Safe measurement-based WCET estimation

Jean-François Deverge and Isabelle Puaut
Université de Rennes 1 - IRISA
Campus Universitaire de Beaulieu
35042 Rennes Cedex, France
{Jean-Francois.Deverge|Isabelle.Puaut}@irisa.fr

Abstract

This paper explores the issues to be addressed to provide safe worst-case execution time (WCET) estimation methods based on measurements. We suggest to use structural testing for the exhaustive exploration of paths in a program. Since test data generation is in general too complex to be used in practice for most real-size programs, we propose to generate test data for program segments only, using program clustering. Moreover, to be able to combine execution time of program segments and to obtain the WCET of the whole program, we advocate the use of compiler techniques to reduce (ideally eliminate) the timing variability of program segments and to make the time of program segments independent from one another.[1]

1. Motivation

Computation of WCET is an important issue for hard real-time systems. Common approaches for WCET computations deal with static analysis of program structures. They rely on hardware models to produce execution time estimations. Latest processors have performance increasing features like caches, branch predictors or multiple-issue pipelines that maintain an internal state that is difficult to predict. As a consequence, these complex hardware models are harder and harder to design [7], leading to *safe* but *pessimistic* estimations.

An alternative approach is to use measurements on real hardware (or a cycle accurate simulator) to obtain WCET estimates. However, exhaustive enumeration of all program inputs is intractable for most programs. Heuristics, like evolutionary algorithms [16], might be used to generate input test data that may cover the worst case path of the program. While yielding realistic WCET estimations, there is *no guarantee* to measure the worst case execution path of the program. Therefore, these methods have almost been used to increase confidence of static WCET analysis methods only [13].

On one hand, program testing may produce unsafe but *realistic* results. On the other hand, static WCET analysis approaches produce *safe* but pessimistic WCET estimations. However, safe and tight estimations of the WCET are highly desirable. Ideally, one would desire WCET tools that produce safe and tight results without harness development of timing models for the next generation processors.

This paper explores the issues to be addressed to design a measurement-based method that produces safe results. We propose to rely on structural testing [20] methods to generate input test data and to exhaustively measure the execution time of program paths. We advocate the use of compiler techniques to reduce (ideally eliminate) the timing variability of program measurements. In Section 2, we outline our method for WCET timing analysis and we give some properties on hardware measurements our method relies on. Section 3 describes how the properties are met, through the control of the unpredictability of some hardware mechanisms, and contains some preliminary results of path measurements on a PowerPC 7450. Related work, some concluding remarks and directions of our ongoing work are given in Section 4.

2. Method outline

One would obtain the program's WCET by measuring all program executions with any of the possible input data for this program. However, exhaustive enumeration of a *program input* is unfeasible for most programs. Another approach is to measure *all paths* of the program. This reduces the number of measurements because a set of possible input data may activate the same program path. However, the path coverage is impracticable for program with unbounded loops, yielding an infinite number of paths [20].

In this paper, we propose to employ structural testing methods [8, 18, 20] to automatically generate input data. These methods generate the input data set from the analysis of the program structures.

A key assumption we make is that the measurement of the executions of the same program path, with different data values, yields the same timing results. Meeting this assumption requires to control the hardware: this issue is discussed in Section 3.

Program clustering. Test data generation methods are mostly based on equations [8] or constraint solving techniques [18]. Due to solver tools and their potentially lack of

[1]This paper was previously published in *Proceedings of the 5th International Workshop on Worst Case Execution Time Analysis, pages 7-10, Palma de Mallorca, Spain, July 2005.*

scalability, the analysis of complete paths of the whole program could be unachievable in practice. Moreover, number of paths could be exponential even for small program. As a consequence, we suggest splitting paths into *segments* to lower the complexity of test data generation.

An example, the program of Figure 1 contains 2^{SIZE} paths. For small values of $SIZE$ (for example $SIZE < 10^4$), it is conceivable to exhaustively make measurements of this code.

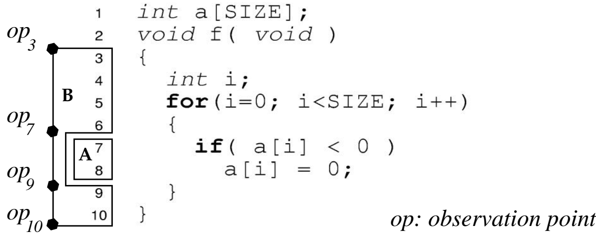

Figure 1. Path clustering.

However, for greater values of $SIZE$, it would be suitable to split the program and apply measurements on segments. This process is called program *clustering*. An intuitive solution to tackle test data generation complexity of the code of Figure 1 is to build two clusters A and B. In this configuration, there are only two paths in cluster A and a single path on cluster B.

Application of clustering on the program works as follows. The automated test data generation is first applied to the whole program. If it produces too many results or if it does not terminate before a limited amount of time, we stop it. We then launch test data generation on smaller parts of the program (e.g. sub trees of the program syntax tree). This iterative process is repeated until segments are small enough to make exhaustive path enumeration inside a segment tractable. We obtain leaf cluster like A.

Program measurement. We focus on the exploitation of program measurements but we don't address methods to obtain program execution times: there exist multiple hardware and software methods described in [11, 14]. *Observations points* provide execution traces and give the execution times of the program units observed [11]. In our approach, we have to place observations points at the cluster boundaries. For example, there are four observation points op_3, op_7, op_9 and op_{10} on Figure 1.

We first measure the two paths of cluster A and we obtain values 70 and 95 cycles for instance. The worst value is the WCET of the cluster A and $WCET_A$ is 95. Then, we execute and we measure the single path of cluster B. Table 1 contains the measurement trace of this execution.

The value of $TIME_B$ is not the WCET for the whole program, because this execution could have covered the shortest path of cluster A. Consequently, we have to add the difference between $WCET_A$ and each $TIME_A$ measured during the execution of cluster B. In this way, we obtain an *upper bound* of the global WCET. Program clustering enables automated test data generation on subprogram paths or *program segments*. The longer the program segments will be, the tighter the WCET estimation will be.

Observation point	Time stamp	Observation interval
op_3	0	
op_7	15	
op_9	85	$TIME_A = 70$
op_7	105	
op_9	200	$TIME_A = 95$
...	...	
op_7	557585	
op_9	557655	$TIME_A = 70$
op_{10}	557695	$TIME_B = 557695$

Table 1. Measurement trace of the path execution of the cluster B.

In this section, we have proposed to assemble WCET of leaves clusters using measurement. We could also investigate for hybrid approach that couples testing and static WCET analysis. In such an approach, we should measure program segments using testing methods and we should use static methods to combine these context-independent segments timings.

3. Obtaining safe program segment measurements

In previous section, we have assumed that any measurements of the different executions of the same program path gives the same results. In this section, we focus on obtaining such *safe* and *context-independent* measurements.

There are three main sources of unpredictability in complex processor architectures:

1. *Global mechanisms*, like caches, virtual memory translation (TLB) or branch predictors. Their internal state and the contents of their tables have direct impact on the execution time of future instructions of the whole program [5, 9].

2. *Variable latency instructions*. Some operations, as the integer multiplication instruction, may have variable timing behaviour because the result should be computed faster on small valued input data operands.

   Processor may partially implement some operations, as the float division or the square root instruction. This means that, in order to support unimplemented operations in hardware, an exception is raised and operation should be computed by an exception handler provided by the operating system.

3. *Statistic execution interference* phenomenon [12], due to unpredictability introduced by DRAM refresh. Similarly to variable latency instructions, load/store operations to the main memory may have varying timing behaviour. Moreover, processors have a built-in multiple level cache hierarchy, and some cache clock speeds may be different to the clock speed of the core processor. A tiny deviation on timings may occur if a load request is received immediately or on the next clock cycle of the slower cache level.

Gaining control of processor unpredictability. Obtaining safe and context-independent measurements requires to eliminate (or at least drastically decrease) the sources of timing variability. For that purpose, we are currently considering a few approaches relying on hardware control and compilation methods.

Regarding the first source of unpredictability, global mechanisms might be disabled or we should clear their history tables before the execution of each program segment. Cache conscious data placement [4] and cache locking [15] reduce varying timings of memory accesses. Likewise, static branch prediction enable to fix behaviour of speculative execution at compilation time [3].

In order to support variable latency operations, [19] proposes to add the difference between the BCET and the WCET of all the operations of the program path. Another approach consists in avoiding the use of these operations and to replace them by predictable instructions.

We could forbid the varying timing behaviour of partially unimplemented instruction by disabling the execution of the exception handler. However, this may affect operational semantics of instructions [10]. It should be preferable to rewrite temporally predictable exception handler and to apply the same strategies as those applied for variable latency operations.

It is not possible to control variability on latency of memory access. However, we feel that such a fluctuation in measurements follows a true statistical distribution. Models to quantify pessimism to apply on results of measurements are related in [2]. In addition, variability of load/store operations latency may be due to the input-dependent memory accesses of the program.

```
1  int a[SIZE], b[SIZE], c[SIZE];
2  void f( void )
3  {
4    int i;
5    for( i=0; i<SIZE; i++ )
6    {
7      a[i] = c[ b[i] % SIZE ];
8    }
9  }
```

Figure 2. Single path program with unpredictable timings of data access.

The sample code from Figure 2 is a single path program. Nevertheless, the number of cache misses on array *c* depends on the contents of *b*. To make this code temporally deterministic, we could disable the cache feature before the memory access to contents of *c* [15]. We could also set the whole array *c* as non cachable introducing program performance penalty. In order to enhance data access latency, we could employ data cache locking [15] or to do scratchpad memory allocation [1] of data subject to *unknown* memory access patterns.

Preliminary experiments. In order to evaluate if the timing variability of program segments can be controlled by software, we conducted experiments on a PowerPC 7450 processor [10]. This 32-bit processor is able to dispatch 3 instructions per cycles on an in-order, seven stage pipeline. It features two dynamic branch prediction mechanisms: a 2-bit prediction scheme with a branch target buffer, and a return stack predictor. Our chip has a 64-Kbyte level-one (L1) cache, and a 256-Kbyte L2 cache. A load will take 3 cycles if the data is in the L1 cache. There is a maximum latency of 9 processor cycles for L1 data cache miss that hits in the L2.

For our preliminary evaluation, we evaluated the impact of hardware control on the execution time of a program segment (*SNU-RT jfdctint*) made of a single path. We achieved these experiments in isolation from asynchronous activities by disabling operating system's context switches and disabling external interrupts.

Figure 3 shows the timings of two sets of 25 measurements. Before each *jfdctint* execution measurement, we first executed one of twenty-five pollution codes: the program itself, random generator, load and writes of big amount of memory, intensive control code, and some code taken from `http://www.c-lab.de/home/en/download.html`.

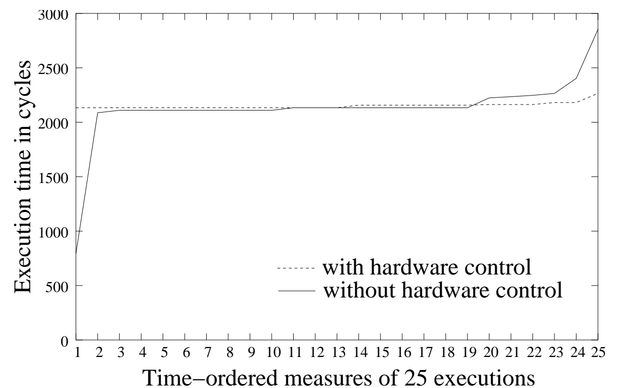

Figure 3. Measurements of jfdctint execution times.

The first set of measurements are made with hardware control. After execution of the pollution code, we have cleared branch predictor buffers, and we have flushed caches (TLB, L1 and L2). The second set of measurements is made without any hardware control.

We can note that the variability of program running times is largely reduced with hardware control. We observe that measurement variability is decreased from 2000 cycles to 150 cycles. Without hardware control, the best case execution time is obtained after the execution of the program itself (warm caches effect). The worst case execution time is due to a pollution code that fills the entire data cache with dirty lines. Consequently, for many data accesses of the measured program, the processor had to update the memory with the victim cache contents before its replacement with program data.

We have investigated the sources of variability on measurement with hardware control. The memory performance-monitoring counters on the PowerPC 7450 reveal that main sources of variability, for program measurements with hardware control, are due to main memory and L2 variable access latency.

It can be remarked that average program running times are almost the same with or without hardware control.

There is no performance degradation on that specific experiment because we did neither deactivate the cache nor the branch predictor. This suggests that *long* program segments can take advantage of dynamic mechanisms if history tables or related internal states could be cleared before execution.

4. Conclusion and future work

In this paper, we have proposed to compute the WCET from execution measurements. We advocate the use of structural testing methods and program clustering to enable measurements of the worst case execution path. This measurement-based approach would produce *safe* and *tight* results.

Recently, the use of another software unit test approach has been proposed in [17]. Model checking methods produce input data to exhaustively cover paths of automatically generated programs from MatLab/Simulink specifications. This approach enables to measure WCET of straight-line C programs with no loops.

Previously, [19] has used data flow analysis to detect single feasible path segments of the program. In their approach, only single path segments are measured, and static WCET analysis is employed on the rest of the program. [19] gives conditions to obtain safe measurements on processors with cache.

Clustering techniques have been applied to static WCET analysis methods to enhance their scalability [6]. The clustering is applied on the syntax tree of the program and the main criterion used is a limit on the number of generated constraints. We propose to apply a similar strategy in our approach, our objective being to reduce the complexity of test data generation.

Traditional static WCET analysis and measurement are combined in [2]. There is no control of the hardware and statistical models are applied, thus providing a probabilistic safety on the global WCET [2]. The combination of test data generation methods and these techniques would represent a fruitful area of study.

Our method has to control any processor features like cache or branch prediction to reduce the unpredictability of these advanced processors mechanisms. We plan to further study the balance between hardware control, necessary yielding negative performance impact on execution time, and the benefit with respect to measurements variability.

References

[1] O. Avissar, R. Barua, and D. Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM Transactions on Embedded Computing Systems*, 1(1):6–26, Nov. 2002.

[2] G. Bernat, A. Colin, and S. M. Petters. WCET analysis of probabilistic hard real-time system. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, pages 279–288, Austin, TX, Dec. 2002.

[3] F. Bodin and I. Puaut. A WCET-oriented static branch prediction scheme for real time systems. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, pages 33–40, Palma de Mallorca, Spain, July 2005.

[4] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 139–149, San Jose, CA, Oct. 1998.

[5] J. Engblom. Analysis of the execution time unpredictability caused by dynamic branch prediction. In *Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 152–159, Toronto, Canada, May 2003.

[6] A. Ermedahl, F. Stappert, and J. Engblom. Clustered calculation of worst-case execution times. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 51–62, San Jose, CA, Oct. 2003.

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

[8] G. Lee, J. Morris, K. Parker, G. A. Bundell, and P. Lam. Using symbolic execution to guide test generation. *Software Testing, Verification and Reliability*, 15(1):41–61, Mar. 2005.

[9] T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, pages 12–21, Phoenix, AZ, Dec. 1999.

[10] *MPC7450 RISC microprocessor family processor manual revision 5*. Freescale Semiconductor, Jan. 2005.

[11] S. M. Petters. Comparison of trace generation methods for measurement based WCET analysis. In *Proceedings of the 3rd International Workshop on Worst Case Execution Time Analysis*, pages 61–64, Porto, Portugal, June 2003.

[12] S. M. Petters and G. Färber. Making worst case execution time analysis for hard real-time tasks on state of the art processors feasible. In *Proceedings of the 6th International Workshop on Real-Time Computing and Applications Symposium*, pages 442–449, Hong Kong, China, Dec. 1999.

[13] P. Puschner and R. Nossal. Testing the results of static worst-case execution-time analysis. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 134–143, Madrid, Spain, Dec. 1998.

[14] B. Sprunt. The basics of performance-monitoring hardware. *IEEE Micro*, 22(4):64–71, July 2002.

[15] X. Vera, B. Lisper, and J. Xue. Data cache locking for higher program predictability. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 272–282, San Diego, CA, June 2003.

[16] J. Wegener and M. Grochtmann. Verifying timing constraints of real-time systems by means of evolutionary testing. *Real-Time Systems*, 15(3):275–298, Nov. 1998.

[17] I. Wenzel, B. Rieder, R. Kirner, and P. Puschner. Automatic timing model generation by CFG partitioning and model checking. In *Proceedings of the Conference on Design, Automation, and Test in Europe*, pages 606–611, Munich, Germany, Mar. 2005.

[18] N. Williams, B. Marre, and P. Mouy. On-the-fly generation of k-path tests for C functions. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, pages 290–293, Linz, Austria, Sept. 2004.

[19] F. Wolf, R. Ernst, and W. Ye. Path clustering in software timing analysis. *IEEE Transactions on Very Large Scale Integration Systems*, 9(6):773–782, Dec. 2001.

[20] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, Dec. 1997.