Scope-tree: a Program Representation for Symbolic Worst-Case Execution Time Analysis

Antoine Colin   Guillem Bernat

Department of Computer Science, University of York
York, YO10 5DD, United Kingdom
Email: {acolin,bernat}@cs.york.ac.uk

Abstract

Most WCET analysis techniques only provide an upper bound on the worst case execution time as a constant value. However, it often appears that the execution time of a piece of code depends on the sizes or values of its input data or local parameters. The WCET of a function call may vary depending on the caller and parameters. We propose an approach to express the WCET of a program or sub-program as a symbolic expression. The obtained parametric WCET can then be later evaluated using the knowledge of input data and system configuration parameters.

In this paper we present the concept of scope-tree as a generalisation of the traditional syntax tree representation of programs. In addition to their WCET, scopes are associated with an expression stating their maximum execution frequency and some variable declarations. These variables may be used for example to express data-dependent number of iterations or non-rectangular loops. We also present how the scope tree may be used to express inter-scope relations (e.g. mutually exclusive paths, loop down-sampling). Finally, this paper presents the use of scope-trees and scope-tree modifications on an example.

Keywords: Real-time systems, WCET, symbolic WCET analysis.

1 Introduction and motivation

Real-time systems differ from other systems by a stricter criterion of the correctness of their applications. Actually, the correctness of a real-time application does not only depend on the delivered result, but also on the time when it is produced. In *hard* real-time systems, missing task deadlines can be catastrophic. As a consequence, knowing task worst-case execution times (WCET) is of prime importance for the timing analysis of such systems. The purpose of worst case execution time analysis is to estimate *a priori* (before execution) the WCET of a given program on a given processor.

WCET analysis is usually done at two levels [15]. The low-level provides the execution time of *basic blocks*[1] taking into account the effect of the hardware. On the other hand, the high-level identifies the longest execution time of the program using a representation of the program (control-flow graph or syntax tree). A main issue in program timing analysis is to avoid pessimism in timing evaluation by improving WCET bounds' tightness. One part of the pessimism of WCET analysis is due to the presence of some hardware features, such as caches, branch prediction mechanisms and pipelines, that affect the execution time of instructions. So the timing analysis of these features is an important research topic in the real-time systems area. However, they are not the only source of overestimation in WCET analysis. To achieve a tight WCET estimation we need information about the program behaviour such as infeasible paths and maximum number of iterations of loops [16, 13, 17]. This kind of information is usually provided by the user in the source code but may also be automatically obtained. Using this information makes it possible to tighten the estimate of the worst case behaviour of programs and to avoid to take into account infeasible behaviours leading to WCET overestimation.

Motivations and Paper contributions

In this paper we focus on the high-level source of pessimism. We address the problem of obtaining high level tight estimations of the WCET by characterising the context in which the analysed code will be executed. This can be achieved by representing the WCET as a parametrised expression - a function of some external variables - instead of a constant value. This technique allows to tighten the WCET of pieces of code for which execution time may vary depending on input data, local parameters or system wide configuration. This is the situation found during the analysis of the RTEMS kernel[6] where the WCET of run-queues management functions depends on the maximum number of tasks allowed to be created in the system. The introductive example in [1] shows how the WCET of a function may depend on its parameters. The example is the `pow(f,n)`

---

[1]A basic block is a sequence of one or more instructions with a single entry point and a single exit point.

function that computes f^n using a loop that iterates n times. This function may be called in a loop with a variable parameter:

```
for n = 1..10 do pow(2,n)
```

In such situation we would like to express the WCET of `pow` as a function of its parameter n and not as a constant value computed considering the worst case value of n (*i.e.*, 10).

In tree-based analysis [16, 14], the WCET of a program is computed using the syntax tree of the program and the knowledge of the WCET of its basic blocks. A set of rules (called *timing schema*) are used to translate the syntax tree, the WCET of basic blocks and some user-provided information into an equational system which, when solved, provides the WCET. The program representation we present in this paper (called *scope-tree*) is a generalisation of the traditional syntax tree representation of programs and is intended to be used in symbolic WCET analysis. A scope-tree represents the syntax structure of the analysed code, the WCET of basic blocks, the user-provided information and the timing schema all at once.

The motivations for the definition of this new representation are the following:

- **Extensibility:** as the timing schema is implicitly described by the scope-tree itself, the computation of the WCET does not rely on the use of a fixed timing schema. Thus, the set of expressions used to express the WCET is not limited.

- **Symbolic WCET**: the proposed representation allows to describe WCET using some variables. It allows us to represent a variable WCET (see the `pow(f,n)` example above) by a symbolic expression. As we rely on computational algebra systems (like Maple [3], or Mathematica [19]) to manipulate, simplify and evaluate these expressions, the only limitation is the power of the chosen symbolic computation tool.

- **Out of hierarchy relationship**: there exist cases where particular assertions have to be made on parts of a program as for example *mutual exclusive paths* or loop *down-sampling*[2].

- **Genericity**: finally, the proposed technique allows to express all that can be expressed by other tree-based techniques, and more.

Note that, as a first step, we consider that the WCET of basic blocks are constant and independent. By doing this we ignore the effects of some architectural features which add some variability to the execution time of instructions. Such assumption leads to some pessimism, but it is required to keep the presentation clear and focused on the high-level

---

[2]Down-sampling: executing one part of the body of a loop less often than the rest of the body.

analysis. We are currently working on the adaptation of the proposed symbolic WCET computation method to take into account basic blocks with path-dependent WCET.

Paper organisation

The remainder of the paper is structured as follows. The next section introduces related work on high level WCET analysis. We give (in section 3) our definitions of *scope* and *scope-tree* and we detail how we intend to use them to express tight WCET. Section 4 presents an example. This example is used to illustrate the representation of data-dependent WCET using scope-tree (*i.e.*, the declaration and use of variables in scope expressions), and to explain how a scope-tree may be modified to express inter-scope relations (*e.g.* mutually exclusive paths, loop "down-sampling"). The scope-tree based computation of the symbolic WCET of the example is also presented. Finally, we present some conclusions as well as current and future work.

2   Related work

Static WCET analysis has been the subject of much research, and substantial progress has been made in the area over the last decade (see [15] for a review of WCET). The WCET estimation is a twofold issue: (i) possible execution paths have to be determined to calculate the execution time along the worst of these paths, (ii) execution time for each atomic unit of a path has to be accurately estimated. In this paper we will only focus on the first issue. There are two ways to determine what sequence of instructions will be executed. On one hand, it is possible to consider all paths implicitly by using integer linear programming (ILP) [10, 17, 12, 7]. On the other hand, the enumeration of program paths can be explicit as in tree-based methods [16, 14, 11, 5].

ILP techniques transform the control-flow graph representing a program into a constraint system and a WCET function, which is maximised by an ILP solver that provides an integer value. All the variables of the equation system have to be constrained, and are given integer values by the ILP solver. So, this kind of techniques can not provide symbolic expressions as WCET results. Thus, the ILP approach is clearly not suitable for symbolic WCET purpose. Tree-based techniques rely on the use of *timing schemas*. A timing schema is a set of rules used to translate the syntax tree of a program into an equational system that must be solved to produce the WCET. Some approaches for dealing with so-called *non-rectangular* loops [8, 2, 9, 4] were developed based on transforming nested loops into nested summations and evaluate and simplify such expressions to obtain WCET estimates. However, all of these approaches rely on the fact that loops are bounded with constant values and the WCET is in the end a constant number. The symbolic WCET analysis proposed in [1] and the parametric timing analysis presented in [18] both propose to compute a

symbolic or parametric WCET, *i.e.*, an algebraic expression of some variables.

The proposed technique is intended to express all the results obtained so far in tree-based approaches and to take into account some situations that are traditionally only taken into account by ILP approach (*e.g.* mutual exclusive paths).

3   From syntax tree to scope tree

The syntax tree is a representation of the program whose nodes describe the structure of the program in the high-level language and whose leaves represent basic blocks. The most often, four types of tree nodes are defined: *sequence*, *loop*, *conditional* and *basic block* (which is a node with no child). The timing schema describes the translation of each node into an equation that expresses its WCET based on the WCET of its children nodes. The timing schema contains a fixed number on "translation rules" (one per node type).

In the approach proposed in this paper, we generalise this notion by allowing arbitrary expressions to express the WCET and also by parametrising such expressions with variables. Moreover the WCET of a sub-tree may not only depend on the WCET of its sons, but may also use the WCET of any arbitrary sub-tree.

3.1   The scope tree representation

We define the concept of scope-tree as a generalisation of the traditional syntax tree used in tree-based analysis techniques. A syntax tree can be viewed either as a collection of nodes and edges or as a hierarchy of sub-trees. It is this second view that fits best the concept of scope-tree. Scope-trees are made of *scopes*, which are equivalent to sub-trees of the syntax tree (see an example on figure 2).

Definition of a scope

A scope represents a sub-tree of the syntax tree, its behaviour and its value (*i.e.*, its WCET). In the following, we use uppercase latin letters to name scopes. As for nodes of the syntax tree, a scope S has only one parent scope ,$\widehat{S}$, and may have some children scopes ,S_i.

The WCET of a scope is described using two expressions: φ and ω. The ω expression represetns the WCET of the scope and φ is its worst case execution frequency. φ takes at least one parameter (*e.g.* an execution frequency of 10 would be described by $\varphi(\alpha) = 10 \times \alpha$). Then, the WCET of a scope as seen by a parent scope is $\varphi(\omega)$, the WCET of the scope modified by its execution frequency function.

More formally, a scope S is defined as the collection of:

- A WCET symbolic expression, ω_S, called ω expression, which expresses the WCET of S based on the WCET and execution frequencies of its sons S_i.

- An execution frequency expression ,φ_S^R, which defines the number of executions of S regarding an outer scope

R. We will normally only express the frequency of a scope regarding its parent scope. For this reason $\varphi_S^{\widehat{S}}$ will be abbreviated as φ_S.

- A set of defined variables: $\mathcal{D}_S$, *i.e.*, variables that *may be* used in the ω expressions of S and all of its sub-scopes, and in the φ expressions of the sub-scopes of S.

- A set of used variables: $\mathcal{U}_S$, *i.e.*, variables which *are* used in scope expressions (ω and φ) of S and sub-scopes of S.

The ω expression of scope S describes how to compute the WCET of S knowing the WCET of all its sons and how many times each of them will be executed (*i.e.*, the execution frequency of its sons). All declared variables (*i.e.*, variables in $\mathcal{D}_S$) may be used in the ω_S expression. So, the resulting WCET may be a parametric WCET.

The execution frequency expression expresses the execution frequency of a scope regarding another scope using scope variables. For instance, the execution frequency expression φ_S^R represents the number of times the scope S is executed for each execution of an outer scope R (called the reference scope). Execution frequency expressions are very flexible and allow the definition of very tight frequencies of exotic scopes. For example, consider three scopes R, S and T with T being a son of S, which is a sub-scope of R. Assume that S is executed 10 times for one execution of R, and that T is executed 2 times for one execution of R (see figure 1).

```
for i=0..9 do
    if (i==3) or (i==5)
        then do something
```
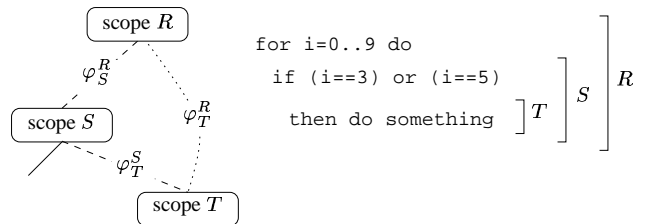
Figure 1. Example of relative execution frequency: $\varphi_S^T = 1$ and $\varphi_R^T = 2$ (instead of 10).

With a tree-based WCET analysis technique that cannot handle down-sampling, the worst case iteration number of T would be 10 times for each execution of R. We want to express the fact that T is executed only 2 times in R. This is done by relating the execution frequency of the scope T to the outer scope R (instead of its direct parent S). The WCET of scope T is ω_T, but from the point of view of R it is $\varphi_T^R(\omega_T)$ where $\varphi_T^R(\alpha) = 2 \times \alpha$. More generally, when the φ expression of a scope S is φ_S^R with $R \neq \widehat{S}$, it means that the WCET of the scope S should not be used to

3

express the WCET of scope $\widehat{S}$ but the WCET of scope R. This kind of situation and the way they can be solved are detailed in paragraph 4.4 and 4.5. All declared variables of the reference scope (*i.e.*, variables in $\mathcal{D}_R$) may be used in the expression φ_S^R.

Associated to each scope S are two sets of variables $\mathcal{D}_S$ and $\mathcal{U}_S$. $\mathcal{D}_S$ is the set of variables defined in scope S. The final WCET of S will be defined as a function of some variables of this set. $\mathcal{U}_S$ is the set of used variables. As scope expressions can be only defined in terms of scope variables of outer scope, $\mathcal{U}_S$ should always be included in $\mathcal{D}_S$ to ensure that all used variables are defined.

Scope-tree

A program or a section of program to be analysed can then be seen as a hierarchy of scopes: the *scope-tree*. A scope-tree is defined by its root scope, and only the variables defined by the root scope of the scope-tree may appear in the final expression of the WCET of the scope-tree.

It may happen that a sub-part of the scope-tree is unknown when the scope-tree is first constructed. It may be the case of some function calls in the program for which the code and the scope-tree of the called function is not provided (*e.g.* functions which are part of a library).

Let us consider a function call. If the scope-tree of the call function is not provided then function call can be represented by a scope with no son. This scope may declare and give values to some variables: the parameters of the called function. To evaluate the WCET of a program that contains such a function call, a representation of the called function has to be provided either as a scope-tree or as a ω expression representing the WCET of the called function (this expression may be parametrised by the function parameters).

3.2 Representing programs using scope trees

In the previous paragraph, we have defined scopes and scope-trees. We now want to represent a program, or a fragment of program to be analysed, using a scope-tree. As a scope-tree is a generalisation of syntax tree, it can obviously be used to do the same WCET computations as with syntax-tree. We show here how to translate a syntax tree and timing schema into the equivalent scope-tree.

As an example, let us consider a syntax tree made of four types of nodes (*sequence*, *loop*, *conditional* and *basic block*) and the timing schema introduced in [16]. The sub-trees (or nodes) of the syntax trees are represented by scopes. As previously said, scope-trees are not made of a fixed number of scope types. The "type" of a scope is defined by its ω and φ expressions. To represent the syntax tree according to the chosen timing schema, we provide three default kinds of ω expressions and two kinds of φ expressions:

$$\omega_S^+(S_0 \ldots S_n) = \sum_i \varphi_{S_i}(\omega_{Si})$$
$$\omega_S^\diamond(S_0 \ldots S_n) = \text{Max}_i \{\varphi_{S_i}(\omega_{Si})\}$$
$$\omega_S^\bullet() = \text{constant}$$

$$\overline{\overline{\varphi}}_S(\alpha) = \alpha$$
$$\overset{\times}{\varphi}_S(\alpha) = \text{constant} \times \alpha$$

Note that other kinds of ω and φ expressions can be defined, and that these expressions may be symbolic expressions (see § 4.4 for an example).

The ω_S^+ expression is used to represent *sequence* and *loop* nodes. It calculates the sum of the WCET of all the sons, S_i, of S. The $\omega_S^\diamond$ expression is used to represent *conditional* nodes, it chooses the maximum WCET in a list of WCET expressions. Finally, $\omega_S^\bullet$ is used when the considered node is either a *basic block* (in this case $\omega_S^\bullet$ is an integer) or a void sub-tree as, for instance, a non-existent *else* in a conditional construct (in this case $\omega_S^\bullet = 0$).

The role of φ expression is to describe the relative execution frequency of a scope regarding an outer scope. The most widely used φ expression is the identity function $\overline{\overline{\varphi}}_S$ that specifies that a scope S is executed as many times as its parent scope. The other useful expression is $\overset{\times}{\varphi}_S$, which specifies that a scope S is executed n times each time its parent scope is executed. This expression is used to represent loops of the syntax tree.

Table 1 shows the correspondence between types of node and scope expressions. Scopes are defined for each type of node of the original timing schema using the ω and φ expressions defined above. Note that the φ expression of scope S is not defined by the node S but is defined by its immediate outer scope $\widehat{S}$.

A *sequence* node is represented by a scope S whose WCET is the sum of the WCET of its sons (the ω^+ expression is used), and whose sons' execution frequencies are the same as their parent frequency (expression $\overline{\overline{\varphi}}$). The WCET of the scope S representing a *loop* node is the sum of the WCET of its sons (*Test* and *Body*), and the execution frequency of its sons are multiples of the one of S (this is expressed by $\overset{\times}{\varphi}$). A conditional node is represented by a scope[3] S whose WCET is $\omega^\diamond$, the maximum of the WCET of its sons, and whose sons' execution frequencies are the same as their parent frequency. Finally, the WCET of scopes representing basic blocks are constant and are represented by $\omega^\bullet$.

The example in figure 2 shows the translation of a syntax tree into its equivalent scope-tree. To ease the graphical representation of scope-trees, the $\overline{\overline{\varphi}}$ expression can be omitted

---

[3]The test of the conditional construct is not represented in this scope but is represented as a son of the upper "sequence" scope.

4

Sequence $S : S_0, \ldots, S_n$	Loop $S : \text{while}(T) \text{ do } B$	Conditional $\text{if}() \text{ then } T \text{ else } F$	Basic block
$\omega_S = \omega^+$ $\mathcal{D}_S = \mathcal{D}_{\widehat{S}}$ $\mathcal{U}_S = \bigcup_i \mathcal{U}_{S_i}$ $\forall i,\ \varphi_{S_i} = \overline{\overline{\varphi}}$	$\omega_S = \omega^+$ $\mathcal{D}_S = \mathcal{D}_{\widehat{S}}$ $\mathcal{U}_S = \mathcal{U}_T \cup \mathcal{U}_B$ $\varphi_B^S = \overset{\times}{\varphi}$ $\varphi_T^S(\alpha) = \varphi_B^S(\alpha) + \alpha$	$\omega_S = \omega^\diamond$ $\mathcal{D}_S = \mathcal{D}_{\widehat{S}}$ $\mathcal{U}_S = \mathcal{U}_T \cup \mathcal{U}_F$ $\varphi_T^S = \overline{\overline{\varphi}}$ $\varphi_F^S = \overline{\overline{\varphi}}$	$\omega_S = \omega^\blacksquare$ $\mathcal{D}_S = \mathcal{D}_{\widehat{S}}$ $\mathcal{U}_S = \emptyset$

Table 1. Default scope expressions corresponding to node types

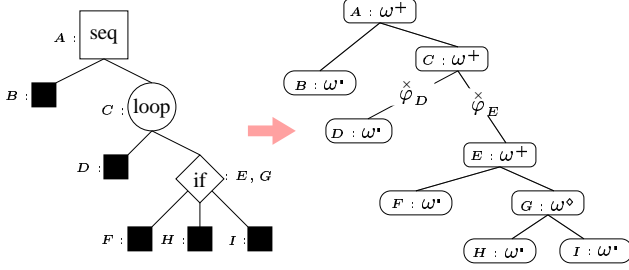in scope-tree representation as it is the default expression.

Figure 2. Syntax tree and equivalent scope-tree

This set of scope expressions is sufficient to represent a traditional syntax tree as a scope-tree. But it is also possible to use some new expressions and variable definitions to represent some more complex situations. The behaviour of a piece of code that cannot be represented accurately in a syntax tree, may be represented by a scope by:

- defining some of its ω expressions to be different from ω^+ and $\omega^\diamond$,

- defining some of its φ expressions to be different from $\overline{\overline{\varphi}}$ and $\overset{\times}{\varphi}$,

- writing scope expressions using some variables defined in outer scopes.

All these possibilities are illustrated in section 4.

3.3 Extracting scopes from the source code

We do not make any assumption on the way to obtain scopes and scope expressions from the source code. The easiest way, from the analysis point of view, is to rely on user-provided annotations, but this requires some extra work from the user. It has been shown in [8, 2, 9] that it is possible for a subset of scope annotations (maximum number of iterations of loops) to be automatically extracted by some analysis tools.

When no particular expressions are specified for a scope, a default ω expression and a default φ expression are used. These default expressions are the ones presented in table 1.

3.4 Scope motion: *shadowing*

As said in the definition of φ, it may happen that the execution frequency of a scope is defined relative to an outer scope which is not its direct parent (φ_S^R where $R \neq \widehat{S}$).

To take into account these relative execution frequencies, the scope-tree has to be modified before the WCET evaluation. The scope-tree is restructured by "moving" S so that it becomes a direct son of R (an example of restructured scope-tree is shown in figure 5). In fact, what we really mean by moving S is simply removing a link in the scope-tree and building a new link from the destination scope to the "moved" scope. Scopes that have been moved are *shadowed*. A new scope, the *shadow* of S, is added as a son of the destination scope. The new link is labelled with a copy of the original φ expression, and the original link to S is removed by setting its φ expression to 0.

The need of shadowing is expressed by the φ_S^R function when R is not the direct parent of S. So, no additional annotation is needed to specify where to perform shadowing.

The semantics of the worst case execution time are preserved by this scope-tree transformation because the number of executions of each scope remains the same in the new scope-tree. Scope expressions can be only defined in terms of scope variables of an outer scope. This has to be verified when moving scopes. Basically, we must verify that the set of used variables of the shadowed scope is included in the set of defined variables of the destination scope (*i.e*, the scope receiving the shadow). If this condition is not verified, the annotations must be re-designed or the scope motion dropped.

3.5 Scope-tree based WCET evaluation

The evaluation of the WCET of a program is done by transforming its scope-tree into a set of WCET functions. Each scope produces two functions corresponding to the ω and φ expressions. The parameters of these functions are the variables of the $\mathcal{U}$ set of the scope. Note that the functions obtained from the φ expressions have an additional parameter: α. We propose to use a computational algebra system like Maple to perform the simplification and evaluation of the expressions. The Maple code of some example WCET functions can be found in appendix A.

The WCET for each scope are then computed recursively by symbolically evaluating the WCET functions in

a bottom-up manner thanks to the scope-tree. After the WCET evaluation, the WCET of each scope of the scope-tree is either a numeric value or a symbolic expression.

3.6 On line scheduling

The result of the evaluation of the WCET expression of a program may be an integer value or an expression of some variables (configuration variables). In the later case, the expression may then be used on-line to compute the actual WCET when the exact value of the variables are known. As the function can be arbitrary complex, we suggest to approximate it with a polynomial function that envelops the WCET expression. Therefore the WCET of the on-line evaluation of the expression itself can be bounded off-line.

4 Explanation of the scope-tree concept on an example

To ease the reading of this paper, we choose to explain the concept of scope-tree and show its expressiveness power on an example. Note that the situations that can be represented using scope-trees are not limited to the ones presented here. We illustrate the use of scope-trees to compute symbolic WCET on the example proposed in [16]. The original example considers the problem of the localisation of a can on a conveyor belt.

4.1 Program specifications

"Tin cans are transported on a conveyor belt. A robot arm seizes the cans and puts them into boxes. The computer controlling the robot arm is connected to a video camera in order to determine the position of the can on the belt"[16]. The image provided by the camera is shown in figure 3.
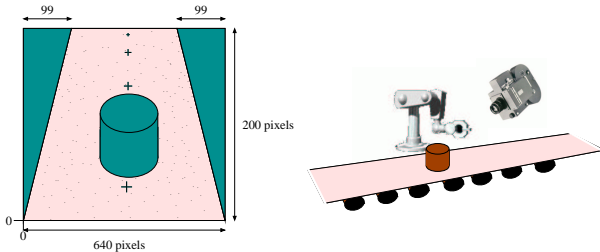

Figure 3. Image of a can on the conveyor provided by the camera

The specifications are the following:

- The image read by the camera is presented in an array of 640*200 black or white pixels.

- The maximum area covered by the image of the can depends on the type of cans conveyed and is known as N. We assume that the value of N may be changed at runtime.

- There may be some noise in the image data. The maximal noise ratio which is tolerated is 10% of N.

- In order to steer the robot arm to the right location, the program has to calculate the center of the marked area.

- It may happen that the camera gives a totally blank image (without any black pixel). In such case, the camera has to be reinitialised.

- Finally, it was not possible to put the camera exactly above the robot arm. So, due to the camera angle, the image provided by the camera is similar to the one presented in figure 3. As the floor is visible on both sides of the conveyor belt, and as the floor is dark enough to perturb the computation, the algorithm should only be applied to the portion of the image defined by $\lfloor y/2 \rfloor < x < 640 - \lfloor y/2 \rfloor$.

A (partial) listing for the camera application is shown in figure 4. Note that, as the issue of annotating the code to define scopes is not discussed in this paper, this listing does not contain any annotation. This example has some interesting characteristics:
- The number of times the section of code between line 12 and line 17 is executed depends on the value of N, which is unknown at analysis time.
- The section of code 12-17 is "down-sampled". It is executed at most N+N/10 times instead of $640 \times 200 = 128000$ times.
- The two loops of this program are non-rectangular loops. The loop called `loop x` iterates $640 - 2\lfloor \frac{y}{2} \rfloor$ times where y is the counter variable of the outer loop (`loop y`).
- Finally, the section of code 12-17 and the 31$^{\text{st}}$ line of the source code are mutually exclusive.

We want to compute the WCET of the example program using all the information that can be extracted from its specifications. Representing the program using a scope-tree will allow us to take advantage of this knowledge to tighten to compute a tight WCET. The resulting WCET will be function of N (see end of § 4.6).

The scope tree of this program is shown in figure 4.

4.2 Data-dependent WCET

It may happen that the execution time of a piece of code depends on the input data of the program, on the system configuration or on some local parameters. This kind of situation is refered as *data-dependent* WCET. In our example, the size (N) of the maximum area covered by the image of the can is unknown at analysis time but will be statically known before the execution of the program (it depends on the system configuration, *i.e.*, the size of the can to localise).

The data-dependent number of execution of the section of code between line 12 and line 17 is expressed using the φ_M^A expression of scope M in the scope-tree shown in figure 4. This expression states that the number of executions

6

```
0    int calc_center(image,x_center,y_center)
1    char image[640][200];
2    int *x_center, *y_center;
3    {
4      int pixel_count, x, y, x_sum, y_sum, black_pixel;
5      black_pixel = pixel_count = x_sum = y_sum = 0;
6      for(y=0;y<200;y ++)              /* loop y */
7      {
8        for(x=y/2;x<640 -(y/2);x++)    /* loop x */
9        {
10         if(image[x][y])
11         {
12           int weight;
13           weight = calc_weight(image ,x,y);
14           x_sum += x * weight;
15           y_sum += y * weight;
16           pixel_count  += weight;
17           black_pixel ++;
18         }
19       }
20     }
21
22     if (pixel_count)
23     {
24       *x_center = x_sum / pixel_count
25       *y_center = y_sum / pixel_count
26     }
27     else
28       *x_center = *y center = 0;
29
30     if (! black_pixel)
31       reinit_camera();
32
33     return 1;
34   }
```
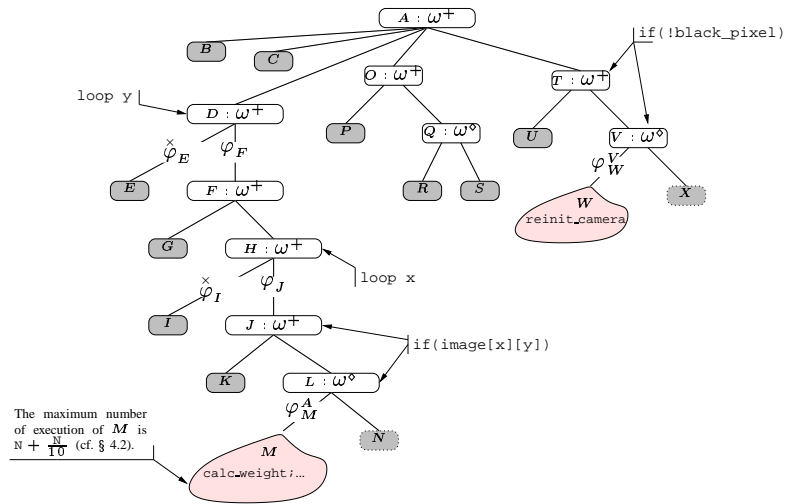
Figure 4. Source code and *scope-tree* of the camera example program

of scope M is $N+N/10$ (this information is provided as an annotation of the *if* construct). The variable N, which is declared by the root scope of the scope-tree(A), will remain in the final WCET expression. Therefore:

$$\mathcal{D}_A = \mathcal{D}_{\widehat{A}} \cup \{N\}$$
$$\varphi_M^A(\alpha) = \left(N + \frac{N}{10}\right)\alpha$$
$$\mathcal{U}_M = \bigcup_i \mathcal{U}_{M_i} \cup \{N\}$$

4.3 Non-rectangular loops

In the case of nested loops, it may occur that the maximum number of iterations of an inner loop depends on the iteration number of an outer loop (*i.e., non-rectangular loops*). In such situation, we would like to express the WCET of the inner loop as a function of the loop counter of one (or several) outer loop.

The nested loops (loop x and loop y) that enumerate all the (x, y) pixels of the image that have to be analysed are non-rectangular. The number of iterations of the loop x depends on the value of y. In the scope-tree shown in figure 4, non-rectangular loops are described using the execution frequency expression φ_F and φ_J. The y variable is declared in scope D (its value is fixed by φ_F) and used in φ_J and φ_I. All the values of the variable y (0..199) are enumerated by the summation used in φ_F.

$$\mathcal{D}_D = \mathcal{D}_{\widehat{D}} \cup \{y\}$$
$$\varphi_F(\alpha) = \sum_{y=0}^{199} \alpha$$
$$\varphi_J(\alpha) = \left(640 - 2\lfloor\frac{y}{2}\rfloor\right)\alpha$$
$$\varphi_I(\alpha) = \varphi_J(\alpha) + \alpha$$

The WCET of scope H is a function of y, but the WCET of D is not because y has been valuated by the sum used to express φ_F.

$$\omega_D = 200\omega_E^{\bullet} + \sum_{y=0}^{199}\left(\omega_G^{\bullet} + (641 - 2\lfloor\frac{y}{2}\rfloor)\omega_I^{\bullet}\right)$$
$$+ (640 - 2\lfloor\frac{y}{2}\rfloor)\omega_J)$$
$$= 108200\omega_J + 200(\omega_E^{\bullet} + \omega_G^{\bullet}) + 108400\omega_I^{\bullet}$$

4.4 Down-sampling

By "down-sampling" we mean executing one part of the body of a loop less often than the rest of the body. For instance, in the following code, the execution frequency of B is smaller than the execution frequency of A and C. Thus, B is down-sampled.

```
for(i=0;i<100;i++) {
   A; if(modulo(i,2)==0) B; C;
}
```

We want to represent the fact that we know that the maximum number of execution of the section of code 12-17 is $(N + N/10)$ for each execution of scope A. This situation is represented in the scope-tree by stating that the execution frequency of M (its φ expression) is relative to the scope A (and not to L, its direct parent).

$$\varphi_M^A(\alpha) = (N + N/10) \times \alpha$$

The down-sampling of this section of code is taken into account by moving the scope M so that it becomes a son of A. This is represented by the α to β arrow in figure 5. On this example, the scope M has been *shadowed* and the *shadow* of M (called M') is added as a son of A. The new link from A to M' $\varphi_{M'}^A$ is a copy of φ_M^A and the
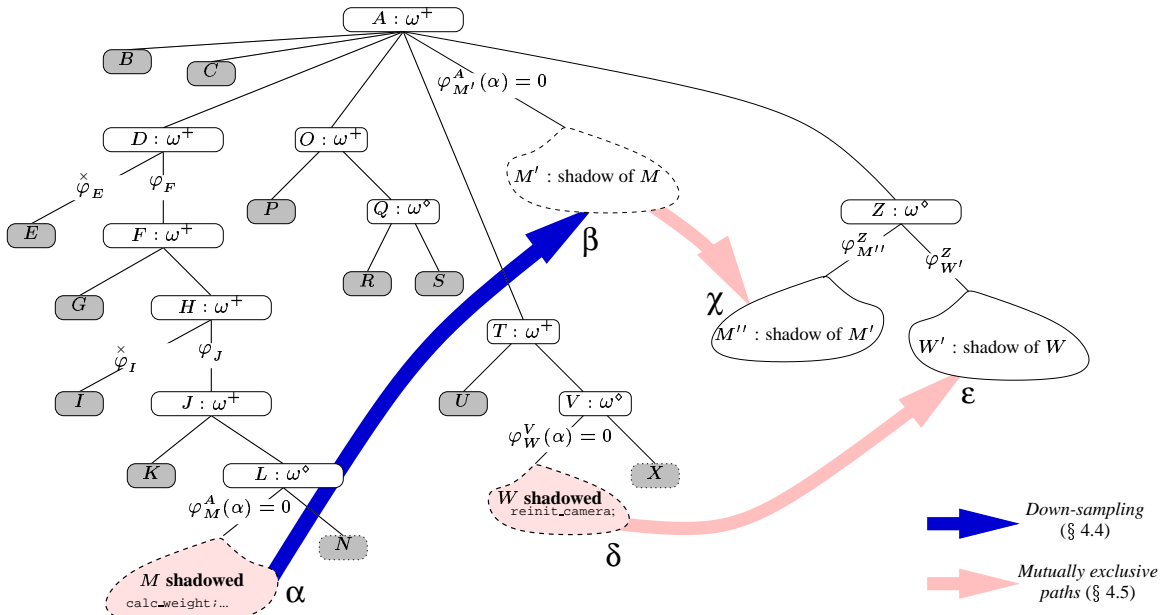
Figure 5. Restructured *scope-tree* of the camera example program

original link between L and M as been removed by setting $\varphi_M^A(\alpha) = 0$.

4.5 Mutually exclusive paths

In addition to the definition of properties among scopes in a hierarchy (*i.e.*, defining the behaviour of a scope based on the information of one or more of its outer scopes), there exist cases where particular assertions have to be made on scopes which are not linked by a parent-son relation. For instance, we would like to express the fact that two "brother" scopes are mutually exclusive, *i.e.*, that the execution of one scope forbids the execution of the other one and *vice-versa*.

Considering two mutually exclusive paths within a loop. It would be possible to specify the number of times each path is executed as presented in the previous paragraph. But the scope motion offers a better solution which does not require to specify precisely the number of executions of each path.

In our example, the reinitialisation of the camera has to be conducted only if no black pixel has been encountered during the image processing. Which means that the execution of the section of code 12-17 (scope M) excludes the execution of line 31 (scope W) and *vice-versa*. accurately. The relation between these two scopes can only been expressed at the level of their common parent, which is scope A. To represent the mutual exclusion, the scope-tree is restructured as follows:

- A new scope, Z, is defined as a son of A. Its WCET is the maximum of the WCET of its sons (using the default ω and φ expressions: $\omega_Z = \omega^\diamond$ and $\varphi_Z = \overline{\overline{\varphi}}$).
- The scope M' is *shadowed* and its *shadow* (called M'')

is set to be a son of Z (see the β to χ arrow on the figure). This is done by defining a new link $\varphi_{M''}^Z$ and removing the old one ($\varphi_{M'}^A(\alpha) = 0$).

- The *shadow* of W (called W') is added as a son of Z (δ to ϵ arrow). The original W scope is *shadowed*, a new link is built ($\varphi_{W'}^Z$) and the original link is removed

Note that there is still only one occurrence of scopes M and W in the scope-tree. M' and W'' are only pointers to shadowed scopes, and M'' is a pointer to pointer of shadowed scope.

In the end, the WCET of Z (which is a part of the WCET of A) is:

$$
\begin{aligned}
\varphi_{M''}^Z(\alpha) &= (\text{N} + \frac{\text{N}}{10})\alpha \\
\varphi_{W'}^Z(\alpha) &= \alpha \\
\omega_Z &= max(\varphi_{M''}^Z(\omega_M), \varphi_{W'}^Z(\omega_W)) \\
&= max((\text{N} + \frac{\text{N}}{10})\omega_M, \omega_W)
\end{aligned}
$$

Note that to express the mutual exclusion of two scopes, they have to be moved to become children of their common parent. This may be impossible if a scope uses some variables which are not defined in the common parent.

More complex relationship ranging from non-reflexive exclusive paths to forced paths and relations involving more than two conditional constructs could also be taken into account by defining the according tree modification rules.

4.6 WCET evaluation

The bottom-up traversal of the modified scope-tree provides an equational system in which some variables (the one

defined by the root scope) may remain in the final solution. The equational system obtained from the scope-tree of figure 5 is represented in table 2. The φ expressions that are equal to $\bar{\bar{\varphi}}$ have been omitted to ease the reading of WCET expressions. The corresponding maple code is shown in appendix A.

ω	φ
$\omega_A = \omega_B^\bullet + \omega_C^\bullet + \omega_D + \omega_O$ $+\, \omega_T + \varphi_{M'}^A(\omega_M) + \omega_Z$	$\varphi_{M'}^A(\alpha) = 0$
$\omega_D = \overset{\times}{\varphi}_E(\omega_E^\bullet) + \varphi_F(\omega_F)$	$\overset{\times}{\varphi}_E(\alpha) = 200\alpha$ $\varphi_F(\alpha) = \sum_{y=0}^{199} \alpha$
$\omega_F = \omega_G^\bullet + \omega_H$ $\omega_H = \overset{\times}{\varphi}_I(\omega_I^\bullet) + \varphi_J(\omega_J)$	$\overset{\times}{\varphi}_I(\alpha) = (y+1)\alpha$ $\varphi_J(\alpha) = y\alpha$
$\omega_J = \omega_K^\bullet + \omega_L$ $\omega_L = max(\varphi_M^A(\omega_M), \omega_N)$ $\omega_N = 0$	$\varphi_M^A(\alpha) = 0$
$\omega_O = \omega_P^\bullet + \omega_Q$ $\omega_Q = max(\omega_R^\bullet, \omega_S^\bullet)$	
$\omega_T = \omega_U^\bullet + \omega_V$	
$\omega_V = max(\varphi_W^V(\omega_W^\bullet), \omega_X^\bullet)$	$\varphi_W^V(\alpha) = 0$
$\omega_Z = max(\varphi_{M''}^Z(\omega_M), \varphi_{W'}^Z(\omega_W))$	$\varphi_{M''}^Z(\alpha) = (\text{N} + \frac{\text{N}}{10})\alpha$ $\varphi_{W'}^Z(\alpha) = \alpha$

Table 2. WCET of the camera application

As said in the previous paragraph, scope M and W have been moved from their original positions and new φ expressions ($\varphi_{M'}^A$, $\varphi_{M''}^Z$ and $\varphi_{W'}^Z$) have been defined and the original frequency expressions (φ_M^A and φ_W^V) have been set to 0.

We now detail the computation of the WCET of scopes D (the WCET of scope Z has been expressed in the previous paragraph).

$$\omega_N = 0$$
$$\omega_L = max(0,0) = 0$$
$$\omega_J = \omega_K^\bullet + 0$$
$$\omega_H = (641 - 2\lfloor\tfrac{y}{2}\rfloor)\omega_I^\bullet + (640 - 2\lfloor\tfrac{y}{2}\rfloor)\omega_K^\bullet$$
$$\omega_F = (641 - 2\lfloor\tfrac{y}{2}\rfloor)\omega_I^\bullet + (640 - 2\lfloor\tfrac{y}{2}\rfloor)\omega_K^\bullet + \omega_G^\bullet$$
$$\omega_D = 200\omega_E^\bullet + \sum_{y=0}^{199}\Big((640 - 2\lfloor\tfrac{y}{2}\rfloor)\omega_I^\bullet$$
$$+ (640 - 2\lfloor\tfrac{y}{2}\rfloor)\omega_K^\bullet + \omega_G^\bullet\Big)$$

Finally, knowing the WCET of each basic block of the program, we can express the WCET of the program , $\bar{\bar{\varphi}}(\omega_a)$, using the variable N ($a$, b and c are some constants).

$$a + max\big((\text{N} + \frac{\text{N}}{10})b, c\big)$$

5 Conclusion and future work

The new program representation (scope-tree) that has been presented in this paper aims at representing both the analysed code and extra-information (data dependent number of iterations, non-rectangular loops, ...) that may be provided by the user or automatically extracted from the source code. The most important aspects of scoped-based WCET analysis is that it provides a way to describe relationships between scopes which are not parents (either directly or indirectly), and that these relationships can be defined using variables. By representing the WCET as a symbolic expression parametrised by some variables, the WCET can be tightened by exploiting the knowledge of the specific execution context of the analysed code.

The scoped-based WCET analysis can be seen as a tree transformation approach in which some nested scopes may be moved up in the tree. The bottom-up traversal of the modified scope-tree provides a symbolic expression of the WCET of the analysed code, *i.e.*, an expression parametrised by variables defined in the root of the scope-tree.

We are currently working on the adaptation of this approach so that the assumption we made on the WCET of basic blocks (*i.e.*, constants and independents) can be relaxed. This will allow us to take into account some hardware features such as caches, branch prediction and pipelines that introduce some variability into the WCET of basic blocks and make them depend on the previously executed basic blocks (*i.e.*, the execution path). The best way to provide the necessary information as annotations in the source code is also investigated. The annotation system should allow the user to define scopes, scope variables, and scope expressions. But it could also be used to specify new scope-tree modification rules.

References

[1] G. Bernat and A. Burns. An approach to symbolic worst-case execution time analysis. In *25th workshop on real-time programming*, Palma, Spain, May 2000.

[2] J. Blieberger. Discrete loops and worst case performance. *Computer Languages*, 20(3):193–212, 1994.

[3] B. W. Char, K. O. Geddes, and G. H. Gonnet. *MAPLE V language reference manual*. Springer-Verlag, 1991.

[4] A. Colin and I. Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems*, 18(2-3):249–274, May 2000.

[5] A. Colin and I. Puaut. A modular and retargetable framework for tree-based wcet analysis. In *Proc. of the 13th Euromicro Conference on Real-Time Systems*, pages 37–44, Delft, The Netherlands, June 2001.

[6] A. Colin and I. Puaut. Worst-case execution time analysis of the RTEMS real-time operating system. In *Proc. of the 13th*

Euromicro Conference on Real-Time Systems, pages 191–198, Delft, The Netherlands, June 2001.

[7] J. Engblom and A. Ermedahl. Modeling complex flows for worst-case execution time analysis. In *Proceedings of the 21th IEEE Real-Time Systems Symposium (RTSS00)*, Orlando, Florida, Dec. 2000.

[8] C. Healy, M. Sjodin, V. Rustagi, and D. Whalley. Bounding loop iterations for timing analysis. In *Fourth IEEE Real-Time Technology and Applications Symposium*, pages 12–21, June 1998.

[9] C. Healy, R. van Engelen, and D. B. Whalley. A general approach for tight timing predictions of non-rectangular loops. *WIP Proceedings of the 1999 Real-Time technology and Applications Symposium*, pages 11–14, June 1999.

[10] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In R. Gerber and T. Marlowe, editors, *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, volume 30 of *ACM SIGPLAN Notices*, pages 88–98, New York, NY, USA, Nov. 1995. ACM Press.

[11] S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, S.-M. Moon, and C.-S. Kim. An accurate worst case timing analysis for RISC processors. In *Proceedings of the 15th IEEE Real-Time Systems Symposium (RTSS94)*, pages 97–108, Dec. 1994.

[12] G. Ottosson and M. Sjödin. Worst-case execution time analysis for modern hardware architectures. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools Support for Real-Time Systems (LCTRTS'97)*, June 1997.

[13] C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5(1):31–62, 1993.

[14] C. Y. Park and A. C. Shaw. Experiments with a program timing tool based on source-level timing schema. *IEEE Computer*, 24(5):48–57, May 1991.

[15] P. Puschner and A. Burns. A review of worst-case execution-time analysis. *Real-Time Systems*, 18(2-3):115–128, May 2000. Guest Editorial.

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

[17] P. Puschner and A. V. Schedl. Computing maximum task execution times – a graph based approach. In *Proc. of IEEE Real-Time Systems Symposium*, volume 13, pages 67–91. Kluwer Academic Publishers, 1997.

[18] E. Vivancos, C. Healy, F. Mueller, and D. Whalley. Parametric timing analysis. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, June 2001.

[19] S. Wolfram. *Mathematica: A System for Doing Mathematics by Computer*. Addison-Wesley, Reading, MA, USA, 1988.

A WCET calculation

Notations

$$\omega_A \dots \omega_Z \to \text{Wa} \dots \text{Wz}$$

$\varphi_{M'}^A \to \text{Fm1a}$ $\overset{\times}{\varphi}_E \to \text{Fe}$ $\varphi_F \to \text{Ff}$

$\overset{\times}{\varphi}_I \to \text{Fi}$ $\varphi_J \to \text{Fj}$ $\varphi_M^A \to \text{Fma}$

$\varphi_W^V \to \text{Fwv}$ $\varphi_{W'}^Z \to \text{Fw1z}$ $\varphi_{M''}^Z \to \text{Fm11z}$

Numeric WCET

`Wb`, `Wc`, `We`, `Wg`, `Wi`, `Wk`, `Wm`, `Wn`, `Wp`, `Wr`, `Ws`, `Wu`, `Ww` and `Wx` represent numeric values.

Maple code and results

```
Wa := Wb + Wc + Wd(N) + Wo + Wt + Fm1a(Wm) +
Wz(N):
Fm1a := (a) -> 0;

Wd := (N) -> Fe(N,We) + Ff(N,Wf(N,y)):
Wf := (N,y) -> Wg + Wh(N,y):
Wh := (N,y) -> Fi(N,y,Wi) + Fj(N,y,Wj(N,y)):
Wj := (N,y) -> Wk + Wl(N,y):
Wl := (N,y) -> max(Fma(Wm),Wn):
Wn := 0:

Fe := (N,a) -> Ff(N,a) + a:
Ff := (N,a) -> sum(a,y=0..199):
Fi := (N,y,a) -> Fj(N,y,a) + a:
Fj := (N,y,a) -> (640 - 2*floor(y/2))*a:
Fma := (N,y,a) -> 0:

Wo := Wp + Wq:
Wq := max(Wr,Ws):

Wt := Wu + Wv:
Wv := max( Fwv(Ww) , Wx ):
Wx := 0;

Fwv := (a) -> 0:

Wz := (N) -> max( Fm11z(N,Wm) , Fw1z(Ww)):

Fw1z := (a) -> a:
Fm11z := (N,a) -> (N+N/10)*a:

Wd(N);

=>  201 We + 200 Wg + 108400 Wi + 108200 Wk}

Wz(N);

=>          11
=>  max(Ww, -- N Wm)
=>          10

Wa;

=>  Wb + Wc + 201 We + 200 Wg + 108400 Wi + 108200 Wk
=>
=>                                          11
=>  + Wp + max(Wr, Ws) + Wu + max(Ww, -- N Wm)
=>                                          10
```