TACLeBench Flow Facts Documentation

(v1.2 / 11.9.2013 / H. Falk)
(v1.1 / 27.7.2010 / T. Kelter)
(v1.0 / 8.8.2008 / R. Pyka)
Partial translation of Thesis by D. Schulte, 2007 [1]

Definition of Flow Facts

Flow Facts have been defined by R. Kirner in his Ph.D. Thesis [2] as follows:

Definition 1:

> Flow facts are meta-information which provide hints about the set of possible control flow paths of a program *P*. The corresponding set of control flow paths is denoted as $CFP_{ff}(P)$

Furthermore, Kirner defines two types of flow facts. The first type describes properties which are implicit due to the structure and semantics of the program, the second type of flow facts is explicitly annotated by the user:

Definition 2:

> Flow facts which exist due to the structure and semantics of the program code are called implicit flow facts (ff_{impl}). Flow facts which are provided by the user are called annotated flow facts (ff_a)

The option for user-provided flow facts is important for a precise WCET-analysis, because the implicit flow facts usually have only a limited quality. This is due to the limitations of analysis tools and due to the fact that the control flow of a program depends on the input data, which is not considered in the computation of implicit flow facts.

There is no clear distinction between ff_{impl} and ff_a. Both sets are not required to be disjoint. The representation of flow facts in TACLeBench is basically the representation of ff_a. Nevertheless, the flow facts could be generated automatically.

Basic aspects of TACLeBench flow facts

In the current literature, flow fact information is attached to the edges of the control flow graph. With respect to annotated flow facts, this has several disadvantages. Control flow edges are not explicitly expressed in the C program code and are therefore not obvious to the user. The natural way of annotating flow facts in a C program would be to define them in relation to program statements. There are three types of flow facts which are used to annotate the TACLeBench source codes: Flow restrictions, loop bounds and entry points.

Flow restrictions and Loop bounds

Flow restrictions allow to express the execution count of one statement in relation to the execution count of other statements. This annotation method would provide sufficient information to express any kind of linear dependency.

Flow restrictions are expressed in terms of inequations. Each side of an inequation is a weighted sum of execution counts of statements. The operator can be one of the following: less-than-equal, equal or greater-than-equal. The semantics of such an inequation is slightly different from what is expected from regular mathematical inequations.

Example: A flow restriction

$1 * stmt_1 <= 15 * stmt_2$

states that one execution of $stmt_1$ is limited by 15 times the execution count of $stmt_2$. The notion becomes more intuitive if $stmt_1$ and $stmt_2$ are not considered to be the statements themselves, but variables in an inequation system holding the actual execution count of its corresponding statement.

Besides the flow restrictions, TACLeBench uses another annotation method. Loop bounds are a simplified method for loop annotation. Loop bounds provide an upper and lower bound for the number of iterations of the annotated loop.

Entry points

Entry points denote points in a program's control flow graph (CFG) where the control flow may start. Typically, this is the "main" function of the program, but in a (possibly interrupt-driven) multi-task system, there may be multiple entry points in a single set of source files. These entry points may even share some common code. In order to express such task entries, each function of a multi-task application where a task begins can be marked as an entry point.

Specification of flow facts in the source code

According to the previous section, flow facts are attached to program statements. TACLeBench uses ANSI-C pragmas for this purpose. Flow facts encapsulated into pragmas can be annotated in the source code without introducing incompatible changes to the syntax of the C program. Therefore, even flow fact annotated code can be processed by legacy compilers and tools which do not take advantage of this information.

The C standard offers two styles of pragmas:

```
#pragma ANNOTATION ↵
```

or

```
_Pragma("ANNOTATION")
```

where ANNOTATION expands as follows in the case of flow facts:

```
ANNOTATION        |=  MARKER | FLOWRESTRICTION | LOOPBOUND |
                      ENTRYPOINT
```

Besides flow restrictions and loop bounds, additional annotations are required to express flow facts in the source code. So-called markers are used to identify program statements in flow restrictions.

```
MARKER            |= marker NAME
NAME              |= Identifier
```

Each marker relates an identifier to the corresponding statement where the marker-pragma has been placed. If the pragma is found to be attached to an expression, the marker is related to the parent statement of that expression. Some restrictions apply to the place where markers can be set:

- Compound statements cannot be marked. This is not a limitation, since compound statements do not match exactly the control flow graph of the program. Flow facts are more precisely expressed in relation to a single statement, which in turn have an exact matching to a corresponding basic block.
- Each function has a default marker with the function name as identifier. It gets related to the entry basic block of this function.
- Markers placed at loop statements are related to their conditional expressions. Therefore,

unconditional loops cannot be marked that way. Nevertheless, the loop body statements can be marked.

The identifier name is an arbitrary alphanumerical string. From the pure semantical point of view, the marker represents a decision variable which provides information about the execution counts of that statement on a control path. The set of values is restricted in loop bounds and flow restrictions which define that decision variable.

Flow restrictions

Flow restrictions allow to define execution counts of statements in relation to execution counts of other statements.

```
FLOWRESTRICTION |=  flowrestriction SIDE COMPARATOR SIDE
COMPARATOR      |=  >=  | <= | =
SIDE            |= SIDE + SIDE | NUM * REFERENCE
NUM             |= Non-negative-Integer
REFERENCE       |= Identifier | Function-name
```

References to statements are expressed in terms of identifiers, where the identifier is equal to the identifier in the marker referencing the statement. Function names are implicitly defined and do not require any marker.

Example: Flow restrictions can be used to express precise upper execution bounds for statements located inside nested loops. Furthermore, multi-entry loops, goto-loops etc. can also be annotated. The basic approach to define maximum iteration counts for loops is to mark a statement outside the loop nest and one inside. The inner statement has to be executed in each loop iteration, regardless of any other condition. Afterwards, a flow restriction has to be defined which limits the maximum execution count of the inner statement.

“flowrestriction 1 * *inner-marker* <= max * *outer-marker*”

Where 'max' is the actual maximal execution count of the inner statement.

A simple example showing the use of flow restrictions and markers in that example scenario is shown in the following listing:

```
_Pragma( "marker outer-marker" )
Stmt-A;
for ( i = 0 ; i < 10 ; i++ )
     for ( j = i ; j < 10 ; j++ )
          _Pragma( "marker inner-marker" )
          Stmt-B;

_Pragma( "flowrestriction 1*inner-marker <= 55*outer-marker" )
```

As the flow restriction states, Stmt-B is being executed no more than 55 times. This is more precise than what could be annotated using loop bounds, since loop bounds can only define the bounds per loop, therefore booth loops have to be annotated with their maximum iteration counts separately, which is 10 in both cases. Using a loop bound annotation, only a less precise maximum execution count for Stmt-B of no more than 100 executions can be deduced.

In a similar way, the maximum recursion depth of recursive functions can be limited. In that case, the call-expression statement has to be marked (i.e. 'call-marker'). The second marker is not required, because the reference points to the function entry which has an implicit marker. A flow restriction limits the recursion

depth in a similar way as for a loop:

```
"flowrestriction 1 * recfunc <= max * call-marker"
```

Flow restrictions are a powerful description method. They allow the description of even more exotic dependencies like mutually exclusive execution of statements. Nevertheless, it is not possible to express deeply nested dependencies as shown in the following example.

```
if ( cond )
     x = true ; // Stmt-A
for ( ... )
     if ( x )  Stmt-B;
```

The execution of Stmt-A implies the execution of statement Stmt-B which cannot be expressed using flow restrictions.

Loop bounds

Loop bounds allow to specify the minimum and maximum iteration counts for each loop.

```
LOOPBOUND        |= loopbound min NUM max NUM
NUM              |= Non-negative-Integer
```

It is not required to mark (loop-) statements for loop bound annotation, since each loop bound applies to exactly that statement where the pragma has been defined. Loop bound pragmas can be annotated only to for-, while- and do-while-statements. Other loop types (e.g., goto-loops) or loops without an exit condition can only be annotated using flow restrictions.

Entry points

Entry points allow to specify that the control flow may initially enter the code at the marked function. If no entry point annotation is given, the "main" function implicitly becomes the only entry point.

```
ENTRYPOINT       |= entrypoint
```

The annotation must be put in front of the name of the function which is to be marked, just after the return value. Both, the declaration and the definition of the function can be marked, which has the same effect.

References

[1] Schulte, Daniel: *Modellierung und Transformation von Flow Facts in einem WCET-optimierenden Compiler.* Dortmund, Technische Universität Dortmund, Dipl. Thesis, May 2007.

[2] Kirner, Raimund: *Extending Optimising Compilation to Support Worst-Case Execution Time Analysis*. Wien, Technische Universität Wien, Ph.D. Thesis, May 2003.