Eliminating Annotations by Automatic Flow Analysis of Real-Time Programs

Jan Gustafsson*
Department of Computer Engineering, Mälardalen University
Box 883, S-721 23 Västerås, Sweden.
jan.gustafsson@mdh.se

Abstract

There is an increasing demand for methods that calculate the worst case execution time (WCET) of real-time programs. The calculations are typically based on path information for the program, such as the maximum number of iterations in loops and identification of infeasible paths. Most often, this information is given as manual annotations by the programmer.

Our method calculates path information automatically for real-time programs, thereby relieving the programmer from tedious and error-prone work. The method, based on abstract interpretation, generates a safe approximation of the path information. A trade-off between quality and calculation cost is made, since finding the exact information is a complex, often intractable problem for non-trivial programs.

We describe the method by a simple, worked example. We show that our prototype tool is capable of analyzing a number of program examples from the WCET literature, without using any extra information or consideration of special cases, needed in other approaches.

*The work is performed within the competence centre Advanced Software Technology (ASTEC, http://www.docs.uu.se/astec), supported by the Swedish National Board for Industrial and Technical Development (NUTEK, http://www.docs.nutek.se) and Mälardalen Real-Time Research Centre (MRTC, http://www.mrtc.mdh.se), supported by the Knowledge Foundation (KKS, http://www.kks.se).

1. Introduction

To guarantee timeliness, it is necessary to know the *execution times* for the programs in a real-time system. As execution time varies, the *worst case execution time (WCET)*, i.e., the longest execution time for a program for all possible input data on a given hardware, is used as a safe upper limit for the execution time. Since it is in practice often impossible to measure the execution time for all inputs, a WCET estimate, calculated by static analysis, is used as an approximation. The WCET estimate must be safe, i.e., must never underestimate the real WCET. To avoid waste of processing resources, it should be as tight as possible.

Manual annotations. In most existing WCET calculation methods, *manual annotations*, given by the programmer, are required. We will concentrate on the following two main types of annotations:

a) The *maximum number of iterations* in loops. This information is *mandatory*, since it is necessary for calculation of the WCET.

b) Information about *infeasible paths*, i.e., paths that never can be executed for any input data. Excluding these paths from the calculation can make the WCET tighter, if the infeasible path is a part of the estimated WCET path.

Example. The code fragment below contains the maximum number of iterations for a `for`-loop, using the syntax for manual annotations in [15].

```
for (a = 1; a < 4; a++)  MAX_COUNT(3) ...
```

The manual annotation (in capital letters) is an extension of the C syntax. The value 3 is inserted by the programmer, but any expression that can be statically calculated by the compiler can be used. □

To give manual annotations is not always a simple task. It is easy to find small examples where, e.g., the maximum number of iterations in a loop is hard to calculate, and where dependencies are complicated. Therefore, we consider it as a drawback of the existing methods that manual annotations are required. We see two main problems:

1. They give *extra workload* on the programmer.
2. The annotations can be *wrong*, i.e., they may invalidate the WCET calculation results.

Instead, we would like the manual annotations to be replaced by *automatically* calculated values.

Context-sensitive WCET calculations. Most methods calculate just one WCET for a program or a function, disregarding the fact that the execution time can depend heavily on the input parameters, which may give large overestimations. Instead, we would like the calculations to be *context-sensitive*. Inside a program, the input limits to a specific instance of a function may be determined by the context of a call. To be able to calculate as tight WCET as possible, these limits should be calculated automatically by the method.

Of course, limits for external input values, like physical inputs from the environment (ranges for sensor values, etc.), have to be given as input to the analysis, to make results as tight as possible.

2. Contributions

We describe an *automatic flow analysis* method which, by static analysis, calculates path information for real-time programs. This information can be used to eliminate the need for manual annotations to be given by the programmer. The method uses *semantic analysis* based on *abstract interpretation* to calculate the path information. The method was introduced in [7, 9, 10], and is presented in detail in the author's Ph.D. thesis [8].

The main advantages of the method are:

1. It derives safe estimations of the *minimum and maximum number of iterations* in loops.
2. It also identifies *infeasible paths* in programs.
3. It is *context-sensitive*, i.e., input parameters are taken into consideration and parameters used in calls are estimated.

The estimated values are safe but not necessarily exact, since a number of approximations are made during the analysis. This is because finding the exact run-time behaviour of general programs can be a very time-consuming task. A trade-off balance between analysis speed and quality is achieved by introducing certain approximations in the analysis.

The main contribution of this paper is to show that the method is capable of analyzing a number of programs from the WCET literature without using any manual annotations or consideration of special cases. We will use our new *prototype tool* to show the usefulness of the method.

Paper Outline. The paper is organised as follows: Section 3 presents related work. Section 4 presents the basics of abstract interpretation and gives an outline of the method, illustrated by a worked example. Section 5 gives the analysis of a number of examples from the literature. Finally, Section 6 presents our conclusions.

3. Related Work

A number of other research projects have studied the problems with manual annotations and have developed methods for automatic flow analysis ([8] contains a detailed analysis of related work).

The SPARK Ada WCET analysis [3, 4] is based on symbolic execution, and performs context-sensitive calculations. The method calculates the maximum number of iterations in loops and finds infeasible paths, but it suffers from the fact that some manual annotations (i.e., `pre`, `mode`, and `post` annotations, see [4]) are still needed. Our method does not rely on such manual annotations.

The CHaRy WCET method [1, 2] uses symbolic execution to calculate the maximum number of iterations in loops in C programs without manual annotations. The method also finds infeasible paths. However, the calculations are not context-sensitive. Also, the coarse way the values are stored during analysis gives less precision than our method.

The WCET research group at Florida State University, Tallahassee, USA, have have presented two different approaches, both based on dataflow analysis. The first method, described in [11], analyzes three special types of loops in C. When applicable, the method efficiently calculates tight approximations of the maximum number of iterations for the loops. It is possible to manually enter limits for variables associated with loops. The second paper [12], describes a technique, based on value-dependent constraints, to calculate the minimum and maximum number of iterations in loops. The results are found without use of annotations. If the tool fails to calculate the number of iterations, it prompts the user to give these values. The limitation of their approaches is that they are not general. In contrast, our approach, based on a general theory, is valid for any program construct.

The WCET analysis method from Chalmers University of Technology, Göteborg, Sweden [13] is based on an instruction-level simulator. It calculates the number of iterations in loops without any manual annotations. Since only the possible executions are simulated, infeasible paths are excluded automatically. One problem with this method is the lack of portability, since the flow analysis and the machine-code analysis are tightly integrated. Our method supports portability, since it allows for a separate low-level analysis.

4. Program Analysis using Abstract Interpretation

Our aim is to calculate as much run-time behavior information as possible without having to run the program on *all* input data, and while guaranteeing termination of the analysis. One such technique is *abstract interpretation* [5], in which the program behavior is calculated using value descriptions or *abstract values* instead of real values. The price to be paid is loss of information; the calculation will sometimes give only approximate information. The abstract interpretation theory guarantees, however, that the information is *safe*. Safe in this context means that the results of all possible real executions of the program are "included" in the calculated results, i.e., the calculated results form a superset of the real results.

Abstract interpretation has three important features:

1. It yields an *approximate* but *safe* description of the program behavior.
2. It is *automatic*, i.e., the program does not have to be annotated.
3. It works for *all* programs in the language.

It is important that the approximations for the concrete values are selected to *reduce the necessary calculations* in each step. But, loss of precision is often the consequence of less calculation.

To be able to calculate iteration counts and identify infeasible paths, we want to obtain information about the execution history of the program. This is done by extending the standard program semantics with additional loop and path information. This semantics is called an *instrumented semantics*. The purpose of the additional information is to keep track of the "execution path" during the analysis, by recording:

- the selected edge label at each selection;
- the loop label at the start of loops;
- the iteration counter at the start of a new iteration of a loop; and
- the termination of loops.

4.1. Outline of Our Method

The method consists of the following parts:

- *Program instrumentation*. The original program is modified to an instrumented, semantically equal, program, as described above.
- *Path analysis*. The analysis performs calculation of iteration count and identification of paths while doing abstract interpretation of the program. The abstraction is based on intervals as abstract values for the variables.
- *Merging*. During the analysis, intermediate results are merged at certain program points (at the end of loop bodies, and after termination of loops). The reason is to avoid an exponential growth of paths to be analyzed.

The result of the analysis is a list of infeasible paths, and a "loop history" for the program. In the loop history, the minimum and maximum number of iterations in the loops can be found. For nested loops, it is also possible to find the possible number of iterations for the inner loops for each instance of the outer loops. The method also calculates possible values (intervals) for the variables at all program points. This can be very useful, since it can be used to calculate tight results for subsequent code and to estimate value ranges of parameters.

Example. As an example of the type of path information that can be generated, consider the code fragment in Figure 1, where `l1` is a loop label, and `e1` to `e4` are edge labels. We will analyze the program using our automatic flow analysis method.

Figure 2 shows the structure of the analysis. In the figure, the analysis starts at the top with the initial value for x which is assumed to be in the inter-

```
while (x < 4) [l1] {
 if (x < 3) [e1] x = x * 2;
 else [e2] x = x + 1;
 if (x == 1) [e3] x = x + 2;
 else [e4] x = x + 1;
 }
```

Figure 1. Code example.

val [0..3]. As the while-loop cannot terminate at this point for any possible input value, the analysis continues in iteration #1. The infeasible loop termination path is indicated with a dashed line.

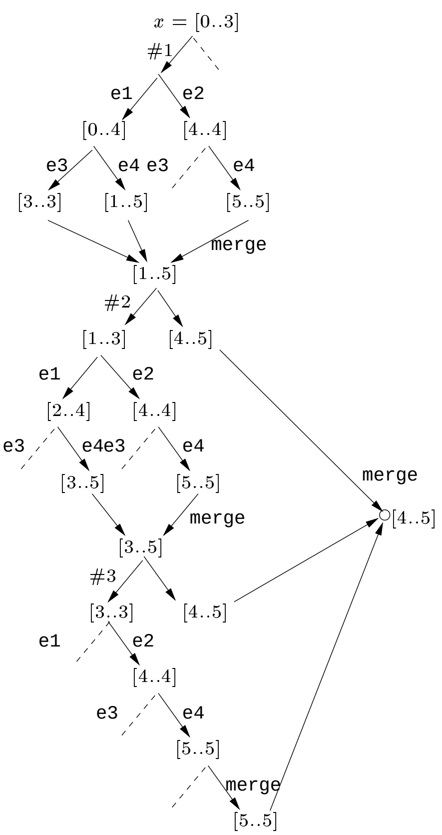

Figure 2. Analysis structure.

Since both edges are feasible in the first if-statement, the analysis continues in both. In `e1`, x will assume the value [0..2] initially and [0..4] at the end of the edge. In `e2`, the corresponding values will be [3..3] and [4..4].

Two possible x values will encounter the second if-statement. For the first, $x = [0..4]$, both `e3` and `e4` are feasible, leading to two possible end cases where x is [3..3] and [1..5], respectively. In the second case, `e3` is infeasible, since $x \neq 1$, i.e., we have found the infeasible path `e2` $\rightarrow$ `e3`, marked with a dashed line in the figure. Thus, only `e4` is analysed further, with an end value $x = [5..5]$.

The analysis of the first loop iteration is finished. Our method now *merges* the values to form the union of all variable values. The purpose of this is to reduce the complexity of the calculation. We can see that the merged value of x becomes [1..5].

The analysis then continues both in the next iteration #2, where $x = [1..3]$, and with the termination of the loop, where $x = [4..5]$.

The analysis of iterations #2 and #3 are performed similar to #1. We can see that four additional infeasible paths are found. The fact that the loop cannot continue with iteration #4 is marked with a dashed line.

When the analysis of the loop is finished, we have three possible final values for x. These are merged to form the final result (the small circle).

Our method gives the following information: the final value of `x` is in the range $4 \leq$ `x` ≤ 5, the loop may iterate one to three times, and the path `e2` $\rightarrow$ `e3` is infeasible. Furthermore, in iteration two the path `e1` $\rightarrow$ `e3` is infeasible, and the edge `e1` cannot be executed in the third iteration.

Executing the program for all possible inputs (possible for this small input set), we can see that the analysis result is safe but somewhat pessimistic, since the path `e1` $\rightarrow$ `e3` is infeasible in iteration #1. This is not discovered by our analysis.

5. Selected Examples from the WCET Literature

This section contains the results when analyzing a number of programs from the WCET literature. In the figures, the output from the tool has been edited to enhance readability.

We have analyzed the programs with our new prototype tool for RealTimeTalk/Smalltalk programs. The example programs, which are coded in a simplified C language, have been ported to RealTimeTalk and analyzed by the tool. Since none of the examples use any object-oriented features, the porting poses no difficulties. The prototype tool and more analysis examples are presented in [8].

Mok example. The program in Figure 3 is fetched from [14]. The program iterates 100 times, but enters the true edge only 10 times. The other 90 times it executes the empty false-edge. In order to find a safe and tight execution time, it is important both to find the maximum number of iterations, and the number of times the faster path is taken.

Mok uses manual annotations to specify this behaviour, while our tool finds it automatically.

```
i = 0; j = 0;
while (i < 100) [l1] {
  if (i < 10) [e1] j = j+1; else [e2];
  i = i+1;
  }
```

```
Infeasible paths:
(l1 1 e2.
(l1 1 2 e2.
 .
 .
 .
(l1 1 2 3 4 5 6 7 8 9 10 e2.
(l1 1 2 3 4 5 6 7 8 9 10 11 e1.
 .
 .
 .
(l1 1 2 3 4 5 6 7 8 9 10 11 ... 98 99 e1.
(l1 1 2 3 4 5 6 7 8 9 10 11 ... 98 99 100 e1.
Loop history:
(l1 1 2 3 4 5 6 7 8 9 ... 98 99 100 = max)
Final state:
 vars:
  i = [100..100]
  j = [10..10]
The time for the analysis was 6 seconds.
```

Figure 3. Mok example with tool output.

Puschner and Koza example. The program in Figure 4 is a simplified illustration of the *loop sequence* concept introduced in [15].

An upper limit of the sum of the individual loop limits in the program is 700, due to the use of the iteration counter `i`. However, value dependencies between `x`, `y`, and `z` force the loops to iterate only 100 times each, with a sum of 300. Puschner and Koza suggest a special type of manual annotation, `LOOP-SEQUENCE ITERATION_SUM(300)`, to reduce this possible source of overestimation. Our tool, however, finds the correct maximum number of iterations automatically.

Healy, Sjödin, Rustagi and Whalley example. The program in Figure 5 is fetched from [11]. The authors show that their method can find the ranges of iterations (in this case, [26..100]), for this loop without manual annotations. Our tool calculates the same result automatically.

```
x = 0; y = 0; z = 0;
i = 0;
while (x < 100 && i < 100) [l1]
  { x = x+1; y = x; i = i+1; }
i = 0;
while (y < 200 && i < 200) [l2]
  { y = y+1; z = x+y; i = i+1; }
i = 0;
while (z < 400 && i < 400) [l3]
  { z = z+1; i = i+1; }
```

```
Infeasible paths:
Loop history:
(l1 1 2 3 4 5 6 7 8 ... 97 98 99 100 = max)
(l2 1 2 3 4 5 6 7 8 ... 97 98 99 100 = max)
(l3 1 2 3 4 5 6 7 8 ... 97 98 99 100 = max)
Final state:
 vars:
  x = [100..100]
  y = [200..200]
  y = [400..400]
  i = [100..100]
The time for the analysis was 11 seconds.
```

Figure 4. Puschner and Koza example with tool output.

```
/* 0 ≤ somecond ≤ 1 */
for (i = 0, j = 1; i < 100; i++, j+=3) [l1]
  if (j > 75 && somecond || j > 300)
    [e1] break;
  else [e2];
```

```
Infeasible paths:
(l1 1 e1.
(l1 1 2 e1.
 .
 .
 .
(l1 1 2 3 4 5 ... 23 24 25 e1.
Loop history:
(l1 1 2 3 4 5 6 7 ... 25 26)
 .
 .
 .
(l1 1 2 3 4 5 6 7 8 ... 97 98 99)
(l1 1 2 3 4 5 6 7 8 ... 97 98 99 100 = max)
Final state:
 vars:
  i = [26..100]
  j = [79..301]
  somecond = [0..1]
The time for the analysis was 39 seconds.
```

Figure 5. Healy, Sjödin, Rustagi and Whalley example with tool output.

6. Conclusions

We have shown that our method is capable of eliminating the need for manual annotations, by demonstrating that our prototype tool calculates safe path information for a number of program examples from the WCET literature. We do this without any types of manual annotations or consideration of special cases, needed in other approaches.

For the Mok example, which is a "classical" example of manual loop and path annotations, we showed that their manual annotations can be calculated automatically using our method. We also show that introducing new types of manual annotations, like in the Puschner example, is not necessary, either. Finally, in the Healy example, we showed that our general method calculates the same results as a method, specialized to calculate the path information for specific cases.

An important benefit of our method is that abstract interpretation, correctly used, guarantees that the results are safe. Another benefit is that since the basic theory is general, our method can be adapted for any programming language.

The method is suitable to include as a part of a general WCET tool, thus automating and simplifying the work of the real-time programmer. To this end, we cooperate with IAR Systems in Uppsala, Sweden, to integrate WCET analysis in their embedded systems development environment [6].

References

[1] P. Altenbernd. On the false path problem in hard real-time programs. In *Proceedings of the Eighth Euromicro Workshop on Real-Time Systems, L'Aquila, Italy*, pages 102–107, June 1996.

[2] P. Altenbernd. *Timing Analysis, Scheduling, and Allocation of Periodic Hard Real-Time Tasks*. PhD thesis, Department of Mathematics and Computer Science, University of Paderborn, Germany, October 1996.

[3] R. Chapman. *Static Timing Analysis and Program Proof*. PhD thesis, Department of Computer Science, University of York, England, 1995.

[4] R. Chapman, A. Burns, and A. Wellings. Integrated program proof and worst-case timing analysis of SPARK Ada. In *Proceedings of ACM Sigplan Workshop on Language, Compiler and Tool Support for Real-Time Systems*, June 1994.

[5] P. Cousot and R. Cousot. Abstract interpretation: A unified model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[6] J. Engblom, A. Ermedahl, M. Sjödin, J. Gustafsson, and H. Hansson. Towards industry strength worst-case execution time analysis. Technical report, Department of Computer Systems, Uppsala University, Sweden, 1999. ASTEC Report 99/02.

[7] A. Ermedahl and J. Gustafsson. Deriving annotations for tight calculation of execution time. In *Proceedings of the Third International European Conference on Parallell Processing (EuroPar'97), Workshop 20, Real-Time Systems and Constraints, Passau, Germany*, Lecture Notes in Computer Science, pages 1298–1307. Springer, August 26–29 1997.

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

[9] J. Gustafsson and A. Ermedahl. Automatic derivation of path and loop annotations in object-oriented real-time programs. In *Proceedings of the Joint Workshop on Parallel and Distributed Distributed Real-Time Systems, between the Fifth International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS'97) and the Third Workshop on Object-Oriented Real-Time Systems (OORTS'97) at the 11th IEEE International Parallel Processing Symposium (IPPS'97), Geneva, Switzerland*, April 1997.

[10] J. Gustafsson and A. Ermedahl. Automatic derivation of path and loop annotations in object-oriented real-time programs. *Journal of Parallel and Distributed Computing Practices (PDCP)*, 1(2):61–74, June 1998.

[11] C. Healy, M. Sjödin, V. Rustagi, and D. Whalley. Bounding loop iterations for timing analysis. In *Proceedings of the IEEE Real-Time Applications Symposium (RTAS'98)*, June 1998.

[12] C. Healy and D. Whalley. Tighter timing prediction by automatic detection and exploitation of value-dependent constraints. In *Proceedings of the Fifth IEEE Real-Time Applications Symposium (RTAS'99)*, 1999.

[13] T. Lundqvist and P. Stenström. Integrating path and timing analysis using instruction-level simulation techniques. In *Proceedings of ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'98), Montreal, Canada*, Lecture Notes in Computer Science (LNCS) 1474. Springer-Verlag, June 1998.

[14] A. K. Mok, P. Amerasinghe, M. Chen, and K. Tantisirivat. Evaluating tight program bound of programs by annotations. In *Proceedings of the Sixth IEEE Workshop on Real Time Operating Systems and Software, Pittsburgh, USA*, pages 74–80, May 1989.

[15] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *The Journal of Real-Time Systems*, 1(2):159–176, September 1989.