Static Determination of Probabilistic Execution Times

Laurent DAVID[(1)] - Isabelle PUAUT[(2)]

[(1)] INRIA - IRISA [(2)] INSA - IRISA
Campus Universitaire de Beaulieu
Avenue du Général Leclerc
35042 RENNES Cedex - France
E-mail: {Laurent.David|Isabelle.Puaut}@irisa.fr

Abstract

Most previous research done in probabilistic schedulability analysis assumes a known distribution of execution times for each task of a real-time application. This is however not trivial to determine it with a high level of confidence. Methods based on measurements are often biased since not in general exhaustive on all the possible execution paths, whereas methods based on static analysis are mostly Worst-Case Execution Time – WCET – oriented. Using static analysis, this work proposes a method to obtain probabilistic distributions of execution times. It assumes that the given real time application is divided into multiple tasks, whose source code is known. Ignoring in this paper hardware considerations and based only on the source code of the tasks, the proposed technique allows designers to associate to any execution path an execution time and a probability to go through this path. A source code example is presented to illustrate the method.

Keywords: *Probabilistic execution times, Static analysis, Soft real-time systems.*

1. Introduction

Real-time systems are widely used in many industrial sectors. They address not only safety-critical applications like nuclear power control or flight management systems, but also multimedia applications like video on demand, video conference, virtual reality and so forth. The first kind of applications, for which all the timing constraints are to be met, is known as hard real-time system. For the second kind of applications, known as soft real-time systems, a temporal fault would just result in a mere decrease of the quality of service (QoS).

Facing the concurrent access to few resources like processors, real-time applications are usually divided into multiple tasks and require during their development stage, a schedulability analysis [5, 14]. This analysis consists in computing – on-line or off-line – the ordering of tasks to be executed on the processor(s) such that all the timing constraints, or a reasonable part of them, are met during runtime. In a hard real-time context, the knowledge of the WCETs of each task is a key point to produce a correct task ordering, and many results on WCET determination have been produced [18]. To match the growing complexity of processor architecture, the WCET estimation methods are getting more and more complex as they take into account instruction and data caches [17, 15], pipelines [11] and branch prediction [7]. While these methods are necessary for hard real-time systems, they are less justified for soft real-time systems, where methods based on probabilistic schedulability analysis are more and more studied. In this latter context, many contributions for probabilistic schedulability analysis [13, 9, 16, 4] assume that the execution times of the tasks are probability distributions.

Using statistical methods, the authors of [12] have proposed an estimation of execution times. However, such techniques can not provide a reliable probabilistic execution time, since they cannot be in general exhaustive in terms of execution paths. To our knowledge, there are no other significant studies only examining the distribution of execution times. Most of the work done is indeed either on WCET or on mean execution time estimation. Leaving apart the deterministic estimation of WCET [18], a recent work [3] from Bernat *et al.* has proposed an estimation of WCET, using probabilistic schemes, and can even be useful for the hard real-time systems where a probability closer enough to 1.0 could be suitable for particular applications. In their theory, the different execution paths are still deterministically evaluated, only the low-level WCETs are given in probabilistic terms. Based on random samples of measured execution times, Edgar *et al.* have also proposed an approach [10] to statistically estimate the WCET with a confidence level. Using these WCET estimates, they have besides dis-

Proceedings of the 12th 16th Euromicro Conference on Real-Time Systems (ECRTS'04)
1068-3070/04 $20.00 © 2004 **IEEE**

cussed on the schedulability analysis of tasks. In their work, WCET estimations come from computation time distribution based on measurements. On the contrary in our work, probabilistic considerations are only applied to the source code and its different execution paths. Note that the theory on average case complexity [2] can also give an answer in estimating average execution times of many algorithms, but once again, it does not deal with execution time distributions.

This paper presents a framework to compute for any possible execution path of the source code, on the one hand the probability to go through this path during runtime and on the other hand, the corresponding execution time. The approach is based on source code analysis. Compared to statistical methods, this static analysis provides a high level of confidence in the proposed method since all the execution paths are considered.

This paper is organized as follows. First, notations and definitions are presented. Secondly, considering straight-line code and then code with loop structures, we present the paper contributions. After a section with an illustrative source code example, we deal with some implementation considerations. Conclusions and perspectives are finally given.

2. Definitions and notations

Real-time applications often communicate with an environment which may use some multimedia or physical data, depending on the application field. These data considered as external data for the real-time system may have a probabilistic representation. In this work, we suppose that each external variable of a task goes along with a probabilistic representation of its values. No assumption is drawn on the availability of the variable in terms of time. These external variables are assumed to be random variables and are denoted E_i with i naturals ($i \in 1, 2, ..., p$). In the source code of the task, some local or global internal variables may depend on the above mentioned random variables. These local or global variables are then also random variables, and let I_i with i naturals ($i \in 1, 2, ..., q$) denote these internal variables.

Any random variable (external or internal one) goes along with a probability mass function (*pmf*), denoted in the following by f. For example, $f_{E_i}(x)$ is the probability that the random variable E_i take the value x. This probability is also noted $P(E_i = x)$ ($P(E_i = x) = f_{E_i}(x)$). The same notations apply to I_i variables. The cumulative function of any random variable E_i is by definition:

$$F(x) = P(E_i \leq x) = \sum_{\substack{j \in Dom(E_i) \\ j \leq x}} P(E_i = j).$$

where $Dom(E)$ denotes the domain of E. We also assume that the domain of E_i or I_i are described only with discrete values. The *pmf* of external variables E_i are supposed to be known whereas those of internal variables I_i are not. They can however be determined if necessary and further details will be given in the next section.

The determination of the task execution time distribution turns out to be a quite complex issue, and one hypothesis we draw to propose a first solution is the independence of external variables E_i.

The proposed method applies to a single task and uses a static analysis of its source code. Furthermore, any source code can be considered as an assembly of conditional, sequence and loop constructs. The following grammar, written in BNF notation – Backus Naur Form –, is the grammar used in this article to describe any source code[1]:

$$\begin{aligned} S \quad ::= \quad & BB; \mid \\ & If\ (Cond)\ then\ S\ else\ S\ endif; \mid \\ & while\ (Cond)\ loop\ S\ endloop; \mid \\ & S\ \{S\} \end{aligned}$$

where BB, $Cond$, S and $\{S\}$ represent respectively a basic block (branch-free sequence of instructions), a condition (boolean expression), a construct, and a sequence of constructs of any type. In the following, a logical condition may also be represented by this kind of notations: C_i, C_{if}, C_{else}.

The proposed method does not take into account hardware considerations yet; the execution time of any instruction is intentionally assumed to be constant and context-independent. Furthermore, we assume no optimization mechanism in the compilation process.

3. Probabilistic execution times

Given the source code of a task and for any execution path, our objective is to compute on the one hand the probability to go through this path and on the other hand, its associated execution time. Our approach is a tree-based method – introduced initially by [19]. We propose to associate to any execution path an execution time and a set of conditions to satisfy to go through the considered path. This association is obtained by applying recursively some formulae to the tree representation of the source code.

The conditions we deal with may be either deterministic or probabilistic. We subsequently assume that any condition can be viewed as a probabilistic condition, associated with a probability that can be equal to 0.0, 1.0 or any value included within 0.0 and 1.0.

[1] This grammar is intentionally simplified.

Construct (S).	Possible Execution Times for S. Notation: T(S).	Possible Conditions to run S. Notation: C(S).	Resulting sequence of basic blocks.
BB (basic block)	$T(BB)$	$C(BB) = True$	$r(BB;) = BB;$
$S_1; ...S_n;$	$\sum_{i=1}^{n} T(S_i)$	$\bigcap_{i=1}^{n} C(S_i)$	$r(S_1; ...; S_n;) = r(S_1); ...; r(S_n)$

if (C_{if}) then
$S_t;$
else
$S_e;$
end if;
	$T(C_{if}) + T(S_t)$
	
	$T(C_{if}) + T(S_e)$
		$C(S_t) \cap C_{if}$
		
		$C(S_e) \cap \neg C_{if}$
			$r(S) = r(S_t);$
			
			$r(S) = r(S_e);$

Table 1. Execution Times and Conditions

Regarding *loop* structures, execution time evaluation requires to determine the number of loop iterations. To gradually introduce our method, we first focus on straight-line code, and then take into account *loop* structures in the source code of a task.

3.1. Straight-line code

Two main steps have been identified in our method: a first step in which a tree-based technique provides for each possible execution path, its corresponding execution time and its corresponding logical conditions to satisfy to go through this path during runtime, and a second step in which conditions are probabilistically evaluated.

Tree-based method

Our tree-based method associates to any node of the tree-based representation of the source code an execution time and a logical condition. This association is reported in table 1 (column 2 and 3) through some recursive formulae, making possible to evaluate any node from its child nodes. In table 1:

- $T(S_i)$ represents the execution time for a construct S_i, and $T(C_{if})$ the execution time of C_{if} evaluation. As we leave apart hardware considerations, we assume that $T(BB)$ and $T(C_{if})$ are fixed and known.
- $C(S_i)$ represents the conditions to run S_i. While for a basic block BB, the related condition is equal to $True$ ($C(BB) = True$), for a sequence $S_1; S_2; ...; S_n;$ of constructs, the conditions to run the sequence is the (logical) intersection of conditions to run each S_i ($i \in 1...n$). In some particular cases, the conditions are trivial: for instance if each $S_i = BB_i$, then the condition results in $C(S_1; S_2; ...; S_n;) = \cap_i C(Si) = \cap_i True = True$[2]. Dealing with conditional constructs the condition $C(S)$ produces two branches resulting in two distinct conditions for the two possible execution paths: $C(S_{if}) \cap C_{if}$ and $C(S_{else}) \cap C_{\neg if}$.

The last column of the table will be useful in the following when dealing with the probabilistic evaluation of conditions. Let us just mention that this column allows us to build the resulting sequence of basic blocks associated with the considered execution path. It transforms the representation of the considered path from a tree into a flat representation (sequence of basic blocks).

Applying recursively the table formulae results in a set of different execution times, a set of condition logical intersections, and a set of basic block sequences. Every execution path in the source code is associated with these three features.

Probabilistic evaluation of conditions

Assuming that above recursive formulae are completed, let Ch_i be an execution path and let $\cap_{i=1}^{n} C(S_i)$ be the set of conditions to satisfy to go through Ch_i. The probabilistic evaluation of the execution time consists then in evaluating:

$$P\left(\bigcap_{i=1}^{n} C(S(i))\right)$$

Besides, any condition can be written as an expression of the following form $[C(S(i))] \equiv [f_i(E_1, E_2, ..., E_p, I_1, I_2, ..., I_q) \propto 0]$ where $\propto = \{<, >, =, \leq, \geq\}$, where E_j are the external variables of the source code, and where I_j are the internal variables of the source code. As the sets are supposed discrete, we have:

$$P\left(\bigcap_{i=1}^{n} C(S(i))\right) = \sum_{\mathcal{D}} P(E_1 = e_1, ..., E_p = i_p, I_1 = v_1, ..., I_q = v_q)$$

where $\mathcal{D}$ represents all the $(p + q)$ tuples $e_1, e_2, ..., e_p, i_1, i_2, ..., i_q$ belonging to the domain $Dom(E_1) \times ... \times Dom(E_p) \times Dom(I_1) \times ... \times Dom(I_q)$ such that $\bigcap_{i=1}^{n} f_i(E_1, E_2, ..., E_p, I_1, I_2, ..., I_Q) \propto 0$ and where the notation $Dom(X)$ represents the domain of random variable X.

[2] $\cap$ is used here as the logical AND.

Though complex, the summation can be automatized by using a full exploration of $\mathcal{D}$, and verifying for each element of $\mathcal{D}$ whether the latter expression is true. If so, the corresponding probability is used in the summation. What is more annoying is that we cannot write a mere product of individual probabilities because of the possible interdependance of I_i variables. Our method proposes then to replace every internal variable I_i with a quantity which only depends on the external variables E_i, assumed to be independent. For this purpose, we use the last column of table 1, where $r(S)$ (S is a construct of any type) represents the resulting flat representation of constructs. If S is a basic block BB for example, we trivially have: $r(S) = BB$. In a conditional construct however, two different flat representations may be obtained: one for the if branch ($r(S_t)$), the other for the $else$ branch ($r(S_e)$). Through a recursive application of the table formulae, this column provides then the resulting sequence of basic blocks of any execution path.

In this basic block sequence, specific to Ch_i, let us point out two facts:

- for any assignment concerning an internal variable I_i, it may report the knowledge of I_i to the knowledge (1) of another internal variable I_i, (2) of a combination of external variables E_i only, or (3) of a combination of internal and external variables,
- in the case I_i does not depend on external variables, the value of I_i can be computed for the considered execution path, using a partial execution of the basic block sequence.

To determine the relation giving I_i from the external variables, we propose algorithm 1. For a given execution path, this algorithm consists in going through the induced basic block sequence (in reverse execution order), and in keeping up to date the history of the assignments involving the internal variables. The algorithm operates as follows: for any execution path, say Ch_i, we assume that the associated logical conditions and their internal variables I_i are identified. These variables are gathered in a set $\mathcal{I}$. The related basic block sequence is examined in the reverse order (lines $1, 2, 10$ and 11). For each basic block we point to, we check whether this basic block contains an assignment involving an element of $\mathcal{I}$ (line 3). If so, we memorize the label of this element and the right member expression of the assignment in a variable called $Lvect$ (line 4). We then withdraw this element from $\mathcal{I}$ (line 5). If some new internal variables appear in this expression, we add them into $\mathcal{I}$ (lines 6 and 7). The storage (in $Lvect$) of labels and the right member expression of assignments, in others words, the "history" of initial I_i variables, then provides the expression of I_i function of external variables E_i.

The probabilistic evaluation of the conditions associated with Ch_i is then very close. As we now know the

Alg. 1 Determining I_i in terms of the external variables E_i.

Require: Ch_i is given, associated conditions are identified, and all the I_i included in these conditions are identified. We store all these I_i in a set $\mathcal{I}$

local Lvect {$Lvect$ is a dynamic list of 2-dimensional vectors: one dimension for a label, and one dimension for a regular expression}

1: We start at the end of the induced basic block sequence;
2: **while** (We are not at the beginning of the induced basic block sequence) **or** (There is still element in $\mathcal{I}$ to determine) **do**
3: **if** (The basic block we are pointing to contains an assignment involving one element – say I_i – of $\mathcal{I}$) **then**
4: We store in Lvect the label of I_i and the expression of the assignment right member;
5: We withdraw I_i from $\mathcal{I}$;
6: **if** (Some new internal variables appear in the expression of the assignment right member) **then**
7: We add these internal variables as some new elements of $\mathcal{I}$
8: **end if**
9: **end if**
10: We go to the next basic block (in reverse execution order);
11: **end while**
12: **if** There is still an element in $\mathcal{I}$ **then**
13: **RAISE** an exception;
14: **end if**

Ensure: For the given execution path, $Lvect$ provides the history of internal variables I_i, and allows us to set up the dependence related to E_i variables.

expression giving all the internal variables I_i according to the external variables E_i, we can rewrite the expression of f_i introduced in terms of the variables E_k solely. Let g_i be this modified function such that $[C(S(i))] \equiv [g_i(E_1, E_2, ..., E_p) \propto 0]$. We finally write:

$$P\left(\bigcap_{i=1}^{n} C(S(i))\right) = \sum_{\mathcal{D}} P(E_1 = e_1, ..., E_p = e_p)$$

where $\mathcal{D}$ represents all the p tuples $e_1, e_2, ..., e_p$ belonging to the domain $Dom(E_1) \times ... \times Dom(E_p)$ such that $\bigcap_{i=1}^{n} g_i(E_1, E_2, ..., E_p) \propto 0$. As we assume that external variables are mutually independent, we write:

$$P\left(\bigcap_{i=1}^{n} C(S(i))\right) = \sum_{\mathcal{D}} \prod_{j=1}^{j=p} P(E_j = e_j) \quad \textbf{(1)}$$

where all the E_j *pmf* are known. The probability that Ch_i will be executed during execution phase is then determined. Moreover the execution times have been determined using

Construct (S).	Possible Execution Times for S. Notation: T(S).	Possible Conditions to run S. Notation: C(S).	Resulting sequence of basic blocks.
For all the possible values of the number of *loop* iterations: $\{p_1, p_2, ..., p_n\}$ (n integer).			
	$\left[\sum_{k=1}^{p_1} T(C_w(k)) + T(S_w(k))\right] + T(C_w)$	$\left[\bigcap_{k=1}^{p_1} C(S_w(k))\right] \cap (N_{It} = p_1)$	$r(S_w(1)); ...; r(S_w(p_1));$
while $C_w(k)$	$\left[\sum_{k=1}^{p_2} T(C_w(k)) + T(S_w(k))\right] + T(C_w)$	$\left[\bigcap_{k=1}^{p_2} C(S_w(k))\right] \cap (N_{It} = p_2)$	$r(S_w(1)); ...; r(S_w(p_2));$
loop			
$S_w(k)$;	$\left[\sum_{k=1}^{...} T(C_w(k)) + T(S_w(k))\right] + T(C_w)$	$\left[\bigcap_{k=1}^{...} C(S_w(k))\right] \cap (N_{It} = p_{...})$	$r(S_w(1)); ...; r(S_w(...));$
end loop;			
	$\left[\sum_{k=1}^{p_n} T(C_w(k)) + T(Seq_w(k))\right] + T(C_w)$	$\left[\bigcap_{k=1}^{p_n} C(S_w(k))\right] \cap (N_{It} = p_n)$	$r(S_w(1)); ...; r(S_w(p_n));$
k is supposed to be the loop index ranging from 1 to the number of *loop* iterations.			

Table 2. Execution Times and Conditions

the formulae of table 1. Hence, we have the probabilistic execution time of Ch_i. For all the feasible paths, we finally get the probabilistic execution times of the source code.

3.2. Code with loop structures

In this subsection, we deal with source code that may contain loop structures. The main refinement of the previous section consists in taking into account code included in *loop* structures and executed several times.

In a *loop* structure, we can always take under consideration a *loop* index k that describes the iteration number. This number may be on the one hand deterministic (k boundaries can be deterministically evaluated as well as its step[3]), or on the other hand probabilistic (one of both boundaries of k may be random variables, and its step may be also probabilistic). As it is difficult to be exhaustive in the range of *loop* index k, we have chosen to restrict our work to a common *loop* condition case: $(I_i \leq k \leq I_j)$, with I_i, I_j two internal random variables and with a constant and unit step for the k range. Cases $(k \leq I_j)$ and $(k \geq I_I)$ are of course particular cases of this latter case.

The successive executions of the loop body imply that all the conditions within loops are evaluated several times. In table 2, we report the recursive formulae to be applied to the source code containing loop structures. In column 2 for example, not only conditions to run the structures of the loop body, but also a condition on the number of times the *loop* body is executed, are used.

Related to the number of loop iterations, the condition to be evaluated is of type $N_{It} = p$. Due to the particular choice of loop condition, it can be written as the evaluation of $(I_j \geq p) \cap (I_i \leq p)$. Then, we can write that $[N_{It} = p] \equiv [(f_1(I_j) \geq 0) \cap (f_2(I_i) \geq 0)]$, where f_1 is the function $x \mapsto x - p$ and f_2 is the function $x \mapsto -x + p$. To determine I_i and I_j function of external variables only, we apply algorithm 1. The same method as in the straight-line code context is then applied to get the probabilistic execution times of the source code.

[3]The difference of k values between two consecutive iterations.

4. A Source Code Example

In this section, a task source code example is scrutinized using the method presented above. Our example deals with the control of a physical experiment. This experiment requires a minimum of power generated by electromagnetic (or acoustic) waves. At each periodical execution of the task, a treatment is conducted to confirm that the physical experiment is still under nominal functioning. This kind of experiment may be used for example to keep a confined area under specific conditions, to test a mobile phone emission/reception or to test a high performance microphone. Let E_x, E_y and E_z be three independent power quantities resulting from the acquisition of three independent signals. Each external variable E_i has a *pmf* supposed to be known. The signals may correspond to three isotropic signals acquired in a three dimensional Cartesian space. Assuming that signal is not taken under consideration when its power is above a threshold, we suppose that this threshold is also a random variable and corresponds to the power of an electromagnetic (or an acoustic) noise. Let T be the random

variable associated with this threshold. The task execution time depends indirectly on the values of E_i and T, as illustrated in part of the task C source code of algorithm 2.

Alg. 2 Source code example.

TASKPOWERALERT := proc E[N], T, PowerMin

Require: N is a global integer variable, $E[N]$ is a table of N floats, T and $PowerMin$ are float. The distribution of E and of T are known and $PowerMin$ is given.

```
local  float P; int j; {Three dimension space.}
1: P=0.0;
2: for (j=0; j<N; j++;) do
3:     if (E[j]>T) then E[j]=T; end if; {Ignore signal power
       above threshold value.}
4:     P = P + E[j]; {Compute the total power.}
5: end for
6: if (P<PowerMin) then Treatment; end if; {The alert is
   treated through a specific code.}
```

Ensure: If the resulting power is under PowerMin then run Specific Code.

In this code, the conditions $(P < PowerMin)$ and $(E[j] > T)$ $(i = 1, 2, 3)$ result in $2^3 * 2 =$ different execution paths. Each path can be identified by the values of the 4 conditions. For instance, the condition to $-1-$ NOT go through the *then* branches within the *for loop* body and $-2-$ to go through the *then* branch of *if* structure, is $(\bigcap_{i=1}^{i=3} E[i] \leq T) \cap (P < PowerMin)$. The probability evaluation of this condition consists then in computing:

$$Pb = P(\bigcap_{i=1}^{i=3}(E[i] \leq T) \cap (P < PowerMin))$$

Besides, the resulting basic block sequence associated with this execution path and coming from the recursive application of formulae given in the last column of tables 1 and 2:

```
1: float P; int j;
2: P=0.0;
3: P = P + E[1];
4: P = P + E[2];
5: P = P + E[3];
6: Treatment;
```

Applying algorithm 1 results in the determination of P: the different assignments give the relation $P = 0.0 + E[1] + E[2] + E[3]$. We then write that $Pb =$

$$P(\bigcap_{i=1}^{i=3}(E[i] \leq T) \cap (E[1] + E[2] + E[3] < PowerMin))$$
$$= \sum_{\mathcal{D}} P(T = t) \times \prod_{i=1}^{i=3} P(E[i] = e_i)$$

where $\mathcal{D}$ represents all quadruples (e_1, e_2, e_3, t) that belong to the set $Dom(E_1) \times Dom(E_2) \times Dom(E_3) \times Dom(T)$ and such that $(e_1 \leq T) \wedge (e_2 \leq T) \wedge (e_3 \leq T) \wedge (e_1 + e_2 + e_3 < PowerMin)$. The probability associated with the considered path can then be computed. The same method is applied to the other execution paths.

As far as execution times are concerned, we apply the recursive equations given in table 1 and 2 to the source code. We then gather in a set the different execution paths having the same execution time. Therefore, as the possible execution path set is discrete, the summation of their associated probabilities results in the probability that the source code take this time to execute.

To experimentally confirm the correctness of our method, we have compared the probabilistic proposed approach with a stochastic simulation of the source code execution. We choose a normal distribution for $E[j]$ and T, and we analyze the execution times of the 16 different execution paths. For the different $E[j]$, the same normal distribution is chosen: a mean of $\mu = 2$ and a variance of $\sigma = 1$. For T, we choose a mean of $\mu = 4$ and a variance of $\sigma = 1$. The value for *PowerMin* is 8. Reporting the results in figure 1, we observe that for each computed execution times, the probabilistic analysis gives closely the same results than a stochastic simulation[4] of the source code execution. It confirms on the example the correctness of the proposed theory. Note that a statistic test has been conducted to confirm that our samples follow a normal distribution.

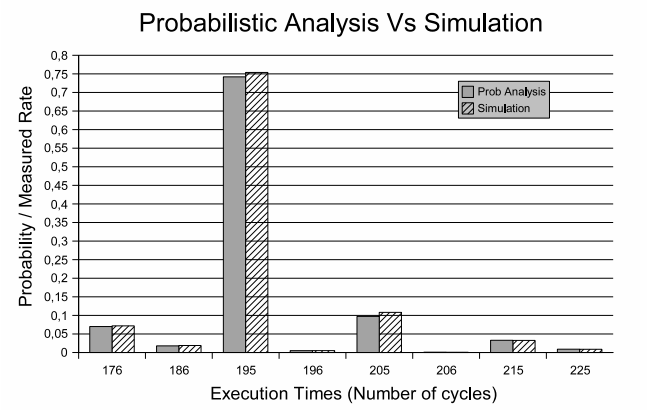

Figure 1. Probabilistic analysis vs. Simulation

[4]Note that the simulation operates on 10^6 different executions.

5. Implementation considerations

Complexity and automation

Considering complexity issues, two issues must be distinguished: one deals with the determination of execution times and their associated conditions, and the other one deals with the probabilistic evaluation of these conditions.

For the first part, at any stage of the execution time analysis, all the execution scenarios must be examined. The complexity is then greater than the complexity of tree-based static WCET analysis. At a given node of the tree, our tree-based technique produces either two leaves when an *if* structure is examined, or p leaves when a *loop* structure is examined with p representing the possible number of iterations. Same considerations also apply to the determination of the condition combination associated with each execution path.

As regards the probabilistic evaluation of conditions, the complexity is clearly exponential and strongly depends on:

1. the relation between the internal I_i and external random variables E_i,

2. the domain width of external variables (through the exploration of the whole domain $\mathcal{D}$).

Nevertheless these two parts can be automatized. An adaptation of WCET tools [8, 6] can first be achieved to implement the tree-based method part (execution times, and conditions). Then, implementation of probability evaluation consists in the implementation firstly of algorithm 1, and secondly of equation (1). Note that the complexity of this latter equation is based mainly upon the exploration of the domain $\mathcal{D}$.

Probabilistic evaluation: some particular cases

As underlined above, the probabilistic evaluation of conditions is exponential. However, its computation can be simplified in some particular – but rather interesting – cases. For example, conditions of type: $[f_i(E_1, E_2) \propto 0]$ with E_1 and E_2 two external variables and with $\propto = \{<, >, =, \leq, \geq\}$, can be rapidly evaluated using a probabilistic algebra. Provided that random variables E_1 and E_2 are independent, this algebra can be used after or before algorithm 1, making, in this latter case, this algorithm unecessary.

For example, probability evaluation involving arithmetic operators is nearly straightforward. The addition operator for instance uses the convolution product of the two *pmf* from the two discrete random variables E_1 and E_2. We can write indeed the following equation [1]: $P(E_1 + E_2 = k) = \sum_{j=min(E_1)}^{k-min(E_2)} P(E_1 = j) \times P(E_2 = k - j)$, where $min(X)$ represents the minimum value that random variable X can take. The evaluation of the probability of condition $E_1 + E_2 = k$ is then reported to the *pmf* of E_1 and E_2, supposed to be known. The same kind of results can be applied to $P(E_1 - E_2 = k)$, $P(E_1 \times E_2 = k)$ and $P(E_1/E_2 = k)$. The scheme used to write these probabilities function of E_1 and E_2 *pmf* lays on the condition partitioning mechanism.

Let for example scrutinize the case $P(E_1 \times E_2 = k)$, with k (integer) ranging from $min(E_1 \times E_2)$ to $max(E_1 \times E_2)$ (not necessarily equal to respectively $min(E_1) \times min(E_2)$ and $max(E_1) \times max(E_2)$ depending on E_1 and E_2 signs). One possible partition of event $E_1 \times E_2 = k$ is $\bigcup_{i=min(E_1)}^{max(E_1)} (E_1 = i) \cap (E_2 = k/i)$. Hence, we can write:

$$\begin{aligned} P(E_1 * E_2 = k) &= P\Big(\bigcup_{i=min(E_1)}^{max(E_1)} (E_1 = i) \cap (E_2 = k/i)\Big) \\ &= \sum_{i=min(E_1)}^{max(E_1)} P((E_1 = i) \cap (E_2 = k/i)) \end{aligned}$$

As E_1 and E_2 are independent, the product of probabilities can then be written, making possible to compute $P(E_1 * E_2 = k)$. If E_1 and/or E_2 take the value 0, special attention is required. The corresponding probability can for example be equal to 0.0 or 1.0 (if k is equal to 0.0, the probability is then equal to 1.0: it is certain that $0 = 0$!). Anyway, leaving apart these special cases, the above expression determines the probability $P(E_1 * E_2 = k)$ in terms of the *pmf* of E_1 and E_2.

With such basic operators, and provided that the condition of type $[f_i(E_1, E_2) \propto 0]$ uses arithmetic operators and that E_1 can be dissociated from E_2 (and *vice versa*), the probabilistic evaluation of the condition is reported to the knowledge of the respective *pmf* of E_1 and E_2 .

Let us just mention that any evaluation of probability $P(f_i(E_1, E_2) \propto 0)$ with $\propto \in \{<, >, \geq, \leq\}$ can also be reported to the evaluation of $P(f_i(E_1, E_2) = k))$ (notion of cumulative probability).

All these operators on random variables can then be used to compose a probabilistic algebra, and to provide a different way to compute the probability of a condition to be satisfied. It is of interest to mention it for a validation issue: indeed, it may give some ways to compute the probability for an execution path to be followed, and could then partially contribute to the validation of a tool implementing the proposed technique.

6. Conclusion

Using a static source code analysis, this paper shows that it is possible for a real-time application designer to determine the probabilistic distribution of task execution times.

The proposed technique requires that for each task of the system, its external random variables be mutually independent and that their respective probability distributions be known. A tree-based technique has been proposed in order to build for each possible execution path, its corresponding execution time, its corresponding logical conditions to satisfy to go through this path during runtime, and lastly its resulting basic block sequence. Once these three features are identified for one execution path, an algorithm is used to look for relations giving internal source code variables in terms of the external variables. Finally, these relations combined with the *pmf* of the external variables provide the probability to go through this execution path during runtime. Generalizing this process to all possible execution paths results in the probabilistic distribution of execution times.

The proposed method is about to be entirely implemented: the tree-based method part can indeed be slightly derived from WCET tools already implemented, and probabilistic evaluation, though exponential, reveals interesting characteristics to be rapidly automatized. Once implemented, we are going to check the feasibility of the method on concrete source code. It is as well of great interest to scrutinize the complexity issue in order to decrease the computational cost of our proposed method. We shall also focus on more intricate *loop* conditions to take under consideration more general source code. Moreover, as no hardware accelerating features is yet taken into account in this work, we shall also examine what would be appropriate to face, for examples, instruction and data caches, pipelines or branch prediction. Another point which requires further research is the assumption of independence between external random variables. While for particular applications, this assumption is suitable, it is more controversial in general. Jointly distributed random variables could however be useful to carry on computing the probabilities.

References

[1] A. O. Allen. *Probability, Stastitics, and Queuing Theory*. Academic Press, 1978.

[2] S. Ben-David, B. Chor, and O. Goldreich. On the theory of average case complexity. In *Journal of Computer and System Sciences*, volume 44, pages 204–216, 1989.

[3] G. Bernat, A. Colin, and S. Petters. Wcet analysis of probabilistic hard real-time systems. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS02)*, pages 279–288, Austin, Texas, USA, Dec. 2002.

[4] A. Burns, G. Bernat, and I. Broster. A probabilistic framework for schedulability analysis. In *Proc. of the 3rd international workshop on embedded software (EMSOFT 2003)*, volume 2855 of *Lecture Notes in Computer Sciences*, pages 1–15, Philadelphia, PA, USA, 2003. Springer-Verlag.

[5] G. Buttazzo. *Hard real-time computing systems*, chapter Periodic task scheduling, pages 77–108. Kluwer Academic Publishers, 1997.

[6] A. Colin and G. Bernat. Scope-tree: a program representation for symbolic worst-case execution time analysis. In *Proc. of the 14th Euromicro Conference on Real-Time Systems*, pages 50–59, Vienna, Austria, June 2002.

[7] 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.

[8] 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.

[9] J. L. Diaz, D. F. Garcia, K. Kim, C. Lee, L. L. Bello, J. M. Lopez, S. L. Min, and O. Mirabella. Stochastic analysis of periodic real-time systems. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS02)*, pages 289–300, Austin, USA, Dec. 2002.

[10] S. Edgar and A. Burns. Statistical analysis of WCET for scheduling. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS01)*, pages 215–224, London, UK, Dec. 2001.

[11] C. A. Healy, D. B. Whalley, and M. G. Harmon. Integrating the timing analysis of pipelining and instruction caching. In *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS96)*, pages 288–297, Dec. 1996.

[12] M. Iverson, F. Ozguner, and L. Potter. Statistical prediction of task execution times through analytic benchmarking for scheduling in a heterogeneous environment. *IEEE Transactions on Computers*, 48(12):1374–1379, Dec. 1999.

[13] A. Leulseged and N. Nissanke. Probabilistic analysis of multi-processor scheduling of tasks with uncertain parameters. In *The 9th International Conference on Real-Time Embedded Computing Systems and Applications.*, Taiwan, R.O.C. 2003.

[14] J. W. Liu. *Real-Time Systems*. Prentice Hall, Apr 2000.

[15] T. Lundqvist and P. Stenstrom. An integrated path and timing analysis method based on cycle-level symbolic execution. *Real-Time Systems*, 17(2-3):183–207, Nov. 1999.

[16] S. Manolache, P. Eles, and Z. Peng. Memory and time efficient schedulability analysis of task sets with stochastic execution times. In *Proc. of the 13th Euromicro Conference on Real-Time Systems*, pages 19–26, Delft, The Netherlands, June 2001.

[17] F. Mueller. Timing analysis for instruction caches. *Real-Time Systems*, 18(2):217–247, May 2000.

[18] 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.

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