A Prototype Tool for Flow Analysis of C Programs

Jan Gustafsson, Björn Lisper, Nerina Bermudo, Christer Sandberg, Linus Sjöberg
Department of Computer Engineering
Mälardalen University, Västerås, Sweden
{jgn, blr, nbo, csg}@mdh.se, lsg98020@idt.mdh.se

Abstract

We describe a prototype tool for flow analysis. The purpose of the tool is to statically analyse C programs in intermediate code format, and to calculate flow information, like loop bounds. This information will be used by a subsequent low-level analysis to calculate a final worst case execution time. We describe the main steps of the tool, and analyse a simple example to illustrate our method.

1 Introduction

Predicting the Worst Case Execution Time (WCET) of programs is an essential step in designing real-time systems, especially hard real-time systems. Methods based on static analysis can guarantee the safeness of the predicted WCET, while measurements, in the general case, can not.

In the presence of loops and recursion, finite iteration bounds must be given to the WCET calculation method. Most often, they are given as *manual annotations* by the programmer. Optional annotations (like information on infeasible paths) may also be given, to reduce the overestimation of the calculated WCET. The annotations can be supplied as comments or in a separate file.

A problem with manual annotations is that the calculation of these are often time-consuming and error-prone. It would be advantageous if these annotations could be calculated automatically. This is the aim of the project described in this paper.

The WCET project is a sub-project within CODER (Cluster on Distributed Embedded Real-Time Systems) in the ASTEC [AST01] competence center. The project consists of two groups, one at Uppsala University (low-level analysis) and one at Mälardalen University in Västerås. The flow analysis research is an activity of the Västerås group.

The flow analysis tool is a part of a planned, complete WCET tool (see [EES+01] for details). The flow analysis part will calculate the possible flow of the analysed program. This information will, together with the results from the low-level analysis, be used to calculate a final WCET.

2 Overview of the Tool

2.1 The Input of the Tool

The tool analyzes C-programs in intermediate code format. We will use the NIC (New Intermediate Code) format (developed within CODER). The full ANSI C language will be supported (including pointers, recursion and unstructured code).

We assume that the code represents a syntactically and logically correct program. For example, we assume that array indices are within bounds. We also assume that the control flow graph of the analysed program is the same as in the final machine code, i.e., that the final steps to machine code does not change the control flow.

Manual annotations are used as a complement when the automatic flow analysis fails.

2.2 The Calculation

There will be a possibility to choose between a slower but more exact analysis or a faster but less accurate. It will also be possible to change certain compiler- or system-specific data used in the analysis (like integer type sizes).

2.3 The Output

The purpose of the tool is to calculate flow information ("flow facts", see [EE00]), like number of iterations and recursion levels, infeasible paths etc., that will be used in the subsequent low-level analysis. The flow facts are attached to the scope graph [EE00] of the analysed program. A scope graph is a partition of the program into scopes; a scope is a part of the program where certain flow facts are valid.

2.4 Flow Analysis Overview

Basically, the analysis of a C program is performed using the steps described in Figure 1.

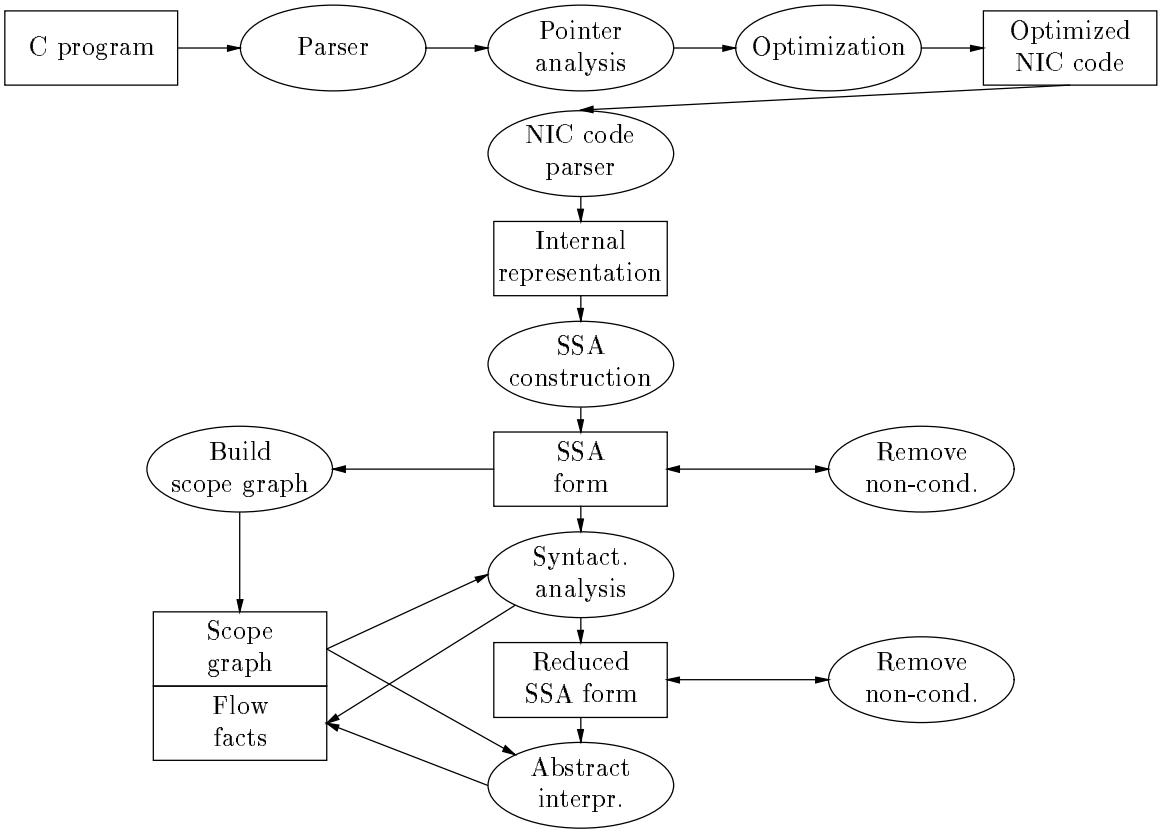


Figure 1: Basic analysis steps.

- Parser. The C code is parsed to produce a NIC file.
- Pointer analysis. Pointers in the program are analyzed. Information about the resulting points-to sets are stored in the NIC file.
- Optimization. The NIC code is optimized. These first three steps are developed within the WPO project.

- NIC code parser. The optimized NIC code is parsed to produce an internal representation. This internal format is the basis for all subsequent analysis steps.
- An SSA (Static Single Assignment) conversion is performed. The calculated data is added to the existing internal representation.
- Non-conditionals are removed. All assignments to variables that do not affect control flow (transitively) are identified and removed from the program. If all references to a variable are removed, the variable will be removed completely. The reason is to simplify and speed up the rest of the analysis.
- Scope graph construction. The scope graph is constructed using the control flow that can be extracted from the internal representation.
- Syntactical analysis. The code is "scanned" for simple, recognizable loop constructs and the corresponding loop counts are calculated, if possible. The loops are replaced with assignments to the final values for the variables updated in the loop, resulting in a simpler program to analyze in the following step.
- The removal of non-conditionals is run again, since variables may become non-conditional during syntactical analysis.
- Abstract interpretation. The remaining code (after the previous step) is analysed using abstract interpretation. The resulting flow facts are appended to the results file.
- If there are constructs for which the abstract interpretation fails, the user is asked for manual annotations for these. The analysis continues with these two last steps until the complete code is successfully analysed.

3 Complete Example

The code in Figure 2 contains a simple and motivating example, activating all the steps of our tool. For simplicity reasons, it does not contain pointers, arrays, or unstructured code. The variable `i` is assumed to receive a value between 0 and 5 by `get_value()`.

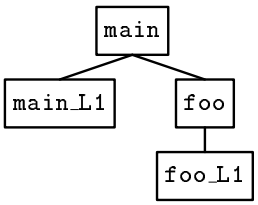
```
int main(void) {
  int i, j, k, n = 10, c = 2, p;
  for (i = get_value(); i <= n; i = i + c) {
    p = i - c * 2; result += 2;
    if (p) k = i + 2;
    }
  j = i + k;
  j = foo(p);
  return(0);
  }
```

```
int foo(int j) {
  int i, result = 0;
  for (i = 0; i < 100; i++) {
    result += 2;
    }
  return(result);
  }
```

Figure 2: Example program

We first parse the code to a NIC file. Conversion to SSA form and removal of non-conditionals (`j`, `k`, `p`, and `result`) yields a NIC code that is equivalent to the C code in Figure 3.

Next step is to calculate the scope hierarchy below. We see that each function and loop constitutes a scope.



```
int main(void) {
  int i, n = 10, c = 2, p;
  for (i = get_value(); i <= n; i = i + c) {
    p = i - c * 2;
    if (p) {}
    }
  foo(0);
  return(0);
  }
```

```
int foo(int j) {
  int i;
  for (i = 0; i < 100; i++) {}
  return(0);
  }
```

Figure 3: Example program after removal of non-conditionals

The syntactical analysis will recognize the loop in `foo` as analyzable and output the flow fact

$$\texttt{foo\_L1:[ ]:}x_{\texttt{header(foo\_L1)}} = 100$$

which means that the loop in scope `foo_L1` iterates exactly 100 times. The notion $x_{\texttt{header(foo\_L1)}}$ refers to the iteration count of the loop header. The function `foo` will be changed by the syntactical analysis as shown below. We see that the loop has been replaced by an assignment.

```
int foo(int j) {
 int i = 100;
 return(0);
 }
```

A new run of removal of non-conditionals removes the variable `i` in `foo` since it does not affect the control flow.

Abstract interpretation of the remaining program will yield the following flow facts for the remaining loop:

1. $\texttt{main\_L1:[ ]:}x_{\texttt{header(main\_L1)}} \geq 3$
2. $\texttt{main\_L1:[ ]:}x_{\texttt{header(main\_L1)}} \leq 6$
3. $\texttt{main\_L1:<4..6>:}x_{\texttt{true}} = 1$

The first two flow facts means that the loop in `main` iterates between 3 and 6 times. The second means that the program will always take the true edge in the if-statement in iterations 4 to 6 of the loop.

References

[AST01] ASTEC (Advanced Software TEChnology) WWW Homepage. URL: `http://www.astec.uu.se/`, November 2001.

[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.

[EES+01] J. Engblom, A. Ermedahl, M. Sjödin, J. Gustafsson, and H. Hansson. Worst-case execution-time analysis for embedded real-time systems. *Springer International Journal of Software Tools for Technology Transfer, (STTT)*, 2001.

[EG97] A. Ermedahl and J. Gustafsson. Deriving Annotations for Tight Calculation of Execution Time. In *Proc. 3rd International European Conference on Parallel Processing, (Euro-Par'97), LNCS 1300*, pages 1298–1307, August 1997.