Elimination of Unstructured Loops in Flow Analysis

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

June 25, 2003

Abstract

A static WCET analysis should ideally be able to handle most kinds of program constructs. Unstructured loops are usually avoided nowadays in code written by humans, but they may be generated by tools, e.g. state machine code generators. They may also be introduced by an optimizing compiler. Therefore, the WCET analysis should be prepared to handle unstructured loops.

Most of todays flow analysis requires code without unstructured loops. This paper proposes that unstructured loops should be transformed into structured loops as a preprocessing step to the actual flow analysis. The advantages with elimination of unstructured loops are:

- The various steps of the analysis does not need to handle unstructured code separately. This includes e.g.
  - Syntactical analysis for finding loops that are possible to reduce
  - Finding merge points for flow information
  - A single upper bound of a loop
- Tree-based calculation can be used without modifications, since it assumes all loops to be structured

1 Introduction

The task of estimating WCET using the static approach can be divided into three phases:

- Flow analysis
- Low level analysis
- Calculation

We are developing a tool that implements these phases in a modular manner [GLB+02, Erm03]. The goal is to estimate the WCET for all program constructs (provided that the program terminates) based on an automatic analysis, i.e. without the need for manual annotations. The purpose of the flow analysis is to detect infeasible paths and to calculate upper loop bounds. There are different approaches with different properties to such an analysis, see [Erm03]. We have chosen to use abstract interpretation on an intermediate code level. One advantage of working with intermediate code is that a certain amount of optimizations may have been done before the flow analysis take place.

The abstract interpretation calculates safe values of variables with respect to loop bounds. Abstract interpretation may require a lot of computational power, the amount depending on the analyzed program as well as its input data. In order to simplify the calculations we can do a number of approximations, e.g. merging of states and simplified calculations of loop bounds. If approximations are made this may lead to overestimations (a less tight WCET) due to the requirement of being safe.

Two of the problems that a WCET tool that is useful in practice has to face are: the computational requirements of the analysis and the accuracy of the result. The occurrence of unstructured loops in the analysis may enlarge these problems.

1.1 Reducing Computational Requirements

In order to reduce computational requirements, we use the following methods [GBS03]: representing abstract values by a single interval, merging states, collapsing loops and reducing

the information needed for iteration count of loops.

The abstract values are represented by a single interval. An abstract value is the set of all possible distinct values that a variable can hold at a certain point of the execution. In order to decrease the amount of data to process we store these values as a single interval in the current implementation of our tool. In case a value of a variable controls the number of iterations of a loop this approximation does not need to decrease the tightness of the WCET value.

Merging of states. Suppose there are two distinct sub paths in the flow graph, p_1 and p_2, that have at least the start and end nodes in common. For each of these paths, the abstract interpreter will calculate different states, i.e. different abstract values of the involved variables. To reduce the amount of data to be processed, these states can be merged to a single state in one of the common nodes of p_1 and p_2, meaning that for each variable in any of the states the abstract value in the new state is calculated as the interval containing all values. We call such common nodes merge points. In general a merge may lead to a larger overestimation but on the other hand it can speed up the analysis time considerably. For example, the following positioning of merge points can be considered:

- at function end
- at loop termination
- at each loop iteration
- after if statements

The alternative is selectable in our tool. A suitable merge point that is expected to have a big effect is once in each iteration of a loop in the flow graph. Since the loop header node is the only node in a loop that is guaranteed to be taken in every iteration, it can be used as a merge point.

Collapsing loops. By use of syntactical analysis we can identify a certain set of loop constructs that can be transformed to a closed form expression. Depending on the analyzed program, this may have a big influence on the need of computation.

Iteration count of loops. The iteration counts of inner loops may be either calculated separately from the outer loop or as a sum of all iterations of the outer loop. Again, the reduction of data needed to be handled in calculation when choosing the latter is payed for as a possibly less tight final result. However, for simple loops there need not be any reduction of the tightness.

1.2 Scope Graph

We create a *scope graph* to provide our tool with a structure of the program on a higher level than the flow graph [GBS03]. The scope graph is a directed acyclic graph of scopes. Each scope is a container for a certain possibly looping program construct like a loop or a function. The scope graph is therefore based on a call graph merged with the flow graphs [EE00]. The scope graph can support both the flow analysis and the low level analysis with structure information of the program. A scope contains a non-empty set of nodes that refers to basic blocks in the flow graph.

Figure 1 depicts a loop with the single node 5, constituting a scope. The function itself will constitute a scope, which subordinates the loop scope. Scopes forms a tree hierarchy.

The flow information like iteration counts of loops and edges is expressed using so called flow facts, see [EE00]. Each loop needs to be annotated with a flow fact giving the upper bound of the iteration count. In addition other flow facts can be created to obtain a more tight final WCET. The flow facts can also be bound to certain conditions.

An important feature of the scope graph is that the nodes in the scopes are not the actual nodes of the flow graph, but rather pointers to these (in figure 3 these nodes are annotated with the numbers of the nodes in the control flow graph that they refer to). This gives a flexibility to express the flow information in a context sensitive manner, since a single flow graph node can be pointed to from different scope nodes. This feature will be used in the following.

2 The Problem of Unstructured Loops

As explained in the introduction we have to manage unstructured loops. An unstructured loop is a loop without a single header, i.e. a node (basic block) in the flow graph that dominates all nodes in the loop body [ASU86].

To analyze unstructured loops one of the following may be a solution:

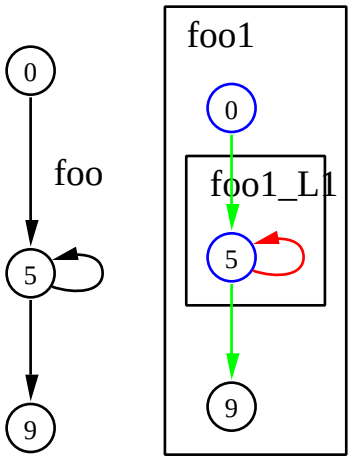

Figure 1: The flow graph and the corresponding scope graph for a function containing a loop.

- [GBS03] describes a method based on attaching one iteration couter to each loop header. This may lead to overestimation.
- Attaching one iteration counter to each basic block. This will give a lot of flow information.

Both solutions have clear disadvantages. Therefore, we have developed a method to transform unstructured loops into structured.

```
int irr(int x)
{
   if (x <  0)
      goto L1;
   do {
      x++;
     L1:
      x+=2;
   } while (x<10);
   return x;
}
```

Figure 2: A simple C-function containing one unstructured loop.

3 Eliminating Unstructured Loops

Unstructured loops in a control flow graph can be seen as loops with entry edges to more than one node in the loop. Actually, an unstructured loop can be seen as different loops sharing (parts of) the same code. In some cases the unstructured loop may actually have been created from different structured loops, merged together by a code size optimizing compiler. Therefore a straightforward way to eliminate unstructured loops is to reverse that step: for each entry of an unstructured loop we create a scope that contains only that entry edge ignoring the other, hence resulting in a scope that has a single header, see Figure 3. This can easily be done because of the scope graph construct proposed by [EE00]. Since each scope node is a pointer to a basic block rather than the basic block itself, it is possible to have duplicate scopes refering to the same piece of code.

Elimination of unstructured loops in this way can be done in a straightforward manner by use of well-known techniques. The advantages with elimination of unstructured loops are:

- Merge points in abstract interpretation can be found without extra effort in the analysis. Since the unstructured loops will be transformed to structured loops there is no difference in the handling.

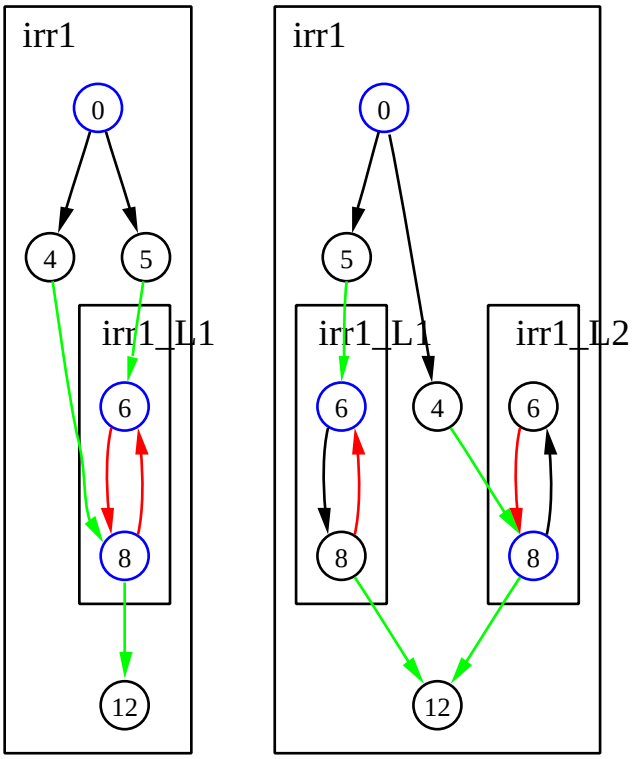

Figure 3: Scope graphs for the unstructured program in Figure 2. The left graph maps directly to the unstructured control flow graph. On the the right hand side there is a scope graph for the same program where the unstructured loop is replaced by two structured.

- The upper bound of originally unstructured loops will not be less tight, which they would have been if the loop had been analyzed in unstructured form.
- It makes tree-based calculation possible to use. The tree-based calculation is not able to handle unstructured loops.
- The various steps of the analysis will be simpler if they can assume that all loops are structured.

There exists several algorithms that can be used to identify unstructured loops [Ram02]. In our implementation we use DJ-graphs, but another can be used as well. One advantage of DJ-graphs is that they are relatively simple to implement. One disadvantage is that the elimination of multiple entry edges of an unstructured loop may expose nested loops, structured or unstructured, that was not discovered by the algorithm.

4 Conclusions

This paper suggests transforming unstructured loops to structured in the flow analysis part of a WCET calculation tool. The method is based on known technologies. We have implemented this as a preprocessing step in our tool for WCET calculation and verified that the abstract interpretation is capable of analyzing originally unstructured code, without any special modification except the preprocessing step. We have also verified that the final WCET calculation, including path-based calculation, works as expected.

5 Future Work

In case the elimination of unstructured loops exposes loops that was not found by the DJ-algorithm, our calculations will fail. To solve all cases of unstructured loops we need to refine the implementation of the loop analysis in our tool. One straightforward method would be to apply the DJ-algorithm recursively, once for each copy of an unstructured loop. Another solution can be to implement the node splitting method, see [ASU86] and [UM02].

References

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986. Generally known as the "Dragon Book".

[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.

[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] Jan Gustafsson, Björn Lisper, Nerina Bernmudo, Christer Sandberg, and Linus Sjöberg. A Prototype Tool for Flow Analysis of C Programs. In *Proc. 14th Euromicro Conference of Real-Time Systems, (ECRTS'02)*, pages 9–12, 2002.

[Ram02] G. Ramalingam. On Loops, Dominators, and Dominance Frontiers. *ACM Transactions of Programming Languages and Systems*, 24(5):455–490, September 2002.

[UM02] S. Unger and F. Mueller. Handling Irreducible Loops: Optimized Node Splitting versus DJ Graphs. *ACM Transactions of Programming Languages and Systems*, 24(4):299–333, 2002.