Test Generation Strategies to Measure Worst-Case Execution Time

Nicky Williams, Muriel Roger
CEA LIST, Software Reliability Laboratory, 91191Gif sur Yvette, FRANCE
nicky.williams@cea.fr, muriel.roger@cea.fr

Abstract

Under certain conditions, the worst-case execution time (WCET) of a function can be found by measuring the effective execution time for each feasible execution path. Automatic generation of test inputs can help make this approach more feasible. To reduce the number of tests, we define two partial orders on the execution paths of the program under test. Under further conditions, these partial orders represent the relation between the execution times of the paths. We explain how we modified the strategy of the PathCrawler structural test-case generation tool to generate as few tests as possible for paths which are not maximal in these partial orders, whilst ensuring that the WCET is exhibited by at least one case in the set. The techniques used could also serve in the implementation of other test generation strategies which have nothing to do with WCET.

1. Introduction

It is very important to know the Worst-Case Execution Time (WCET) of real-time software in order to schedule different tasks. However, recent developments in processor architectures [1] complicate the task, even when it is restricted, as it usually is, to sequential, uninterrupted, portions of code. Events such as data-cache misses and bad branch prediction use up many more cycles than individual instructions but the precise behaviour of these prediction mechanisms is not usually divulged by the manufacturer and is subject to continual innovation.

We propose an approach based on the measurement of the effective execution time on the target processor when the code is run on each test-case in a certain set. We will not discuss here how to execute the code and measure the execution time but these tasks can clearly be automated. We define a test set which will guarantee that under certain conditions the longest execution time of the cases in the set is the WCET of the function under test when it is run uninterrupted. We will not go into the details here either of the conditions under which our approach is justified and how they can be checked or ensured. Our subject in this paper is the automatic generation of the defined test set. We explain how we modified the test generation strategy of the PathCrawler prototype tool to do this.

2. The basis of our approach: path testing

Structural testing provides the first step in our approach. The 100%-feasible-path structural test criterion guarantees that at least one test case is executed for each feasible execution path in the source code of the program under test. Let us suppose that execution of the same path in the source code, starting from the same initial state of the machine, will always give the same execution time. Then if we run the function under test on a test set satisfying the all-paths criterion, and measure the execution time for each test-case, the longest execution time measured will be the WCET.

In fact, as explained in [2], we need to ensure that the following conditions are satisfied to be sure that a safe WCET is obtained in this way.

1. Each feasible execution path in the source code gives rise to at most one feasible execution path in the binary code (even if it is not the same path).
2. The execution time of a feasible execution path in the binary code is the same for all input values which cause the execution of this path.
3. For each test case it is possible to set the machine to some worst possible initial state concerning cache behaviour, branch prediction, etc before running the test.
4. Variations in external system behaviour such as bus activity, DRAM refresh, do not influence execution time.

```
  f(int x, int y){
1     z = x;
2     if (x < 0)
3      z = 0;
4     if (y < 5)
5      z = 2z;
6     if (z > 0)
7      z = z + 1;
8     else
9      z = z - 1;
      }
```

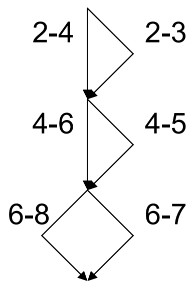

Figure 1: Example source code and CFG

In a similar approach described in [3], a path predicate is found for each execution path by a combination of dataflow analysis and slicing. The feasibility of the path is checked by linear constraint solving. However, the WCET of the path is not measured, as in our work, but estimated by abstract interpretation of a model of the microprocessor. The data cache and branch prediction are not modelled.

3. Towards fewer execution paths

Many real-life programs have far too many feasible execution paths for the measurement of the execution time of each one to be envisaged, even if the process is fully automated. We must then reduce the size of our test set whilst ensuring that it still contains at least one path exhibiting the WCET.

The prediction mechanisms of recent microprocessors mean that we cannot always assume that the path with the most instructions has the longest execution time, even if instructions are weighted according to their relative execution times. However, we can make certain generalisations about the prediction mechanisms which enable us to compare the execution times of certain pairs of paths.

We have therefore defined two partial orders on execution paths based on assumptions about their relative execution times. This means that in a set of feasible execution paths, the path (or paths) 'with the longest execution time must be maximal in one of these partial orders. We then only need to measure the execution times of paths which are maximal in these partial orders. As our partial orders are based on the control structures which contribute to the combinatorial explosion in the number of execution paths, the size of the test sets is significantly reduced if they only include maximal paths.

4. Our first partial order

Our first partial order is based on if-the-else (ITE) structures. We illustrate it on the very simple example of a C function whose source code and control flow graph are shown in Figure 1.

Let us define an execution *path* by the sequence of conditional branches it takes, after unrolling all loops, in-lining all function calls and replacing control-structures such as `case`, as well as multiple conditions, by the equivalent ITE branches. We call a path fragment which starts at entry to the program but ends before exit from the program, a *partial path*. Given a path *P* and the partial path, *PP*, which is a *prefix* of *P*, the *suffix of PP in P* is obtained by removing *PP* from *P*.

The execution paths of our example are shown in Figure 2. The branches are labelled by the line number at their start and end points: the first branch in P1 goes from line 2 to line 4 so is labelled 2-4. We note the whole path P1 2-4:4-5:6-8. Note that two of the paths in Figure 2 are infeasible: they cannot be activated by any test-case because their branch conditions are inconsistent.

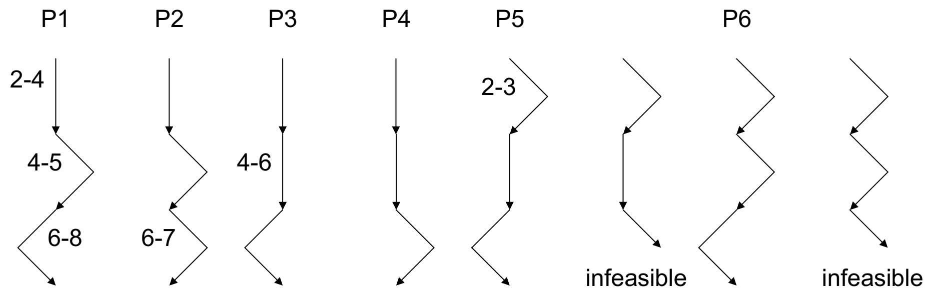

Figure 2: Default strategy paths of the example

An *ITEpath* is a path fragment going from the beginning of an ITE structure to the end of the structure. It may be one branch or a sequence of branches containing paths of other, nested, ITEs. An ITEpath is defined as *empty* if it contains no assignments, no function calls, and no other ITE structures or loops. It can contain an unconditional jump to the instruction after the ITE. Note that even non-empty ITEpaths may only consist of a single branch. An *ITEE* is an ITE in which one path is empty.

Our example contains 3 ITE structures of which the first two are ITEEs (empty ITEpaths are shown as vertical in Figures 1 and 2). We now define the following strict partial order on execution paths:

Path *Pi* is *empty-ITE-path-slower-than* path *Pj*, noted *Pi* > *Pj*, if the only difference between *Pi* and *Pj* is that one or more ITEpaths are empty in *Pj* but not in *Pi*. This partial order can be extended to partial paths.

Here are the empty-path-slower-than relations in our example:

P6 > P5 > P3
P6 > P1 > P3
P2 > P4
P2 and P6 are maximal.

We hypothesise that if feasible execution path *Pi* is empty-ITE-path-slower-than feasible execution path *Pj* then the execution time of Pi will be greater than that of *Pj*.

Let us briefly consider whether this hypothesis is justified for two prediction mechanisms which make WCET prediction particularly difficult.

The first is the data cache. Many data cache algorithms exist but they are all based on comparing the address of the data referenced in the current instruction to that of data referenced in recently executed instructions. Executing a non-empty path in an ITEE instead of the empty path will not only cause more instructions to be executed, with a corresponding increase in execution time, but may also reference new data, increasing the chances of a cache miss, and another increase in execution time. However, if the non-empty path changes the value of a pointer which is later used to access data, it is possible that with this new value a data cache miss is avoided after the non-empty path, but occurs after the empty path. To be sure that we can apply the partial order in the presence of a data cache, we should check for each execution path whether any pointer values are changed in the non-empty path through the ITEE and then de-referenced after the ITEE. This check can be carried out using PathCrawler.

The other mechanism is branch prediction. This can be based not only on the history of past branches taken, but also on a default prediction which is often the "true" branch. In C code, most empty ITEpaths will be "else" branches (because it is far more natural to code an empty "else" in C than an empty "then"), for which default branch prediction will fail, imposing a penalty on the execution time which could be longer than that of the "then" path. We cannot use this partial order on machines which may use this type of branch prediction.

5. PathCrawler's default strategy

To implement the automatic generation of test sets which cover the fewest possible paths which are not maximal in one of the partial orders above, we modified the generation strategy of the prototype PathCrawler tool.

We originally developed PathCrawler to automatically generate test inputs to cover 100% of feasible execution paths in a C program. It takes as inputs the C source code and a specification of the legitimate input values, as described in [4], and outputs a set of test inputs with the execution path covered by each.

PathCrawler was one of the first in a growing number of tools which are based on both symbolic and concrete execution of the function under test and on the modification, and constraint resolution, of the predicate of a previously covered path. However, unlike many other concolic tools [5][6], PathCrawler tries to produce a test set whose coverage of feasible execution paths is complete unless constraint resolution times out. Further comparison with similar tools can be found in [7].

Let us start by describing PathCrawler's default test-case generation strategy, before explaining how we modified it to take account of the partial orders above. The default strategy is illustrated on our example in Figure 2.

By default, PathCrawler generates test-cases by a depth-first exploration of the entire binary tree of feasible execution paths in which the order, left-right or right-left, is determined by constraint resolution and so is effectively non-deterministic. PathCrawler's strategy is implemented using constraint logic programming [8] and it makes extensive use of the backtracking which is built-in to this language.

Given a C function, PathCrawler arbitrarily chooses a first legitimate input vector which will activate some

```
cover_all(P,i,PP) =
    if $Bi_P$ is_last_branch_in P
    then cover_subtree(PP:$\underline{Bi_P}$,i)
    else
     1st pass : cover_all(P,i+1,PP:$Bi_P$)
     on backtrack : cover_subtree(PP:$\underline{Bi_P}$,i)

cover_subtree(PP,i) =
   if gen_test(PP) = P′
       then if  $Bi_P′$ is_last_branch_in P′
               then backtrack
               else cover_all(P′,i+1,PP)
   else backtrack
```

Figure 3: Default strategy pseudo-code

feasible execution path, *P1*. It then behaves as though it performed the call *cover_all(P1,0,empty_path)* of the recursive function *cover_all(P,i,PP)* whose pseudo-code is shown in Figure 3.

Notation: In the pseudo code, *P* and *P′* are paths, composed of branches $B1_P$, $B2_P$,…,Bi_P,… and $B1_P'$, $B2_P'$,…,Bi_P',…, respectively, where 1, 2, …, *i*,… are branch indices and *PP* is a partial path. $\underline{Bi_P}$ is the branch which is the opposite of branch Bi_P. Predicates such as *is_last_branch_in_path* are written in infix form: Bi *is_last_branch_in_path P*. The concatenation of a branch to the end of a path is denoted with a the symbol *:* and actions are connected in a sequence by the symbol *;*. If one action is performed in the first pass through a particular point and then another on back-tracking, the actions will be labelled *1st pass* and *on backtrack*. *gen_test(PP)* designates an attempt to generate a test to cover *PP*. If *PP* is feasible, *gen_test(PP)* succeeds and its result is the whole path covered but if *PP* is infeasible *gen_test(PP)* has no result.

In our example, let us suppose that *P1* is the first path. Its first branch, $B1_{P1}$, is 2-4, which is not also its last branch so 2-4 becomes the first branch in *PP*. Similarly, 4-5 becomes the second branch in *PP*. $B3_{P1}$ is 6-8 and is the last path in *P1* so $\underline{B3_{P1}}$, which is 6-7, is added to *PP*, which becomes 2-4:4-5:6-7.

The test-case generated for *PP* activates path *P2* in which (because *P2* is identical to *PP*), $B3_{P2}$ is the last branch so the default strategy backtracks over the previous treatment of branch 4-5 in *P1*, replacing it in *PP* by its opposite so that *PP* becomes 2-4:4-6. Let us suppose that the test-case generated for this *PP* activates path *P3*. Branch $B3_{P3}$ is 6-7. This is the last branch in *P3* so its opposite, 6-8, is added to the new partial path so that *PP* becomes 2-4:4-6:6-7. However, this partial path is infeasible so Pathcrawler backtracks over the previous treatment of branch 2-4 in *P1*, replacing it by its opposite as first branch in *PP*, which becomes 2-3. The test-case now generated for *PP* activates path *P5*, and so on.

Indeed, when PathCrawler succeeds in generating a test case for a value of *PP* formed by replacing a branch in path *P* by its opposite, the path, *P′*, covered by the new case may be exactly the same as *PP*, leading from the replaced branch straight to exit from the function under test (as in the case of *P2* in our example, which is exactly the partial path formed by replacing the last branch in *P1*) or may be composed of other branches before exit from the function (as in the case of *P3*). Indeed, *PP* may be common to several paths (as in the case of the partial path formed by replacing $B2_{P2}$, this partial path is common to *P3* and *P4*). PathCrawler generates an input vector which may activate any one of these continuations after the replaced branch. The path activated depends on the constraint resolution strategy used by PathCrawler to generate the input vector so is effectively non-deterministic.

In conclusion, depth-first exploration ensures that when PathCrawler forms a feasible partial path *PP* by replacing a branch Bi_P in *P* by its opposite, $\underline{Bi_P}$, it covers the whole sub-tree of feasible execution paths rooted in *PP* before backtracking over the treatment of $B(i\text{-}1)_P$. This is the property that will now be used to modify the test generation strategy.

6. Strategy to minimise empty ITEpaths

6.1 Introduction

We modified PathCrawler's default strategy so as to generate tests for a subset of the feasible execution paths containing all paths which are maximal in our first partial order and the fewest possible non-maximal paths. The default strategy is modified in three ways:

1. to memorise feasible paths and infeasible partial paths ;
2. to explore all non-empty paths of ITEEs before any empty paths ;
3. for any partial path ending in an empty ITE-path, to only try to generate tests to cover maximal paths by comparing this partial path to the feasible paths and infeasible partial paths which have already been found.

6.2 Illustration on the example

We first illustrate the modified strategy on our example, before presenting the new algorithm. The paths explored by the modified strategy are shown in Figure 4. We suppose that the first path generated is still *P1*. The modified strategy first memorises *P1*, and then starts to treat it. Its first branch, 2-4, is an empty ITE-path and, instead of adding this branch to *PP* in the same way as the default strategy, the modified strategy forces exploration of a non-empty ITEpath first by adding the opposite, branch 2-3, to *PP* and then trying to generate a test-case to cover this partial path 2-3. Let us suppose that the new test-case covers path *P6* (and not *P5*, which also has 2-3 as a prefix). The modified strategy memorises this feasible path and proceeds to treat the branch after 2-3, branch 4-5. This is a non-empty ITEpath and the modified strategy adds this to the new partial path, in the same way as the default strategy. The next branch, 6-8, is not a branch of an ITEE so the modified strategy treats it in the same way as the default strategy, creating the infeasible partial path 2-3:4-5:6-7. The modified strategy memorises this infeasible partial path and then backtracks over the treatment of branch 4-5 in *P6*, adding its opposite, 4-6, to the new partial path as for the default strategy to obtain partial path 2-3:4-6. However, branch 4-6 is an empty ITEpath so instead of just trying to generate a test case for the new partial path, the modified strategy compares it to the memorised infeasible partial paths (just 2-3:4-5:6-7 in this case) and feasible paths (*P1* and *P6*). Indeed, if a test were generated to cover the new partial path 2-3:4-6, it would cover path *P5*, and the already-generated *P6* is empty-ITE-path-slower-than *P5*. However the only partial path starting with 2-3 and taking branch 6-7 tried so far was the infeasible partial path 2-3:4-5:6-7 and so a feasible path starting with 2-3:4-6:6-7 could be maximal. The modified strategy therefore adds the suffix 6-7 to the partial path 2-3:4-6 and tries to generate a test for the resulting partial path 2-3:4-6:6-7. This is infeasible but it doesn't need to be memorised because 2-3:4-5:6-7 is already memorised. There are no more continuations of 2-3:4-6 leading to maximal paths so the modified strategy now backtracks over the treatment of branch 2-4 in *P1*. This empty ITEpath was replaced by its opposite in *P6*, so on backtracking it is the original branch 2-4 which becomes the first and only branch of the new partial path. Once again, the new partial path ends in an empty ITEpath so the modified strategy compares it to the memorised infeasible partial path and feasible paths and deduces that a feasible path starting with 2-4:4-5:6-7 would be maximal. It therefore adds the suffix 4-5:6-7 to the partial path 2-4 and tries to generate a test for the resulting partial path 2-4:4-5:6-7. This is *P2*, so a test is generated which covers exactly the partial path, which is memorised as feasible. Now there are no new branches to be explored, no more suffixes of 2-4 which could be maximal paths and no treatments of branches to backtrack over so test-case generation stops.

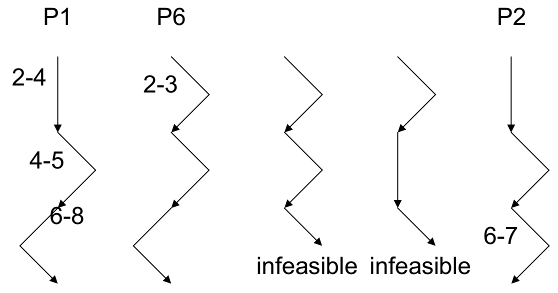

Figure 4: Paths for the modified strategy

6.3 Properties

Like the default strategy, the modified strategy carries out a depth-first exploration of the tree of all feasible execution paths but it forces the left/right choice (to explore the non-empty ITEpaths first) in the case of empty ITEpaths. Depth-first exploration ensures that when the strategy backtracks over the treatment of the first branch in a non-empty path of an ITEE, all feasible and infeasible suffixes of the ITE are known, because the feasible part of the sub-tree rooted in the exit from the non-empty ITEpath has been fully explored. This is the basis for our first claim, which is that our modified strategy

1. generates tests for all maximal paths

Note that in this example only 3 test-cases were generated, successfully covering the 2 maximal paths, and both infeasible partial paths were explored.

Obviously, the performance of the modified strategy compared to the default strategy depends on the element of non-determinism. However, we also claim that our modified strategy

2. reduces (or, at worst, does not increase) unnecessary exploration of the tree of execution paths and
3. reduces (or, at worst, does not increase) generation of tests for paths which are not maximal.

6.4 Techniques

The implementation of the modified strategy is based on two techniques:

The first is the storage of infeasible partial paths in the form of "abstract partial paths". These are (partial) paths which abstract the precise path taken in each ITEE, as explained below. They identify, as the same abstract path, the paths which only differ in their traversal of the ITEEs. The modified strategy uses them to recognise and manage ITEEs, ensuring that partial paths are explored in the right order.

The second technique is the concatenation, to certain non-maximal partial paths, of the suffixes from "compatible" partial paths previously found to be infeasible. The "compatible" partial paths are those with the same abstract prefix as the current partial path. This concatenation replaces complete depth-first exploration of the sub-tree of execution paths rooted at the current partial path.

6.5 The detailed algorithm

We now present the detailed algorithm of the modified strategy in order to demonstrate that it has the three properties mentioned above.

We first define the abstract partial paths used in the modified strategy:

> An *abstract path* is a (concrete) path in which path fragments corresponding to a complete path in an ITEE are replaced by an *abstract ITEE* containing both the empty ITEpath and an abstract non-empty ITEpath. This means that for each abstract ITEE in an abstract path there are two concrete instantiations, one containing the empty ITEpath and one containing the non-empty ITEpath. Note that in each abstract path, each ITEE structure will be represented by one abstract ITEE for each feasible non-empty ITEpath. The definition of abstract paths is extended to abstract partial paths.

Extra notation: the modified strategy needs a set of infeasible abstract partial paths, denoted *infeas* and initialised to the empty set, and also a set of feasible paths, denoted *feas* and also initialised to the empty set. Assignment is written *:=*, set union is denoted by U and set membership is written ϵ. We use the symbol $\cdot$ instead of *:* for concatenation of a branch to the end of an abstract partial path as a reminder that abstract paths contain abstract ITEEs. The addition of a branch which is the end of a non-empty ITEpath causes the whole non-empty ITEpath to be replaced in the abstract partial path by an abstract ITEE. Iteration is written *foreach*, the operation *foreach_in_order* iterates over paths ordered according to the partial order: for each successive ITEE, paths taking a non-empty ITEpath, themselves ordered according to the remaining ITEEs they contain, are treated before the others.

The modified strategy corresponds to the call *cover_max(P1,1,empty_path,empty_path)* of the recursive function *cover_max(P,i,PP,APP)* whose pseudo-code is presented in Figure 5.

As shown by the pseudo-code, if the current branch is the last in the path and not the first (and only) branch in a non-empty path of an ITEE then the default strategy is just extended to memorise feasible paths and abstract infeasible partial paths (by calling *cover_max_subtree* instead of *cover_subtree*). However, if the current branch is the last in the path and also the first (and only) branch in a non-empty path of an ITEE then there is no need to generate a test-case for the path taking the alternative, empty, branch so the default strategy backtracks immediately over the treatment of the previous branch.

If the current branch is not the last branch in the path nor an empty ITEpath nor the first branch in a non-empty path of an ITEE then the default strategy is just modified as above to call *cover_max_subtree*.

However, if it is the first branch in a non-empty path of an ITEE, then when exploring its opposite (i.e. the empty ITEpath) on backtrack, the modified strategy calls *cover_rest_subtree* instead of *cover_max_subtree*. Indeed, if two partial paths, *PP1* and *PP2*, are identical up to an ITEE, but *PP1* takes a non-empty path through the ITEE and *PP2* an empty path, then potentially, ignoring feasibility, *PP1* and *PP2* have identical subtrees of path suffixes after the ITEE. Each of these suffixes will form a path, *P1*, when concatenated to *PP1* which is empty-path-slower-than the path, *P2*, which is formed when the same suffix is concatenated to *PP2*. However, some of these suffixes may be infeasible if they are concatenated to *PP1* but not if they are concatenated to *PP2*. There is no need to cover the path formed by concatenating to *PP2* a suffix which, when concatenated to *PP1* formed a feasible path. This is why *cover_rest_subtree* looks for infeasible partial paths which have a prefix with the same abstract representation (i.e. are identical except for ITEEs) as the current partial path. These are partial paths which were infeasible with a non-empty ITEpath but may be feasible with an empty ITEpath. The situation is more complicated if the current partial path contains several ITEES because a partial path with, for example, one empty ITEpath followed by one non-empty ITEpath is not comparable, in our partial order, with the partial path which is identical except that it

```
cover_max(P,i,PP,APP) =
if $Bi_P$ is_last_branch_in P
then {
    if $Bi_P$ starts_non_empty_ITEE_path
    then backtrack
    else
      cover_max_subtree(PP:$\underline{Bi_P}$,APP·$\underline{Bi_P}$,i) }
else if $Bi_P$ starts_non_empty_ITEE_path
      then {
        1st pass :
            cover_max(P,i+1,PP:$Bi_P$,APP·$Bi_P$)
        on backtrack :
            cover_rest_subtree(PP:$\underline{Bi_P}$,APP·$\underline{Bi_P}$,i) }
      else if $Bi_P$ is_empty_ITEE_path
             then {
              if gen_test(PP:$\underline{Bi_P}$) = P′
              then {
                 1st pass :
                   feas := feas ∪ P′ ;
                   if  $Bi_P$′ is_last_branch_in P′
                   then backtrack
                   else cover_max(P′,i+1,PP:$\underline{Bi_P}$,APP·$\underline{Bi_P}$)
                 on backtrack :
                   cover_rest_subtree(PP:$Bi_P$,APP·$Bi_P$,i)}
               else infeas := infeas ∪ APP·$\underline{Bi_P}$ ;
                      cover_max(P,i+1,PP:$Bi_P$,APP·$Bi_P$) }
             else {
               1st pass :
                   cover_max(P,i+1,PP:$Bi_P$,APP·$Bi_P$)
               on backtrack :
                   cover_max_subtree(PP:$\underline{Bi_P}$,APP·$\underline{Bi_P}$,i)}

cover_max_subtree(PP,APP,i) =
    if gen_test(PP) = P′
        then feas := feas ∪ P′ ;
               if  $Bi_P$′ is_last_branch_in P′
               then backtrack
               else cover_max(P′,i+1,PP,APP)
        else infeas := infeas ∪ APP ;
               backtrack

cover_rest_subtree(PP,APP,i)=
foreach APPext ϵ infeas {
  foreach_in_order PPext concretises APPext {
    if PP is_a_prefix_of PPext
    then {
      if not slower_feas_paths(Pext,APPext)
      then {
        if gen_test(PPext) = P′
        then {
          feas := feas U P′ ;
          if  $Bi_P$′ is_last_branch_in P′
          then backtrack
          else cover_max(P′,i+1,PP,APP) }
        else backtrack }  } } }

slower_feas_paths(Pext,APPext) if
  exists Pext′, PPext′ such_that (
    Pext′ ϵ feas
    and PPext′ is_a_prefix_of Pext′
    and PPext′ concretises APPext
    and Pext′ empty_ITE_path_slower_than Pext )
```

Figure 5: Pseudo-code of the modified strategy

contains a non-empty path through the first ITEE and an empty path through the second. This is why the *slower_feas_paths* predicate is necessary.

If the current branch is not the last branch in the path but is an empty ITEpath, then the modified strategy overrides the left-right non-determinism of the default strategy to force exploration of the non-empty ITEpaths first (unless they are all infeasible) and the empty ITEpath itself on backtrack. This ensures that when suffixes of the empty ITEpath are explored by *cover_rest_subtree*, all infeasible suffixes of the non-empty ITEpaths have been found.

Note that we have made several optimisations to the algorithm as defined above but to improve clarity these are not described.

7. Our second partial order

Our second partial order is based on loops. If a loop has a number of iterations which varies with the value of the inputs, then two paths which are identical except for the number of iterations of this loop may not have the same execution time and must be treated as two separate paths. However, under certain conditions we can consider that of these two paths, the one which makes the most iterations of the same loop will have the longest execution time. We can suppose that this is the case if the following conditions are satisfied:

1) the same branches (if any) are taken in each loop iteration ;
2) the data accessed in successive loop iterations have addresses which are close enough to ensure that executing a particular iteration will not avoid a future data-cache-miss ;
3) as for our first partial order, we must check the effect of the successive loop iterations on pointer values which are de-referenced after loop exit.

Indeed, branch prediction mechanisms should not present any problems in this case; they should always succeed in each iteration until the last, when they will always fail. As for data cache behaviour, if the same branches are taken in each iteration then the same data will be referenced, unless it is referenced using pointers whose value changes from iteration to iteration, hence the second condition.

It so happens that many real-life programs present in real-time systems do contain loops respecting the conditions above. In particular, programs written with the aid of mathematical modelling software often contain loops which traverse arrays in order to obtain a linearised approximation (or interpolation) of the continuous graph of a mathematical function. These loops typically search for the two x-coordinates of the discrete graph which surround a given input value and then return the corresponding y-coordinate of the linearised approximation. If the graph has another dimension, then it will be treated in a similar way. Typically, the search is not optimised but just consists of inspecting one by one the array elements which represent the x-coordinate values until the appropriate one is found and then exiting the loop after a variable number of identical iterations.

Let us define a second partial order on execution paths:

Path *Pi* is *identical-iteration-slower-than* path *Pj* if

- *Pi* and *Pj* are identical except for the number of iterations of certain loops and
- each of these loops has at least one iteration and
- in each of these loops, all iterations are identical and
- for each of these loops, *Pi* has more iterations than *Pj*.

The partial order is restricted to loops with at least one iteration because zero iterations of the loop may result in different data cache behaviour than at least one iteration of the loop because of the data references in the body of the loop. Note that nested loops are not considered and nor are loops with a mixture of an unbroken sequence of identical iterations and some other, non-identical, iterations too.

To generate tests to cover paths that are maximal in this partial order, we have further modified PathCrawler's generation strategy in a similar way to that described for the first partial order. This modification is not described in detail here because it is more complicated than for the first partial order, but it is based on the same two basic techniques described in 6.4.

For this partial order, the modified strategy forces exploration of an additional loop iteration first, in a similar way to the exploration of non-empty ITEpaths. When an additional loop iteration is infeasible, we have reached the maximum number of iterations for the given path so far. This must be memorised so that on backtrack it can be compared to the current number of loop iterations in similar paths. Indeed, loop exit is only explored on backtrack and if it is exit from the maximal number of iterations. In that case, the whole sub-tree of feasible execution paths is explored. Otherwise, the infeasible partial paths and feasible paths found so far are studied to decide which suffixes to try and cover, just as for our first partial order. The number of iterations of each loop with identical iterations is abstracted in the abstract path and for each such loop in the abstract path suffix, concrete loops are tried in descending order of the number of iterations.

The first difficulty is in the analysis of the loop. Identifying identical iterations is relatively straightforward. However, the loop head may have a complex condition made up of multiple sub-conditions combined in conjunctions and disjunctions. The loop may also contain `break` instructions with equally complicated conditions. In this case, we have to identify the sub-conditions that immediately precede the start of a new loop iteration and those that immediately precede exit from the loop. We also have to identify the sub-conditions whose negation immediately precedes the start of a new loop iteration or loop exit and those whose negation can lead to either. Note that if a path contains a loop with just one iteration, then it is not possible to know until backtrack whether it will need to be compared to other paths in which there are more, identical, iterations.

8. Perspectives

We tried our default and modified strategies on an example of industrial real-time software comprising 1512 lines of commented C source code and 89 conditional instructions but no loops. The default strategy took several days to generate 846975 test-cases and our modified strategy took about one day to generate 6554 test-cases. Geensys, our partners in the MaSCotTE project, designed and implemented a test bench and ran the two test sets on a HCS12X simulator to measure the effective execution time of each test-case. The path with the longest execution time in the first set was indeed covered by a test-case in the second set.

In fact, the modified strategy generated all the test-cases in a few hours and the rest of the time was spent trying in vain to find solutions for continuations of partial paths ending in empty ITEpaths. We are currently studying how to reduce the number of continuations explored by taking into account the dependencies between the code in the ITE body or loop iteration and the final branch condition in the infeasible partial path.

The techniques used to generate paths which are maximal in our partial orders could certainly be developed further and more widely used to implement other structural testing strategies which have nothing to do with WCET, but where certain paths, or path fragments, do not need to be covered. Abstracting the path fragment which traverses a control structure (ITE, loop or function call) helps to avoid unnecessary exploration of this part of the program under test. For example, a path-coverage test criterion may not impose coverage of all paths which only differ in the number of identical iterations of a certain loop or in the path taken through a called function. The memorisation of infeasible partial paths ensures that at least one instance of each feasible path containing the loop or function call is tested. Indeed, we already used the memorisation of infeasible partial paths to treat function calls in [7] but in that approach the traversal of the function was “abstracted” by its specification. In an alternative approach to the treatment of function calls, [9] store not infeasible partial paths but the result of symbolic execution of already-explored paths. In [10], memorisation of execution paths and simple dependency analysis is used to eliminate path fragments which have already been tested.

Acknowledgements

We would like to thank our colleagues Nikolaï Kosmatov and Bernard Botella for their helpful advice. This work is partially funded by the French Agence Nationale pour la Recherche (PREDIT and Num@tec Automotive project MaSCotTE).

References

[1] R. Heckmann, M. Langenbach, S. Thesing and R. Wilhelm, *The influence of processor architecture on the design and the results of WCET tools*, In Proceedings of the IEEE 91(7): 1038-1054 (2003)

[2] N. Williams,*WCET Measurement using modified Path Testing*, In Proc. WCET’05, Palma de Mallorca, Spain, July 2005

[3] Meng-Luo Ji, Ji Wang, Shuhao Li and Zhi-Chang Qi, *Automated WCET Analysis based on Program Modes,* In Proc. AST’06, Shanghai, China, May 2006

[4] N. Williams, B. Marre, P. Mouy and M. Roger, *PathCrawler: Automatic generation of path tests by combining static and dynamic analysis*, In Proc. EDCC-5, Budapest, April 2005

[5] K. Sen, D. Marinov, and G.Agha, *Cute: A concolic unit testing engine for C*, In Proc. ESEC/FSE'05, Lisbon, Portugal, September 2005

[6] C.Cadar, V.Ganesh, P.M.Pawlowski, D.L.Dill, and D.R.Engler, *Exe: automatically generating inputs of death*, In Proc. ACM Conference on Computer and Communications Security, 2006

[7] P. Mouy, B. Marre, N.Williams and P. Le Gall, *Generation of all-paths unit test with function calls*, In Proc. ICST’08, Lillehammer, Norway, 2008

[8] M. Wallace, S. Novello and J.Schimpf, *ECLiPSe: A platform for Constraint Logic Programming*, IC Parc, Imperial College, London, Aug 1997

[9] S. Anand, P. Godefroid, N. Tillmann, *Demand-Driven Compositional Symbolic Execution*, In Proc. TACAS 2008, Budapest, Hungary, 2008

[10] P. Boonstoppel, Cristian Cadar, Dawson Engler, *RWSet: Attacking path explosion in constraint-based test generation,* In Proc. TACAS 2008, Budapest, Hungary, March-April 2008