Automatic Timing Model Generation by CFG Partitioning and Model Checking *

Ingomar Wenzel, Bernhard Rieder, Raimund Kirner and Peter Puschner
Institut für Technische Informatik
Technische Universität Wien
Treitlstraße 3/182/1, 1040 Wien, Austria
{ingo,bernhard,raimund,peter}@vmars.tuwien.ac.at

Abstract

In this paper we present a new measurement-based worst-case execution time (WCET) analysis method. Exhaustive end-to-end measurements are computationally intractable in most cases. Therefore, we propose to measure execution times of subparts of the application. We use heuristic methods and model checking to generate test data, forcing the execution of selected paths to perform runtime measurements. The measured times are used to calculate the WCET in a final computation step. As we operate on source code level our approach is platform independent except for the run time measurements performed on the target host.

We show the feasibility of the required steps and explain our approach by means of a case study.

1 Introduction

Due to the temporal constraints required for correct operation of a real-time system, predictability in the temporal domain is a stringent imperative to be satisfied. Therefore, it is necessary to determine the timing behaviour of the tasks running on a real-time computer system. Worst-case execution time (WCET) analysis is the research field investigating methods to assess the timing behaviour of real-time tasks.

A central part in WCET analysis is to model the timing behaviour of the target platform. However, hardware modelling is a time-consuming task for modern processor hardware [6]. In order to avoid this effort and address the portability problem in an elegant manner, we developed a new hybrid WCET analysis approach performing execution time measurements on the instrumented application executable to obtain the required hardware timing model [10]. From the approach presented in [10], two challenges arise: (i) the granularity of the items subject to measurements has to be determined and (ii) the measurement subsystem has to force the corresponding measurements on the target hardware.

The first contribution of this paper is the development of an automatic control flow graph (CFG) partitioning method to identify the units subject to these measurements (Section 2). Second, we introduce model checking for generating test data that allows the systematic measurement of the selected paths. We show that there exists a reasonable tradeoff between the number of instrumentation points and the number of measurements necessary in order to cope with the complexity inherent in real-time program code (Section 3). The practical applicability of our method is illustrated by a case study from the automotive domain. It consists of an application automatically generated by the *TargetLink* code generator from *dSpace* [5] (Section 4).

2 Control Flow Graph Partitioning

In this section, we present our method for automatically partitioning the control flow graph into these subparts that will later be used as smallest granularity items in our execution time model.

2.1 Basic Concepts

Before proceeding, we introduce some terms.

A *basic block* denotes a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end, without the possibility of branching except at the end of the basic block [1].

The *control flow graph* (CFG) is a directed graph

*This work has been supported by the FIT-IT research project "Model-based Development of distributed Embedded Control Systems (MoDECS-d)".

that consists of linked basic blocks with one distinguished *initial node* (whose first instruction is the first instruction in the program).

Due to the fact that it is impossible to use end-to-end measurements covering all paths, we partition the code using the notion of *program segments* (PS). A PS is a subgraph of the CFG that can be entered only via the transition of a single control edge, multiple exit edges are possible. A *structured program segment* (SPS) is a PS that has only a single exit edge.

After partitioning the program into program segments, *instrumentation points* are introduced before and after the program segments. Execution time measurements of individual PS are used to calculate the WCET bound of the whole program.

2.2 Partitioning the Control Flow

The CFG is partitioned into PS following the abstract syntax tree. This partitioning is then used to determine the necessary instrumentation points and the number of measurements needed to evaluate the program. During the partitioning step the CFG is traversed hierarchically top-down. The decision whether a PS is analysed as a whole (i.e., all paths of the PS are measured by instrumenting the PS at its beginning and end) depends on the number of paths inside the PS. Whenever a PS is identified to be analysable as a whole, no further analysis of subordinated PS is performed.

The partitioning algorithm works as follows. As a starting point, the whole function subject to the analysis process is considered as one PS.

Whenever the number of paths within a PS is less or equal some given bound b, measurement points are inserted at the beginning and the end of the PS. Thus, the number of instrumentation points ip increases by 2 and the number of measurements m is increased by the number of paths within that PS.

If the number of paths within the PS is higher than path bound b then the PS is decomposed and the execution times of its constituents (nested PS; the smallest unit of PSs are basic blocks) are measured.

Figure 1 presents an example code listing with its associated CFG. The nodes in the CFG of Figure 1 are labelled with the line numbers of the first instruction of the respective basic block.

Table 1 shows the estimated measurement effort for the analysis of the sample program. For bound $b = 1$ the number of instrumentation points equals 22 since every single basic block is measured. For $b = 2$, an `if` statement containing 2 paths is measured as a whole PS. Thus, the four basic blocks having the *id* values

```
1  int main() {
2      int i;
3      printf1();
4      printf2();
5      if (i==0)
6      {
7          printf3();
8          if (i==0) {
9              printf4();
10         } else {
11             printf5();
12         }
13     }
14     if (i==0)
15     {
16         printf6();
17         printf7();
18     }
19     printf8();
20 }
```

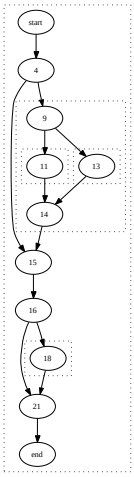

Figure 1. Example listing with CFG

Bound b	Instr. Points ip	Measurement m
1	22	11
2	16	9
3	16	9
4	16	9
5	16	9
6	2	6
7	2	6

Table 1. Measurement effort with different path bound b

$6, 3, 4, 5$ need not to be instrumented (this reduces ip by $4 * 2 = 8$), but the PS between node 4 and 15 is instrumented by 2 instrumentation points. Thus, we get $ip = 22 - 8 + 2 = 16$. The number of measurements equals $m = 11 - 4 + 2 = 9$ in this case. Finally, when the path bound is increased to 6, all paths in the function are covered by measurements. In this case, only two instrumentation points and 6 measurements are necessary.

2.3 Tradeoff between Instrumentation Points and the Number of Measurements

We examined the relationship between the path bound b, the number of instrumentation points ip and the number of measurements m when using an existing automotive application code which is generated by the *TargetLink* code generator [5] out of *Matlab/Simulink* models. The source files of this application, with all

include files resolved, have an average size of approximately 5000 lines of code, the analyzed functions have around 800 basic blocks and about 300 conditional branches.

Figure 2 outlines the relationship between the path bound b and the number of instrumentation points ip (note that the scaling of the x-axis is logarithmic). For $b = 1$ the number of instrumentation points equals the doubled number of nodes in the CFG (i.e., $857 * 2 = 1714$), which is due to the stand-alone instrumentation of each basic block. When incrementing the path bound b, the number of instrumentation points decreases. Considering the right tail of the line, even huge increments of the bound b result only in minor instrumentation point reductions.

It is important to bear in mind that this plot – as well as all the subsequent ones – depicts the situation for a specific piece of code.

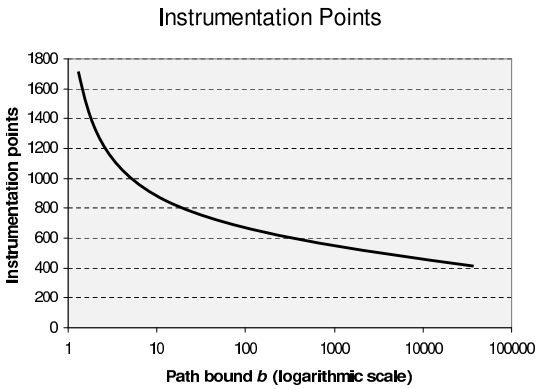

Figure 2. Number of instrumentation points over path bound b

The relation between the number of instrumentation points ip and the number of measurements m is depicted in Figure 3. From higher to lower numbers of instrumentation points an explosion in the number of required measurements can be observed. End-to-end measurements would be performed at the point where $ip = 2$, increasing m to an computationally intractable value. The arrows inside the diagram outline the changing costs between instrumentation and measurement.

Our first implementation of a simple code partitioning algorithm was able to keep the number of instrumentation points as low as 500[1].

We are currently extending the CFG partitioning algorithm to produce a general PS partitioning. This is expected to result in improvements in the number of instrumentation points at low measurement cycle costs.

[1]When using "intelligent" instrumentation, this number might be considered to be $500/2 + 1 = 251$ by fusing two consecutive instrumentation points.

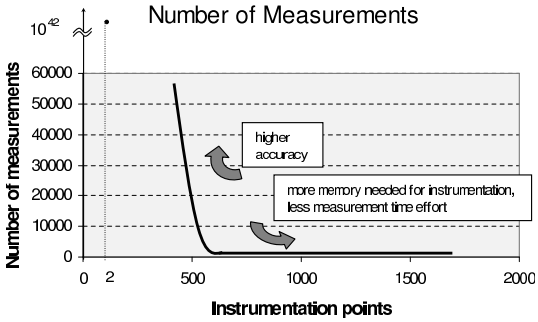

Figure 3. Measurement cycles and instrumentation points

3 Test Data Generation

From the static code analysis performed during the control flow partitioning the paths to be measured are known. To perform measurements of selected program segments we need test data that lead to the execution of the selected path segments.

A method of generating test data is model checking, see [9]. If there exists a test data pattern that leads to the execution of a distinct path it will always be found with model checking. The drawback of model checking is its calculation cost. For this reason a hybrid approach has been chosen: first, test data are generated using heuristic methods (i.e. genetic algorithms) until a given coverage bound is reached. A possible bound could be that no new paths have been reached with the last 10^6 generated data patterns. The bound is necessary to stop calculation at a point where the costs to find new paths with heuristic methods equal the costs of finding new paths using model checking. According to [11] we expect heuristic methods to generate more than 90% of the required test cases. In a second step the remaining test data are generated using model checking. If no data pattern is found for a selected path the path is deemed infeasible.

We use the *SAL* model checker from SRI Labs [4] for test data generation. For this purpose, the C source code is converted into the *SAL* Language.

3.1 State Space

An important goal for the C-to-*SAL* conversion is to keep the state space of the model as small as possible. Assuming a model with n variables x_i with domain D_i, the domain of the state vector $\vec{x} = (x_1, \ldots, x_n)$ is

$D = D_1 \times D_2 \times \ldots \times D_n$. The reachable state space D_R is a subset of D. D_R depends on the set of initial values of the variables, which is denoted as D_I. According to the communication from a developer of the model checker SAL, the number of bits required to encode the state vector $\vec{x}$ should not exceed 700 to get acceptable performance on currently available personal computers.

Direct conversion without any semantic knowledge is very inefficient: In C, boolean values are mostly encoded as 16 bit integers where a single bit would suffice. If boolean values are encoded in the SAL model in the same way, i.e. as integers, about 44 boolean values would fill up the complete state space, whereas 700 booleans could be processed if a single-bit encoding were used.

3.2 Optimisations to reduce the State Space

Basically there are two properties that affect the efficiency of a model: The size of the state space and the number of transitions that are required to reach the goal (i.e. to generate a counter example). We use optimisations to reduce either of these values. The optimisations presented in this paper not only increase the performance of the model checker but also allow model checking of programs that would be too big otherwise. The presented optimisations do not change the model. Instead, we use these optimisations to make the representation of the model more compact, thus reducing the state space and increasing the model checker performance.

Some of the techniques presented here are also applied in compiler technology and are comprehensively described in [1]. The majority of optimisation techniques described in [1] introduce new temporary variables. When they are used for model checking this would increase the state space and therefore degrade the performance.

3.2.1 Reverse CSE

This optimisation is the contrary to *Common Subexpression Elimination (CSE)* known from compilers.

Temporary variables containing intermediate results are replaced by the values that are assigned to them. For instance, the sequence `a=b+1; c=a+b; d=a*2;` is replaced by `c=(b+1)+b; d=(b+1)*2;` The performance loss from recalculating the subexpression is small compared to the gain from the reduced state space.

3.2.2 Live-Variable Analysis

This is a common optimisation technique for compilers that can directly be applied to model checking. Multiple variables can share the same memory location if they are not used at the same time. A variable is "alive" if a value assigned by a declaration is used by a subsequent statement (definition-usage-pairs). The effect of this optimisation technique on model checking depends highly on the structure of the source code but it will always shrink the state space and improve performance, unless the unlikely case occurs that no variables can be reused. This optimisation technique is also used to remove unused variables.

3.2.3 Statement Concatenation

The basic idea is to combine as many C statements as possible into a single *SAL* block, thus reducing the number of transitions to be executed by the model checker. If a model contains two C statements in each transition it requires only half the number of transitions compared to a model of the same program that contains only a single statement in each transition. The prerequisite for this optimisation is that the variables in the C statements are independent.

3.2.4 Variable Range Analysis

As mentioned above, analysing the value range of a variable can be used to keep the number of bits used for their representation small (1 bit vs. 16 bits for boolean expressions). For this optimisation a semantic analysis is necessary. This applies not only for boolean variables but also for integer variables as the model checker does not limit the variable size to multiples of eight bits. For integer variables not used as booleans this might require assistance from the user or, preferable, annotations made by the code generator. The code generator will have this information from the *MatLab/Simulink* model in most of the cases.

3.2.5 Variable Initialisation

All variables contained in the model that are not input variables are uninitialised. The model checker can assign any value to them and therefore produces a large set of D_I. Assigning initial values to these variables does not reduce the size of the state space $|D|$ but speeds up model checking by reducing $|D_R|$.

3.2.6 Dead Variable and Code Elimination

Since we are not interested in the data flow but only in the control flow, all variables that do not affect the

control flow directly or through assignments to other variables can be removed. Even code segments that do not affect variables involved in the control flow can be removed, as long as we are not looking for test data to reach these paths. This optimisation has a direct impact on the state space since it reduces the number of variables.

3.3 Evaluation of Optimisation Techniques

The current version of our C to *SAL* conversion tool does not yet support all the described optimisations. The structure of the example optimisations had to be manually applied to the generated *SAL* model. As hand-optimising the generated *SAL* model is a laborious and error prone task we used a short example code for this experiment instead of our real case study. The C source code for the evaluation consists of 105 lines without comments and empty lines, four boolean and thirteen byte variables from which three can be substituted by "Reverse CSE", three are not affecting the control flow and three are not used at all.

Table 2 shows the impact of the individual optimisations on the simulation time and on the memory footprint of the model checker. The last column shows the number of simulation steps that were performed. All evaluated optimisation methods except "combination of subsequent statements" and "initialising of undefined variables" reduce the state space directly by reducing the number of bits used for variable representation. It can be seen that reducing the state space has the strongest impact, especially the variable range analysing. By default all variables created by our C to SAL translator are 16 bit signed integers and therefore the usage of 8 bit signed integers or booleans, which are expressed with a single bit, cuts down the bits used for variable representation down to less than the half.

optimisation technique	simul. time [s]	memory use [kb]	steps
unoptimized	283.4	229,360	28
all optimisations used	2.2	26,580	13
Variable Initialisation	172.7	173,334	28
Variable Range Analysis	12.7	59,492	28
Reverse CSE	25.3	71,620	26
Statement Concatenation	22.5	61,444	18
Dead Variable Elimination	44.2	99,444	28
Live-Variable Analysis	10.8	41,856	28

Table 2. Impact of optimisations on model checking time and memory usage

Combining subsequent statements reduces the number of necessary transitions to about 65% of the original value, thus the model checker can find the results faster. This is an explanation for the lower time effort but not for the smaller memory footprint. The smaller memory footprint seems to be based on internal optimisations of the model checker. Obviously, the *SAL* model checker can reduce the state space when transitions are combined in the model.

From other experiments we have seen that the simulation time and memory footprint rises exponentially with the number of variables. Therefore, we expect the effect of these optimisations to be even more significant with larger programs. On the other hand, all optimisations in this evaluation are done by hand. It is reasonable to expect the optimisations done by a tool not to reach the same quality as when done by hand. This will result in a little degradation of the optimisations.

Other projects researching model checking for ANSI-C programs that should be mentioned for the interested reader are CBMC [3], the BOOP Toolkit [12], BLAST [7], SPIN [8], and SLAM [2].

4 Case Study

Mainly, we investigated the concepts of our approach on several real-sized industrial applications. The most important results we gained from this work have been incorporated in Section 2 and Section 3. However, due to intellectual property issues we cannot publish these applications. To provide a convenient view on the big picture of our approach, we use a case study that is a magnitude smaller but has a similar structure to the original application. Also an almost identical design process and the same employment background as the industrial applications is used.

Our case study is an automotive wiper control application. The controller inputs are a two-step speed selector (off, slow and fast) for the wipers, a button to switch on the water pump and an end position switch to indicate the neutral position of the wipers.

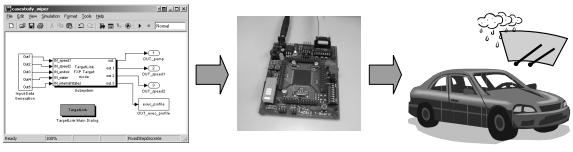

Figure 4. Case Study

The design flow is illustrated in Figure 4. We modelled the application in *MatLab/Simulink* . The *Stateflow* chart has 9 states, the complete *MatLab/Simulink*

model contains around 70 blocks. Using the *TargetLink* code generator we created the application code automatically. The whole functionality is encapsulated in a single function `wiper_control`. This function is subject to our analysis method.

We instrumented the application with instructions starting and stopping an internal cycle counter register. Basically, the code consists of nested `switch` and `if` statements. Using the generated test data we enforce the execution of particular paths and measure the execution time of each `case` statement block.

Then the application is compiled and linked for the *Motorola HCS12* platform and uploaded to the evaluation board. To enforce our test data on the input parameters and the state of the application we use the glue code functionality of *MatLab/Simulink* and *TargetLink* involving minimal user interactions for the transfer of the obtained execution time values.

Due to the small input space we could also evaluate the WCET to 250 cycles in exhaustive end-to-end measurements. But we partitioned the application so that each `case` block equals one PS and calculated a WCET bound of 274 cycles using the measured execution times and a simple timing schema approach. We did not apply any optimisations like semantic analysis or considering hardware effects to obtain this level of quality. One advantage of our method is that even if no semantic analysis is attached to the analysis process, many semantic dependencies are covered implicitly by the systematic partitioning.

5 Conclusion

In this paper we presented the results of the application of our measurement-based WCET analysis method described in [10].

We have shown that it is possible to reduce the high complexity inherent in real-sized application code by applying a simple partitioning algorithm (Section 2).

We showd how model checking can efficiently be used to generate test data to execute individual paths of an application and can guarantee that test data patterns can be found to reach all feasible paths. This method can be used for testing because various structural code coverage criteria may be satisfied using this approach. Driven by the huge state spaces arising from the analysis of complex applications we introduced a number of new highly efficient optimisation techniques and presented a detailed evaluation of their capabilities.

Finally, we investigated a number of real automotive applications (Subsection 2.3) and illustrated the complete application of the WCET analysis method in a clearly arranged case study (Section 4).

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers - Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] T. Ball and S. K. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. In *Workshop on Model Checking of Software, SPIN 2001*, pages 103–122, 2001.

[3] E. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[4] L. de Moura, S. Owre, H. Ruess, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari. SAL 2. To be presented at CAV 2004, jul 2004. Available at `http://www.csl.sri.com/~rushby/abstracts/sal-cav04`.

[5] dSpace GmbH. dSpace TargetLink. Available at `http://www.dspace.com`.

[6] R. Heckman, 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*, volume 91:7, pages 1038–1054, July 2003.

[7] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *Proceedings of the 29th Annual Symposium on Principles of Programming Languages*, pages pp. 58–70. ACM Press, 2002.

[8] G. J. Holzmann. The Model Checker Spin. In *IEEE Transactions on Software Engineering, Vol. 23, No. 5*, pages 279–295, 1997.

[9] H. S. Hong, S. D. Cha, I. Lee, O. Sokolsky, and H. Ural. Data Flow Testing as Model Checking. In *IEEE 25th International Conference on Software Engineering*, pages 232–242, 2003.

[10] R. Kirner, P. Puschner, and I. Wenzel. Measurement-Based Worst-Case Execution Time Analysis using Automatic Test-Data Generation. In *WCET'04 Proceedings*, 2004. To appear.

[11] N. Tracey, J. Clark, J. McDermid, and K. Mander. *Systems Engineering for Business Process Change: New Directions*, chapter A search-based automated test-data generation framework for safety-critical systems, pages 174 – 213. Springer-Verlag New York, 2002.

[12] G. Weißenbacher. An Abstraction/Refinement Scheme for Model Checking C Programs, 2003.