AN APPROACH TO SYMBOLIC WORST-CASE EXECUTION TIME ANALYSIS

Guillem Bernat Alan Burns

Real-Time Systems Research Group
University of York
YO1 5DD - York (England)
{bernat,burns}@cs.york.ac.uk

Abstract: Using the constant value for the worst case execution time for all calls to a subprogram is safe but pessimistic. We address the problem of tailoring the WCET of each call to a subprogram by building the WCET as an algebraic expression parameterised with some of the subprogram parameters. The WCET of each call may then be constrained to particular ranges of values that lead to tighter WCET. The technique relies on the power of computational algebra systems to perform sophisticated simplifications and evaluation of the intermediate expressions. The same approach also allows to obtain tighter estimations of nested loop iterations which may also be tailored for different execution calls. This approach requires the description of the non-functional behaviour of the code with annotations included as specially formatted comments.

Keywords: Real-time, timing-analysis.

1. INTRODUCTION

Worst case execution time analysis computes upper bounds on the execution time of tasks in a system. As the general problem of determining the WCET of an arbitrary piece of code is undecidable (it is very similar to the halting problem) one can not expect to obtain the exact value, instead, only an upper bound can be provided. However, it is desirable for the WCET to be as tight as possible.

WCET analysis is usually done at two levels (Puschner and Burns 2000). The low-level, which is done at the object code, considers the effects of hardware level features (like cache and pipelining) (Healy *et al.* 1999*a*). On the other hand, high level analysis is performed at the source code and it focuses on characterising possible execution paths. It usually relies on annotations provided by the user to describe execution frequencies (like maximum number of loop iterations) (Puschner and Koza 1989), (Puschner and Schedl 1991).

We address the problem of obtaining high level tight estimations of the WCET by characterising the context in which the code is executed. With a single technique based on symbolic expressions of the WCET we address two sources of overestimation of high level analysis: subprogram calls and number of loop iterations.

Traditional techniques compute the WCET of a subprogram and then use this value in all the calls to it. Thus not exploiting the fact that a particular call may have a much tighter WCET than others. In the same way, a maximum loop bound is used to obtain the worst case execution time of a loop, ignoring the fact that this loop may be executed by less than this maximum number in different contexts. Consider for example the pow(f,n) function that computes f^n with a loop. The worst case execution time depends on the exponent n which dictates the number of iterations of the loop. An upper bound on the maximum number of iterations of the loop can be easily provided

[1] This work has been funded by the DERA with project CSM/1254 and with a grant from the *Ministerio de Educacion y Cultura Español*

with an annotation or by examining the code with, for instance, range analysis and determining that the maximum exponent is, for instance, 10. However, a call to Pow(Pi,3) has a shorter WCET that the one that considers that all calls loop for 10 times.

The main difference between symbolic WCET and other WCET techniques is that it represents the WCET of a piece of code as an algebraic expression instead of a single number. For example, the WCET of the pow function could be specified as something like $1474 + 434n$ where n is the exponent. This expression is left unevaluated until the actual value of n is known in a particular call to the function, or substituted by the maximum for all n if the parameter is not known. The same mechanism can be used to describe the number of iterations of loops (and nested loops). The number of iterations of non-rectangular loops can be expressed as nested summations.

We rely on the power of computational algebra systems like Mathematica or Maple to manipulate, simplify and evaluate these expressions.

Note that previous WCET is focused on local analysis and barely considers the context in which a piece of code is executed. The technique presented here is holistic in the sense that it specifically addresses dependencies among code sections.

Note that this technique complements low-level WCETA and therefore can use the accurate estimates of the effects of pipelines and caches.

We present the idea of this work around an example that illustrates two important points: the mapping between program constructs and algebraic expressions and the evaluation of such expressions in the context in which they appear. For this purpose the rest of the paper is structured as follows: the next section outlines the related work in WCET. We then present an example code from which we show that it has a known WCET but for which current techniques are very pessimistic. We then expose our framework and analysis technique. We finally apply our technique to a code example.

2. RELATED WORK

A good review of WCET can be found in (Puschner and Burns 2000). Also, recent work on abstract interpretation can be found in (Ermedahl and Guftafsson 1997) and symbolic execution in (Lundqvist and Stenstrom 1998)(Liu and Gomez 1998). The main drawback of these techniques is that they simulate all paths of a loop for every loop iteration. Other approaches for dealing with so-called non-rectangular loops (Healy *et al.* 1998), (Healy *et al.* 1999*b*), originally in the context of paralelising compilers (Sakellariou 1997) was 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 single number.

There are few exceptions like the Spats approach in Spark Ada of (Chapman *et al.* 1996) in which modes of a subprogram can be defined, for which particular constraints apply. However, this is based on an a-priori enumeration of all possible modes.

The technique presented here does not intend to replace any of these, it is in fact a complementary technique to obtain tighter estimates by evaluating loop bounds expressions and by considering the context in which this code is executed.

3. CODE EXAMPLE

Consider the code example in figure 1 which describes a simple version of the pow function adapted from (Chapman *et al.* 1996) that computes the n-th power of a float number f (if n is negative, then the result is $1/f^{\mathsf{abs}(n)}$). The numbers in parenthesis are the worst case execution time of each particular basic block. Using, for example, range analysis it is determined that the range of the exponent is $[-10, 10]$ and therefore the maximum number of iterations of the loop is 10. A simple approach to the analysis of the WCET of this function by adding the execution times of all sections and looking for the longest path in conditional branches results in:

$$W_{\mathsf{Pow}} = 50 + 111 + max(301, 583) + 270 + 117 +$$
$$+10(117 + 317) + 99 + max(560, 0) + 244 = 6374$$

Note that the WCET holds a dead path as the branches of the two ifs with the maximum WCET may never be taken together for a single call. Also the WCET depends on the number of iterations of the loop, however the maximum value, 10, is used.

The idea developed in this paper is to formulate the WCET as an algebraic expression. In this case, we could formulate the WCET of the function as a function of the exponent e. The WCET can be rewritten as: [2] .

$$\begin{aligned} W_{\mathsf{Pow}}(e) = 50 + 111 + [e < 0]301 + [e >= 0]583 + \\ +270 + 117 + \sum_{i=1}^{abs(e)} (117 + 317) + \\ +99 + [e < 0]560 + [e >= 0]0 + 244 \end{aligned}$$

which Maple V is able to simplify to:

$$W_{\mathsf{Pow}}(e) = \begin{cases} 1752 - 434e & e < 0 \\ 1474 & e = 0 \\ 1474 + 434e & e > 0 \end{cases} \quad (1)$$

[2] $[E]$ is 1 if the expression in brackets is true, 0 otherwise, $[\bar{E}]$ is a shortcut for $[\mathtt{not} E]$

```
function pow (f:float; e :integer) return
float is
   result      : float;
   times       : integer;
   exchange  : boolean;
begin
   exchange := true;
   if e <0 then
      times   := -e;
   else
      exchange := false;
      times   := e;
   end if;
   result := 1.0;

   for i in 1 .. times loop
      result := result *f;
   end loop;

   if exchange then
      result := 1.0/result;
   end if;
   return (result);
end pow;
```

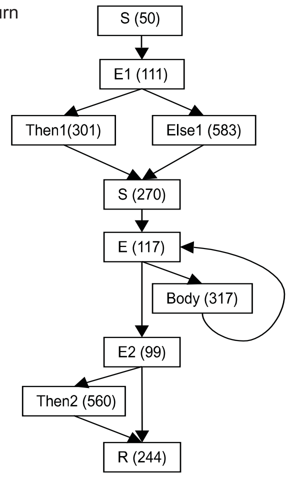

Fig. 1. Pow function. (WCET in parenthesis)

The maximum of $W(e)$ for $e \in [-10, 10]$ is 6092, which happens when $e = -10$. Note that this value is tighter than the value obtained before because it actually represents the path exclusion between the two if constructs. However, the important fact is that the expression is left parameterised until a call to the function is found. For example, the WCET of the following procedure:

```
Procedure usepow(n: integer) is
   f,g,h: float;
begin
   f:= pow(Pi,2);
   g:= pow(E,n);
   h:= pow(f+g,-2);
end usepow;
```

would be:

$$W_{\mathsf{usepow}}(n) = W_{\mathsf{pow}}(2) + W_{\mathsf{pow}}(n) + W_{\mathsf{pow}}(-2) = \\ = 4962 + W_{\mathsf{pow}}(n)$$

where $W_{\mathsf{pow}}(2) = 2342$ and $W_{\mathsf{pow}}(-2) = 2620$.

The WCET of usepow still depends on n. If a call to this function is actually done with a particular value, then this one will be used to obtain the final WCET. However, if it is not possible to determine the value of n we should rely on range analysis to determine the maximum and minimum values of n, say $[-10, 10]$ and find $max_{n\in[-10,10]}\{W_{\mathsf{pow}}(n)\}$. This maximisation can also be done algebraically in most of the cases. If algebraic evaluation is not possible, then enumeration methods like abstract interpretation and symbolic execution should be used.

In order to be able to build such expression, annotations are required in the source code, but not for expressing maximum loop bounds, but to describe the expressions that determine the dependency of a particular path with the parameters of the function. The annotations required for the pow function are shown in the following section.

4. SYMBOLIC WCET

Current programming languages are conceived to describe the functional aspects of a problem. However, WCET analysis tries to identify the non-functional aspects of it. The problem for WCET analysis is that the non-functional aspects may be impossible to derive simply because they are not captured by the programming language. One alternative is to use a programing language which captures non-functional aspects, the other approach is to complement the program with information of the non-functional behaviour. The former is likely to have little acceptance as it requires specific compiler, tools, etc. The latter seems more easy to apply and it is widely used.

The additional information to describe the non-functional behaviour of the system is obtained from the analysis of specially formatted comments (like the Spats ones) or with calls to specific procedures that describe the WCET properties. In any case, a tight prediction of the WCET requires the collaboration of the programmer. Without this additional information it is impossible to determine the exact WCET because it is generally not specified in the functional description of the code. Our approach is to complement the functional code with statements that describe its non-functional behaviour in a way that is flexible and allows a high-level manipulation with a computational algebra system.

We assume that there exists a low level WCET analysis that is able to provide the WCET of each basic block. Our objective is to build an expression of the WCET of the whole program. The symbolic analysis of the WCET is therefore based on three elements:

- The support of an annotation mechanism for particular constructs (conditions regarding the then or else parts of an if construct), or to define abstract expressions for loop bounds.

- A mechanism for building expressions for whole pieces of code by combining the expressions of the building blocks.
- An evaluation engine (highly tied to a computational algebra system) that performs abstract simplification and evaluation of the expressions to finally obtain the WCET.

We view each of these components in more detail.

4.1 *Annotations and WCET parameters*

Annotations are required to define the parameters that affect the WCET of a piece of code and the expressions that describe the WCET of a section of code. We have introduced a notation similar to the one in (Chapman *et al.* 1996) but adaptation to other WCET annotation systems is straightforward. It is easy to see them with an example. This is the annotated code of the pow function:

```
function pow (f:float; e :integer)
              return float is
   result    : float;
   times     : integer;
   exponent  : boolean;
begin
   --{ mode  (e)
   exchange := true;

   if e <0 then  --{ [e <0]
       times    := -e;
   else
       -- e>=0 assumed
       exchange := false;
       times    := e;
   end if;
   result := 1.0;

   --{ loop i in 1 .. abs(e)
   for i in 1 .. times loop
       result := result *f;
   end loop;

   if exchange then   --{ [e <0]
        result := 1.0/result;
   end if;
   return (result);
end pow;
```

Annotations are required for:

- Defining the parameters of a subprogram the WCET depends on. (i.e. –{ mode (e) and induction variables of loops. Additional information on ranges of values can be specified. These parameters of an expression are limited to subprogram parameters and to induction variables. We call these parameters *WCET parameters*.
- Boolean expressions in if-then-else constructs (i.e. –{ [e<0]) to relate particular paths with parameters of the expressions. These are expressions that evaluate to true or false and that are expressed only in terms of WCET parameters.
- Induction variables and limits of loops (i.e. –{ loop i in 1 .. abs(e)) or simply a constant maximum number of iterations (i.e. –{ loopbound n) that describe the induction variable and the expressions that determine the loop bounds. The expressions of the loop bounds can also describe relationships with other WCET parameters, for instance of outer loops for non-rectangular loop analysis (see section 5).

In this context, WCET expressions are of three types

With these three basic annotations it is possible to build complex expressions for nested loops and if constructs that are parameterised for each procedure call.

4.2 *Expressions*

We assume that each basic block has either a constant WCET (for example obtained by a low level WCET analysis) or an expression which is parameterised by $e_1, \ldots, e_n$, where e_i is defined in the context of the basic block. It is easy to show that for a structured program, an expression of a section of code can be build using basic patterns for the typical constructs. Assuming that for each basic block a we have $W_a(e_1, \ldots, e_n)$ which is the expression of the WCET of a, in terms of the parameters e_i, then the three basic patterns for a sequential composition, loops, and if-then-else constructs are:

- Sequence: The expression of a sequence of instructions A;B is simply $W_\mathsf{A} + W_\mathsf{B}$.
- If: The expression of an if construct has two variants. If there is a condition in the construct like:

```
A=if X then
  --{ [E]
  B;
else
   C;
end if;
```

  where [E] is an expression of WCET parameters that evaluates to true or false. The WCET of this code fragment is therefore

$$W_\mathsf{X} + [E]W_\mathsf{B} + [\bar{E}]W_\mathsf{C} = \\ = W_\mathsf{X} + \begin{cases} W_\mathsf{B} & \text{if } E \\ W_\mathsf{C} & \text{otherwise} \end{cases}$$

  Otherwise, if there are no annotations, the WCET expression is simply $W_\mathsf{X} + max(W_\mathsf{B}, W_\mathsf{C})$ which evaluates to a constant.
- Loop: The expression of a loop, depends on whether an abstract evaluation of the loop is desired. A complex loop with a loop iteration annotation like:

```
--{ loop i in E1 .. E2
--{ loopbound E3
for i in a .. b loop
   D;
end loop
```

where E1, E2 and E3 are WCET expressions that evaluate to an integer. The WCET of the loop is

$$W_{\mathsf{pre}} + \sum_{i=E_1}^{E_2} (W_{\mathsf{D}}(i) + W_{\mathsf{post}})$$

or if only the loopbound E3 is used/known, it is simply:

$$W_{\mathsf{pre}} + E_3(W_{\mathsf{D}} + W_{\mathsf{post}}))$$

where W_{pre} is the initialisation of the loop (initialisation of induction variable and first evaluation of the exit condition) and W_{post} is the WCET of the induction variable increment plus the evaluation of the exit condition.

4.3 *Evaluation*

The evaluation of the WCET involves the manipulation and construction of the WCET expressions for each of the nodes of the context tree. We again distinguish three special nodes. Subprograms, loops and conditional constructs. This is done in two stages.

(1) Expression building. The context tree is visited bottom-up and expressions for each node are build from the expressions of the node. After each expression is build abstract simplifications are applied to reduce them to a compact form.
(2) Instantiation. The context tree is visited again, but top-down to instantiate each expression and to evaluate it to a constant value. This is, in fact, an evaluation of the expressions.

The end result is a context tree, annotated with the worst-case execution time of each node, where each call to a procedure has a tailored WCET that depends on the context in which it is called.

We stress the fact that the whole process of annotation extraction, expression building, instantiation, evaluation and simplification is performed automatically.

5. LOOP BOUNDS

One of the powerful capabilities of the framework described here is for tight loop iterations identification. There is a vast knowledge on how to evaluate nested finite summations which is already available in most computational algebra systems like Mathematica or Maple. We can exploit this power to obtain expressions for the number of iterations of nested loops with if constructs in them.

It is outside of the scope of this paper to present a formal treatment of loop bounds and therefore we only present the principal properties of this approach.

The rules for building expressions are enough for specifying expressions that can be suited for automatic detection of loop bounds. The applicability of the technique developed here is also shown with an example. Consider the following function:

```
procedure foo(i : integer) is
begin
   for j in i .. i + 100 loop
       if j>=40 then A; end if;
       if j<50 then B; end if;
   end loop;
end foo;
```

The exact worst case occurs for a subset of values of i, any loop enumeration technique or loopbound annotation will necessary lead to an overestimation of the WCET. However, the abstract expression of this procedure is given by:

$$\begin{aligned} W_{\mathsf{foo}}(i) &= W_{\mathsf{pre}} + \\ &+ \sum_{j=i}^{i+100} (W_{\mathsf{E1}} + [j \geq 40] W_{\mathsf{A}} + \\ &+ W_{\mathsf{E2}} + [j < 50] W_{\mathsf{B}}) = \\ &= W_{\mathsf{pre}} + 101(W_{\mathsf{E1}} + W_{\mathsf{E2}}) + \\ &+ \sum_{j=i}^{i+100} \left(\left\{ \begin{array}{ll} W_{\mathsf{A}} & \text{if } j \geq 40 \\ 0 & \text{otherwise} \end{array} \right\} + \right. \\ &\left. + \left\{ \begin{array}{ll} W_{\mathsf{B}} & \text{if } j < 50 \\ 0 & \text{otherwise} \end{array} \right\} \right) \end{aligned}$$

If i is known then a substitution allows to calculate W_{foo} without problems. However, if the values of i are unknown the question is to compute $\max_{\forall i}\{W_{\mathsf{foo}}(i)\}$.

A simple approach is not to consider the specific loop and if annotations which leads to the value

$$\begin{aligned} W_{\mathsf{foo}}(i) = W_{\mathsf{pre}} + 101(W_{\mathsf{E1}} + \\ + W_{\mathsf{A}} + W_{\mathsf{E2}} + W_{\mathsf{B}}) \end{aligned}$$

However, Maple V is able to determine that maximum happens to be the same for all $i \in [40, 50]$. However, we note that in the general case it is not always possible to obtain such a tight bound and we may have to rely on the value obtained in equation (2).

6. FUTURE WORK

This research is still under development. Future work includes the formalisation of the translation procedure and the development of a prototype tool that implements the technique.

Special attention is being taken for identifying specific simplifications of expressions generated by nested loops with if constructs and on algebraically finding the maximum of these expressions.

We have succeeded in obtaining very tight WCET values for very complex code fragments by an adequate annotation mechanism. Special algebraic methods are required for such expressions to identify, firstly, if the code is simplifiable, and if so, to perform such simplification. We have succeeded in determining some of these methods, further work is required to complete them. In fact, our analysis allows an exact determination of the WCET of the foo function described in section 5.

7. CONCLUSION

By representing the WCET of a section of code as an algebraic expression parameterised with some variables much tighter WCET values can be obtained by exploiting the knowledge in which this code may be executed.

The technique presented here will provide a value which is at least as tight as the one obtained with other high-level methods at the extra cost of a more sophisticated analysis. In any case, this symbolic technique complements other types of analysis like range analysis and dead-path elimination, and low-level WCET analysis.

8. REFERENCES

Chapman, R., A. Burns and A. Wellings (1996). Combining static worst-case execution time analysis and program proof. *Real-Time Systems* **11**, 145–171.

Ermedahl, A. and J. Guftafsson (1997). Deriving annotations for tigh calculation of execution time. In: *Proc. of Europeean Conference on Parallel Processing*.

Healy, C., M. Sjodin, V. Rustagi and D. Whalley (1998). Bounding loop iterations for timing analysis. In: *Proceedings of the IEEE Real-Time Technology and Applications Symposium*.

Healy, C., R. D. Arnold, F. Mueller, D. Whalley and M.G. Harmon (1999*a*). Bounding pipeline and instruction cache performance. In: *IEEE Transactions of computers*. Vol. 48.

Healy, C., R. van Engelen and D.B. Whalley (1999*b*). A general approach for the tight timing predictions of non-rectangular loops. In: *WIP Proceedings of the 5th IEEE Real-Time Technology and Applications Symposium*.

Liu, Y. and G. Gomez (1998). Automatic accurate time-bound analysis for high-level languages. In: *ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*.

Lundqvist, T. and P. Stenstrom (1998). Integrating path and timing analysis using instruction level simulation techniques. In: *ACM SIGPLAN Workshop on Languages, Compilers and Tools for Embedded Systems*.

Puschner, P. and A. Burns (2000). A review of WCET analysis. guest editors introduction. *Real-Time Systems (To appear)*.

Puschner, P. and A. B. Schedl (1991). Computing maximum task execution times - a graph based approach. *Real-Time Systems*.

Puschner, P. and C. Koza (1989). Calculating the maximum execution time of real/time programs. report. Technische Uiversitat Wien.

Sakellariou, R. (1997). Symbolic evaluation of sums for parallelising compilers. In: *Proceedings of the 15th IMACS World congress on Scientific Computation, Modelling and Applied Mathematics*.