IST-004527 ARTIST2
http://www.artist-embedded.org/FP6/

http://www.cordis.lu/ist/home.html

5TH INTERNATIONAL WORKSHOP ON WORST-CASE EXECUTION TIME ANALYSIS (WCET2005)

Satellite Event to ECRTS'05, Palma de Mallorca, Spain

Reinhard Wilhelm (Workshop Chair)

Message from the Workshop Chair

You have in front of you the proceedings of the 5th International Workshop on Worst-Case Execution Time (WCET) Analysis. The workshop was held on the 5th of July 2005 as a satellite event to the 17th Euromicro Conference on Real-Time Systems (ECRTS 2005) in Palma de Mallorca, Spain.

It was the fifth event in the series after the successful meetings in Delft (Holland) in 2001, Vienna (Austria) in 2002, Porto (Portugal) in 2003 and Catania (Italy) in 2004. The goal of these workshops is to bring together people from academia, tool vendors and users in industry that are interested in all aspects of timing analysis for real-time systems. The workshops provide a relaxed forum to present and discuss new ideas, new research directions, and to review current trends in this area. It consisted of short presentations that should encourage discussion by the attendees. The topics of the 2005 workshop included paper on the following topics:

- Measurement-based timing-analysis methods,
- Experience from industrial case studies,
- Architectural issues, and
- Timing analysis in real-time education.

In addition, there was an invited talk by Lothar Thiele, ETH Zuerich, on Composable Real-Time Analysis. There is no paper about this talk contained in the proceedings.

The industrial case studies showed that the techniques have matured to industrial applicability. Better results are achieved if the methods and tools are integrated into the development process. Measurement-based methods were controversially discussed. Further talks showed that much support is needed to deal with architectural features that endanger timing predictability.

Reinhard Wilhelm (Workshop Chair)

Program Committee

- Andreas Ermedahl (Univ. Mälardalen, Sweden)
- Tulika Mitra (National University of Singapore)
- Isabelle Puaut (University of Rennes/IRISA)
- Lothar Thiele (ETH Zuerich, Switzerland)
- David Whalley (Florida State University, USA)
- Reinhard Wilhelm (Saarland University, Germany)

Steering committee

- Guillem Bernat, University of York. England UK, bernat@cs.york.ac.uk
- Jan Gustafsson, Mälardalen University, Sweden, jan.gustafsson@mdh.se
- Peter Puschner, Technical University of Vienna, Austria, peter@vmars.tuwien.ac.at

Table of Contents

Issues using the Nexus Interface for Measurement-Based WCET Analysis

Adam Betts and Guillem Bernat
Real-Time Systems Research Group
Department of Computer Science
University of York, UK
{abetts, bernat}@cs.york.ac.uk

Abstract

Hardware debug interfaces such as Nexus have the power to unleash the full potential of measurement-based WCET approaches due to the passive nature in which timing data are collected from the processor. However, difficulties arise as a result of their restrictive nature, thus disallowing true user freedom in the selection of instrumentation point placement. This paper elaborates on the problems encountered when using the Nexus interface in our measurement-based WCET framework, and how some of these issues can be resolved, particularly that of irreducibility.

1 Introduction

Measurement-based (MB) WCET analysis techniques are being embraced as the predictability of state-of-the-art processors diminishes due to modern speed-up features, e.g. cache, branch prediction, out-of-order execution, etc. Real-time hardware architects are increasingly looking towards such features as the thirst for performance enhancement grips the embedded market [5]. However, the resulting effect is instruction latencies that are difficult to model *statically*, thus resulting in pessimistic assumptions about speed-up features' behaviour that ultimately leads to loose WCET estimates. MB approaches, on the other hand, permit tighter WCET estimates [2] by testing the program on its actual hardware platform.

The main hindrance in using MB approaches is that timing data need to be collected whilst the program executes: either processor simulation or software probing performs the desired task. Cycle-accurate simulators capture both functional and temporal aspects of a processor, which are usually constructed by scrutinising the processor's user manual. Many factors render themselves crucial in the accurate design of a processor simulator for which failure can produce unsafe WCET estimates. Engblom [3] has cited potential sources of error in simulator construction and he concludes that user manuals are generally not trustworthy and that complex processors do not lend themselves to ease of construction. Alternatively, software monitoring inserts instrumentation points (ipoints) in the program in order to accumulate timing data during its execution. The clear advantage of this technique is program execution on its intended hardware, thus overcoming intrinsic difficulties in modelling the processor and peripheral hardware. However, a negative phenomenon widely known as the *probe effect* ensues whereby ipoints disturb the temporal nature of the program, i.e. the execution time of the program differs when the software ipoints are removed.

A solution for the seemingly unavoidable probe effect has arrived in the form of hardware debug interfaces, such as Nexus [6] which is discussed in section 3, and the ARM embedded trace macrocell (ETM) [4]. The fundamental aspect of these interfaces is that data are collected *passively* from the processor during program execution. Problems do arise through the restrictive nature that is imposed on data collection, which restricts true user freedom in the selection of instrumentation point placement, thus requiring new techniques to handle these hurdles. In this paper we discuss such problems and techniques in the context of Nexus, primarily because it is now an IEEE-ISTO standard, although they are equally applicable to the ETM.

In this paper we discuss how this restriction naturally leads to the problem of irreducibility in the data structure that is employed in our measurement-based framework, the instrumentation point graph (IPG), which is briefly introduced in the next section. We will show how some irreducible issues can be resolved by the relationship that exists between the program's control flow

graph (CFG) and the IPG. We further discuss how trace data loss and out-of-order execution affects the computation of WCET estimates. Finally, we outline some conclusions and future areas of work.

2 Instrumentation Point Graphs

Our approach combines timing data collected during measurement through high-level static techniques that reconstruct the longest path through the program, independent of the type of monitoring employed. This is accomplished using the IPG in which the atomic unit of computation is the time that is observed between ipoints, instead of basic blocks, which is the case of the CFG. In essence, the IPG arranges the possible transitions among ipoint pairs in the CFG into structural form. The transitions among ipoints in the IPG thus represent sequences of code exercised when a particular edge is followed. Timing measurements for these sequences are obtained by executing the program using test-data generation algorithms.

Once the IPG has been built, we adopt traditional calculation methods found in the literature, i.e. tree-based, path-based and IPET, revising them accordingly so that they pertain to the IPG. As hierarchical representations have difficulty modelling *irreducible* regions of code, the reducibility property of an IPG is a central issue in tree-based methods. Software instrumentation can guarantee the reducibility of the resultant IPG, details of which can be found elsewhere [1]; however, as true user freedom in ipoint placement is disallowed in Nexus-monitored programs, this property is only in evidence in trivial CFGs. An example later in the paper will help clarify this issue.

3 The Nexus Standard

Nexus has been introduced in response to the ever-increasing subtle nature of software and hardware bugs [6, 7]. This subtlety has arisen as a result of the complexity of modern processors: more transistors, faster clock-rates, multiple-level on-chip caches, multi-core processors, etc. All these intricacies mask the visibility of bugs and render traditional methods of debugging, e.g. in-circuit emulators and logic analysers, inappropriate. Nexus uses JTAG ports to communicate between a debug tool and the processor, and is increasingly being supported by chip manufacturers such as Motorola and STMicroelectronics.

The principal way to extract WCET data through the Nexus interface is by utilising its program trace feature, *branch trace messaging* (BTM). This allows time stamps to be recorded when sequential program flow *discontinues*, i.e. at taken branches and exceptions. In the case of taken *direct* branches, Nexus includes the number of sequential instructions that were executed since the last taken branch or exception, including those direct branches that resolved to untaken and indirect branches. The address of the branch target and branch-condition predicate bits can also be derived by using the more refined *historical* BTM feature [7]. At first sight it would appear that only monitoring taken branches in unnecessarily restrictive. The motivating factor for this is to lower the burden placed on the Nexus interface: too many requested debug reports results in trace buffer overflow. The BTM feature is the principal means of extracting WCET data, so we now qualify how its restrictive properties impacts the computation of MB WCET estimates.

Irreducible IPGs

In optimising compilers and WCET analysis, the reducibility of a CFG is a key property as loop identification techniques, for example, are greatly simplified. A loop is irreducible when there are multiple entries into the loop [9], so that no single node dominates all nodes in the loop body. In these cases it is often difficult to compute the nodes that are contained in the body, the nesting relationship among loops, and even the number of loops. However, using the Nexus interface in our framework ensures a much higher prevalence rate of irreducibility in the IPG, even for relatively simple CFGs. Worse yet, the irreducibility properties are vastly more complex than an optimising compiler would typically introduce on a CFG, in the sense that reducibility encapsulates much larger subgraphs of the IPG.

The cause of irreducibility emanates from the *virtual* nature of Nexus ipoints in contrast to the *physical* nature of software probes. In the latter, all executions of a program that invoke a set of basic blocks include the execution of ipoints belonging to those basic blocks. This is not the case in the BTM, in which only the invocation of a transition among basic blocks triggers the time stamp. Therefore, unless an edge e that includes a virtual ipoint dominates all other edges E including virtual ipoints on each path to exit in the CFG, it is possible that the flow of control skips around e to each edge in E. This point is illustrated in figure 1: there is a CFG with virtual ipoints on all transi-

tions for which Nexus would record a time stamp[1], as well as the resulting IPG that consists of ipoints as nodes. The edge that includes ipoint i_1 clearly dominates edges that include ipoints i_2 and i_3 in the CFG, thus ensuring that all paths from *start* pass through this edge to reach i_2 and i_3. On the other hand, there is no dominance relation among i_5 and i_6 and they both reside in a natural loop structure. Therefore, the set of ipoints on paths from *start* to this loop have a corresponding edge to i_5 and i_6 if *start* is their immediate dominator; the complexity evidently intensifies as the size of this set increases as well as the number of ipoints in the loop body. Another complexity that emerges from figure 1 is that of misidentification of loop nesting structures. The natural loop structure containing i_5 and i_6 might incorrectly be identified as a nested loop whereby i_5 is the outer loop header and i_6 is the inner loop header. In fact, all edges $i_5 \rightarrow i_6$, $i_6 \rightarrow i_5$ and $i_6 \rightarrow i_6$ are loop backedges since they correspond to another iteration of the loop in the CFG. This problem is also in evidence in the *while* loop structure that contains i_2 and i_3.

We utilise two interlinked techniques in handling IPG irreducibility that is generated by virtual ipoint placement. Firstly, by using the relationship between the CFG and the IPG it is relatively straightforward to ascertain when simulated nest loops actually conform to a single loop. This is primarily carried out by inspection of the dominance relation that exists among basic blocks in the CFG with respect to that of the ipoints in the IPG. Secondly, we can reduce the intricacy of IPG irreducibility, and complexity in general, by means of *edge pruning*. The relationship between the CFG and the IPG is yet again central to the accomplishment of this task, as well as the familiar WCET principle that a program's WCET occurs under maximum loop iteration. In general, any edge in the IPG that bypasses the execution of a loop in the CFG can be pruned. Although both of these techniques reduce some irreducibility aspects, the general case remains an open problem.

Trace data loss

As we discussed above, Nexus attempts to prevent trace data loss through its BTM scheme by recording data when program flow discontinues. This does not completely guarantee fulfilment of this requirement since tightly grouped sequences of control flow changes might still overwhelm the JTAG port. The problem is accentuated by the pulsating increase in microprocessor performance that results in higher instruction throughput. Multiple-issue processors, more accurate branch prediction schemes, and more elaborate speculative execution techniques ensure a faster turnover of branch instructions. In sharp contrast to this, nobody is suggesting that the average size of basic blocks, typically six instructions, is on the verge of increasing.

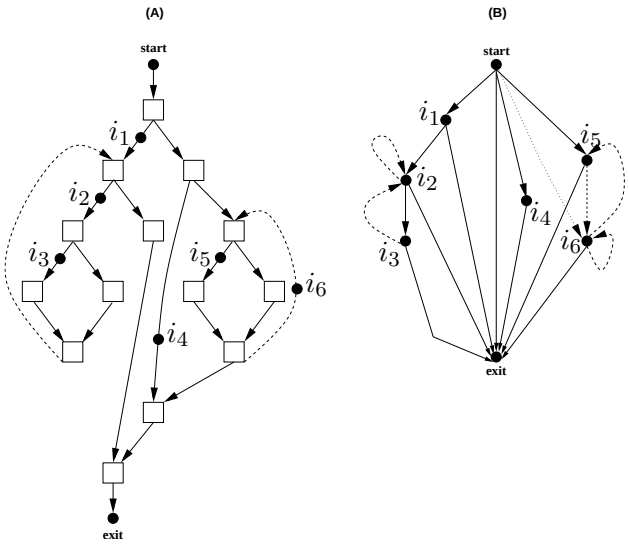


Figure 1: (A) CFG with virtual ipoints on edges Nexus monitors and (B) Resultant IPG generated

The loss of trace data has varying consequences on the computation of WCET estimates. Incomplete information about the branch target address will evidently result in difficulties in reconstructing the path that was executed in the program. Consequently, the time that is observed between recorded pairs of ipoints can lead to inaccurate WCET estimates as it will include the execution of a path that includes another unrecorded ipoint. If measurement is being used to ascertain loop bounds then data loss inevitably leads to the possibility of underestimation since fewer iterations will be observed than the actual number. The irreducibility problem of IPGs further complicates trace data loss: others [10] have shown that it is possible to reconstruct the path through a program by instrumenting the leaves of the CFG's dominator tree. However, the dominator tree of the IPG is extremely shallow as a result of irreducibility, thus eliminating this possibility.

The key to reducing trace data loss is to minimise the data rate out of the CPU. In particular, Nexus does permit the monitoring of a set of addresses within a specified range. Hence it is possible to guide the instrumentation process given static and dynamic properties of the program. For example, some studies [8] have shown that only a

[1] We include start and exit nodes as ipoints

small number of branches are dynamically invoked, thus it may be more pertinent to observe these locations. Other properties such as whether a branch appears on the worst path, the size of the program etc., have equal bearing.

Out-of-order execution

Of all the contemporary hardware features, out-of-order execution causes the greatest distress within the field of WCET due to the sheer complexity required in the analysis - it has also inhibited MB analysis [2]. The crux of the problem is that out-of-order execution permits instructions to execute in a different order to that described in the program, thus it is difficult to determine the correspondence between timing data and the instructions it encompasses. This obstacle is independent of the basic unit of computation employed, so it equally applies to the IPG. Moreover, since Nexus targets modern microprocessors there is a greater likelihood that out-of-order execution problems arise as this technique creeps into the embedded market.

Finding solutions for the out-of-order problem that do not result in undue pessimism is a difficult task. However, as we indicated above, Nexus does provide additional information that might disambiguate some of these issues. Knowledge such as the number of sequential instructions executed since the last program flow discontinuity could provide valuable insight.

4 Conclusions and Future work

Hardware debug interfaces such as Nexus appeal greatly to MB WCET analysis techniques due to the passive collection of timing data from the processor, thus eliminating the probe effect. However, new problems surface as a result of the restrictions imposed by these interfaces on the placement of virtual instrumentation points.

In particular, we have demonstrated how irreducibility quickly becomes problematic even for relatively small CFGs, and that irreducibility is decidedly more complex. On the other hand, we have highlighted how the relationship between the CFG and our underlying data structure, the instrumentation point graph, can be exploited to overcome some of these issues. The focus of future work is to formalise these techniques and to extend them in order to handle a larger subset of irreducible graphs. We also showed how trace data loss and out-of-order execution negatively impacts the computation of WCET estimates. Future work in this area will quantify these effect and propose some solutions.

References

[1] A. Betts and G. Bernat, "Instrumentation Point Graphs for WCET Analysis", Technical report, Department of Computer Science, University of York, April 2005.

[2] A. Colin and S. Petters, "Experimental Evaluation of Code Properties for WCET Analysis", *In proceedings of the* 24^{th} *Real-Time Systems Symposium (RTSS'03)*, December 2003.

[3] J. Engblom, "Processor Pipelines and Static Worst-Case Execution Time Analysis", PhD Thesis, Uppsala University, Uppsala, Sweden, April 2002.

[4] ARM development tools. At `http://www.arm.com`.

[5] G. Frantz, "Digital Signal Processor Trends", IEEE Micro, Vol. 20, No. 6, pages 52-59, November 2000.

[6] The Nexus 5001 forum. At `http://www.nexus5001.org`.

[7] J. Turley, "Nexus Standard Brings Order to Microprocessor Debugging", August 2004. At `www.nexus5001.org/nexus-wp-200408.pdf`, March 2005.

[8] S. Sechrest, C-C. Lee and T. Mudge, "Correlation and Aliasing in Dynamic Branch Predictors", *In Proceedings of the* 23^{rd} *annual international symposium on Computer architecture*, May 1996.

[9] V. Sreedhar, G. Gao, and Y. Lee, "Identifying Loops using DJ graphs", *In ACM transactions on Programming Languages and Systems*, 18(6):649-658, November 1996.

[10] M.M. Tikir and J.K. Hollingsworth, "Efficient Instrumentation for Code Coverage Testing", *In proceedings of the international symposium on Software testing and analysis*, July 2002.

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. 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. 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 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.

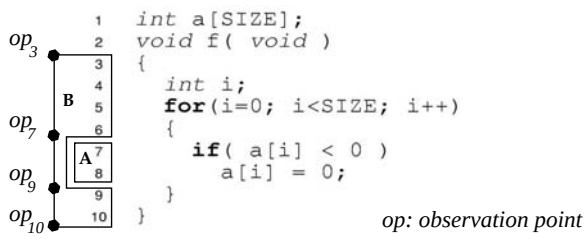


Figure 1. Path clustering.

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.

We propose to cluster the program 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.

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.

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.

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`.

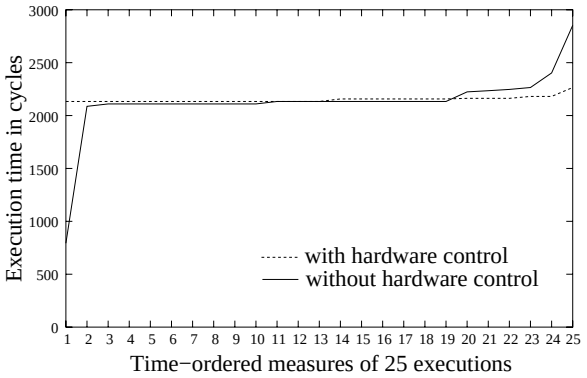


Figure 3. Measurements of jfdctint execution times.

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*, Palma de Mallorca, Spain, July 2005. To appear.

[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, 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.

WCET measurement using modified path testing

Nicky Williams
CEA/Saclay
DRT/LIST/DTSI/SOL/LSL
91191Gif sur Yvette, FRANCE
Nicky.Williams@cea.fr

Abstract

Prediction of Worst Case Execution Time (WCET) is made increasingly difficult by the recent developments in microprocessor 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.

1 Introduction

Prediction of Worst Case Execution Time (WCET) is made increasingly difficult by the recent developments in processor architectures [3]. 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 bad branch prediction use up many more cycles than individual instructions. These events are difficult to predict by static analysis of the program code, especially when the precise architecture of the processor is not divulged by the manufacturer. Moreover, static analyses must continually keep up with the latest architectural innovations.

2 First step to a solution : path testing

The structural testing field provides a first step towards an alternative to pure static analysis. 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 assumptions, 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): this is true for most compilers and options.
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*: in fact 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 penalty to the measured execution time for each such instruction in the path. Cache behaviour may also cause this condition to be violated, e.g. if an execution path contains successive references to elements of a data structure with variable indices and the data structure is large enough to provoke a cache miss for some values of the indices but not for others. Input values should therefore be chosen so as to maximise the difference between index values of the same data structure used in neighbouring references.
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*: in fact, it is impossible to prove which initial state is the worst possible for a given test case without detailed knowledge of the microprocessor. However, by using our knowledge of the test case (branches, addressed variables,…) and broad characterisations of cache and dynamic branch prediction behaviour we

can construct a program to be run just before the program under test and put it into what should be the worst possible state. Cache and branch prediction algorithms are based on past behaviour so such a program might aim to fill the caches with useless data and instructions and repeatedly run the program under test on paths with branches which are the opposite of those in the test case. Note that 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. In this case, all possible sequences of previous test cases which create the initial conditions for a given case can be either run (if not too numerous) or analysed to find which seems to be the worst.

4. *Variations in external system behaviour such as bus activity, DRAM refresh, do not influence execution time*: in fact, the measured time may have to be weighted to account for such aspects [1][7].

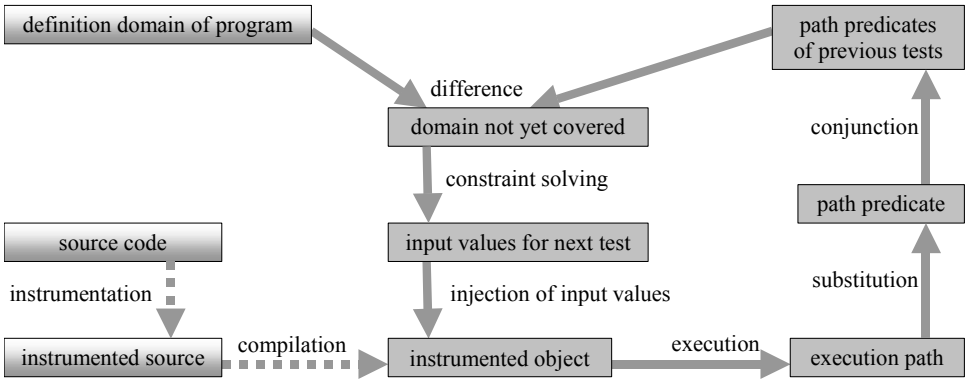
3 The PathCrawler tool for automatic generation of path tests

We have developed the PathCrawler prototype 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 and dimensions they may have, 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. In this way the user can also define different modes or scenarios for which the WCET is to be measured. 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. Note that no annotations of source or object code are necessary. 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

Figure 1 : the PathCrawler test generation process



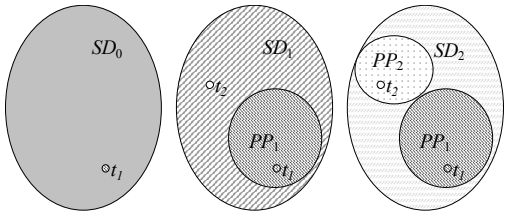


Figure 2:Incremental coverage of the input domain

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 set of legitimate inputs, t_1 is the first test case generated, PP_1 is defined by 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 defined by the predicate of the path covered by t_2 and SD_2 is the difference between SD_1 and PP_2.

PathCrawler could be implemented to treat source code in any imperative programming language. The current prototype [8] treats a wide range of ASCII C programs, which may include arrays and pointers but it cannot yet treat type unions, pointers to functions and recursive functions.

The inputs for each successive test case are found using constraint solving techniques. For integer, Boolean and character 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 is why the user does not need to provide 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. Also, constraint solving based on real numbers has a complexity which 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, as explained in [8]. The result is a very efficient generation of test inputs: for one example program described in [8] 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.

4 The next step: measuring fewer execution paths

Path testing avoids exhaustive testing of all inputs but some programs 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 [4][6].

We are exploring a different approach which is not based on path decomposition but instead takes advantage of the fact that PathCrawler's test generation strategy can be modified so as to decide 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 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 strategy 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 new 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 execution 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 fewer iterations has a shorter execution time than the other one. If the execution time of the path with more 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 maximum: if (a > b) then max = a else max = b;

b) a limit: if (a > limit) then a = limit;

(but note that 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. 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 this partial order to exclude paths from test generation.

For example, to eliminate paths differing only in the number of iterations of a certain loop, we annotate loop head instructions during instrumentation. PathCrawler's strategy is first 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 favours early generation of paths with more loop iterations. Secondly, the strategy excludes from future test generation all the paths identical to the already generated ones except for a lower number of loop iterations.

5 Conclusions

Using PathCrawler to generate a test set to measure WCET promises the following advantages :

- no need for code annotations;
- most ANSI C programs can be treated;
- to measure the execution time of the whole program only two observation points are necessary ;
- no need to predict micro-architectural events such as cache miss, pipeline stall, out-of-order execution, as long as they are deterministic for a given execution path in the source code.

The feasibility of the approach we suggest relies on the extent to which the number of tests can be reduced using simple hypotheses about e.g. cache behaviour and branch prediction. The safety of the WCET obtained also depends on the validity of using such hypotheses to exclude paths and define the initial state(s) for each test case. Further analysis, as well as experiments on different program examples, are needed in order to evaluate the categories of program or micro-architecture for which 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 and obtain a safe WCET.

References

[1] Pavel Atanassov and Peter Puschner, *Impact of DRAM Refresh on the Execution Time of Real-Time Tasks*, In Proc. IEEE International Workshop on Application of Reliable Computing and Communication, Dec. 2001

[2] Guillem Bernat, Antoine Colin and Stefan M. Petters, *WCET Analysis of Probabilistic Hard Real-Time Systems*, In Proc. 23rd IEEE Real-Time Systems Symposium (RTSSO2), Austin, Texas, December 2002

[3] Reinhold Heckmann, Marc Langenbach, Stephan Thesing and Reinhard 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)

[4] 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

[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, 200

[6] 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

[7] Jürgen Stohr, Alexander von Bülow and Georg Färber, *Controlling the Influence of PCI DMA Transfers on Worst Case Execution Times of Real-Time Software*, In Proc. WCET'04, Catania, June 2004

[8] 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

Computing the Worst Case Execution Time of an Avionics Program by Abstract Interpretation

Jean Souyris* (jean.souyris@airbus.com),
Erwan Le Pavec* (erwan.lepavec@airbus.com),
Guillaume Himbert* (guillaume.himbert@airbus.com),
Victor Jégu* (victor.jegu@airbus.com),
Guillaume Borios** (guillaume.borios@airbus.com)

*Airbus France, **Atos Origin Integration,

Reinhold Heckmann*** (reinhold.heckmann@absint.com)

*** AbsInt GmbH

Abstract

This paper presents how the timing analyser aiT is used for computing the Worst-Case Execution Time (WCET) of two safety-critical avionics programs. The aiT tool has been developed by AbsInt GmbH as a static analyser based on Abstract Interpretation

1 Introduction

In the field of safety-critical avionics applications, verifying that a program exhibits the required functional behavior is not enough. Indeed, it must also be checked that its timing constraints are satisfied. Generally these constraints are expressed in terms of the **Worst Case Execution Times** (WCETs) of program tasks.

The WCET of a program (or piece of a program, like a routine, a task, etc) is the maximum of the execution times of all program runs. Unfortunately, finding the program run that leads to the WCET is impossible for real-life programs. What is achievable is computing an upper bound of the WCET, i.e., a time greater than the real but incomputable WCET. In that case, the WCET bound is said to be sound, and the less it is away from the real WCET – from above – the better it is. A good method for estimating the WCET of a program must be sound and precise, i.e. yield tight results, with the ability to demonstrate both properties.

For the most safety-critical avionics programs, a pure measurement-based method is not acceptable, especially if the dynamic structure of the program was not made as deterministic as possible, by design.

This paper will show how it became almost impossible to compute the WCET of two safety-critical avionics programs by Airbus' traditional method, due to the complexity of the modern processor they execute on (sections 2 and 3). Section 4 presents AbsInt's aiT for PowerPC MPC755, a tool that computes the WCET of a program by analyzing its binary file. Section 5 describes how aiT is currently used for computing the WCET of the two avionics programs we take as examples along the paper. Finally, section 6 mentions future work and concludes.Instructions

2 A challenging hardware

Sometime ago, processors behaved in a very deterministic way. The latency of an instruction was a constant, i.e., it did not depend on what happened before the execution of that instruction. This was the case for internal instructions (add, mul, or, etc.) as well as for those that access external devices like memory or IO. In order to increase their average computation power, modern processors are endowed with accelerating mechanisms causing variable execution times of instructions. Hence, the duration of an instruction depends on what was executed before it. This "effect of history" can be very deep and without logical correlation to the instruction it affects. One example of such a mechanism is the cache. Indeed, depending on the execution path leading to, say, a load instruction, the memory line containing the data to be loaded may already be in the data cache (HIT), or not, be it that it was not yet loaded (MISS) or already removed (MISS due to replacement). There are many other accelerating mechanisms like out-of-order execution, branch prediction, speculative accesses, "superscalarity", duplication of processing units (e.g., two Integer Units), Store Buffers, pipelining of addresses, etc.

The board on which the analysed programs execute is made of a PowerPC MPC755, SDRAM, and IO peripherals connected to a PCI bus. The MPC755 is a "modern" processor in the sense that it employs the kind of accelerating mechanisms mentioned above. SDRAM is also modern because of the sort of cache associated to each bank of memory: The SDRAM page containing the most recently accessed memory address is kept "open", allowing for faster access in the future. The chipset making these components "talk" to each other is modern as well, since, for instance, it optimizes accesses to SDRAM by buffering certain kinds of writes or by taking profit of the address pipelining mechanism of the MPC755.

As if this was not enough, the refresh of the SDRAM and the so-called Host Controller (connected to the PCI bus) steal cycles from the processor asynchronously.

3 A traditional method very hard to apply now

Before the new method based on aiT for PowerPC MPC755 was introduced, Airbus used some "traditional" method for computing the WCETs of previous generations of the two avionics programs. This method was a mix of measurement and

llectual analysis. Let's briefly describe how it worked for each of the two programs.

First avionics program. Basically, the way the WCET is obtained takes profit of the structure of the program, which is produced by an automatic code generator taking a graphical specification (SCADE) as input. The generated program almost entirely consists of instances (or routine calls to) a rather limited set of small snippets like logical and comparison operators (multiple input AND, OR, Level Detector), arithmetical operators (ADD, MUL, ABS, etc), digital filtering operators, etc. The small size of these basic components makes it possible to measure their WCETs, as long as the initial execution environment for each measurement is the worst possible with respect to execution time. One must also notice that the generated program is very linear since all conditionals and loops occur in the small snippets, thus being very local. The program being made of instances of (or calls to) code snippets, the WCETs of which are available after the measurement campaign, a simple formula (more or less a sum) is applied for inferring the whole WCET, knowing that each (instance of a) code snippet is executed once and only once.

Second avionics program. The second program also consists, for the main part, of basic processing units, each based on a small number of patterns. The size and limited execution paths inside these patterns make it also possible to measure their WCETs in the same conditions as for the first program. But the main difference to the first program is that the basic units form the implementations of some abstract machine instructions collected in an instruction table (also called configuration table). The execution of the program is driven by a high-level loop, reading instructions and parameters from the configuration table, and then calling the appropriate basic processing units implementing the instructions. Each basic processing unit being called multiple times, the resulting WCET is the sum of the basic WCETs of the instructions present in the configuration table.

The traditional approach is correct with "deterministic processors". As stated in section 1, the computed WCET must be sound, i.e., an upper bound of the real WCET, and tight, i.e., not too far from the real WCET. With processors having no – or few – history-based accelerating mechanisms, both criteria are met without too much difficulty since finding the worst possible initial environment for the measurement of the WCET of each snippet and producing a – non-formal – demonstration that the computed WCET is an upper bound of the real WCET is accessible to a human being, and the overestimation is acceptable.

Modern processors. But even for such deterministic programs, the legacy methods are no more applicable when using hardware components like the ones mentioned in section 2. The first reason is that it is a lot harder to find the worst possible initial environment for measuring the WCETs of the small snippets, and also to get sure that combining these WCETs for computing the WCET of the entire program is safe. The second reason is that even if we could solve the first difficulty (worst initial environment and safe combination), the resulting WCET would be too much overestimated for being useful. Both reasons originate from the history-dependent execution time of each instruction that makes it impossible to deduce a good global WCET from any local measurements or computations.

3.1 Asynchronous extra time (SDRAM refresh and Host Controller (for Input/outputs) activity).

When measuring the WCET of the small snippets the whole program is made of, only the effect of SDRAM refreshes is measured. It is therefore not possible to infer a WCET of the program that encompasses the effect of the Host controller

4 aiT for PowerPC MPC755

AbsInt's aiT tools form a family of WCET tools for different processors, including PowerPC MPC755. aiT tools get as input an executable, an .ais file containing user annotations, an .aip file containing a description of the (external) memories and buses (i.e. a list of memory areas with minimal and maximal access times), and a task (identified by a start address). A task denotes a sequentially executed piece of code (no threads, no parallelism, and no waiting for external events). The names of the various input files and other basic parameters are bundled in a project file (.apf file) that can be loaded into a graphical user interface (GUI).

All instances of aiT determine the WCET of a task in several phases: **CFG building** decodes, i.e. identifies instructions, and reconstructs the control-flow graph (CFG) from a binary program. User annotations may help aiT in identifying the targets of computed calls and branches. **Value analysis** computes value ranges for registers and address ranges for instructions accessing memory. **Loop bound analysis** determines upper bounds for the number of iterations of simple loops. Such upper bounds are necessary to obtain a WCET. Loop bounds that cannot be determined automatically must be provided by user annotations. **Cache analysis** classifies certain memory references as cache hits. **Pipeline analysis** predicts the behavior of the program on the processor pipeline and so obtains the WCETs of the basic blocks. **Path analysis** determines a worst-case execution path of the program.

Value analysis is based on an abstract interpretation of the operations of the analyzed task, taking into account variable values specified by the user (e.g. to restrict the analysis to a certain operation mode of the analyzed software). The results of value analysis are used to determine loop bounds, to predict the addresses of data accesses and to find infeasible paths caused by conditions that always evaluate to true or always evaluate to false. Knowledge of the addresses of data accesses is important for cache analysis. Value analysis usually works so good that only a few indirect accesses cannot be determined exactly. Address ranges for the remaining accesses can be provided by user annotations.

Cache Analysis uses the results of value analysis to predict the behavior of the (data) cache. The results of cache analysis are used within pipeline analysis allowing the prediction of pipeline stalls due to cache misses. The combined results of the cache and pipeline analyses are the basis for computing the execution times of program paths. Separating WCET determination into several phases makes it possible to use different methods tailored to the subtasks. Value analysis, cache analysis, and pipeline analysis are done by abstract interpretation. Integer linear programming is used for path analysis.

5 WCET computation with aiT for PowerPC MPC755

First avionics program. We first present some more details about the structure of the program. It consists of 24 uninterruptible tasks that are activated one-by-one by a real time clock in a fixed schedule: task 1 to task 24, then task 1 again, and so on until the electrical power of the aircraft is switched off. This time-triggered scheduling method requires that the WCET of

each task must be less than the period of the real-time clock. The call graph of each task is basically organized in three layers. The first layer contains 4 calls to so-called sequencers, which for each task are selected from a list of 38 possibilities. These sequencers allow for the activation of pieces of code at different rates, i.e., 1 over 2 ticks, 4 ticks, 8 ticks or 24 ticks. Still in this highest layer, some system routines are called before and after the four sequencer calls. The second layer consists of the routines containing the actual operation code composed of "calls" to code macros, which form the basic components referred to in section 3. The third layer consists of the input/output routines called by some of the basic components present in the second layer.

As described in section 4, aiT requires some global parameter settings in the graphical user interface, in a project file (.apf file), or in a parameter file (.aip file), and user annotations in an annotation file (.ais file). Global parameters are quite basic and mainly describe the hardware, so let us concentrate on the work involved in writing the annotations. As described in section 4, annotations are mandatory when value analysis fails to find the iteration bound of some loop or the computed address range for an external access (load or store) is too imprecise. We now present the annotations required in the analysis of the first avionics program. As stated above, its functional code consists of operators implemented as macros and a few input/output and "system" routines.

Annotations in operators:

- cos(x) and sin(x): The values of these functions are extracted from some table. The index into this table is computed from the floating-point number x given as argument. As aiT does not "cover" floating-point calculus, its safe reaction is not to compute any bounds for the value of the index. Hence the address range of the table lookup is unknown. This range, which is in fact a function of the size of the table, is provided by a user annotation.

- delay family (4 operators): These operators are called once in the course of each task to store a value into some array. They also update a static variable containing an index to this array. Since the periodic updates of the index are triggered by the real time clock and not by a loop inside the code, aiT cannot compute the range of addresses for this index. Thus an annotation has to be provided.

If the basic operators were implemented as routines, the number of annotations as described above would have been very limited and, thus, affordable easily. Yet the operators are implemented as code macros. Consequently, what would be a single annotation in case of a routine splits into as many annotations as there are instances of the macro in the code, which is quite huge.

An automatic generator of annotations. For solving this industrial problem, it was decided to use the same technique for annotating as for the coding itself, namely automatic generation. Indeed, an automatic generator of annotations has been implemented that reads a generic description of an annotation in a given macro (basic operator) and produces the actual instances of this annotation for all occurrences of the macro in the binary.

Annotations in I/O or "system" routines:

- Communication drivers (USB, AFDX): a few pointers involved in the data exchanges via these communication channels are too dynamic for being computed by aiT automatically. The relevant address ranges must be provided via annotations.

- Loops: Annotations are required for a small number of loops whose iteration bounds cannot be determined by aiT.

The second avionics program is also based on uninterruptible tasks activated by a real time clock. The tasks must complete in the allowed time to guarantee proper functioning of the program. Each task consists of a high-level loop that reads a list of operation codes with parameters from a configuration table, and for each operation code, calls the basic blocks implementing the operation. These basic blocks are highly dependent on parameterisations provided by the configuration table. The parameterisations influence the timing behaviour because they may induce specific processing or affect loop bounds and addresses of external accesses.

Therefore, it is of first importance for aiT to keep track of these parameters as they are copied from the configuration table to temporary variables and registers. In the first place, the configuration table has to be recognized by aiT. The configuration table is located in a specific area of a binary file. aiT supports an annotation saying that a data area in the analyzed binary is "read-only" so that the values found there can be used by value analysis. However, in the program considered here, this constant area is provided as a separate binary because the configuration tables are separate loadable parts. Since aiT can analyse only single binaries, the binary with the configuration table (this is Mbytes of data) is translated into .ais file annotations specifying the contents of the table. As the same basic blocks are called multiple times in one task activation, either directly from the main loop, or because they are low-level shared services, some instruction cache hits can be guaranteed. However these cache hits depend on the ordering of the operation codes in the configuration table and cannot be taken into account by the standalone basic block WCET measurements of Airbus' "traditional" approach (simple sum of basic block execution times). Hence this method cannot take advantage of the improvements of modern processors, and thus cannot provide WCET results compatible with the allowed execution time.

The major part of the factors affecting the WCET (conditions, loop bounds, pointers, etc) is found automatically by aiT, either by code inspection or from the annotations describing the configuration table. Yet some factors are outside aiT's knowledge and capacities, and annotations have to be provided to bound the analysis and achieve a result. These factors are lower and upper bounds on input data, static data from previous task activations, or data provided by devices outside of processor knowledge (DMA for example). For these, maximum loop iterations, values read from memory, branch exclusions, etc, have to be specified to aiT.

There are 256 different configuration tables, corresponding to different tasks of the same computer, and of different computers in the aircraft. One objective of the separation of software in configuration tables (lists of operation codes with parameters) and the code for interpreting these tables is to shorten (in terms of delay between specification and deliveries) the development and validation cycles for the software. This objective requires an automatic computation of the WCET without direct human involvement.

WCET determination for the combination of the interpreter code with each of the 256 configuration tables consists of determining the longest execution path in a quite complex piece of code. These WCET analyses are to be performed in a period of time compatible with the industrial constraints associated with the short development cycle. This however is possible due to

fact that the analyses for the 256 tables are independent from each other and can be performed in parallel on many computers.

6 Considerations about the results

The WCETs of the two programs are not yet publishable for general confidentiality reasons; the authors apologize for that and hope the reader will understand.

Nevertheless some considerations about the results and how they were obtained are presented in this section.

First avionics program. This program has a very linear structure and limited sources of jitters (in number and amplitude). This allows for valid comparisons between the figures obtained by measurement during real executions of the tasks and those produced by using aiT. The comparison shows that the WCET of a task typically is about 25% higher than the measured time for the same task, the real but non-calculable WCET being in between.

Another comparison is worth to mention: the one between aiT's results and those of Airbus' "traditional" method. As predicted when the decision for using aiT was made, the figures obtained by the traditional approach are a lot higher than those produced by the aiT-based method. Actually the overestimation is such that the "traditional" figures are useless.

Second avionics program. Despite the fact that this program has not a linear structure and contains a great number of loops and branches, the sources of jitter are limited. Most of the loop and branch conditions can be evaluated knowing the configuration table. There are still more sources of jitter than in the first program, but comparisons between the figures obtained by measurement of real executions and results produced by aiT are still possible. The WCETs computed by aiT are about 50% higher than the corresponding measured execution times. These higher margins can be explained (at least in part) by the difficulties to set up an environment for running the task that is sufficiently close to the environment leading to the real WCET.

More importantly, the WCETs computed by aiT are much lower than the results from the "traditional" method and are compatible with the program's timing constraints.

Status with respect to some general acceptance criteria.

- Soundness: First of all, aiT was developed in the framework of Abstract Interpretation [2], which is very good for the soundness of the underlying principles. The soundness of the actual implementation of the tool, as used on the real avionics programs, cannot be assessed formally. The way of getting confident in the tool and its method of use is the Validation/Qualification process sketched in section 7.
- Ease of use: Users, i.e., normal engineers, always worked with aiT in an autonomous way. Furthermore, they were able to develop utilities for making aiT easier to use in the industrial context, like the automatic generator of annotations (see section 5).
- Resource needs: The WCET computations for the second avionics program (the most demanding one) take an average of 12 hours (on a 3+ Ghz Pentium 4) per task (there are 256 tasks). But this workload is dispatched on many computers and is not a real problem. The main concern is about the space requirement that is, for some analyzed tasks, close to the current 3 Gbytes limit of the 32-bit architecture. To address this point, a migration to the 64-bit architecture is under investigation.

7 future work

Validation. By this term the authors mean "getting a high level of confidence" in the results produced by aiT as used on the avionics programs referred to in this paper. Although some data collected during the first industrial usage of aiT are already available for validation, most of the work is still do be done. It will have four basic objectives: soundness of the underlying principles (a), correctness of the models (b), soundness of the method of use (c), and validity of the results (d).

(a) and (b): most of the analyses performed by aiT (see section 3) have theoretical principles (models and sometimes proofs) precisely described in theses or scientific papers. Some effort has already been spent in reading this documentation. This process will carry on together with some checking against hardware manuals, e.g., the PowerPC MPC755 manual [3].

(c): the objective here is mainly to check that the annotations (loop bounds, register contents at certain program points, etc) do not make aiT compute an unsafe WCET.

(d): the very detailed results produced by aiT make it possible to perform useful and automated checks. An example of such checks is whether real execution traces (got via a logic analyzer) belong to the set of those computed by aiT.

Qualification. This term comes from DO178B. In the case of aiT, it is about the qualification of a verification tool. Current practices consider that the qualification is twofold: in-service history and qualification tests. Both items of the qualification will be built from the outputs of the validation process. Also, one should notice that the qualification of a tool is per avionics program the tool is used on.

References

[1] Stephan Thesing, Jean Souyris, Reinhold Heckmann, Famantanantsoa Randimbivololona, Marc Langenbach, Reinhard Wilhelm, and Christian Ferdinand. An Abstract Interpretation-Based Timing Validation of Hard Real-Time Avionics. In *Proceedings of the International Performance and Dependability Symposium (IPDS)*, June 2003

[2] Patrick Cousot. Interprétation abstraite. *Technique et Science Informatique, Vol. 19, Nb 1-2-3.* Janvier 2000, Hermès, Paris, France. pp. 155-164.

[3] Motorola. *MPC750 RISC Processors User's Manual*.

Experiences from Industrial WCET Analysis Case Studies

Andreas Ermedahl, Jan Gustafsson and Björn Lisper
Dept. of Computer Science and Electronics, Mälardalen University
Box 883, S-721 23 Västerås, Sweden

Abstract

Static Worst-Case Execution Time (WCET) analysis is currently taking a step from research to industrial use. We present a summary of three case studies where static WCET analysis has been used to analyse production code for embedded real-time systems. The primary purpose has not been to test the accuracy of the obtained WCET estimates, but rather to investigate the practical and methodological difficulties that arise when applying current WCET analysis methods to these particular kind of systems.

In particular, we have been interested in how labor-intensive the analysis becomes, for instance by estimating the efforts to study the analysed code in detail, and measuring the number of manual annotations necessary to perform the analysis. From these observations, we draw some conclusions about what would be needed to turn static WCET analysis into a useful tool for embedded and real-time systems software development.

1 Introduction

To give timing guarantees for embedded and real-time systems, a key parameter is the *worst-case execution time* (WCET) of the executing tasks. Until now, the common method (if any) in industry to derive WCET values has been by measurements. A wide variety of measurement tools are employed in industry, including emulators, logic analyzers, oscilloscopes, and software profiling tools [14]. This is labor-intensive and error-prone work, and even worse, it is difficult to guarantee that the WCET has been found.

Static WCET analysis is an alternative method to determine the WCET of a program, relying on mathematical models of the software and hardware involved. The analysis avoids the need to run the program by considering the effects of all possible inputs, including possible system states, together with the program's interaction with the hardware. Given that the models are correct, the analysis will derive a timing estimate that is safe, that is greater than or equal to the actual WCET. The static WCET analysis research community has developed a number of prototype tools during the last couple of years, for example SWEET [18] and Heptane [10]. Recently also commercial WCET tools, such as aiT from AbsInt GmbH, Germany [1] and Bound-T from Tidorum, Finland [3], have appeared.

In this paper, we present experiences from three case studies where WCET analysis tools have been used. In the two first case studies, we analysed time-critical parts in a real-time operating system (Sections 3, 4). The third case study targeted code controlling automotive data communication (Section 5). We also report from two on-going case studies (Section 6).

All of these case studies have been performed as MSc theses works. The students can spend about five months on their work, and they are no experts on the code at the beginning. This means that these results can be seen as typical for WCET analysis made by an well-educated but external person; the work should probably have taken less time if an expert or the programmer had performed it.

We believe that doing case studies, with careful evaluations, provides valuable input both for WCET research and WCET tool development. Our hypothesis is that the studied software is representative for a large class of industrial embedded real-time code, making our results applicable to similar systems.

There are only a few other case studies of using WCET analysis in practice known to us; reporting on the use of Heptane [6], aiT [15], and Bound-T [11].

To make static WCET analysis industrially useful, it is desirable to automate the process on a "one-click-analysis" basis. Consequently, one of the main topics has been to investigate how labour-intense practical WCET analysis actually becomes.

We were also interested in the characteristics of the obtained WCET values. Most scheduling theories assume that each task has a single fixed WCET, and we wanted to find out whether this assumption is valid in

real industrial settings.

The rest of the paper is organized as follows. In Section 2, we present the used tools. In Section 3, we describe a case study with our own research prototype, SWEET (SWEdish Execution time Tool). Sections 4 and 5 describe two case studies where we used aiT to analyse commercial code. Section 6 presents two ongoing case studies. In Section 7 we draw some conclusions, and in Section 8, we point out further research.

2 Tools Used

SWEET is a prototype WCET tool developed at Uppsala and Mälardalen University [18]. SWEET consists of three main parts; a flow analysis which detects program flow constraints, a low-level analysis, and a final WCET calculation. The flow analysis part of SWEET analyses intermediate code produced by a research compiler. Our current focus is to develop automatic flow analysis methods, such as abstract interpretation-based methods [9].

The aiT tool is a commercial WCET analysis tool from AbsInt GmbH [1]. In contrast to SWEET, aiT does not rely on any specific compiler, but analyses executable binaries, with support for a number of target architectures. The tool also includes an automatic loop bound analysis, which can catch simple cases.

3 Case Study 1: Using SWEET to Find Time Bounds For DI Regions

This case study was performed with the low-level and calculation parts of an earlier version of SWEET. The purpose was to find upper bounds of the execution time for a number of Disable Interrupts (DI) regions in the delta kernel (ARM9 version) of the Enea OSE operating system [7]. The OSE operating system is a real-time operating system used in embedded applications, for example in mobile phones and aircrafts. The case study is described in closer detail in [4].

Having short DI regions is important, since the execution of these regions can potentially delay any other activity in the system. The goal of the study was to investigate if WCET analysis could provide a feasible way to bound the execution times of the DI regions at a reasonable cost.

The study was done in the following steps:

1. The DI regions was extracted from the binaries.
2. The control flow graph for these regions was constructed.
3. The WCET tool was used to calculate upper bounds for the WCET of each region.

We identified 612 DI regions in the Delta OSE kernel. Most of these were very simple. We selected ten DI regions that were potentially challenging for WCET analysis for a closer investigation. These regions had a more complex control structure than the others, and several contained loops. To find upper loop iteration bounds sometimes posed a problem, since no automatic loop bound analysis was available, and it was hard to deduce loop bounds manually from the code.

Experiences and conclusions. A lot of effort was used to identify DI regions and construct their control flow graphs. The tools developed to do this had some shortcomings. Even with these problems, there are some interesting conclusions we can draw:

- The problem of defining upper loop iteration bounds depends on the special type of code analysed here. Operating systems are often run in certain modes which may affect loop bounds, and therefore the WCET is typically mode-dependent. One would thus like to have different, tight WCET bounds for different modes, rather than a single WCET bound valid for all modes.
- The usefulness of analyses such as WCET analysis grows fast with the level of automation. In our experiment, even simple means of automation made a huge difference in the amount of engineering work.

4 Case Study 2: Using aiT to Find Time Bounds For Time Critical Code

The Enea OSE operating system for the ARM processor was studied also here. Some of the tools developed in the first case study were re-used in this case study. 180 of the previous DI regions were analysed, as well as four system calls. In this study, we used the aiT tool [1]. This commercial tool has a richer set of processor timing models, and a better user interface, than our prototype tool. The case study is described in closer detail in [13].

The aiT ARM7 tool analyses executables. This information is, however, often not sufficient to yield a good WCET bound for the analyzed code. In particular, information about program flow, such as bounds to loop iteration counts not caught by the loop bounds analysis, and knowledge of infeasible paths, has to be provided by the user. Therefore, aiT supports a set of *user annotations* to provide external information to the analysis [8]. Some of the more important annotations are: *loop bounds*, *maximal recursion depth*, *dead code*, and (static) *values of conditions*.

Experiences and conclusions. We soon discovered that the execution time of the system calls depended on many parameters. A global WCET bound, valid for all possible parameter values, could become very poor for for actual configurations and standard running modes.

We dealt with this problem in our experiments by assuming some "typical" scenarios for parameters affecting the WCET (after correspondance with the OSE designers). We also excluded uninteresting execution paths from the analysis by manual annotations.

We made the following observations:

- A significant amount of annotations were required for each system call; for the analysed routines of sizes between 78 and 143 instructions, the number of annotations were between 10 and 33.
- Another observation is that excluding the error handling code in the OSE system calls yielded significantly smaller code to analyze.
- Many loops in the OSE kernel depends on dynamic data structures. This had the consequence that the aiT loop bound analysis did not perform well for these loops.
- Providing upper bounds manually for these loops required a deep understanding of the code. Consequently, the analysis was quite labor-consuming, even if the analyzed code was small. Also, the analysis relied on information from the OSE designers.

We conclude that the usefulness of WCET analysis would improve with a higher level of automation and support from the tool. Especially, it would be important be to develop advanced flow analysis methods, that could find complex loop bounds automatically. Another important conclusion made is that absolute WCET bounds are not always appropriate for real-time operating system code. The reason is, as mentioned, that the WCET often depends on dynamic system parameters. An absolute WCET bound, covering all possible situations, will provide a gross overapproximation.

5 Case Study 3: Using aiT for Time-Critical Parts of Automotive Code

This case study targeted automotive code, namely the Volcano Tool Suite for design and implementation of in-vehicle communication over CAN and/or LIN networks. The company Volcano Communications Technologies AB (VCT) [16] provides tools for embedded network systems, principally used within the car industry. The Volcano LIN Target package (LTP) was selected as a suitable part of the Volcano LIN tool suite to analyse. The work is described in closer detail in [12].

The microcontroller used in this study was a MC9S12DP256 from Motorola, which includes a 16-bit Star12 CPU of the MC68HC12 family.

Results from analysis of nine different LIN API functions were presented. We were able to obtain WCET values for all analyzed functions. However, these values were often not a constant single value, but depended on some system parameters. Also, all functions needed manual annotations to be analysed. The number of annotations ranged between 6 and 14 for functions of sizes between 2 kb and 14 kb.

Experiences and conclusions. As for the OSE code, the WCET for the studied LIN functions often depends on some specific system configuration parameters and modes. Similarly, a single WCET bound valid for all parameter values would provide a very poor estimate in most situations. A mode- and input-sensitive WCET analysis would obtain a better resource utilization and provide better understanding of the system's timing characteristics.

For many parts of the LIN API it was possible to manually create parametrical WCET formulas. It seems interesting to develop methods to automatically derive these parametrical formulas.

Much work was required to set annotations manually. To do this required an understanding of the meaning of the code.

There is a need for ways to automate the analysis. For example, better flow analysis methods would be useful to avoid manual calculation of loop bounds.

After discussions with the VCT employees it turned out that not only the WCET, but also the jitter of a piece of code, is of large interest. (The jitter is the largest execution time variation a function can experience, that is the difference between the best-case execution time (BCET) and the WCET.)

6 On-going work

We are currently performing two case studies which are summarized below.

Comparison of Different Methodologies for Obtaining WCET Values. Two MSc students are currently studying real-time embedded systems code from CC-Systems [5], using aiT for the Infineon C167 processor. CC-Systems develops embedded software and hardware for welding machinery, as well as for trucks, ships, trains and other vehicles. This case study will compare the different methodologies for obtaining timing values, that is static analysis and measurement-based methods.

Evaluation of static WCET Analysis Methods for Time-Critical Real-Time Embedded Code. We have also just started a new case study at Volvo Construction Equipment (Volvo CE) [17]. Volvo CE uses the Rubus real-time operating system from Arcticus [2] in their embedded, time-critical systems for trucks and other vehicles. The target processor will be Infineon C167 and (if possible) Infineon XC161. The precision of the WCET analysis will be evaluated against the measurements which are performed regularly by Volvo CE.

7 Conclusions

Sections 3 to 5 provided a number of detailed results and experiences. Some common conclusions can be drawn from our case studies.

It is possible to apply static WCET analysis to code with properties similar to the analysed code. The tools used performs well, once the necessary preparatory work, such as defining annotations, has been done. However, the WCET analysis process is not automated on a 'one-click-analysis' basis. Much manual intervention, and detailed knowledge of the analyzed code, is required to perform the analysis.

A higher degree of support from the tool, for example with automatic loop bounds calculation, would be desirable. A graphical interface is also valuable, to obtain an overview of the analysed code and see how it executes.

Absolute WCET bounds are not always sufficient. Support for some type of parametrical WCET calculation is sometimes needed.

8 Future Work

We intend to continue with WCET analysis case studies. One direction is to use the flow analysis developed in our own tool, SWEET, both as a stand-alone tool and used in cooperation with the commercial tools. We are members of the Compilers and Timing Analysis cluster in the ARTIST2 Network of Excellence on Embedded Systems Design. One of the aims of the work in this group is to define common formats for cooperation between different parts of WCET tools.

Acknowledgements

This work was performed within the ASTEC competence center, `www.astec.uu.se`, supported by the Swedish Agency for Innovation Systems (VINNOVA), `www.vinnova.se`. We thank AbsInt GmbH for giving us access to their WCET analysis tool. We are also grateful to the ENEA, CC-Systems, Volcano, Volvo CE, and Arcticus companies for their support.

References

[1] AbsInt company homepage, 2005. `www.absint.com`.

[2] Arcticus Systems homepage. URL: `http://www.arcticus.com`, 2005.

[3] Bound-T tool homepage, 2005. `www.tidorum.fi/bound-t/`.

[4] M. Carlsson, J. Engblom, A. Ermedahl, J. Lindblad, and B. Lisper. Worst-case Execution Time Analysis of Disable Interrupt Regions in a Commercial Real-Time Operating System. In *Proc. 2nd International Workshop on Real-Time Tools (RT-TOOLS'2002)*, 2002.

[5] CC-Systems AB homepage. URL: `http://www.cc-systems.com`, 2004.

[6] A. Colin and I. Puaut. Worst-Case Execution Time Analysis for the RTEMS Real-Time Operating System. In *Proc. 13th Euromicro Conference of Real-Time Systems, (ECRTS'01)*, June 2001.

[7] Enea. Enea Embedded Technology homepage, 2004. URL: `http://www.enea.com`.

[8] C. Ferdinand, R. Heckmann, and H. Theiling. Convenient User Annotations for a WCET Tool. In *Proc. 3rd International Workshop on Worst-Case Execution Time Analysis, (WCET'2003)*, 2003.

[9] J. Gustafsson. *Analyzing Execution-Time of Object-Oriented Programs Using Abstract Interpretation*. PhD thesis, Department of Computer Systems, Information Technology, Uppsala University, May 2000.

[10] Homepage for the `Heptane` WCET analysis tool, 2005. `www.irisa.fr/aces/work/heptane-demo/heptane.html`.

[11] M. Rodriguez, N. Silva, J. Esteves, L. Henriques, D. Costa, N. Holsti, and K. Hjortnaes. Challenges in Calculating the WCET of a Complex On-board Satellite Application. In *Proc. 3rd International Workshop on Worst-Case Execution Time Analysis, (WCET'2003)*, 2003.

[12] S. Byhlin, A. Ermedahl, J. Gustafsson, B. Lisper. Applying Static WCET Analysis to Automotive Communication Software. In *Proc. 17th Euromicro Conference of Real-Time Systems, (ECRTS'05)*, July 2005.

[13] D. Sandell, A. Ermedahl, J. Gustafsson, and B. Lisper. Static Timing Analysis of Real-Time Operating System Code. In *Proc. 1st International Symposium on Leveraging Applications of Formal Methods (ISOLA'04)*, Oct 2004.

[14] D. B. Stewart. Measuring Execution Time and Real-Time Performance. In *Proceedings of the Embedded Systems Conference (ESC SF) 2002*, Mar 2002.

[15] S. Thesing, J. Souyris, R. Heckmann, F. Randimbivololona, M. Langenbach, R. Wilhelm, and C. Ferdinand. An Abstract Interpretation-Based Timing Validation of Hard Real-Time Avionics Software. In *Proc. of the IEEE International Conference on Dependable Systems and Networks (DSN-2003)*, June 2003.

[16] Volcano Technologies Communications AB homepage. `www.volcanoautomotive.com`, 2005.

[17] Volvo CE (Construction Equipment) homepage, 2005. URL: `http://volvo.com/constructionequipment`.

[18] Worst Case Execution Times (WCET) project homepage, 2005. `www.mrtc.mdh.se/projects/wcet`.

Using a WCET Analysis Tool in Real-Time Systems Education

Samuel Petersson*▯, Andreas Ermedahl*▯, Anders Pettersson*, Daniel Sundmark*, and Niklas Holsti†
*Dept. of Computer Science and Electronics
Mälardalen University, S-72123 Västerås, Sweden
spn99007@student.mdh.se, andreas.ermedahl@mdh.se
anders.petterson@mdh.se, daniel.sundmark@mdh.se
†Tidorum Ltd
Tiirasaarentie 32
FI-00200 Helsinki, Finland
niklas.holsti@tidorum.fi

Abstract

To reach a more widespread use, WCET analysis tools need to be a standard part in the education of embedded systems developers. Many real-time courses in academia use Lego Mindstorms, an off-the-shelf kit of Lego bricks for building and controlling small prototype robots. We describe work on porting the Bound-T WCET analysis tool to the Lego Mindstorms microprocessor; the Renesas H8/3292. We believe that this work will make students, and indirectly the industry of tomorrow, aware of the benefits of WCET analysis tools.

We also present the real-time laboratory framework in which this WCET analysis tool will be used. The framework has been developed with schedulability and timing predictability in mind, and is already used in a number of real-time courses given at Mälardalen University in Sweden. The developed WCET tool and the real-time laboratory framework will be freely available for academic use.

1 Introduction

Today, tools for static *Worst-Case Execution Time* (WCET) analysis, such as Bound-T [4] and aiT [1], are starting to be used in embedded system development and timing verification [9, 10, 14, 15, 18]. We believe that such tools have a potential to be part of the embedded real-time developer's tool chest, in the same way as profilers, hardware emulators, compilers, and source-code debuggers already are today. By providing easier verification of timing behavior they should provide improvements in product quality and safety, as well as reduced development time.

Unfortunately, too few embedded system developers are yet aware of WCET analysis tools and the functionality they offer. This article describes an attempt to improve this situation. We are currently porting an existing WCET analysis tool, Bound-T, to the Renesas H8/3292 microprocessor. This microprocessor is used in Lego Mindstorms [11], an off-the-shelf kit of Lego bricks for building and controlling small prototype robots. This kit is used in many real-time courses in academia. Thereby static WCET analysis could be regularly used in the education of the embedded system developers of tomorrow.

However, for achieving system predictability and to make best use of calculated WCET estimates, the complete system needs to be developed with timing predictability in mind. To make students aware of this fact, the developed WCET analysis tool will be used in a real-time laboratory framework targeting such system predictability. This framework is already used in a number of real-time courses given at Mälardalen University in Sweden. It consists of a small real-time micro-kernel and an operating system called Asterix [16], a configuration tool called Obelix, and a GNU GCC cross-compiler for the H8/3292.

Both the WCET analysis tool and the real-time laboratory framework will be freely available for academic use. This should provide a valuable foundation for teaching students how to construct better and safer real-time systems.

2 WCET Analysis for Mindstorms

Lego Mindstorms is an off-the-shelf kit of Lego bricks for building and controlling small prototype robots. The simplicity in the design of Lego Mindstorms makes it suitable for educational purposes in ages from 12 years and up. Out of the box, the Mindstorms kit is not considered to be a platform for real-time systems. However, the set of Lego bricks includes sensors and actuators such as pressure sensors, a light sensor and small motors, i.e., the fundamental necessities for real-time systems. Also, the sparse hardware resources and few interfaces for hardware access places the Mindstorms construction in the embedded system category.

The RCX, illustrated in Figure 1, is the processing unit

This research has been supported by the Advanced Software Technology Center (ASTEC) in Uppsala [2]. ASTEC is a Vinnova (Swedish Agency for Innovation Systems) initiative [19].

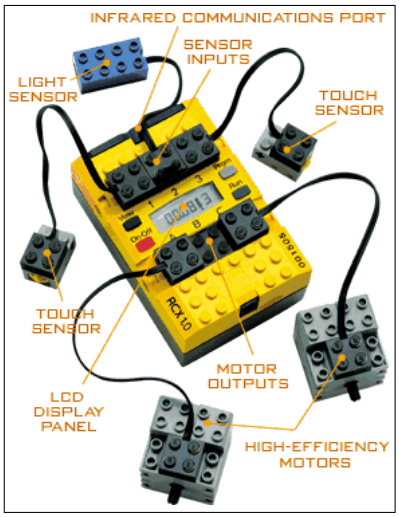


Figure 1. The Lego Mindstorms RCX unit

of Lego Mindstorms. The RCX is based on the single-chip H8/3292 [8], a RISC microcomputer running at 16 MHz. It features a H8/300 CPU core and a complement of on-chip supporting modules. The H8/3292 has 16 kBytes of read-only memory (ROM) and 512 bytes of on-chip random-access memory (RAM) and an additional 16 kBytes of external RAM in a separate circuit. The ROM includes several functions for reading of sensors, controlling the motors, display segments and numbers on a LCD-display. Located on-chip are one 16-bit timer, two 8-bit timers, a watchdog-timer, a serial communication interface and an 8-channel 10-bit analog-digital converter. The RCX also contains an IR-transceiver, useful for downloading programs and for communicating with other RCX units.

2.1 H8/300 Hardware Timing

Instructions in the H8/300 architecture generally have a fixed execution time, the exception being the `EEPMOV` instruction that copies a block of data in memory. There is no cache (at least not on-chip) and no visible pipeline. The fastest instructions take two clock cycles; for example an `ADD.B` or `ADD.W` executed from the on-chip memory using register operands. Complex instructions that access external memory may have execution times of 20 or more cycles, depending on the length of the instruction, the addressing mode, the data width and the memory areas that are accessed. The time does not depend on execution history. Two instructions, `MOVFPE` and `MOVTPE`, synchronize with the ”peripheral” clock and have somewhat variable execution time.

The simple instruction timing means that the high-level flow-path analysis becomes the main problem in WCET analysis for Lego Mindstorms. We chose the Bound-T WCET tool [4] as the basis for our work because its low-level analysis is sufficient for the H8/300, it has a fairly powerful high-level analysis, and its modular structure let us divide the porting work between the WCET analysis group at Mälardalen University and the company behind Bound-T, Tidorum Ltd [4].

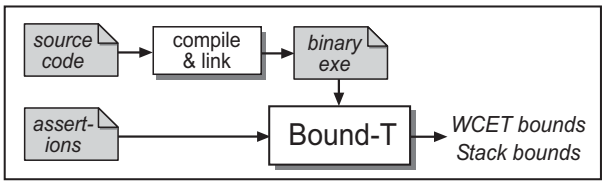


Figure 2. The Bound-T WCET analysis tool

2.2 The Bound-T WCET Analysis Tool

Bound-T, see Figure 2, performs WCET analysis from machine code (binary, linked executables). To find loop bounds Bound-T models the computations and branch conditions with Presburger arithmetic. Bound-T examines the model to find loop-counter variables and computes a bound on loop iterations from the counter’s initial value, its final value implied by the exit condition, and its change (step) in the loop body.

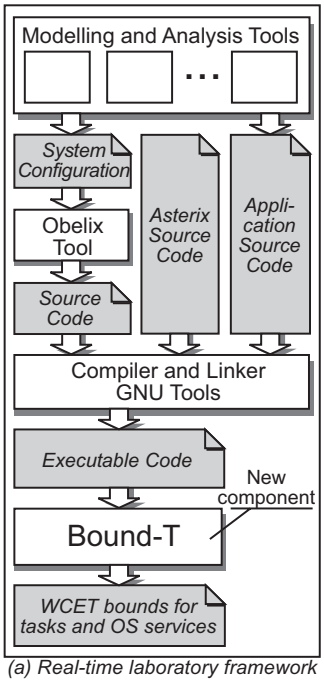
Loop bounds can be context-sensitive, i.e., dependent on the call-path. The Presburger model is used also to resolve dynamic jumps, for example from switch/case statements, and to compute stack usage bounds. The worst-case path is found with implicit path enumeration (IPET).

Bound-T is implemented as a single Ada program with a strict division into modules specific to the target processor (e.g., instruction decoding) and modules independent of the target processor (e.g., analysis of loop bounds). For Presburger analysis Bound-T uses the Omega Calculator [13]. For IPET the lp_solve program [3] is used. Control-flow and call graphs can be emitted in DOT form [6]. The automatic loop analysis can be supported or replaced by user assertions in a separate input file (not as source annotations).

Target processors supported by Bound-T include Intel 8051 [9], SPARC V7 (in its ERC32 implementation) and Analog Devices 21020 DSP [10]. Ports to ATMEL AVR and ARM7 are under way.

2.3 Porting Bound-T to the H8/300

Bound-T is based on an internal model of the target program as a set of control-flow graphs (one for each subprogram) connected into a call-graph. The flow-graph nodes have attributes for the execution time and the arithmetic (computational) effect of the node. The structure of the program model is independent of the target processor but the model is parameterized by target-specific types and operations that are defined in target-specific Ada packages.



(a) Real-time laboratory framework

```
SYSTEMMODE = NORMAL;
RAM = 512000;
MODE mode_1 {
  RESOLUTION = 1000;
  HARD_TASK ht_1 {
    ACTIVATOR  = 100; //period time
    OFFSET     = 0;
    DEADLINE   = 50;
    PRIORITY   = 10;
    STACK      = 50;
    ROUTINE    = ht_1_routine;
    ARGUMENTS  = "1, 2, 3";
    ERR_ROUTINE= ht_1_err_routine;};
  HARD_TASK ht_2 {
    ACTIVATOR  = 50; //period time
    OFFSET     = 0;
    DEADLINE   = 20;
    PRIORITY   = 20;
    STACK      = 50;
    ROUTINE    = ht_2_routine;};
  SOFT_TASK st_1 {
    ACTIVATOR  = sig_1; //trigger
    OFFSET     = 0;
    DEADLINE   = 10;
    PRIORITY   = 10;
    STACK      = 50;
    ROUTINE    = st_1_routine;};
  SIGNAL sig_1 {
    USER       = ht_1;
    USER       = st_1;};
  SEMAPHORE sem_1 {
    USER       = ht_1;
    USER       = ht_2;};
};
```

(b) Example configuration

```
#include <os_kern.h>
#include <sys_spec.h>
void ht_1_routine(void *ignore)
{
     getSemaphore( sem_1 );
     outputLCD( counter1 );
     counter1++;
     releaseSemaphore( sem_1 );
     if( counter1 > 30 ){
         getSemaphore( sem_1 );
         counter2 = 0;
         releaseSemaphore( sem_1 );
     }
}
```

(c) Example task code

Figure 3. The real-time laboratory framework

To port Bound-T to a new target processor, one must implement these target-specific types and operations. The main operation is the one that decodes instructions and inserts them in the program model. This operation is given the address of an instruction and must ”fetch” the binary instruction from the program’s memory image (a COFF file from GCC in our case), decode the instruction to find out the instruction type and the operands, and call Bound-T operations that create new nodes and edges in the control-flow graph.

The porting of the H8/300 to Bound-T took about five months and was performed by the first author, Samuel Petersson, as an MSc project in his computer science studies [12]. The instruction decoding process was divided into two steps: first from the binary instruction to an ”abstract” instruction, and then from the abstract instruction to the Bound-T model. The abstract instruction is a model of the H8/300 architecture built from Ada types. This model depends only on the H8/300, not on Bound-T.

The decoding from binary instructions to abstract instructions included a lot of processor manual reading. It was a considerable job because there are 57 different instructions which can execute in eight different addressing modes. Furthermore, no instructions allow all the addressing modes.

The next step was the conversion of abstract instructions to the Bound-T model. This included creation of flow-graph nodes with timing information and arithmetic effects of the included instructions. The decoding process expands the (abstract) `EEPMOV` instruction into three flow-graph nodes that model the block-copy loop. Bound-T’s usual loop-bound analysis applies here. For the `MOVFPE` and `MOVTPE` instructions we assume the worst-case execution time.

The first version of Bound-T for the H8/300 supports the H8/3297 chip series (which includes the 3292). Tidorum plans to support other H8/300 chips and perhaps other members of the H8 family such as the 32-bit H8/300H processor.

3 Mindstorms in RT Systems Education

The Bound-T tool will be used in a real-time laboratory framework, see Figure 3(a), developed at Mälardalen University. The framework replaces the software architecture and programming environment that are delivered together with the Lego Mindstorms kit. It consists of a small real-time micro-kernel and operating system called Asterix [16], a configuration tool called Obelix, and a GNU GCC cross-compiler for the H8/3292.

3.1 Asterix - the Real-Time Kernel

The Asterix real-time kernel handles execution strategies ranging from strictly statically scheduled systems via fixed priority scheduled systems to event-triggered systems, or any combination of these. To fulfill the needs for embedded systems we have minimized the kernel and the application memory footprints.

A built-in monitoring function facilitates the use of state-of-the-art testing and debugging tools like deterministic testing, replay debugging, and visualization [17]. The kernel also provides on-line facilities for measuring execution times (via testing). For every system reconfiguration the kernel must be recompiled, leading to an efficient usage of memory and other limited resources. Task properties (e.g., deadline, priority, offset) are defined outside the source code in an Obelix configuration file.

3.2 Obelix - the System Configuration Tool

The Obelix system configuration tool allows static off-line definition of system resource demands. The demands are set in a configuration file separated from the source code, clearly separating the system functionality from the system configuration and requirements. Furthermore, Obelix allows cleaner source code in the sense that no special tags and system calls are needed for initialisation and system set up. This gives the possibility of moving the code to the target development environment after testing without modifications.

The configuration files include descriptions of all necessary resource requirements as well as configuration information for e.g., the task schedule, synchronization of tasks, inter-task communication and inter-node communication.

In addition, the configuration files contain all necessary information of task attributes and time resolution. Examples of a configuration file and implemented task functionality are depicted in Figure 3(b) and Figure 3(c) respectively.

3.3 The Real-Time System Student Assignments

On a yearly basis, the laboratory framework is used in two real-time courses (a regular course and a distance course) given at Mälardalen University. For each course, two student assignments are given: a preparatory assignment and a robot project. Because of the straightforward programming (basic C-programming) the preparatory assignments are easily done in a few hours, giving the programming knowledge required for the subsequent robot project. Since the start approximately 500 students have performed 200 robot projects using Lego Mindstorms and Asterix.

Due to the separation of functionality and configuration in the Asterix framework, the students may implement their application in a late stage, and focus more on the theoretical aspects of designing a robust real-time system (e.g., timing analysis, scheduling, etc.). Furthermore, the simplicity of configuring and programming leaves room for proper software design. However, students have previously experienced difficulties to estimate proper execution times by measurements. By instead using Bound-T for this purpose, we believe that the students will be able to derive more accurate timing bounds.

In order to perform response-time analysis and to derive overall system timing guarantees the students need WCET bounds both for tasks and for OS services. To simplify the assignment, WCET bounds for all OS calls will be derived beforehand, and presented in an off-line table. However, the students will be required to use Bound-T to derive WCET bounds for their own robot application task code.

4 Conclusions, Related and Future Work

We have described work on porting the Bound-T WCET analysis tool to the Lego Mindstorms and the H8/3292 microprocessor. The tool will initially be used in assignments in real-time systems courses given at Mälardalen University. This should allow students to get familiar with WCET analysis and should provide valuable feedback on the functionality required for WCET tools to be applicable in real-time system development. The developed WCET tool and the real-time laboratory framework will be freely available for academia and under license for industry.

An alternative OS for Lego Mindstorms, and the developed WCET analysis tool, is BrickOS from Sourceforge [5]. BrickOS supports preemptive multitasking, dynamic memory management, POSIX semaphores, as well as native to display, buttons, IR communication, motors and sensors. However, compared to Asterix, BrickOS is not a hard real-time OS, making it more difficult to provide overall system timing guarantees.

The H8/300 has previously been ported to the Heptane WCET tool [7] and the BrickOS. However, we have not seen any reports on using the tool in education.

Future work includes a systematic validation of the developed Bound-T H8/300 timing model using measurement tools such as oscilloscopes and logic analyzers. This will minimize the possibility of implementation faults and verify that the timing given in the H8/300 processor manual [8] actually corresponds to the real hardware timing.

References

[1] AbsInt company homepage, 2005. `www.absint.com`.

[2] ASTEC homepage, 2005. `www.astec.uu.se`.

[3] M. Berkelaar. *lp_solve: (Mixed Integer) Linear Programming Problem Solver*, 2004. `ftp://ftp.es.ele.tue.nl/pub/lp_solve`.

[4] Bound-T tool homepage, 2005. `www.tidorum.fi/bound-t/`.

[5] BrickOS homepage, 2005. `brickos.sourceforge.net`.

[6] Homepage for the Graphviz tool, Aug 1997. `www.graphviz.org`.

[7] Homepage for the Heptane WCET analysis tool, 2005. `www.irisa.fr/aces/work/heptane-demo/heptane.html`.

[8] Hitachi. Hitachi Single-Chip Microcomputer H8/3297 series. *Hardware manual, 3rd edition*, 2000.

[9] N. Holsti, T. Långbacka, and S. Saarinen. Using a Worst-Case Execution-Time Tool for Real-Time Verification of the DEBIE software. In *Proc. of the DASIA 2000 Conference (Data Systems in Aerospace 2000, ESA SP-457)*, Sep 2000.

[10] N. Holsti, T. Långbacka, and S. Saarinen. Worst-Case Execution-Time Analysis for Digital Signal Processors. In *Proc. of the EUSIPCO 2000 Conference (X European Signal Processing Conference)*, Sep 2000.

[11] Lego Mindstorms homepage, 2005. `www.legomindstorms.com`.

[12] S. Petersson. Porting the Bound-T WCET tool to Lego Mindstorms and the Asterix RTOS. Master's thesis, Mälardalens University, Västerås, Sweden, May 2005.

[13] William Pugh. The Omega test: a Fast and Practical Integer Programming Algorithm for Dependence Analysis. In *Supercomputing*, pages 4–13, 1991.

[14] S. Byhlin, A. Ermedahl, J. Gustafsson, B. Lisper. Applying Static WCET Analysis to Automotive Communication Software. In *Proc. 17^{th} Euromicro Conference of Real-Time Systems, (ECRTS'05)*, July 2005.

[15] D. Sandell, A. Ermedahl, J. Gustafsson, and B. Lisper. Static Timing Analysis of Real-Time Operating System Code. In *Proc. 1^{st} International Symposium on Leveraging Applications of Formal Methods (ISOLA'04)*, Oct 2004.

[16] H. Thane, A. Pettersson, and D. Sundmark. The Asterix Real-Time Kernel. In *Proc. 13^{th} Euromicro Conference of Real-Time Systems, (ECRTS'01)*, June 2001.

[17] Henrik Thane. *Monitoring, Testing and Debugging of Distributed Real-Time Systems*. PhD thesis, Royal Institute of Technology, Stockholm, May 2000.

[18] S. Thesing. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*. PhD thesis, Saarland University, 2004.

[19] Vinnova homepage, 2005. `www.vinnova.se`.

Analysis of Memory Latencies in Multi-Processor Systems

J. Staschulat, S. Schliecker M. Ivers, R. Ernst
Technical University of Braunschweig
Hans Sommer Str. 66, D-38106 Braunschweig, Germany
{staschulat|schliecker|ivers|ernst}@ida.ing.tu-bs.de

Abstract

Predicting timing behavior is key to efficient embedded real-time system design and verification. Current approaches to determine end-to-end latencies in parallel heterogeneous architectures focus on performance analysis either on task or system level. Especially memory accesses, basic operations of embedded application, cannot be accurately captured on a single level alone: While task level methods simplify system behavior, system level methods simplify task behavior. Both perspectives lead to overly pessimistic estimations.

To tackle these complex interactions we integrate task and system level analysis. Each analysis level is provided with the necessary data to allow precise computations, while adequate abstraction prevents high time complexity.

1 Introduction

Memory is a critical bottleneck in embedded systems, as the gap between processor speed and memory access time is increasing. Current chip designs use hierarchical memory architectures with caches, multi-threading and co-processors to reduce memory latency time. Embedded applications often require real time constraints, but performance verification of complex systems is a challenge.

State-of-the-art in industrial practice is using functional test and simulation. Simulation times are often too long for a complete code coverage, which would need exponential time. Therefore, only critical paths are tested, and safe timing bounds cannot be given.

Formal analysis is an alternative. With simplified assumptions the analysis complexity is reduced and safe upper and lower performance bounds are calculated. One such assumption is a constant memory latency, even though it is influenced by the memory hierarchy, bus arbitration, and buffer sizes as well as the background memory latency. Static analysis approaches for single tasks assume constant memory access time, such as [5]. The behavior of caches has been studied as well [5] [3] [8], but these approaches assume a constant cache miss penalty. A small overestimation of memory access time will lead to a high overestimation for worst case execution time of single tasks.

This overestimated task execution time is used for higher level system analysis for resource scheduling or for throughput estimation of heterogeneous multi-processor architectures. Such analyses have been proposed by [7] [6]. Compositional performance analysis methodologies combine local techniques on the system level by connecting their input and output event streams according to the overall application and communication structure [4] [1].

Crowley and Baer propose in [2] an analysis for multi-threaded processors. They identify parallelism in the execution of individual threads on a processor and use special nodes with a negative execution time to model the gain of multithreading.

While task level methods simplify system behavior, system level methods simplify task behavior. Both perspectives lead to overly pessimistic estimations, one reason for the marginal influence of formal methods in industrial system design. To tackle these complex interactions task level and system level analysis are integrated. Each analysis level is provided with the necessary data to allow precise computations, while adequate abstraction prevents state space.

The paper is structured as follows. Section 2 describes the problem statement and Section 3 reviews previous work. Section 4 describes the integrated task and system level analysis approach. Finally, we conclude in Section 6.

2 Problem Statement

A simple multi-processor architecture is given in Fig 1. Two processors are connected by a shared bus with memory and a co-processor. For example, suppose that on CPU_1 runs a heat control application. A sensor on CPU_2 periodically transmits temperature values, which are saved in the shared memory. The application on CPU_1 checks this value but also loads its instructions and other data from this

memory device. The heat control is managed by the co-processor, which is triggered by the application on CPU_1.

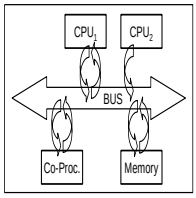


Figure 1. Multi-processor design example.

This simple setup already shows the analysis complexities: Memory traffic by the CPU_1 and CPU_2 uses a shared bus. The memory access time depends on the system state, including bus state, buffers and memory state. We will call a memory or co-processor request latency of a resource as *transaction latency* in this paper. This denotes the end-to-end latency, including the time for requesting the data, transmission over the bus, processing on the remote component and transmission to its source.

The objective is to compute the worst case execution time (WCET) of a task, e.g. the heat control on CPU_1, while analyzing transaction times on system level.

3 Previous Work

3.1 Single Task Analysis

The timing analysis of tasks is separated in two stages: program path analysis and micro-architecuture modeling. Program path analysis is used to determine the path which is executed in the worst case. To derive all the possible program paths the program is transformed into a control-flow graph. Based on this control-flow graph the worst case path which starts at the beginning node and ends at an terminating node of the control-flow graph is determined.

Micro-architectural modeling is understood as the timing analysis of sequences of instructions. Li and Malik [5] integrate program path analysis and micro-architecture modeling to analyze the worst case execution time of tasks under the influence of instruction caches. They use an Integer Linear Programm (ILP) to avoid an explicit enumeration of all program paths. The micro-architectural model is used to derive execution-time of individual basic-blocks and the ILP is used to find the path through the control-flow graph which maximizes the execution time spent. In [8] means are proposed for identifying infeasible paths in the program and the timing analysis is based on single-feasible-paths instead of basic-blocks.

In this paper we use this analysis framework, called SymTA/P, to transform a program into its control flow graph and to compute the WCET by solving the ILP. We assume that the execution time of single-feasible paths are given including micro-architectural influences. SymTA/P assumes that a conservative memory access time, either as cache miss penalty or general memory access time, must be specified a priori. A more precise estimation of the worst case memory access time should rather be done at system level, where timing properties of all other components are available.

3.2 System Level Analysis

In SymTA/S [4] a compositional performance analysis methodology is proposed which integrates different local scheduling analysis techniques through event stream propagation. The local techniques are composed on the system level by connecting their input and output event streams according to the application and communication structure. Instead of considering each event individually, as simulation does, the formal scheduling analysis abstracts from individual events to event streams. The analysis requires only a few simple characteristics of event streams, such as an event period, a maximum jitter and minimum distance. From these parameters, the analysis systematically derives worst case scheduling scenarios.

One way to extend the compositional analysis to memory accesses would be to model each access as an event. However, this would require splitting the task into many smaller atomic tasks and it would lead to a complex task description.

4 Integrated Analysis Approach

Our approach integrates the single task analysis and global system analysis. Fig. 2 shows the workflow of the integrated analysis. The global system analysis consists of tasks and event streams. A task i consumes tokens from an input event stream, E_i^{in} and produces tokens for the output event stream E_i^{out}. The event streams are connected to tasks according to the overall application.

Memory access are modeled with additional communication between task and system level. The task requests a number of memory transactions, T_{mem} from the system level (dashed arrow). The memory request is propagated via a system path. Based on the event model of the remaining tasks, the end-to-end latency of this request $L(T_{mem})$ is computed by the system level analysis. This latency is used by the task level analysis to compute the worst case response time (WCRT), which finally determines its output event model E_i^{out}. All tasks of the system which share event streams on this path, or which are directly affected by the memory access, have to be re-analyzed in the next iteration loop. The analysis will stop when a fixed point is found.

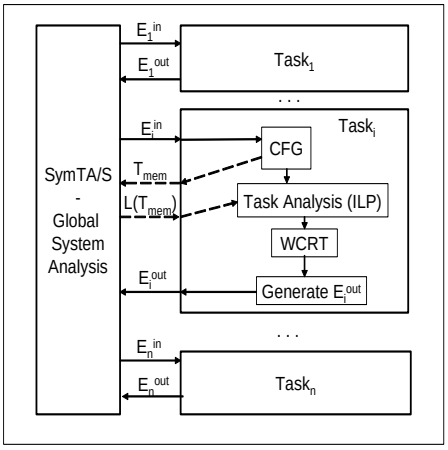


Figure 2. Workflow of integrated analysis.

This abstract modeling allows several compositional implementations. From task level only the memory requests need to be specified. This means any task level analysis which can determine the memory access patterns can be used here. In the next section we describe in detail how we extend our single task analysis. In Section 4.2 we present an extension to our system level analysis, which implements this general framework.

4.1 Extension of Single Task Analysis

Worst case execution time analysis is based on the control-flow graph (CFG) of an application. To include compiler optimizations, the assembly code is parsed and the corresponding CFG is constructed. At this stage we assume that the core execution time has been computed. For each basic block (or single-feasible pathes) the number of memory accesses is extracted. For example, if a basic block i requests three 32 byte memory blocks, then the transaction request is $T_{mem}(b_i) = (32,32,32)$. For each basic block such a request is generated. In the second step all requests are collected and propagated to the system level analysis where the transaction latency $L(T_{mem}(b_i))$ considering the system state is computed. This is described in Section 4.2.

The worst case execution time is computed by solving the ILP, which is constructed from the control flow graph. The general ILP consists of an objective function and several structural and functional constrains. Structural constrains represent possible control flow and functional constraints describe loop bounds or denote infeasible pathes. The sum-of-basic blocks, as proposed by Li and Malik [5] is given by the following objective function: $\max \sum_{i=1}^{n} c_i \cdot x_i$ where c_i denotes the core execution time of basic block i, n the maximum number of basic blocks of the task and x_i the execution count of basic block i. We extend the objective function to $\max \sum_{i=1}^{n} (c_i + L(T_{mem}(b_i))) \cdot x_i$ to include the memory access time of $T_{mem}(b_i)$ memory access during the basic block. Further below we omit b_i and use only the term T_{mem} for simplicity reasons. The ILP is then solved to find the longest execution path in the program.

4.2 System-Level Analysis of Transaction Latency

The basic SymTA/S model of [4] is composed of tasks and event streams. In the following we show how to extend the SymtTA/S framework to compute $L(T_{mem})$. A transaction starts at some task τ_i and is transmitted via a chain of intermediate tasks and ends at the same task τ_i. The total latency of this transaction is denoted by $L(T_{mem})$.

A request T_{mem} has to be translated into some event stream, which SymTA/S supports. An event stream is a tuple $E = \{\mathcal{P}, \jmath, d_{min}\}$, where $\mathcal{P}$ denotes the period, $\jmath$ the jitter and d_{min} the minimum distance between two events. Given these parameters, the system level analysis calculates the worst case response time $R(E)$ for this event stream E along the chain considering every other event stream of the system, which relates to this chain. In this paper we assume that the transactions of other ressources are given and are independent of system behavior. From the response time $R(E)$ the response time of the transaction $L(T_{mem})$ is calculated. First, we give a translation for single transactions. In a second step we translate multiple transactions.

4.2.1 Single Transaction

The idea for a single transaction, such as $T_{mem} = \{32\}$, is to restrict an event stream to a single event by choosing a period greater than the jitter. If the period were smaller then the jitter the second event of the event stream could arrive together with the first one. We define the event stream E for a single transaction by $\mathcal{P} = \jmath'$, $\jmath = 0$ and $d_{min} = 0$. The minimum distance is zero because this parameter is not used. The jitter is set to zero, to exclude any future events. As period we choose a large number, e.g. $\jmath'$. If the jitter along the chain, $\jmath^{max}$, is larger than the initially assumed $\jmath'$, the period will be adjusted to $\jmath^{max} + 1$ and the system analysis is called for a second time.

4.2.2 Multiple Transaction

Multiple events, such as $T_{mem} = \{32,32,32\}$, are modeled as a burst, this means all requests are issued at the same time instant. As period $\mathcal{P}$ we start with a great value, $\jmath'$. The jitter is set to $\mathcal{P} \cdot (|T_{mem}| - 1)$ to guarantee that exactly $|T_{mem}|$ events arrive together. Future events will be excluded if the period is greater than the total jitter $\jmath^{max}$ on the chain. Unfortunately, $\jmath^{max}$ is only available after the first system level analysis iteration. So possibly a second iteration is

necessary. We define E for multiple transactions by $\mathcal{P} = \mathcal{J}'$, $\mathcal{J} = \mathcal{P} \cdot (|T_{mem}| - 1)$ and $d_{min} = 1$.

We assume a minimum distance of one instruction to be conservative. In the future we will analyze the minimum distance between memory accesses within basic blocks to increase analysis precision. We also restrict each event to request the same amount of data for simplicity reasons. For instruction caches this is not a limitation, since entire cache lines are requested.

Now the system analysis can compute the response time $R(E)$ for single and multiple events. In case of single events this is equal to the response time of the transaction $L(T_{mem}) = R(E)$. In the case of multiple events the $R(E)$ denotes the response time of that event, which possesses the worst case response time among all events of this event stream. This includes latencies of previous events. Because it is unknown which event caused the worst case response time, we have to assume that it is the last one. Therefore

$$L(T_{mem}) = (|T_{mem}| - 1) \cdot d_{min} + R(E) \quad (1)$$

The first term denotes the time until the last event is activated. This completes the description of system chain analysis. All tasks belonging to this chain need to be reanalyzed, since their input event model might have changed due to the transaction.

5 Experiments

We applied the analysis to the architecture as shown in Fig. 1. For comparison, we use a simulation environment for a network processor and an isolated approach, where every transaction is assumed to take the maximum time observed during the simulation.

Eight applications are executed on CPU_1. For simplicity we provided some background traffic for CPU_2 and the the Co-processor. Figure 3 compares the worst case response times for constant memory access time (isolated), the WCRT of the analysis described in this paper (integrated) and the maximum WCRT observed in simulation.

The analysis provides a significantly reduced WCRT compared to constant memory access times. As simulation cannot provide the worst case scenario we cannot evaluate the accuracy using simulation.

6 Conclusions and Future Work

In this paper we have integrated a static timing analysis on task level and a schedulability analysis on system level. Current approaches focus only on one level which assume a constant delay for each memory access. In this approach several memory transactions within a basic block

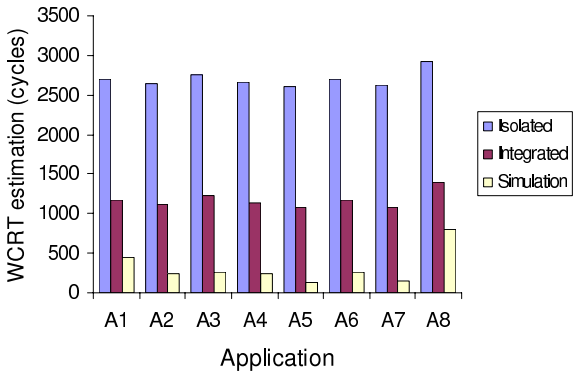


Figure 3. Experiment for WCRT analysis.

are grouped together. In a second step, the total access time of these transactions is determined by system level schedulability analysis. The experiment shows that the analysis precision increases significantly compared to the isolated approach.

Future research includes to consider the type and size of a memory transaction as well as as a greater minimum distance of memory accesses beyond basic blocks.

References

[1] S. Chakraborty, S. Künzli, and L. Thiele. A general framework for analysing system properties in platform-based embedded system designs. In *Design, Automation, and Test in Europe*, Munich, Germany, March 2003.

[2] P. Crowley and J.-L. Baer. Worst-case execution time estimation for hardware-assisted multithreaded processors. In *The Second Workshop on Network Processors*, Anaheim, California, USA, Feb. 2003.

[3] C. Ferdinand and R. Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, 1999.

[4] M. Jersak, K. Richter, and R. Ernst. Performance analysis for complex embedded applications. *International Journal of Embedded Systems, Special Issue on Codesign for SoC*, 2004.

[5] S. Malik and Y.-T. S. Li. *Performance Analysis of Real-Time Embedded Software*. Kluwer Academic Publishers, 1999.

[6] T. Pop, P. Eles, and Z. Peng. Holistic scheduling and analysis of mixed time/event-triggered distributed embedded systems. In *CODES*, Eates Park, Colorado, USA, May 2002.

[7] K. Tindell, A. Burns, and A. Wellings. An extendible approach for analysing fixed priority hard real-time systems. *Journal of Real-Time Systems*, 6(2):133–152, March 1994.

[8] F. Wolf, J. Staschulat, and R. Ernst. Hybrid cache analysis in running time verification of embedded software. *Design Automation for Embedded Systems*, 7(3):271–295, Oct 2002.

Efficient Analysis of Pipeline Models for WCET Computation

Stephan Wilhelm
AbsInt GmbH and Saarland University
Saarbrücken, Germany
sw@absint.com

Abstract

Worst-case execution time (WCET) prediction for modern CPU's cannot make local assumptions about the impact of input information on the global worst-case because of the existence of timing anomalies. Therefore, static analyses on the hardware level must consider a large subset of the reachable states of the underlying hardware model. As the number of states grows, WCET prediction can become infeasible because of the increase in computation time and memory consumption. This paper presents a solution for this problem by defining the static analysis of processor pipelines for WCET computation in terms of operations on binary decision diagrams (BDD's).

1. Introduction

Finding the worst-case execution time (WCET) for all tasks of a software is an important requirement in the design of hard real-time systems. Because the execution time depends on the underlying processor hardware, WCET computation requires a detailed analysis of the hardware behavior for the analyzed task. For CPU's using modern techniques for reducing the average execution time, such as caches, pipelined execution, branch prediction, speculative execution, and out-of-order execution, the WCET cannot be obtained by measurements because it is usually not possible to determine the worst-case inputs manually.

A proven approach for obtaining tight upper bounds of the WCET has been presented in [8]. It employs several semantics-based static program analyses on the assembly level control flow graph (CFG) of the input program. First, the *value analysis* computes the address ranges for instructions accessing memory. In a second step, an integrated *cache-* and *pipeline-analysis* predicts the cache behavior [7] and the behavior of the program on the processor pipeline [13]. The result of the pipeline analysis is the WCET for each basic block from which a subsequent *path analysis* [12] computes the global worst-case path.

Pipeline analysis computes sets of pipeline states that can occur at any point in the program. Imprecisions in its input information, arising from unknown memory accesses or unknown cache behavior (may be cache hit or miss), cause situations where the pipeline analysis must consider several possible successor states for each incoming pipeline state. Unfortunately, it has been proven that for CPU's, using modern techniques for reducing the average execution time, it is not possible to decide locally which element from the input set triggers the global worst-case behavior. E. g. a cache *hit* might contribute to the global worst-case. Such cases have been termed *timing anomalies* [9]. Because of the presence of timing anomalies, pipeline analysis must consider all possible successor states. For complex pipelines with large state spaces, the analysis can become infeasible because of the increase in memory consumption and computation time [13]. This problem is known as *state explosion* and it is also a well known phenomenon in the area of model checking. The use of ordered binary decision diagrams (OBDD's) [4] for symbolic set operations has significantly reduced the state explosion problem for model checking and the size of systems that have been successfully verified by model checking has increased ever since [5]. The key idea of this paper is to define pipeline analysis in terms of BDD [1] operations, similar to symbolic model checking. It can be expected, that this will reduce runtime and memory consumption of pipeline analyses, making the analysis of complex pipelines feasible, even for large programs.

2. Finite state machines

Processor pipelines can be regarded as finite state machines (FSM's) and pipeline analysis can be defined as a computation on sets of states of the FSM for the analyzed pipeline. The efficiency of the presented approach for pipeline analysis relies on the BDD-based representation of FSM's which is introduced in this section.

[1] The terms BDD and OBDD are used interchangeably in this text.

Definition 1 A *Finite State Machine (FSM)* M is a triple, (Q, I, T), where Q is the set of states, I is the set of input values, and $T = (Q \times I \times Q)$ is the transition relation.

Each set of FSM states $A \subseteq Q$ can be associated to its *characteristic function* $\mathbf{A} : Q \rightarrow \{0,1\}$; $\mathbf{A}(x) = 1 \Leftrightarrow x \in A$. In the same way, the transition relation T can be associated to the function $\mathbf{T} : Q \times I \times Q \rightarrow \{0,1\}$; $\mathbf{T}(x,i,y) = 1 \Leftrightarrow (x,i,y) \in T$. It is common practice to represent FSM state sets and the FSM transition relation by their characteristic functions encoded as BDD's. This representation has the advantage of compactly representing a large number of commonly encountered functions. Useful operations such as negation, conjunction and existential quantification can be efficiently performed using BDD's.

A *hardware design* consists of a set of interconnected latches and gates. A design with n latches and m input wires is characterized by an associated FSM with state space $Q = \{0,1\}^n$ and input space $I = \{0,1\}^m$. The transition relation is defined by the corresponding logic. For such models, the variables of BDD's for encoding the characteristic functions of states sets represent the latches of the design.

Definition 2 Given a FSM (Q, I, T) and a set of states $A \subseteq Q$. The *image* of A, $\mathrm{Img}(A) \subseteq Q$, is the set of states that is reachable from A under T.

Image computation is the core operation of symbolic model checking algorithms [10]. Section 4 shows that it can also be used for dataflow analysis of pipeline models.

Image computation can become infeasible for large designs if the transition relation is given as a single BDD [3] but there are efficient algorithms for image computation that avoid building the monolithic transition relation by exploiting the fact that the FSM transition relation can be factored into the transition relations of the involved latches [10]. This technique is known as *conjunctive partitioning* of the transition relation.

3. Specifying pipeline models

Hardware description languages like VHDL or Verilog have been designed for writing concise descriptions of hardware designs in terms of latches and update logic. It has been shown that such specifications can be compiled into (timed) finite state machines [6]. The *VIS* system for model checking and synthesis of hardware designs supports a substantial subset of Verilog extended by an expression for specifying non-deterministic behavior. Specifications in Verilog are compiled using the *vl2mv* compiler and the resulting description of the system as a finite automaton can be used for CTL modelchecking and reachability analysis [11].

```
reg [0:3] cycles;
reg [0:1] instr;
reg [0:1] delay;

initial cycles = 0;
initial instr = 0;
initial delay = 0;

always @(clk_first) begin
   if (cycles == 7)
      cycles = 0;
   else
      cycles = cycles + 1;
end

always @(clk_second) begin
    if (delay == 0)
        delay = get_delay();
    else
        delay = delay - 1;
end

always @(clk_third) begin
    if (delay == 0)
        instr = get_next_instr();
end
```

Figure 1. Example Verilog code for simple FSM.

Low-level HDL specifications, including detailed models of pipeline states and the corresponding logic, are readily available for many CPU's and can be compiled into finite automata by vl2mv. The resulting automata are often too large for most kinds of analyses but the problem can be overcome by applying suitable abstractions to the original description. Automatic abstraction from HDL models is a field of ongoing research [2]. HDL's also support behavioral descriptions of hardware which can be used to describe the (timing) behavior of a design as specified by the manual. Such descriptions are usually more compact and the resulting automata are smaller.

Figure 1 shows an example of Verilog code for a FSM with a two bit delay counter. Note that the declaration `reg[k:l]` denotes a set of $(l - k) + 1$ variables of the FSM state with value 0 or 1. Let us assume that this FSM is a simplified pipeline model [2]. It has an instruction pointer, `instr`, for 3 instructions and a delay counter, `delay`, that is initialized with the delay for

[2] Although the FSM of figure 1 is not a pipeline model, it is sufficient for illustrating the principles of the presented approach for pipeline analysis.

instruction	delay
0	1
1	2
2	{0,1}

Table 1. Example program input for analyzing the model from figure 1.

each instruction. The `cycle` counter counts execution cycles of basic blocks with at most 7 cycles. Execution of one cycle is done in three steps, indicated by the signals `clk_first`, `clk_second` and `clk_third`. Whenever the delay counter reaches 0 and the signal `clk_second` is active, a new value for `delay` is read using the function `get_delay()`. Similarly, the new value for `instr` is obtained from the function `get_next_instr()` when `delay` is 0 and the signal `clk_third` is active.

When generating a FSM for this description, the input functions are modeled as non-deterministic transitions. Thus, image computation for a state where `delay` is 0 will yield the 3 possible successor states where `delay` takes the values 0, 1 or 2.

4. Pipeline analysis

Given a hardware model by an FSM, pipeline analysis performs a fixed point iteration on the domain $P(Q)$ of pipeline states. The least fixed point (LFP) is the solution to the data flow problem containing all FSM states that are reachable for a given program point and also containing the WCET state. Note that the FSM state comprises a counter for execution cycles of basic blocks (the number of execution cycles per basic block is clearly finite). The WCET for each basic block B is found by selecting the state with the highest value for the execution cycle counter from all states where the last instruction belonging to B has finished.

4.1. Transfer functions

The transfer functions for pipeline analysis compute the next states for each FSM transition in all current states. In general, this is an image computation with the restriction that program analysis is only interested in the set of reachable states under the *concrete* inputs of the program. Image computation as defined in section 2 determines the set of reachable states for *all* possible inputs. Let A_0 be the set of initial states of the FSM (Q, I, T). Then, the following fixed point calculation computes the set of reachable FSM states:

$$A_{k+1} = A_k \cup \text{Img}(A_k)$$

The problem of encoding concrete inputs of the analyzed program can be solved by constructing BDD's for the states where inputs are read and BDD's for the concrete inputs themselves. Remember that the BDD variables for a hardware design are the latches of the design. Let $\cdot$ denote the conjunction of BDD variables and $\neg$ is the negation of a variable. For the example of figure 1, the state where the delay for instruction 1 is read by the function `get_delay()` can be encoded as follows:

$$\mathbf{J}_1 = \texttt{instr<0>} \cdot \neg\texttt{instr<1>} \cdot \neg\texttt{delay<0>} \cdot \neg\texttt{delay<1>} \cdot \texttt{clk\_second}$$

This is the state where the instruction pointer is 1, the value of the delay counter is 0 and the signal `clk_second` is active. Table 1 specifies that the delay for instruction 1 is 2. This concrete input can be encoded as follows:

$$\mathbf{C}_1 = \neg\texttt{delay<0>} \cdot \texttt{delay<1>}$$

The BDD's $\mathbf{J}_1$ and $\mathbf{C}_1$ can be regarded as the characteristic functions of the state sets J_1 and C_1 where the variables have the values encoded in the BDD's. For a set A_k of FSM states, the next states for the concrete input at this program point are then computed by the following formula:

$$A_{k+1,1} = (\text{Img}(A_k \cap J_1)) \cap C_1$$

For n concrete inputs, the next states for the set of states where *no* concrete input information is required is calculated as follows:

$$A_{k+1,-} = \text{Img}(A_k \setminus \bigcup_{0 \le l \le n} J_l)$$

Finally, the fixed point iteration for pipeline analysis under n concrete input informations from the input program can be computed as:

$$A_{k+1} = (\bigcup_{0 \le l \le n} A_{k+1,l}) \cup A_{k+1,-}$$

Please note that imprecise input information can also easily be encoded as a BDD. E. g. the delay information for instruction 2 in Table 1 is either 0 or 1. The BDD for this input is simply $\mathbf{C}_2 = \neg\texttt{delay<1>}$. The sequence in which instructions are analyzed is also an input to the model of figure 1. This input can be determined from the program's CFG and encoded in the same way as the delay input.

The success of BDD based algorithms depends on the size of the involved BDD's, which is very sensitive to the ordering of the BDD variables. Finding a minimum sized BDD for a given logic function is algorithmically intractable. However, there are many heuristics for finding good variable orderings [10]. A good variable ordering must only be found once for each pipeline model.

5. Work in progress

A prototype of the presented approach for pipeline analysis has been implemented for a simple pipeline similar to the ARM7 pipeline. This prototype is currently being integrated with the aiT [1] framework for WCET analyses. For the future, we are planning to model more complex CPU's like the Infineon Tricore and the Motorola Power PC family of processors (MPC5xx and MPC755). The MPC755 is the most challenging and interesting target because of the huge state space of the pipeline model. We expect to achieve a significant reduction of computation time and memory consumption compared to the existing implementation.

6. Conclusion

We have shown that the application of well-known techniques for handling large state sets from the area of model checking to the program semantics-based analysis of pipeline models, can help to handle the increasing complexity of modern processor hardware for WCET computation. For large state sets, BDD based algorithms are more space efficient and faster than implementations using an explicit representation of pipeline states.

Furthermore, we have established a connection between pipeline analysis implementation and pipeline specifications written in Verilog or VHDL. Generating the pipeline analysis from the same specification used for hardware synthesis is faster and less error-prone than the difficult way of manual implementation. Finally, the important task of verification of the analysis is also simplified for analyses generated from HDL specifications.

References

[1] http://www.absint.com/aiT/.
[2] http://www.avacs.org.
[3] A. Aziz, S. Tasiran, and R.K. Brayton. BDD Variable Ordering for Interacting Finite State Machines. In *31st ACM/IEEE Design Automation Conference (DAC)*, San Diego, CA, 1994.
[4] R. Bryant. Graph based algorithms for boolean function manipulation. In *IEEE Transactions on Computers*, 1986.
[5] J. Burch, E. Clarke, K. McMillan, D. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. IEEE Comp. Soc. Press, 1990.
[6] S.-T. Cheng. Compiling Verilog into Automata, 1994.
[7] C. Ferdinand. *Cache Behavior Prediction for Real-Time Systems.* PhD thesis, Saarland University, 1997.
[8] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In *Proceedings of EMSOFT 2001, LNCS 2211*, 2001.
[9] T. Lundquist and P. Stenström. Timing Anomalies in Dynamically Scheduled Microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, 1999.
[10] R. Ranjan, A. Aziz, R. Brayton, B. Plessier, and C. Pixley. Efficient BDD algorithms for FSM synthesis and verification, 1995.
[11] The VIS Group. VIS user's manual.
[12] H. Theiling. ILP-based Interprocedural Path Analysis. In *Proceedings of the Workshop on Embedded Software*, Grenoble, France, 2002.
[13] S. Thesing. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*. PhD thesis, Saarland University, 2004.

Classification of Code Annotations and Discussion of Compiler-Support for Worst-Case Execution Time Analysis *

Raimund Kirner, Peter Puschner
Institut für Technische Informatik
Technische Universität Wien
Treitlstraße 3/182/1
A-1040 Wien, Austria
{raimund,peter}@vmars.tuwien.ac.at

Abstract

Tools for worst-case execution time (WCET) analysis request several code annotations from the user. However, most of them could be avoided or being annotated more comfortably if the compilers would support WCET analysis.

This paper provides a clear categorization of code annotations for WCET analysis and discusses the positive impact on code annotations a compiler-support on WCET analysis would have.

1 Introduction

The knowledge of the worst-case execution time (WCET) is a mandatory prerequisite for the design of safety-critical embedded systems, since embedded systems have to fulfill the temporal requirements imposed by their physical environment.

Current research on compilers for embedded systems mainly focuses on issues like reduction of energy consumption, resource-aware code generation, or retargetable code generators. Program execution time is typically covered - as in traditional compiler construction - by performance-oriented code optimizations. The real-time behavior of programs is rarely covered.

As a consequence, a WCET analysis tool has to request the user for numerous code annotations, mostly at object code level, which could be avoided if the compilers explicitly support WCET analysis. Furthermore, the compiler-support would allow to specify code annotations at the source code level instead of burdening the user with object code annotations. Existing mechanisms like debug information is not sufficient in case of code optimizations performed by the compiler.

This paper categorizes in Section 2 the code annotations used by WCET analysis frameworks. Methods for providing such annotations are described in Section 3. The positive impact a compiler providing explicit support for WCET analysis would have on usage of these code annotations is discussed in Section 4.

2 Code Annotations for WCET Analysis

The calculation of the worst-case execution time (WCET) for a piece of code in general requires further information about the possible execution context or runtime behavior of the code. For example, the chosen configuration of the hardware platform has to be specified. Furthermore, the program analysis method may fail to predict the full execution behavior of a program with complex control flow and therefore, explicit assertions about the program behavior are required. These examples give an idea of what information is required by a WCET analysis tool additionally to the input program code. The specification of this additional information is done by code annotations. This section categorizes the different classes of code annotations required for WCET analysis and discusses possible methods to specify them.

Due to limitations on computability, a WCET analysis framework that is capable to analyze industrial code within a realistic software production process requires interfaces for the explicit specification of miscellaneous parameters. Some of these parameters are not

*This work has been partially supported by the FIT-IT research project "Model-Based Development of distributed Embedded Control Systems (MoDECS)" and the ARTIST2 Network of Excellence of IST FP6.

directly related to the WCET calculation itself, but are required to parse and interpret the program code. Therefore, we also looked at code annotation mechanisms provided by commercial WCET analysis tools like aiT[1] [3, 6] or Bound-T[2] [8, 7].

The code annotations for WCET analysis can be categorized as follows:

1. Platform Property Annotations (PPA)
2. CFG Reconstruction Annotations (CRA)
3. Program Semantics Annotations (PSA)
4. Auxiliary Annotations (AA)

Auxiliary annotations are constructs of annotation languages that are used to reference certain locations or control-flow edges in a program code. For example, a symbolic name that will be used later on within other code annotations, is assigned to a specific code location. The other categories of code annotations are described in the following subsections.

2.1 Platform Property Annotations

Platform Property Annotations (PPA) are application-independent annotations, which are used to characterize the target platform. A WCET analysis framework supports one or more target platforms. In case of a strictly static WCET analysis tool, it uses a built-in hardware model for each target platform. However, a computing platform typically can be configured in many ways. For example, there may be caches available with different layouts, or, as another example, the assignment of data and code to the available memory configuration can be done in different ways. Furthermore, to represent the calculated WCET bound as real time instead of processor cycles it is required to annotate the selected clock frequency for the processor.

The PPA annotations described above are used, for example, to parameterize the hardware models of caches and pipelines. Since in this case the annotations are not directly bound to the application code, there is no need of compiler support for such annotations. However, PPA annotations may be attached to the program code for the sake of code optimizations. For example, annotations about the use of read-only or write-only memory regions can be combined with annotations about their assignment to program code. This may allows a compiler to optimize the access operations for these data areas.

[1] http://www.absint.de
[2] http://www.bound-t.com

2.2 CFG Reconstruction Annotations

The CFG Reconstruction Annotations (CRA) are used as guidelines for the analysis tool to construct the control flow graph (CFG) of a program. Without these annotations it may not be possible to construct the CFG from the object code of a program.

On the one side, annotations are used for the construction of syntactical hierarchies within the CFG, i.e. to identify certain control-flow structures like loops or function calls. For example, a compiler might emit ordinary branch instructions instead of specific instructions for function call or return. In such cases it might be required to annotate a branch instruction whether it is a call or return instruction. A work around that sometimes helps avoiding code annotations is to match code patterns generated by a specific version of a compiler. However, such a "hack" cannot cover all situations and may also have the risk of incorrect classifications, for example, if a different version of the compiler is used.

On the other side, annotations may be needed for the construction of the CFG itself. This may be the case for branch instructions where the address of the branch target is calculated dynamically. Of course, static program analysis may identify a precise set of potential branch targets for those cases where the branch target is calculated locally. In contrast, if the static program analysis completely fails to bind the branch target, it has to be assumed that the branch potentially precedes each instruction in the code, which obviously is too pessimistic to be able to obtain a useful WCET bound. In such a case, code annotations are required that describe the possible set of branch targets.

2.3 Program Semantics Annotations

Program Semantics Annotations (PSA) are used to guide the calculation of a program's dynamic behavior. In contrast, the annotations of Section 2.2 and 2.1 provide mostly static information about the program to be analyzed and its intended target platform.

To obtain a precise WCET bound, it is mandatory to accurately calculate the possible dynamic behavior of the program. For example, a static WCET analysis tool calculates the dynamic behavior of the program by *exec-time modeling* and by performing *path analysis* (as described in Chapter 2 of [10]). Exec-time modeling means the assignment of execution time to instructions for a given execution context.

To calculate a WCET bound, it is at least necessary to get iteration bounds for every loop or recursive call structure in the program. A quality improve-

ment of the resulting WCET bound is possible if infeasible paths can be excluded from the calculation of the longest path. Annotation languages that allow the explicit specification of flow constraints are described in [9, 1] (also `aiT` and `Bound-T` allow the specification of flow constraints). However, in case the static analysis of the WCET tool performs a semantic analysis of the program, it may be sufficient to indirectly specify the feasible paths by describing properties like value constraints or invariants of program variables.

Another kind of PSA annotation is the description of possible addresses of memory references. Such annotations may improve the path analysis as well as the exec-time modeling.

3 Annotation Methods

This section discusses different methods how to annotate the code. First of all, to get precise results, it is important that WCET analysis is performed at a program representation level close to the executable program format. We call the program representation level where the analysis is performed *object code* level. Following the ongoing trend in embedded systems development, the representation level where the program is developed is much more abstract. By *source code*, we denote the representation level of program development. The whole tool chain that transforms the program from source code to object code is summarized as *compiler*. Following these definitions we can describe things in common terms without loosing generality.

3.1 Separate Annotation Files

One way to annotate code is to use a separate annotation file. This is especially useful for annotating the object code, as there are no common tools to add such information to the object code. Since `aiT` and `Bound-T` are primarily designed to analyze object code, they both support the use of separate annotation files. Both tools have to provide such an annotation technique, due to the missing compiler-support for WCET analysis. Another reason is that the WCET analysis framework should also be able to analyze code that is only available as object code. The obvious drawback of this procedure is that the developer has to look at and understand the object code, which is only an intermediate representation where code locations might change each time the source code is modified and re-compiled.

The support of annotations referring to code locations relative to symbolic labels reduces the amount of code annotations that have to be checked again whenever a single module has been re-compiled.

A more practical way to refer to the program code is to describe the referring code location structurally. For example, `aiT` allows to refer to loops by their order within a function. `aiT` also allows to annotate loops at the source code, but this represents the same mechanism, since the source code locations of these annotations are translated into structural locations. Further, `Bound-T` provides a quite generic pattern matching language that allows to refer to code locations based on various criteria. Using structural references allow the user to annotate for the object code while looking at the source code. However, the drawback of this technique is that it fails in case that the code optimizations performed by the compiler change the structure of the code.

3.2 Annotations within Program Code

Code annotations within the program code provide the advantage that the developer can annotate the program behavior directly where the program is coded. The preferred annotation method from the developer's point of view is to directly annotate the source code.

The concrete syntactical realization of these code annotations is not of stringent importance within this paper. Even the approach of extending the programming language with code annotation constructs allows the compilation by conventional compilers that do not support these language extensions. This can be realized by deactivating the annotations by a preprocessing pass prior to compilation [9]. The more relevant question is whether the compiler provides support for maintaining the consistency of code annotations in case of code optimizations that change the structure of the code. As shown by Exler, the consistency of code annotations may not be maintained without the help of the compiler in case of code optimizations that change the structure of the code [2].

The code annotation within the source code is especially interesting for PSA annotations since this provides the most seamless annotation interface for analyzing and annotating the code manually by the user. PPA annotations are natural candidates for separate annotation files since they refer to the low-level details of the target platform. As a further argument, the PPA annotations are often application independent.

4 Compiler Support for WCET Analysis

The compiler (and all related tools as defined in Section 3.2) transforms the code from the source code representation level to the object code level, at which

WCET analysis is applied. There are several reasons why a compiler can contribute to and improve the calculation of a WCET bound:

- The compiler has the control and knowledge over all code transformations that are performed before emitting the object code. For a number of code optimizations it is not possible to recognize the effect of the optimization by comparing the structure of the object code with that of the source code.
- The compiler has the view on both, the source code and the object code. Typically, the execution behavior of a program is easier to obtain from the source code than from the object code. This is because the instructions in the object code reflect low-level implementation issues enforced by the characteristics of the target hardware optimized for low resource consumption. For example, distinct variables in the source code can become aliased as spilled registers in the object code.

However, due to their lack of support for WCET analysis, compilers are currently not considered as a helpful tool for calculating a WCET bound. Instead, the policy often is to turn off most of the features of a compiler for the sake of generating object code that maintains properties found in the source code. The result is an object code that shows a poor runtime performance and a WCET that is typically much higher than in the fully optimized code.

The intention of having a compiler supporting WCET analysis is to get WCET analyzable code with a seamless interface for code annotations. The support by the compiler can be twofold. First, a seamless integration of code annotations into the source code representation level can be provided. Second, the need for code annotations can be reduced by emitting properties about the object code by the compiler. The following lists several possibilities how compilers could support WCET analysis.

Emit Description of CFG Structure: A static WCET analysis tool has to use CRA annotations at the object code level for reconstructing the CFG of a program. Using such code annotations is a burden for the user of the tool since it forces him to look at the object code level of a program, maybe each time the code is re-compiled.

The compiler knows about the CFG structure at the same precision as it is given by the syntactic structure of the source code. Therefore, the compiler could automatically annotate the generated object code by CRA annotations that will guide the WCET analysis tool to reconstruct the CFG of the program.

Currently, there is an initiative under way by the cluster *Compilers and Timing Analysis* of the ARTIST2 Network of Excellence of the IST FP6. The aim of this group is to define a common format for the specification of object code and code annotations. As a natural consequence, compilers could be extended to directly generate such a code specification file.

Maintain Consistency for Code Annotations: The natural interface for PSA annotations is the source code representation level, because this would allow the developer to do the implementation of the program logic and the code annotation at the same representation level.

A framework that allows to maintain consistency of control-flow annotations in case of code optimizations performed by the compiler is described in [10]. This framework maintains the consistency of the annotations for arbitrary code transformations. Such a framework can be complemented by static program analysis as a preprocessing step to calculate control-flow annotations from the code semantics and the provided annotations about code invariants. An example for such a static program analysis based on abstract interpretation has been described by Gustafsson and Ermedahl [4].

Emit Properties of Execution Behavior: PSA annotations provide hints about the execution behavior of a program. This information can be used for the *exec-time modeling* and *path analysis* phase of a WCET analysis tool (see Section 2.3).

The compiler may reduce the amount of required PSA code annotations by automatically calculating and emitting some of these code properties. For example, the compiler may know the memory area potentially referenced by a specific pointer operation. Research on compiler extensions to emit code annotations about control flow and memory access addresses is described in [11, 5].

Improve Predictability of Code: A compiler may indirectly support WCET analysis by features not directly related to code annotations. For example, by using the *single path conversion* the execution-time jitter of real-time programs may be reduced while at the same time the WCET analyzability of the program will be improved [12]. This conversion may be also applied to local program seg-

ments instead of the whole program, giving an effect similar to *wcet-oriented programming* [13].

4.1 Using Optimizing Compilers to Produce Safety-Critical Code

There is often the argument that code optimizations have to be prohibited for the production of safety-critical code. This argument is strengthened by the fact that it is very hard to prove formal correctness of a compiler. However, recent research is focusing on analysis techniques to verify the semantic equivalence between the original and the optimized version of a program [14]. Maybe, such verification techniques in the future can be used to weaken the prohibition of code optimizations on safety-critical software, and at the same time providing a stronger argument for the support of WCET analysis by optimizing compilers.

5 Summary and Conclusion

This paper provides an analysis of how compiler-support would improve the use of WCET analysis tools.

First, a categorization of code annotations for WCET analysis has been done, resulting into four categories: platform property annotations (PPA), CFG reconstruction annotations (CRA), program semantics annotations (PSA), and auxiliary annotations (AA).

Second, it has been discussed what impact compiler-support could have on these annotations. PPA annotations are program-independent, therefore no compiler-support is needed to support WCET analysis. However, the compiler may use PPA annotations for platform-dependent code optimizations. The CRA annotations address the object code level. A compiler may be extended to output additional program properties in order to reduce the need for manual CRA annotations. The most important impact a compiler supporting WCET analysis provides, is for PSA annotations. It will free the user from the burden of manually analyzing and annotating the behavior of the object code (provided that the source code of the program is available).

Acknowledgments

The authors would like to thank C. Ferdinand, R. Heckmann, and H. Theiling from AbsInt and N. Holsti from Tidorum for fruitful discussions about code annotations for WCET analysis.

References

[1] J. Engblom and A. Ermedahl. Modeling complex flows for worst-case execution time analysis. In *Proc. 21st IEEE Real-Time Systems Symposium (RTSS)*, Orlando, Florida, USA, Dec. 2000.

[2] M. Exler. Propagierung von Pfadinformation für die Analyse von Programmlaufzeiten. Master's thesis, Technische Universität Wien, Vienna, Dec. 1999.

[3] C. Ferdinand, R. Heckmann, and H. Theiling. Convenient user annotations for a WCET tool. In *Proc. 3rd Euromicro International Workshop on WCET Analysis*, pages 17–20, Porto, Portugal, July 2003.

[4] J. Gustafsson and A. Ermedahl. Automatic derivation of path and loop annotations in object-oriented real-time programs. *Parallel and Distributed Computing Practices*, 1(2), June 1998.

[5] C. A. Healy, R. D. Arnold, F. Mueller, D. Whalley, and M. G. Harmon. Bounding Pipeline and Instruction Cache Performance. *IEEE Transactions on Computers*, 48(1), Jan. 1999.

[6] R. Heckmann and C. Ferdinand. Combining automatic analysis and user annotations for successful worst-case execution time prediction. In *Embedded World 2005 Conference*, Nürnberg, Germany, Feb. 2005.

[7] N. Holsti. *Bound-T Application Note ERC32.* Space Systems Finland Ltd, Espoo, Finland, 1 edition, Jan. 2002.

[8] N. Holsti. *Bound-T User Manual.* Space Systems Finland Ltd, Espoo, Finland, 2 edition, Mar. 2003.

[9] R. Kirner. The programming language WCETC. Technical report, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2002.

[10] R. Kirner. *Extending Optimising Compilation to Support Worst-Case Execution Time Analysis.* PhD thesis, Technische Universität Wien, Vienna, Austria, May 2003.

[11] S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, S.-M. Moon, and C.-S. Kim. An accurate worst case timing analysis for RISC processors. *Software Engineering*, 21(7):593–604, 1995.

[12] P. Puschner. Transforming execution-time boundable code into temporally predictable code. In B. Kleinjohann, K. K. Kim, L. Kleinjohann, and A. Rettberg, editors, *Design and Analysis of Distributed Embedded Systems*, pages 163–172. Kluwer Academic Publishers, 2002. IFIP 17th World Computer Congress - TC10 Stream on Distributed and Parallel Embedded Systems (DIPES 2002).

[13] P. Puschner. Algorithms for Dependable Hard Real-Time Systems. In *Proc. 8th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, Jan. 2003.

[14] R. van Engelen, D. Whalley, and X. Yuan. Automatic validation of code-improving transformations on low-level program representations. *Science of Computer Programming*, 52:257–280, Aug. 2004.

Exploiting Branch Constraints without Exhaustive Path Enumeration

Ting Chen Tulika Mitra Abhik Roychoudhury Vivy Suhendra
School of Computing, National University of Singapore
{chent,tulika,abhik,vivy}@comp.nus.edu.sg

Abstract

Statically estimating the worst case execution time (WCET) of a program is important for real-time software. This is difficult even in the programming language level due to the inherent difficulty in detecting and exploiting infeasible paths in a program's control flow graph. In this paper, we propose an efficient method to exploit infeasible path information for WCET estimation of a loop without resorting to exhaustive path enumeration. The efficiency of our approach is demonstrated with a real-life control-intensive program.

1. Introduction

Static analysis of a program for obtaining the Worst Case Execution Time (WCET) is important for hard real-time embedded systems. WCET analysis typically consists of three phases: *flow analysis* to identify loop bounds and infeasible flows through the program; *architectural modeling* to determine the effect of pipeline, cache, branch prediction etc. on the execution time; and finally *estimation* to find an upper bound on the WCET of the program given the results of the flow analysis and the architectural modeling. In this paper, we concentrate on the estimation problem.

There exist mainly three different approaches for WCET estimation: *tree-based*, *path-based*, and *implicit path enumeration*. The tree-based approach estimates the WCET of a program through a bottom-up traversal of its syntax tree and applying different timing rules at the nodes (called "timing schema") [5]. This method is quite simple and efficient. But it has limitations in exploiting the results returned by flow analysis. In particular, it is difficult to exploit infeasible paths due to branch constraints (dependencies among branch statements) in this approach as the timing rules are local to a program statement. Implicit path enumeration techniques (IPET) [4] represent the program flows as linear equations or constraints and attempt to maximize the execution time of the entire program under these constraints. This is done via an Integer Linear Programming (ILP) solver. Attempts have been made to integrate special flow information in IPET [2]. However, the kind of flow information that can be handled by IPET is inherently limited. This is because the usual ILP formulation introduces formal variables for the execution counts of the nodes and edges in the Control Flow Graph (CFG) of the program. Since the variables denote aggregate execution counts of basic blocks, it is not possible to express certain infeasible path patterns (typically denoting a *sequence* of basic blocks) as constraints on these variables.

Path-based techniques estimate the WCET by computing execution time for the feasible paths in the program and then searching for the one with the longest execution time. Thus, path-based techniques can naturally handle the various flow information. Healy et al., in particular, detect and exploit branch constraints within the framework of path-based technique [3]. Originally, path-based techniques were limited to a single loop iteration. However, Stappert et al. [6] have extended it to complex programs with the help of *scope graphs* and *virtual scopes*.

One of the main drawbacks of path-based techniques is that they require the generation of all the paths. In the worst case, this can lead to 2^n paths where n is the number of decisions in the program fragment. In control-intensive programs, we have encountered up to 6.55×10^{16} paths in a single loop iteration (see Table 1). Research by Stappert et al. [6] has sought to avoid this expensive path enumeration by finding (a) the longest program path π, (b) checking for the feasibility of π, and (c) removing π from CFG followed by the search for a new longest path if π is infeasible. This technique is a substantial improvement over exhaustive path enumeration. However, if the feasible paths in the program have relatively low execution times, then this approach still has to examine many program paths. Indeed, for our benchmark only a small fraction (less than 0.1%) of the paths are feasible, making this approach quite costly (see Table 1).

In this paper, we present a technique for finding the WCET of a program in the presence of infeasible paths without performing exhaustive path enumeration.

2. WCET Estimation Algorithm

In this section, we present a method for finding the WCET path of a single loop. Once this is obtained, the WCET path of the program can be obtained by composing the WCET paths of individual loops through the well-known timing schema approach [5]. Of course, this will mean that infeasible path information across loops cannot be taken into account. We assume that state-of-the-art WCET analyzers such as aiT[1] can be used to estimate the worst-case execution time of each basic block.

Given a fragment of assembly code corresponding to a loop in a source program, we first construct the directed acyclic graph (DAG) capturing the control flow in the loop body (i.e., the CFG of the loop body without the loop back-edge). We assume that the DAG has a unique source node and a unique sink node. If there is no unique sink node, then we add a dummy sink node. Each path from the source to the sink in the DAG is an **acyclic path** — a possible path in a loop iteration. Our algorithm finds the worst case execution path for a single iteration of the loop, i.e., we find the heaviest acyclic path. If the estimated execution time of the heaviest acyclic path is t and the loop-bound is lb, then the loop's estimated WCET is $lb \quad t$.

We find the heaviest acyclic path accurately by taking into account infeasible path information and yet we avoid enumerating all the acyclic paths in a loop. Clearly, the infeasible path information that we work with may not always be complete; so the accuracy of our heaviest acyclic path detection depends on the accuracy of the infeasible path information. First, we discuss the infeasible path information used and then explain how it is efficiently exploited in our WCET calculation.

2.1. Infeasible path information

Our infeasible path information consists of two binary relations capturing conflicting pairs of branches/assignments: *AB_conflict* and *BB_conflict*. The relation *AB_Conflict* is a set of (assignment, branch-edge) pairs, that is, if $a, e \quad AB\_Conflict$ then a is an assignment instruction and e is an outgoing edge from a conditional branch instruction; on the other hand, the relation $BB\_Conflict$ is a set of (branch-edge, branch-edge) pairs.

We do not detect conflicts between arbitrary branches and assignments to avoid an inefficient conflict detection procedure. The only conditional branches whose edges appear in our *BB_conflict* and *AB_conflict* relations are of the form $variable\ relational\_operator\ constant$. Similarly, the only assignments which appear in *AB_Conflict* are of the form $variable := constant$. For such assignments and branches we can define pair-wise conflict in a natural way (see [3] for a full discussion). For example, $x := 2$ conflicts with $x > 3$, but not with $x < 3$; similarly $x > 3$ conflicts with $x < 2$ but not with $x > 5$. Now, for such restricted branches and assignments, we put an assignment a conflicting with a conditional branch-edge e into the $AB\_Conflict$ relation (i.e., $a, e \quad AB\_Conflict$) iff there exists at least one path π from a to e which does not contain assignments to the common variable appearing in a, e. Similarly, we put two conflicting conditional branch-edges $e\ , e$ into the $BB\_Conflict$ relation (i.e., $e\ , e \quad BB\_Conflict$) iff there exists at least one path π from e to e which does not contain assignments to the common variable appearing in $e\ , e$. If $e\ , e \quad BB\_Conflict$, there may be another path π from e to e that contains an assignment to the common variable appearing in $e\ , e$. Such paths should not be considered as infeasible paths.

The computation of the $AB\_Conflict$ and $BB\_Conflict$ relations can be accomplished in $O((|V| + |E|) \quad |E|)$ time where $|V|, |E|$ are the number of nodes and the number of edges in the control flow DAG; this is because for each branch-edge we need to perform a depth-first like search to find conflicting branch-edges and/or assignments.

Example: A loop-free program fragment (which can be the body of a loop) and its control flow DAG are shown in Figure 1. In this example, the relation $AB\_Conflict$ contains only one pair – the assignment at basic block $B6$ (which sets x to 1) and the branch-edge $B7 \rightarrow B9$ (which stands for $x \geq 2$). The relation $BB\_Conflict$ contains the branch-edge pair $B1 \rightarrow B2, B7 \rightarrow B8$ that captures the conditions $x > 3$ and $x < 2$.

2.2. WCET calculation

We now present our WCET estimation algorithm for finding the heaviest feasible path in an iteration of a loop. We *do not* enumerate the possible paths in an iteration and then find the heaviest. At the same time, we do not consider all paths in the loop's control flow graph to be feasible – we consider the infeasible path information captured by $AB\_Conflict$ and $BB\_Conflict$ relations.

Our algorithm traverses the loop's control flow DAG from sink to source. However, to take into account the infeasible path information, we cannot afford to remember only the "heaviest path so far" as we traverse the DAG. This is because the heaviest path may have conflicts with earlier branch-edges or assignment instructions resulting in costly backtracking. Instead, at a basic block v, we maintain a set of paths $paths(v)$ where each $p \quad paths(v)$ is a path from v to the sink node. $paths(v)$ contains only those paths which when extended up to the source node can potentially become the WCET path. For each path $p \quad paths(v)$ we also maintain a "*conflict list*". The conflict list contains

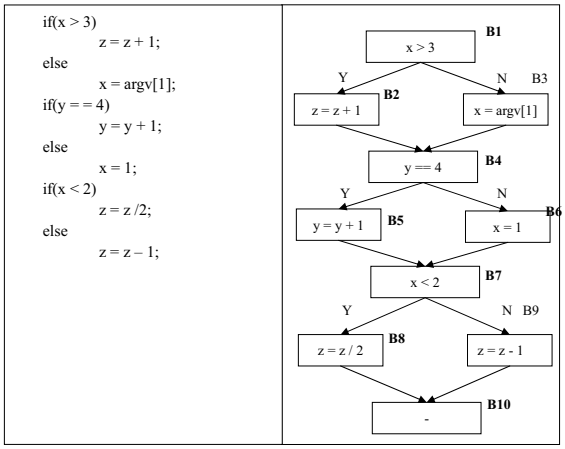


Figure 1: *A loop body with its Control Flow Graph*

the branch-edges of p that participate in conflict with ancestor nodes and edges of v.

During our traversal from sink to source, consider a single step traversal from v to u along the edge $u \rightarrow v$ in the control flow DAG. We first construct $paths(u)$ from $paths(v)$ by adding the edge $u \rightarrow v$ at the beginning of the paths in $paths(v)$. Also, for each path $p \quad paths(u)$ we update its conflict list to contain exactly those edges in p which have conflicts with branch-edges/assignments "prior to" u (in topological order). At this stage we may add the branch-edge $u \rightarrow v$ to p's conflict list or we may remove an edge e from p's conflict list if all branch-edges/assignments conflicting with e appear "after" u (in the topological order). If as a result, we have identical conflict lists for two paths $p, p \quad paths(u)$, then we maintain the heavier path among p and p . Finally, if the conflict list of a path $p \quad paths(u)$ becomes empty and p is the heaviest path in $paths(u)$, we assign the singleton set $\{p\}$ to $paths(u)$. Details of the algorithm are omitted for space considerations.

In the worst case, the complexity of our algorithm is exponential in $|V|$, the number of nodes in the loop's control flow DAG. This is because the number of paths in $paths(v)$ for some block v may be $O(2^{|V|})$ due to different decisions in the branches following v. In practice, this exponential blow-up is not encountered because (a) branch-edges which do not conflict with any assignment/branch-edge do not need to be kept track of, and (b) a branch-edge which conflicts with other branch-edges/assignments need not be remembered after we encounter those conflicting branch-edges/assignments during the traversal.

Illustration We demonstrate our WCET calculation method by employing it on the control flow DAG of Figure 1. As mentioned, we traverse the DAG from sink to source and maintain a set of paths $paths(v)$ at each visited node v. For each path $p \quad paths(v)$ we also maintain the *conflict list*, a set of branch decisions drawn from branch decisions made so far. Thus each path p in $paths(v)$ is written in the form $p_{conflict\ list}$.

Starting from node $B10$ in Figure 1, our traversal is routine till we reach node $B7$ (φ denotes empty set).

$$\begin{aligned} paths(B10) = &\quad \{\ B10\ _{\varphi}\} \\ paths(B9) = &\quad \{\ B9, B10\ _{\varphi}\} \\ paths(B8) = &\quad \{\ B8, B10\ _{\varphi}\} \end{aligned}$$

The outgoing edges from node $B7$ appear in conflict relations capturing infeasible path information. Consequently, our method maintains two paths in $paths(B7)$ — the heaviest path starting with $B7 \rightarrow B8$ and the heaviest path starting with $B7 \rightarrow B9$.

$$\begin{aligned} paths(B7) = \quad \{ \quad & B7, B8, B10\ _{\{B7 \rightarrow B8\}} \\ & B7, B9, B10\ _{\{B7 \rightarrow B9\}}\ \} \end{aligned}$$

Now, from node $B7$ we traverse to nodes $B5$ and $B6$. The assignment in node $B6$ conflicts with $B7 \rightarrow B9$. Therefore, we do not consider any path in $paths(B7)$ which contains $B7 \rightarrow B9$ in its conflict list. This is how infeasible path information is accounted for in our WCET calculation. Thus we have

$$paths(B6) = \{\ B6, B7, B8, B10\ _{\varphi}\}$$

We drop $B7 \rightarrow B8$ from the conflict list of $B6, B7, B8, B10$ as we have encountered an assignment to program variable x in $B6$. The assignment implies that the conflict between $B7 \rightarrow B8$ and $B1 \rightarrow B2$ does not hold along any extension of the partial path $B6, B7, B8, B10$.

At node $B5$, we first add $B5$ to the two partial paths from $B7$. Then, we notice that the edge $B7 \rightarrow B9$ is involved only in a conflict with $B6$ and we have already traversed $B6$. Therefore, we can drop this edge from the conflict list of the partial path $B5, B7, B9, B10$ and this path now becomes completely conflict free. Assuming that

Function	Basic Blocks	Total Paths	Feasible paths	BB-Conflicts	AB-Conflicts	Enumerated Paths
statemate	334	6.55x10^{16}	1.09x10^{13}	74	15	121,831
statemate1	44	19,440	7,440	6	0	15
statemate2	73	902	36	26	0	14
statemate3	161	1,459,364	69,867	15	0	40
statemate4	17	10	10	0	0	1
statemate5	43	256	58	2	0	4

Table 1: *Efficiency of our WCET calculation method.*

Function	WCET Estimation (cycles)	
	w/o infeasibility	with infeasibility
statemate	44,800	41,520
statemate1	29,400	28,960
statemate2	2,750	2,270
statemate3	7,300	7,000
statemate4	1,070	1,070
statemate5	2,370	2,090

Table 2: *Accuracy of WCET estimation with and without considering infeasibility.*

$B5, B7, B9, B10$ is heavier than $B5, B7, B8, B10$, we have

$$paths(B5) = \{\ B5, B7, B9, B10\ _{\varphi}\}$$

On the other hand, if $B5, B7, B8, B10$ is heavier, we still need to maintain both the partial paths (as the heavier path may become infeasible later). Continuing in this way we reach node $B1$; we omit the details for the rest of the traversal. Note that the control flow DAG of Figure 1 has three branches and $2^3 = 8$ paths. However, when we visit any basic block v of the control flow DAG, $paths(v)$ contains at most two paths (*i.e.*, the exponential blow-up is avoided here in practice).

3. Experiments

In this section, we present preliminary experiment results for the proposed method. The benchmark used in our experiment is a car window lift control program taken from the C-Lab benchmark suite. It is automatically generated by the Statemate tool based on a state-chart specification. We report results for the entire program (`statemate`) as well as program fragments corresponding to all the five functions (`statemate1` to `statemate5`).

An enumeration-based WCET estimation method typically examines each possible path, filters out the infeasible paths and selects the feasible path with the maximum execution time. Table 1 shows that the total number of paths through a single iteration of the loop body can be quite large (6.55×10^{16} possible paths out of which at most 1.09×10^{13} are feasible for `statemate`). In fact, a naive WCET calculation method, which enumerates all these possible paths for one loop iteration and chooses the longest feasible one, runs out of memory even on a PC with 1 GB main memory. The column *Enumerated Paths* shows the maximum number of paths that need to be maintained by our estimation technique at any point of time. The results are quite encouraging; even for the entire `statemate` program, we only need to keep at most $121,831$ paths at any point of time during the estimation. As a result, our estimating technique requires less than 1 minute for the entire `statemate` program on a Pentium4 1.7Ghz platform with 1GB memory. Finally, Table 2 shows that, as expected, our method produces more accurate estimation compared to a method that does not take infeasibility information into account. The only exception is `statemate4`, which does not have any infeasible path.

4. Discussion

In this paper, we have reported preliminary results on exploiting (limited) infeasible path information during WCET estimation of a loop without resorting to path enumeration. The efficacy of our technique has been demonstrated on a substantial real-life car window control benchmark. In near future, we will develop WCET estimation methods that can take into account infeasible path patterns of arbitrary length without compromising efficiency.

References

[1] AbsInt. aiT: Worst case execution time analyzer, 2004. `http://www.absint.com/ait/`.

[2] A. Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, 2003.

[3] C. Healy and D. Whalley. Automatic detection and exploitation of branch constraints for timing analysis. *IEEE Transactions on Software Engineering*, 28(8), 2002.

[4] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *DAC*, 1995.

[5] C. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-time Systems*, 5(1), 1993.

[6] F. Stappert, A. Ermedhal, and J. Engblom. Efficient longest execution path search for programs with compelx flows and pipeline effects. In *CASES*, 2001.

Composable Real-Time Analysis (Abstract)

Lothar Thiele, ETH Zürich

Embedded computer systems are getting increasingly distributed. This can not only be seen on a small scale, e.g. in terms of multiprocessors on a chip, but also in terms of embedded systems that are connected via various communication networks. Whereas classical methods from the worst case timing analysis and real-time community focus on single resources, new models and methods need to be developed that enable the design and analysis of systems that guarantee end-to-end properties.

The talk covers a new class of methods based on real-time calculus. They can be considered as a deterministic variant of queuing theory and allow for

(a) bursty input events and event streams,

(b) heterogeneous composition of scheduling methods (EDF, FP, TDMA, WFQ, ...),

(c) distributed computation and communication resources

(d) detailed modelling of event stream correlations and resource behaviour and

(e) hard worst case bounds.

Besides introducing the basic models and methods, some application studies are covered also.

It appears that this class of new methods provide a major step towards the analysis and design of predictable distributed systems.