A Unified Flow Information Language for WCET Analysis

Andreas Ermedahl†
IT-Dept. Uppsala University
Box 337, SE-751 05 Uppsala
Sweden
andreas.ermedahl@it.uu.se

Jakob Engblom†
IAR Systems AB
Box 23051, SE-750 23 Uppsala
Sweden
jakob.engblom@iar.se

Friedhelm Stappert*
C-LAB
Fürstenallee 11, 33102 Paderborn
Germany
friedhelm.stappert@c-lab.de

Abstract

In this paper we raise the question if it is possible to create a unified flow information language that all WCET research groups can agree upon, and that is independent of flow analysis and calculation methods.

We discuss desired characteristics of such a flow information language and describe the type of flows that it should be able to express. We present our previously published flow fact annotation language and discuss how it fulfils the desired language properties.

1. Introduction

A correct WCET calculation method must take into account the possible program flow, like loop iterations and function calls. For expressing program flows numerous annotation languages have been presented in the WCET literature. The expressiveness and the type of flows that can be handled by these languages mostly depend on the characteristics of flow analysis methods used, rather than being targeted for the potential WCET tool user.

To generate a WCET estimate, we consider a program to be processed through the phases of *program flow analysis*, *low level analysis* and *calculation*. Most WCET research groups make a similar division notationally, but sometimes integrate two or more of the phases into a single algorithm.

The program flow analysis phase determines possible program flows, and provides information about which functions get called, how many times loops iterate, if there are dependencies between `if`-statements, etc. The information can be obtained by *manual annotations* (integrated in the programming language [14] or provided separately [6, 9, 19]). The flow information can also be derived using *automatic flow analysis* methods [7, 10, 13, 22].

In the calculation phase a program WCET estimate is derived, combining the information derived in the program flow and low-level analysis phases. There are three main categories of calculation methods proposed in literature: *tree-based*, *path-based*, and *IPET* (Implicit Path Enumeration Technique).

In a *tree-based* approach the WCET is calculated in a bottom-up traversal of a tree generally corresponding to a syntactical parse tree of the program, using rules defined for each type of compound program statement (like a loop or an `if`-statement) to determine the execution time at each level of the tree [1, 2, 16, 20].

In a *path-based* approach the possible execution paths of a program or piece of a program are explored explicitly to find the longest path [10, 12, 22, 23]. The path-based approach is natural within a single loop iteration or function.

In *IPET*, program flow and low-level execution time are modeled using arithmetic constraints [6, 9, 15, 18, 21]. Each basic block and program flow edge in the program is given a time (t_{entity}) and a count variable (x_{entity}), and the goal is to maximize the sum $\sum_{i \in entities} x_i * t_i$, subject to constraints reflecting the structure of the program and possible flows.

2. Representing Program Flow

The program flow phase can be further divided into three different subphases:

1. **Flow analysis**: Obtaining flow information. By manual annotations or automatic flow analysis.
2. **Flow representation**: Representing the results of the flow analysis.

† This work is performed within the Advanced Software Technology (ASTEC, `http://www.docs.uu.se/astec`) competence center, supported by the Swedish National Board for Industrial and Technical Development (NUTEK, `http://www.nutek.se`).

* Friedhelm is a PhD student at C-LAB (`www.c-lab.de`), which is a cooperation of Paderborn University and Siemens.

3. **Calculation**: Using the control flow information (as represented in the flow representation) in the final WCET calculation.

Some WCET methods integrate two or more of the phases. We believe that the separation of the flow analysis from the calculation reduces the complexity of each stage. Also, by keeping the flow analysis phase separate from the flow representation, results from several different flow analysis methods and manual annotations can be integrated and used together in the calculation phase.

When designing a language for expressing flow information there are a number of choices to be made:

- *Expressiveness:* What type of flows should be possible to express? What type of language constructs should be used?
- *Code relation:* How is the information related to different entities in the program code?
- *Calculation conversion:* How should the information be used in the final calculation phase?

2.1. Expressiveness

We first note, that a natural way to give flow information is by constraining the number of times different program entities, e.g. loops, statement, nodes or edges, can be taken. This can either be precise bounds, e.g. that a loop is iterated exactly ten times, or upper or lower bounds, e.g. that node `A` can't be taken more than five times. It is also beneficial if we can relate the executions of different program entities, e.g. that node `A` and node `B` will always be executed together.

The language can consist of named special relations between entities (e.g. using constructs like Parks `samepath(A,B)` and `nopath(A,B)` [19]). An alternative is to use a more generic style based on math, like our flow fact language [6]. The benefit of a generic math-based language is that it can express flows that are hard to put in words and that there is no obvious limit to the types of flows that can be expressed. On the other hand, a special purpose language is easier to understand, but requires that new language constructs are invented in order to express new flows.

The language must reflect the flows found in real-world programs. Researchers have investigated embedded software [4], the RTEMS operating system [3] and common signal-processing algorithms [8]. The results are not in complete agreement on the properties and flows typical for embedded software, showing that more research and knowledge is needed here.

One observation is that flow information is mostly *local* in its nature, specifying something valid for a small part of a program or a particular invocation of a function. Thus, it is not always suitable to specify flow information once for each entity in the program. E.g. we would like to be able to specify that some node `A` can't be executed during the first five iterations of a loop or give a loop bound valid for just some particular executions of a loop. A language should allow for such local flow information to be expressed.

2.2. Code Relation

First we note that it is natural to express flow information in relation to the entities available in the program code. Flow information can be provided in relation to the source code, intermediate code in a compiler, or the object code. If provided on source code level, the information must be mapped to the object code to be used in the WCET calculation. In the presence of optimizing compilers, this problem is non-trivial [5, 17].

Automatic flow analysis is probably easier to perform at the source code or intermediate code, since variables and other entities of interest are harder to identify in optimized object code. Also, for the potential WCET-tool end-user manual annotations are typically easier to provide at the source-code level.

Another issue is if the flow information should be included as a part of the programming language or provided outside the program. The benefit of language inclusion is that it forces the programmer to write code in an analysable manner. However, this requires compiler support and makes it harder to try different scenarios.

Specifying the flow information outside the program source allows it to free itself from the static structure of the program. For example, by using a *call-graph* representation, we can differ between invocations of the same function when called from different places in the code. An example of the extended version is our *scope graph* represention [6].

A good language should provide *stability* in that program changes not related to annotated code should not force the annotations to change. For example, a problem with expressing flow information on the object code level is that the information might need to be regenerated every time the program code changes.

An important issue is the ability to handle *unstructured code*, e.g. due to uses of `goto` and jumps into loops. An optimizing compiler might produce unstructured object code from structured source code, and automatic code for state machines also tends to be unstructured. A general purpose flow information language must be general enough to express flows over such unstructured code.

2.3. Calculation Conversion

Regardless of the flow information language used the extracted flow information must be "compiled" or

```
if(i < 10) A;  // Stmt B and C
else B;        // can not be
if(i <= 7) C;  // taken together
else D;
```

(a) Infeasible path

```
for(i=0;i<10;i++)     // bound: 10
  for(j=i;j<10;j++)   // local bound: 10
    E;                // E executed at
                      // most 55 times
```

(b) Triangular loop

```
if( cond )
  x = true;  // stmt: F
for( ... )   // Execution of G
  if( x ) G; // is implied by F
```

(c) Deeply nested dependency

Figure 1. Example of Code with Different Type of Flows

"adapted" to the calculation method used. The adaptation must be *safe*: never exclude execution paths which are considered possible by the flow information, and *tight*: including as few extra execution paths compared to the provided flow information. Figure 1 gives example code showing that not all calculation methods can take advantage of all types of flow information.

The tree-based method [1, 2, 16, 20] is conceptually simple and computationally cheap, but has problems handling flow information, since the computations are local within a single program statement and thus cannot consider dependencies between statements. For example, the code and flow information in Figure 1(a) causes problems in a tree-based calculation method since the timing of the first if-statement will be calculated in isolation from the second if-statement.

The path-based approach is natural within a single loop iteration or other executions of one loop [11, 23]. The method has problems with flow information stretching over loop borders and/or flow information on the *total* number of times entities are taken. For example, the path-based method has problems handling the "triangular" loop dependency in Figure 1(b). If WCET calculation is performed locally, the WCET calculation for the inner loop will assume 10 iterations, and the WCET calculation for the outer loop will use 10 executions of the inner loop, leading to the body of the inner loop being counted 100 times, when it is actually never executed more than 55 times.

For IPET very complex flows can be expressed using constraints, but all flow information needs to be given on a global program level [6, 9, 15, 18, 21]. This contradicts the need to specify flow information in a local context. As shown in [6], local flows can be handled by unrolling the program and lifting the information to a global level. Since flow information is given as relations over count variables some type of flow implications are problematic to express. E.g. Figure 1(c) shows an example of code where we would like to express an implication dependency like: "if `F` is taken once then (and only then) `G` can be taken several times, but if `F` is not taken then `G` can not be taken either".

3. Our Flow Fact Language

This chapter describes our previously published flow fact annotation language [6] and discusses how it fulfils the desired language properties.

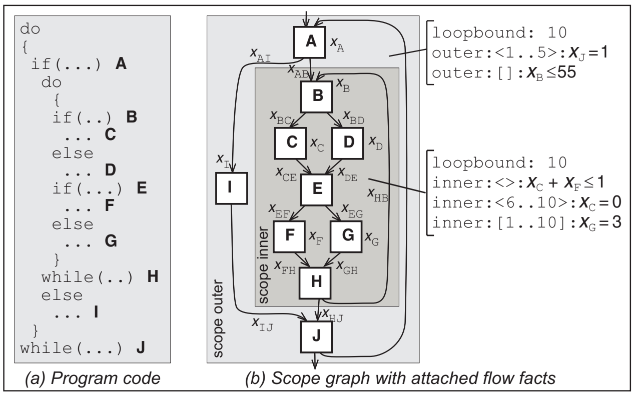

Figure 2. Scopes with Attached Flow Facts

The program representation used is the *scope graph*. It is a hierarchical representation of the dynamic structure of the program. Each *scope* corresponds to a certain repeating or differentiating execution context in the program, e.g. loops and function calls, and describes the execution of the object code of the program within that context. Figure 2(b) shows the scope graph generated for the code in Figure 2(a).

A scope consists of a number of *nodes* and *edges*. A node belongs to exactly one scope, and represents the execution of a certain basic block in the program in the environment given by the scope and its ancestors. For each scope, a header node must be given. If the scope iterates, each iteration must pass the header node, and a bound on the number of iterations has to be provided.

To express more complex program flow information than just basic loop bounds each scope can carry a set of *flow facts* [6]. The flow facts use constraints local to a scope to describe the flow. The constraints can be given for a range of iterations, or all iterations of a certain loop. They can also be local within a single iteration ("foreach facts") or represent a total over all iterations ("total facts").

The scope graph in Figure 2(b) has been decorated with some flow facts.

Flow fact `inner:<>:`$x_{\mathtt{C}} + x_{\mathtt{F}} \leq 1$ is a foreach fact and gives that the nodes `C` and `F` cannot be executed on the same iteration of the scope `inner` (an infeasible path), while the flow fact `inner:<6..10>:`$x_{\mathtt{C}} = 0$ gives that for each entry of `inner`, during iterations 6 to 10 of `inner`, node `C` can not be executed.

Flow fact `inner:[1..10]:`$x_{\mathtt{G}} = 3$ is a total fact that gives that, for each entry of `inner`, during the ten first iterations, node `G` must be taken exactly three times.

Compared to the criteria given above, we note that the flow facts language uses the math-based style and allows us to give local information. The information is given outside the code and uses an expanded version of the call graph (and thus the control flow graph). In its current version, it cannot handle all types of unstructured code due to the need for a header, and since it relates to the object code, it is very sensitive to program changes.

It has been used to perform both IPET- and path-based calculations [6, 23], but not all facts could be used in the path-based approach. It is interesting that the path-based calculation recognized certain types of facts as meaning "samepath" or "not samepath", and exploited these by rewriting the graph.

References

[1] R. Chapman. Program Timing Analysis. Dependable Computing System Centre, University of York, England, May 1994.

[2] A. Colin and I. Puaut. Worst Case Execution Time Analysis for a Processor with Branch Prediction. *Journal of Real-Time Systems*, May 2000.

[3] A. Colin and I. Puaut. Worst-Case Execution Time Analysis for the RTEMS Real-Time Operating System. In *Proc. 13^{th} Euromicro Conference of Real-Time Systems, (ECRTS'01)*, June 2001.

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

[5] J. Engblom, P. Altenbernd, and A. Ermedahl. Facilitating worst-case execution times analysis for optimized code. In *Proc. of the 10^{th} Euromicro Workshop of Real-Time Systems*, pages 146–153, June 1998.

[6] J. Engblom and A. Ermedahl. Modeling Complex Flows for Worst-Case Execution Time Analysis. In *Proc. 21^{th} IEEE Real-Time Systems Symposium (RTSS'00)*, November 2000.

[7] A. Ermedahl and J. Gustafsson. Deriving Annotations for Tight Calculation of Execution Time. In *Proc. Euro-Par'97 Parallel Processing, LNCS 1300*, pages 1298–1307. Springer Verlag, August 1997.

[8] R. Ernst and W. Ye. Embedded program timing analysis based on path clustering and architecture classification. In *International Conference on Computer-Aided Design (ICCAD '97)*, 1997.

[9] C. Ferdinand, F. Martin, and R. Wilhelm. Applying Compiler Techniques to Cache Behavior Prediction. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'97)*, 1997.

[10] C. Healy, R. Arnold, F. Müller, D. Whalley, and M. Harmon. Bounding Pipeline and Instruction Cache Performance. *IEEE Transactions on Computers*, 48(1), January 1999.

[11] C. Healy, M. Sjödin, V. Rustagi, D. Whalley, and R. van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Journal of Real-Time Systems*, May 2000.

[12] C. Healy and D. Whalley. Tighter Timing Predictions by Automatic Detection and Exploitation of Value-Dependent Constraints. In *Proc. 5^{th} IEEE Real-Time Technology and Applications Symposium (RTAS'99)*, pages 79–88, June 1999.

[13] N. Holsti, T. Långbacka, and S. Saarinen. Worst-Case Execution-Time Analysis for Digital Signal Processors. In *Proceedings of the EUSIPCO 2000 Conference (X European Signal Processing Conference)*, September 2000.

[14] Raimund Kirner and Peter Puschner. Transformation of Path Information for WCET Analysis during Compilation. In *Proc. 13^{th} Euromicro Conference of Real-Time Systems, (ECRTS'01)*, June 2001.

[15] Y-T. S. Li and S. Malik. Performance Analysis of Embedded Software Using Implicit Path Enumeration. In *Proc. of the 32:nd Design Automation Conference*, pages 456–461, 1995.

[16] S.-S. Lim, Y. H. Bae, C. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, and C. S. Ki. An Accurate Worst-Case Timing Analysis for RISC Processors. *IEEE Transactions on Software Engineering*, 21(7):593–604, July 1995.

[17] S-S. Lim, J. Kim, and S. L. Min. A Worst Case Timing Analysis Technique for Optimized Programs. In *Proc. of the fifth International Conference on Real-Time Computing Systems and Applications (RTCSA); Hiroshima, Japan*, pages 151–157, Oct 1998.

[18] G. Ottosson and M. Sjödin. Worst-Case Execution Time Analysis for Modern Hardware Architectures. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS'97)*, June 1997.

[19] C. Y. Park. Predicting Program Execution Times by Analyzing Static and Dynamic Program Paths. *Real-Time Systems*, 5(1):31–62, March 1993.

[20] P. Puschner and C. Koza. Calculating the Maximum Execution Time of Real-Time Programs. *The Journal of Real-Time Systems*, 1(1):159–176, 1989.

[21] P. Puschner and A. Schedl. Computing Maximum Task Execution Times with Linear Programming Techniques. Technical report, Technische Universität, Institut für Technische Informatik, Wien, April 1995.

[22] F. Stappert and P. Altenbernd. Complete Worst-Case Execution Time Analysis of Straight-line Hard Real-Time Programs. *Journal of Systems Architecture*, 46(4):339–355, 2000.

[23] F. Stappert, A. Ermedahl, and J. Engblom. Efficient Longest Executable Path Search for Programs with Complex Flows and Pipeline Effects. In *Proc. 4^{th} International Workshop on Compiler and Architecture Support for Embedded Systems, (CASES 2001)*, November 2001.