involved in termination conditions, of these 154 were updated in the loop (table 3), and 3 were updated in different conditional branches of the loop. The operations of the update has not been investigated in detail, but the operands are based on constants only. Note that because of the context sensitive analysis 'loop' and 'variable' should in this paragraph be read as and *instance* of a loop and *instances* of a variable respectively.

In most cases the initial value could be found as a constant in the same function, however in 14 cases in the calling function and in 5 cases the value was obtained from a called function. The value were never found in a cross call-tree node.

The initial values were in 5 cases defined based on variables updated in the enclosing loop (table 4).

Table 3: *Definition from program constant.*

Distance to defining function	Update operand	Initial value
-1	0	5
0	154	151
1	0	14

Table 4: *Definition from induction variable.*

Distance to loop	Update operand	Initial value
0	0	5

5 Conclusions and Future work

The big nesting level indicates that there might be lots of dependencies between loops, as well as a needing to search for loop variable definition in outer functions. However, the subset of loop variables that we were able to investigate does not confirm this. Surprisingly only a few loops actually depend on outer loops and should be simple to bound. Instead, there are problem with dereferenced pointers in lot of cases (48% of the analysed loops). A conclusion is that to be able to bound all loops a flow analysis need to handle pointers in a powerful way. A manual inspection was made on one of the systems to get some hints about the loops with pointers (this was a medium sized system, containing 45 of the 138 dereferenced pointers). It appeared that all of them were of interprocedural type (function parameters or global variables).

To make the results more valuable further inspections need to be done. It is necessary to handle functions in a context sensitive manner without linking them all together. It is also necessary to make a more detailed investigation of the dereferenced pointers.

To increase the usefulness of the inspection more code will need to be collected.

6 Acknowledgments

We would like to thank ESAB and CC Systems for making their code available for our work. This work is performed within the Advanced Software Technology (ASTEC) competence center, supported by the Swedish National Board for Industrial and Technical Development (NUTEK).

References

[CP99] A. Colin and I. Puaut. Worst-Case Execution Time Analysis of the RTEMS Real-Time Operating System. Technical Report No 1277, IRISA, November 1999.

[EE00] J. Engblom and A. Ermedahl. Modeling complex flows for worst-case execution time analysis. In *Proc. 21st IEEE Real-Time Systems Symposium (RTSS'00)*, November 2000.

[Eng99a] J. Engblom. Static Properties of Embedded Real-Time Programs, and Their Implications for Worst-Case Execution Time Analysis. In *Proc. 5th IEEE Real-Time Technology and Applications Symposium (RTAS'99)*. IEEE Computer Society Press, June 1999.

[Eng99b] J. Engblom. Why SpecInt95 Should Not Be Used to Benchmark Embedded Systems Tools. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'99)*. IEEE Computer Society Press, May 1999.

[Erm03] A. Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. PhD thesis, Faculty of Science and Technology, Uppsala University, June 2003.

[GBS03] J. Gustafsson, N. Bermudo, and L. Sjöberg. Flow Analysis for WCET calculation. Technical Report 0547, ASTEC Competence Center, Uppsala University, URL: `http://www.mrtc.mdh.se/publications/0547.ps`, March 2003.

[GLB+02] J. Gustafsson, B. Lisper, N. Bernmudo, C. Sandberg, and L. Sjöberg. A Prototype Tool for Flow Analysis of C Programs. In *WCET 2002 Workshop Wienna*, pages 9–12, aug 2002.

[GSW95] M. P. Gerlek, E. Stoltz, and M. Wolfe. Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA form. *ACM Transactions on Programming Languages and Systems*, 17(1):85–122, January 1995.

[Gus00] J. Gustafsson. *Analyzing Execution-Time of Object-Oriented Programs Using Abstract Interpretation*. PhD thesis, Department of Computer Systems, Information Technology, Uppsala University, May 2000.

[HSRW98] C. Healy, M. Sjödin, V. Rustagi, and D. Whalley. Bounding Loop Iterations for Timing Analysis. In *Proc. 4th IEEE Real-Time Technology and Applications Symposium (RTAS'98)*, June 1998.

[RSE+03] M. Rodriguez, N. Silva, J. Estives, L. Henriques, D. Costa, N. Holsti, and K. Hjortnaes. Challenges in Calculating the WCET of a Complex On-board Satellite Application. In *WCET 2003 Workshop Porto*, 2003.

[San03] C. Sandberg. Elimination of Unstructured Loops in Flow Analysis. In *WCET 2003 Workshop Porto*, 2003.