

WCET measurement using modified path testing

Nicky Williams

CEA/Saclay, DRT/LIST/SOL/LSL, 91191 Gif sur Yvette, France

Nicky.Williams@cea.fr

Abstract

Prediction of Worst Case Execution Time (WCET) is made increasingly difficult by the recent developments in micro-processor architectures. Instead of predicting the WCET using techniques such as static analysis, the effective execution time can be measured when the program is run on the target architecture or a cycle-accurate simulator. However, exhaustive measurements on all possible input values are usually prohibited by the number of possible input values. As a first step towards a solution, we propose path testing using the PathCrawler tool to automatically generate test inputs for all feasible execution paths in C source code. For programs containing too many execution paths for this approach to be feasible, we propose to modify PathCrawler's strategy in order to cut down on the number of generated tests while still ensuring measurement of the path with the longest execution time.

Introduction

Prediction of Worst Case Execution Time (WCET) is made increasingly difficult by the recent developments in micro-processor architectures [8]. This is because the execution time of an instruction in the source code has become strongly dependent on the state of the machine at the time the instruction is executed because events such as data cache misses and pipeline stalls due to bad branch prediction use up many more cycles than individual instructions. These events are difficult to predict by techniques such as static analysis of the program code, especially when the precise architecture of the micro-processor is not divulged by the constructor. Moreover, static analyses must continually keep up with the latest architectural innovations.

An alternative approach, if the target architecture is already available, is not to predict the WCET using techniques such as static analysis but to measure the effective execution time when the program is run. If the target architecture is not yet built but a cycle-level simulator of it is available, then simulation can be used to “measure” the execution time. The disadvantage of direct measurement of the program execution time is that exhaustive measurements on all possible input values are usually prohibited by the number of possible input values. If effective execution time is only measured for a subset of possible program input values, then either we must take a probabilistic approach [1] or else we must guarantee that the measured subset includes program input values which cause the longest execution time.

First step to a solution : path testing

The structural testing field provides a first step towards a solution. 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. Suppose that the longest effective execution time is found for a test set which satisfies this criterion. This will be the WCET if we can 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. In fact, we need to make the following suppositions, each of which requires careful analysis to be sure that a safe WCET is obtained:

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), which is deterministically executed by means of the same sequence of operations at the micro-architecture level: some compiler options may need to be disabled for this to be true we may also need to ensure that the test set data does not result in floating-point operations being dynamically replaced by their integer equivalents because the operands have integer values.
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: the execution time of some instructions such as division or square root may vary for different values of input data but it should be possible to measure this variation and either choose input values accordingly or else add a fixed factor to the measured execution time according to the number of such instructions in the path. This condition may also not be respected if an execution path contains a reference to an element of a data structure with a variable index and the data structure is large enough to provoke a cache miss for some values of the index but not for others. Test data

generation should therefore favour input values which result in the largest possible index values for the data structure references in the path.

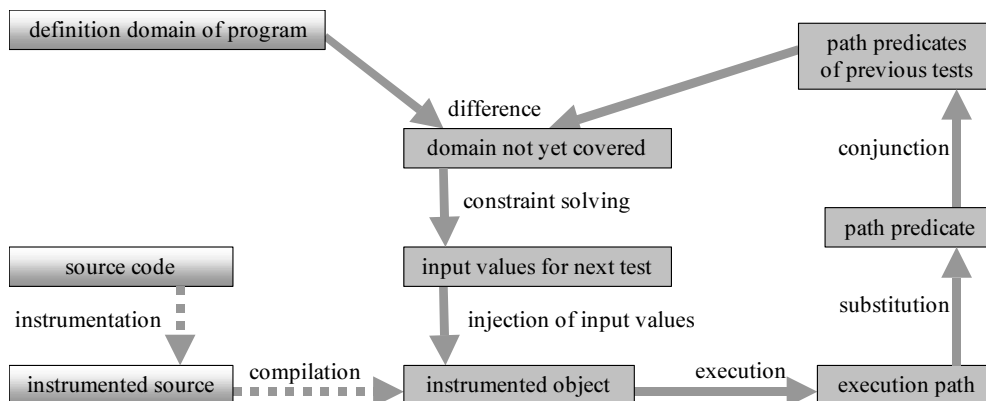
3. Each test case can be run on a machine which is initially set to some worst possible initial state: for a given execution path, the initial state of the caches and DRAM will determine the future behaviour of the caches, DRAM refreshes, and DMA accesses. A minimal knowledge, or hypothesis, concerning the way data is placed in the caches should allow us to design a program to be run just before the program under test in order to fill the caches with useless data or instructions. In the case of embedded, reactive, cyclical systems, the initial state of the machine is often the state in which it was left by the last execution of the same program and an analysis of the program should allow us to define the worst initial state possible.
4. The execution of the test case is not interrupted by another task.

The PathCrawler tool for automatic generation of path tests

We have developed the PathCrawler tool 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. This consists of a list of the input variables and the range of values they may take, as well as any preconditions to avoid run-time errors. Indeed, the effective input parameters of a C function cannot always be deduced from its code: not all of the formal parameters may be effectively referenced, some may have their value changed but their value on input may never be read and, conversely, values of some global variables may be read by the code. Moreover, in the case of structured variables and pointers, it may only be the values of certain elements or fields that are read on input, or the values accessed by pointer de-references. This why PathCrawler currently calculates the set of all possible input parameters (fields, elements, de-references, etc of formal parameters and global variables of the program under test), which may contain many elements which are not in fact input parameters, and then asks for the user's help in reducing the set. PathCrawler also starts with the default input range of each input parameter given by its type declaration. For example, it is supposed that each integer input could take any value from -2^{31} to $2^{31}-1$. However, the user has the opportunity to reduce these ranges if the effective values of the inputs will always be much more restricted. Finally, the program may contain operations which will cause a runtime error if applied to certain values (e.g. division by zero). The user can specify a pre-condition (using a limited form of quantification in the case of array elements) to restrict input values to those which avoid such runtime errors or exclude other illegitimate program inputs. The output of the PathCrawler tool is a set of test inputs with the execution path covered by each.

PathCrawler is based on a novel approach to test case generation which is illustrated in Figure 1. It starts with an instrumentation of the source code so as to recover the symbolic execution path each time that the program under test is executed. The instrumented code is executed for the first time using a "test-case" which can be any set of inputs from the domain of legitimate values. PathCrawler recovers the corresponding symbolic path and transforms it into a path predicate which defines the "domain" of the path covered by the first test-case, i.e. the set of input values which would cause the same path to be followed (see Figure 2). The next test-case is found by solving the constraints defining the legitimate input values outside the domain of the path which is already covered. The instrumented code is then executed on this test-case and so on, until all the feasible paths have been covered. In Figure 2 SD_0 is the domain of legitimate inputs, t_1 is the first test case generated, PP_1 is the predicate of the path covered by t_1 , SD_1 is the difference between SD_0 and PP_1 , t_2 the second test case generated, PP_2 the predicate of the path covered by t_2 and SD_2 is the difference between SD_1 and PP_2 .

Figure 1 : the PathCrawler test generation process



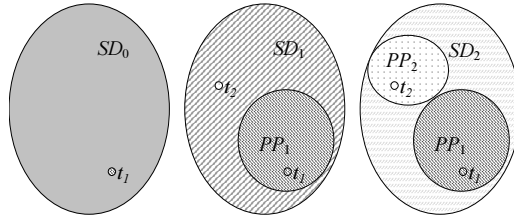


Figure 2 : Incremental coverage of the input domain

The instrumentation phase of the PathCrawler prototype has been implemented using the Cil library [6] and the rest using the Eclipse constraint logic programming environment [9]. It could be implemented to treat source code in any imperative programming language. The current prototype treats a wide range of ASCII C programs, which may include arrays and pointers but it cannot yet treat unions, pointers to functions and recursive functions. The inputs for each successive test case are found using constraint solving techniques.

For integer, Boolean and char variables constraint solving is NP-complete in the worst case, but PathCrawler uses heuristics which give much lower complexity in practice. Note that this is the complexity of the search for inputs for a given path if it is feasible, or of the determination of its infeasibility if not. Constraint solving can determine which paths are infeasible and so can automatically discover the maximum number of iterations of a loop (by determining the infeasibility of paths with too many iterations). This means that the user does not need to annotate loops with the maximum number of iterations.

Current constraint-solving techniques for floating-point variables model them using real numbers, which poses the problem of potential loss of precision during constraint resolution which could result in “solutions” to the constraints which are too imprecise to really satisfy the constraints, or in the solver declaring that no solution exists (the execution path is infeasible) when this is not the case. Also, constraint solving based on real numbers has a complexity which is quadratic for linear constraints but is undecidable in non-linear cases. However, current research on constraint solving for floating point numbers proposes using a finite representation in order to avoid these problems [5].

PathCrawler adopts an approach to test-case generation which combines static and dynamic analysis so as to avoid the problems encountered by other, purely static or dynamic, approaches. Unlike the dynamic approaches based on meta-heuristics such as genetic algorithms, which can require huge numbers of iterations or fail to terminate or to find inputs for a test-case which is in fact feasible, PathCrawler does find a test-case for each feasible path. Compared to approaches based on a global static analysis of the program, PathCrawler’s static analysis of a single, feasible path with unrolled loops is much simpler. This means aliases can be treated with relative ease. PathCrawler also avoids an enumeration of all the paths in the control flow graph, many of which are infeasible in practice. All infeasible path prefixes are detected and eliminated before being developed into a set of infeasible complete paths. The result is a very efficient generation of test inputs: for one example program described in [10] 20993 tests were generated and 15357 infeasible path prefixes detected in approximately 116 seconds of CPU execution time on a 2GHz PC running under Linux.

The next step to a solution : measuring fewer execution paths

Path testing avoids exhaustive testing of all inputs but some programs even have too many feasible execution paths to be able to measure all of them. This is why some hybrid approaches to WCET prediction propose a combination of decomposition of the program and measurement of the effective execution time of each component using techniques such as path testing [3, 7].

We are exploring a different approach, based on a modification of PathCrawler’s test generation strategy. It is not based on path decomposition but instead takes advantage of the fact that PathCrawler can be modified so as to decide dynamically not to generate inputs for certain paths, meaning that the excluded paths will not be tested. The decision can be based on information obtained beforehand (e.g. by static analysis of the control flow graph) or on information obtained dynamically (e.g. by additional instrumentation) when the program is run on the other test cases.

We would like to use this possibility to define a new search strategy for PathCrawler which cuts down the number of paths for which test cases are generated, but guarantees coverage of the path with the longest execution time, so that the WCET is still safe. The idea is to first modify the strategy in order to favour early generation of test cases covering the paths with the most instructions in the hope that these will include some of the paths with longer execution times. Before generating a test for each path prefix, PathCrawler would then determine which paths in the control flow graph could have this prefix. If any of those paths were sure to have a shorter execution time than an already measured path, then they could be excluded from test generation.

The problem is to identify the paths which certainly have a shorter execution time than a given path. Of course this is not always possible but the combinatorial explosion in the number of paths in a program can be

partly due to very minor differences between paths. Even in the presence of features such as memory caches and branch prediction, we can suppose that the execution time of a path depends on the instructions in the path, including the variables referenced by each instruction, and their order. The order of instructions may be modified during compilation or in super-scalar architectures but in some cases we should be able to suppose that it will be modified in the same way for the different paths. In these cases, a path which contains a subset of the instructions and variable references, in the same order, of another already measured path, cannot have more memory cache misses or bad branch predictions than the measured path. It therefore cannot have a longer execution time.

The most obvious example of this is two paths which are identical except in the number of iterations they perform of a loop with a variable number of iterations and with no branches in the body of the loop. Note that the parts of the path before, after and inside the loop must be identical. It seems safe to assume in this case that the path with the least number of iterations has a shorter execution time than the other one. If the execution time of the path with the most iterations has already been measured then there is no need to generate a test for the other path.

Other examples of common code constructions which result in paths which are very similar are :

- a) the implementation of the operation “max” : if ($a > b$) then $\max = a$ else $\max = b$;
- b) the implementation of a limit on a value : if ($a > \text{limit}$) then $a = \text{limit}$;

However, in these cases we may need to take into account the possibility of a bad branch prediction in one of the paths and not the other. We should also be able to identify paths which differ only by instructions which have the same execution time in the same context, e.g. $x = a + b$; and $x = a - b$;

Further study is needed to define a full set of conditions under which one path has a shorter execution time than another, in particular in the case of loops containing branches. Static analysis of the control flow graph can then be used to determine for each path in the graph those paths which have a shorter execution time, i.e. to impose a partial order on the paths in the control flow graph. The instrumentation and the test generation strategy of PathCrawler can then be modified to use the results of the static analysis to exclude paths from test generation. For example, to eliminate paths differing only in the number of iterations of a certain loop, we can annotate loop head instructions during instrumentation. PathCrawler’s strategy would first be modified to use these annotations to ensure that if the path covered by the previous test case contains a loop with a variable number of iterations then the next test generated covers a path with a prefix which increases the number of iterations of this loop. This would favour early generation of paths with the most loop iterations. Secondly, the strategy would use the results of the static analysis (in fact, the static analysis is trivial in this example) to exclude from future test generation all the paths identical to the already generated ones except for a lower number of loop iterations.

Experiments on different program examples will then be necessary in order to evaluate for which categories of program or micro-architecture a sufficient number of paths can be eliminated (in less time than it would take to generate tests for them) to effectively cut the combinatorial explosion in the number of paths.

References

1. Guillem Bernat, Antoine Colin and Stefan M. Petters, *WCET Analysis of Probabilistic Hard Real-Time Systems*, In proc. 23rd IEEE Real-Time Systems Symposium (RTSS02), Austin, Texas, December 2002
2. A. Gotlieb, B. Botella and M. Reuher, *A CLP Framework for Computing Structural Test Data*, CL2000, LNAI 1891, Springer Verlag, July 2000, pp 399-413
3. Markus Lindgren, Hans Hansson and Henrik Thane, *Using Measurements to Derive the Worst-Case Execution Time*, In Proc. 7th International Conference on Real-Time Computing Systems and Applications RTSCA 2000, Cheju Island, South Korea, December 2000
4. C. Michael and G. McGraw, *Automated Software Test Data Generation for Complex Programs*, In Proc. ASE, Oct 1998, Honolulu
5. C. Michel, M. Rueher and Y. Lebbah, *Solving Constraints over Floating-Point Numbers*, CP’2001, LNCS vol. 2239, pp 524-538, Springer Verlag, Berlin, 2001
6. G.C. Necula, S. McPeak, S.P. Rahul and W. Weimer, *CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs*, Proc. Conference on Compiler Construction, 2002
7. Stefan M. Petters and Georg Farber, *Making Worst Case Execution Time Analysis for Hard Real-Time Tasks on State of the Art Processors Feasible*, In Proc. 6th International Conference on Real-Time Computing and Applications RTCSA’99, December 1999
8. Christine Rochange and Pascal Sainrat, *Difficulties in Computing the WCET for Processors with Speculative Execution*, In Proc. 2nd International Workshop on Worst Case Execution Time Analysis, Vienna, June 2002
9. M. Wallace, S. Novello and J. Schimpf, *ECLiPSe: A Platform for Constraint Logic Programming*, IC-Parc, Imperial College, London, August 1997
10. Nicky Williams, Bruno Marre, Patricia Mouy and Muriel Roger, *PathCrawler: Automatic generation of path tests by combining static and dynamic analysis*, In Proc. EDCC-5, April 2005, Budapest