Experimentation of WCET computation on both ends of automotive processor range

H. Cassé P. Sainrat C. Ballabriga M. de Michiel
Université de Toulouse, Institut de Recherche en Informatique de Toulouse (IRIT)
118 route de Narbonne, 31062 Toulouse Cedex 9
+33 561 55 83 32 + 33 561 55 84 25 +33 561 55 83 32 +33 561 55 83 32
casse@irit.fr sainrat@irit.fr ballabri@irit.fr michiel@irit.fr

ABSTRACT
This article presents the results of experimenting our OTAWA tool to compute WCETs on a real automotive embedded application. First, we analyze the application (C source generated from Simulink models) and exhibit specific properties and their implication on the WCET computation. Then, two very different embedded processor architectures are tested and in both cases we show (1) how their specific features are supported by OTAWA and (2) how to configure them to maximize both performances and determinism[1].

Categories and Subject Descriptors
D.2.4 [**Software**]: Software Engineering – *software/program verification*, *Reliability*.

General Terms
Algorithms, Performance, Verification

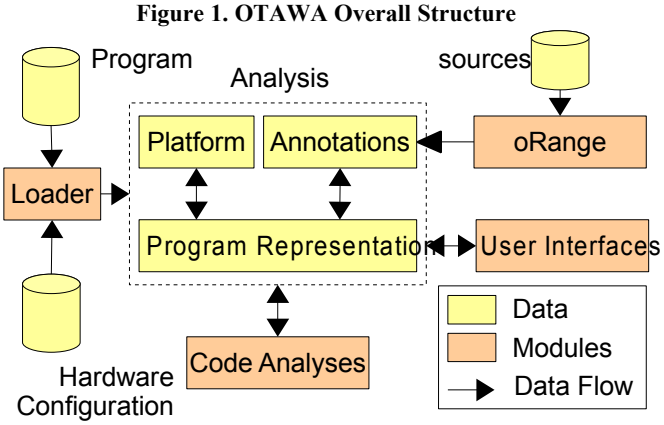
Keywords
Real-time, static analysis, WCET, automotive.

1. INTRODUCTION
Real-time properties verification is essential for embedded automotive applications. A failure may have critical effects on passenger health while software errors fixes is usually hard to perform and may cause economic threat to the company.
A goal of the MASCOTTE [1] project was to prove that tools which check real-time properties was mature enough to be involved in a real automotive application development. A partner of the project proposed a case study application that should be executed on two very different architectures. The first one, the Freescale Star12X is quite new but uses an old-fashioned 16-bit architecture. Its low price makes it a good candidate to be embedded in cars but its computation power might be a bottleneck in the future. On the opposite, the MPC5554, also from Freescale, is a full 32-bit PowerPC ISA (Instruction Set Architecture) exhibiting full power from RISC architecture.
The challenge for our tool, OTAWA [2], was to show its ability to compute the Worst Case Execution Time (WCET) for the provided application and for both architectures. WCET is used to time the execution of parts of a program in order to verify that, at any time of the application, all tasks can be performed without exceeding their dead-line. WCET may be hard to compute according to the complexity of the processed application and to the processor execution model that may implement non-deterministic features.
This article shows the results of experimenting OTAWA on the given automotive case study for both architectures. It presents OTAWA in the next section while section 3 gives information about the proposed case study. Section 4 presents our experience for both architectures.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CARS '2010, April 27, Valencia, Spain
Copyright 2010 ACM 978-1-60558-915-2/10/04... $10.00

[1] Throughout this article, the determinism term qualifies features whose variability is either small, or predictable.

Figure 1. OTAWA Overall Structure



2. OTAWA
OTAWA has been developed in our team since 2004 and has now reached a good level of reliability. It has been involved in several projects such as MASCOTTE.

2.1 Tool Overview
Figure 1 shows the overall structure of OTAWA. As a distinctive feature, it is not devoted to a specific WCET computation method although it emphasizes an important implementation of the Implicit Path Enumeration Technique (IPET) [3] approach. In fact, OTAWA is a generic framework providing facilities (1) to load and to represent a program in machine language, (2) to provide different program representations and (3) to support and combine several static analyses useful for WCET computation.

This flexibility to support and to adapt its computation to any architecture makes it a good candidate for the the processors proposed in MASCOTTE. Therefore, both architectures have been processed with variants of IPET approach, that is currently the more promising method to compute WCET. Most performed analyses have been shared for both processors and few specific analyses have been introduced.

Flexibility and re-usability in OTAWA comes from its internal structure. First, an ISA-dependent loader scans the machine code

program in order to provide an abstract program representation: all following analyses are independent from the ISA. It does not mean that it can not take into account the specific processor features but analyses may be re-used whatever the execution model of the hardware. In fact, the hardware details are made visible in a portable way to all analyses by the platform description. In the next sections, we give more details on the pro-eminent components of OTAWA.

2.2 Plug-ins and Loaders

OTAWA is heavily based on a plug-in system allowing to import easily new analyses or to customize the Integer Linear Programming (ILP) solver used in IPET, the output of statistics, program representations or supported ISA.

The root is the loader responsible of handling the different ISAs. It may be built on any technology but the ones provided with OTAWA are based on GLISS [4], a functional simulator generator that represents the ISA using the SimNML language. The generated code includes a simulator, a binary decoder and a disassembler. OTAWA uses mainly the two latest components to perform instruction decoding and to output a human-readable program representation.

To obtain a full loader, some code is needed to bind together OTAWA library and GLISS generated code. Some information required by OTAWA about instruction types, branch target and so on is added by hand but version 2 of GLISS will automatically generate this information from the SimNML description.

Figure 2. A usual WCET Scenario

Phase 1: Path Analysis
1. instruction decoding
2. loop bounding (oRange)
3. float fact loading
4. CFG building
5. loop detection

Phase 2: Timing Analysis
1. instruction cache analysis
2. flash memory analysis
3. block timing

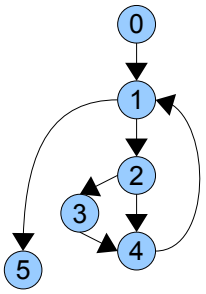
Phase 3: WCET computation
1. adding constraints for CFG
2. adding constraints for cache
3. adding constraint for flash memory
4. system resolution

2.3 Configuration of Computation

From the low-level program representation, OTAWA can derive several program representations like Control Flow Graphs (CFG) or Context Trees according to the needs of the analyses. Yet, all these representations share a common facility: they support annotations. These annotations are the glue to bind together the different analyses. Each one, called a code processor in the OTAWA terminology, can be viewed as a function taking the current program representation and the hooked annotations and producing as output the same program representation improved with new annotations. Therefore, the WCET computation may be viewed as the process of performing a chain of analyses until the so-named WCET annotation becomes available. A typical example of such a chain is given in Figure 2.

Although the scenario in Figure 2 seems easy and linear, it involves many more analyses with complex links. They are based on a producer-consumer scheme: annotations from one analysis are required by another one. To handle automatically these dependencies, OTAWA structures the annotations in a "feature" and expresses relationships between analyses as a set of required and provided features and, according to these requirements, produces a schedule to perform the complete computation. This system helps to design the analysis of a processor and contributes also to the flexibility and the adaptability of the tool: any analysis producing some feature may be replaced by any other one if the new one provides the same features.

Figure 3. Example of IPET



$$W = t_0 x_0 + t_1 x_1 + t_2 x_2 + t_3 x_3 + t_4 x_4 + t_5 x_5$$
$$x_0 = 1$$
$$x_1 = e_{0,1} + e_{4,1} = e_{1,5} + e_{1,2}$$
$$x_2 = e_{1,2} = e_{2,3} + e_{2,4}$$
$$x_3 = e_{2,3} = e_{3,4}$$
$$x_4 = e_{3,4} + e_{2,4} = e_{4,1}$$
$$x_5 = e_{1,5}$$
$$e_{4,1} \leq N \times e_{0,1}$$

2.4 IPET

The IPET approach consists in representing the execution time as an Integer Linear Programming system which maximizes a function, W, representing the WCET under some constraints. Figure 3 shows a small example of WCET. The program is represented as a CFG whose nodes represent instructions blocks and edges control flow transfer, like branches, between blocks. Basically, a variable x_i is associated with a block to count the number of times it is executed on the WCET execution path. Constraints are added to take into account the graph flow: a block is executed as many times as the sum of execution times of its predecessors or successors transitions, $e_{i,j}$ or $e_{j,i}$. The WCET function is the sum of each block variable multiplied by the execution time of a block. More constraints and variables may be added to this minimal system in order to take into account the different features of a processor.

For example, one classic way to handle caches is the category approach [9]. It consists in categorizing each instruction block according to its behavior in the cache. Classic categories includes always-hit, always-miss, persistent (stays in the cache once loaded) and not-classified (none of the other categories). The goal is to evaluate two variables: the number of misses and hits. Then, we add two kinds of constraints: the first one states that the sum of misses and hits is equal to the number of times a block is executed; the second depends on the category (for example, always-miss states that the number of hits is ever equal to zero). The category approach may be extended to other features containing a state.

2.5 oRange

The ILP described in the previous paragraph may contain circularities induced by the loops in the graph. If we do not bound them, the ILP resolution will converge to infinity. To avoid this, the last component involved in OTAWA, oRange [5], is a tool to compute automatically flow facts especially loop bounds. It takes as input a set of C source files, the entry point of the task to compute bounds for and produces, as output, a file with loop bounds according to the call context. As all other tools, oRange does not find all loop bounds (but most of them), so the user has to provide the lacking bounds if any. An example of loop bound constraint is given in the last constraint of Figure 3.

3. MASCOTTE CASE STUDY

Because of non-disclosure agreement in the project, we cannot give all details on the case study but we can outline its structure. Actually, it is enough to understand its specific timing properties.

3.1 Application structure

The case study represents a real embedded automotive application implementing a control law based on sensors and actuators. It has been modeled using Simulink / StateFlow and the sources are produced by an automatic C generator .

From a functional point of view, the case study is composed of 8 tasks i.e. functions that are launched periodically at their own rate according to the functional needs of the device. The tasks are managed by a static scheduler and the tasks execute without interruption. Tasks are called T_0 to T_7 throughout the document.

Table 1. Size of tasks

Task	Size (line)	Task	Size (line)
T_0	40000	T_4	3500
T_1	18000	T_5	2600
T_2	8000	T_6	1100
T_3	3700	T_7	260

These tasks are very different. First, in term of size, Table 1 shows the number of source lines of each task. The tasks may be sorted into three groups: big tasks (T_0, T_1 and T_2) , medium-size tasks (T_3, T_4 and T_5) and very small tasks (T6, T7). One may observe a factor 154 between the biggest task and the smallest task. Yet, although we have to process task with a huge size, the generated sources are not as complex as hand-coded sources and should remain tractable.

Figure 4. Interpolation loop

```
while (x > *((x_table)++))       {
       (y_table)++;
}
```

3.2 Source characteristics

The sources are made of 15 files for a total of 80,000 lines of C. There are only 55 loops and the body of loops is very small: 1 line of body containing one assignment and 2 incrementations as shown in Figure 4. This type of loop is used to compute by interpolation some complex functions of the Simulink model. x_table and y_table are two parallel tables implementing a function y=f(x). The closest greatest value for x is found in x_table containing ordered x values in the functional input interval. At the same position in y_table, we get a value for y that may be adjusted according to the distance of x in the interval formed by current and previous entries in x_table.

100% of loops of the application follow this scheme: the flow profile of the application is very atypical compared to programs on usual workstations or even on embedded application coded by hand. Consequently, most of the control flow is made of conditional structures, about 3200. It remains relatively small compared to the total number of lines in the sources, therefore, most of the code performs computations. If we take a closer look, we can see that most of these computations are simple operations like addition, subtraction, bit shifting and so on, exclusively on integer values. There are very few complex operations and most of them are handled by interpolation functions.

In fact, as the Simulink model comes from a real embedded application, it remains traces from its adaptation to low-power processors. A lot of operations that should have required complex multiplications, divisions and even more complex functions, have been simplified to fit the limitation of the hardware: integer only, minimal memory footprint (most variables have a size of 16 bits), speed (all divisions have been converted by hand to shifts). For example, the first version of the case study was including a function that was performing a division of a 24-bit value by a 16-bit value with a very complex low-level algorithm. The timing of this function was so non-deterministic that it has been replaced by a simple 32-bit division for the purpose of the study.

3.3 Flow facts

Loop bounds of the case study application were relatively easy to obtain (100% solved automatically by oRange). Most loops have been bounded by the size of the traversed interpolation table. In some cases, this bound is an overestimation of the actual bound as the interpolation function may be called in different contexts with different parameter ranges. oRange does not embed a value/interval analysis to detect this but the way the value interval are expressed in Simulink and the generation method with function specialization exclude the possibility to have part of the interpolation tables unused.

Another problem, that would have been solved by value analysis (definition of values handled by the program), concerns the interpolation algorithm form. The first table cell was processed in a conditional causing oRange to overestimate the bound by 1 iteration. However, oRange has provided safe and relatively tight bounds for all program loops.

4. RESULTS

We expose in this section the achievement of OTAWA applied to the case study for both architectures.

4.1 Star12X

The Star12X is the last 16-bit processor of FreeScale from the M68HC family. Implementing a very old CISC ISA, this is a typical old generation processor embedded in automotive: cheap and robust.

The challenge for GLISS was to support a CISC instruction set with variable size instructions, from 1 byte to 18 bytes, with very complex immediate operand encoding. The size of the instruction set was also an issue that has stressed the performances of the tool. Despite this, the adaptation has been successful.

The timing of the instructions was relatively easy to obtain as the Star12X handbook specifies precise timings for each instruction. An issue was the accessed address alignment (inducing 1 more cycle for a word access on an odd address): easily computed for scalar variables, we assumed worst cases for arrays. As array accesses represent only 18% of memory accesses, this caused very few overestimation on WCET.

As the Star12X has a word of 16 bits, emulation functions are used for bigger data types. These functions are usually programmed in assembly for efficiency and therefore not supported by oRange. Their computation, by hand, has shown a big variability in the iteration number according to their arguments. This is especially true for the 32-bit division where the computation time may have a factor of 20 between divisors on more or less than 24 bits. The outcome is an important overestimation of the WCET while only divisions with 16-bit divisors occur.

These two issues would have been resolved by a value analysis giving either intervals for values or alignment for addresses. Therefore, we get 100% precise times at block level but an overall WCET overestimated by 75%. Despite this, we can state that the adaptation of OTAWA to the Star12X was relatively easy and the computation time, ranging from 1s to 58s, very tractable.

4.2 MPC5554

The MPC5554 is a 132 MHz RISC 32-bit processor providing a PowerPC-compatible ISA (without floating-point extension). It is dedicated to automotive applications. This processor exhibits more interesting non-deterministic features (pipeline, flash prefetch and so on) for OTAWA and provides a bunch of flexibility to tune it. The problem is to maximize processor performance and to reduce WCET overestimation.

The 7-stage in-order pipeline is processed by thescribed in XML and by a customization of the graph to handle specific processor features: bypassing, variable duration division, two-stage fetch, etc. This is mainly done by adding some edges in the graph and fixing some duration in graph nodes. As the application is critical, the CrossBar has been configured to give priority to the processor. Therefore, the internal bus access time is 3 cycles.

The case study application has few global variables (about 4Kb) and a small stack footprint. Thus, they have been mapped to an embedded static RAM whose access time is constant. Because of its big size (about 108Kb), the code has been mapped in the flash memory. Dynamic RAM has not been used as it is known that the analysis of such a memory conducts to an intractable computation time cause of its refresh cycles.

As the flash memory has a relatively long latency, a prefetch front-end with different policies is in this processor. Prefetch policies are not standardized but we have been able to adapt a flash prefetch analysis implemented for the WCET Challenge'08 [8] on an ARM processor. The MPC5554 includes two 32-bit prefetch lines configured to prefetch the next line after each line access that allowed (1) to run the processor at its full speed and (2) to have a predictable behavior. When a data is prefetched (hit), accesses are 4 cycles. Otherwise (miss), 7 cycles are required. OTAWA analysis of the flash prefetch uses the category approach [9] and gives 25% of always-hit, 65% of always-miss and only 10% of not-classified accesses.

As accesses to instructions in the flash are 4 cycles, the use of an instruction cache was required. The MPC5554 provides a customizable 32 Kb unified cache with 8-way associativity. Its replacement policy is pseudo-round-robin: all sets share the same counter. It is known that the analysis of such a cache conducts to an intractable computation time, the solution was thus to configure the cache such that only one way was used for instructions reducing it to a direct-mapped cache of 4 Kb. Yet, due to the properties of the application, it does not impair the performance: loop bodies are very small and are easily contained in the new cache configuration. On the opposite, code out of loops only benefits from space locality that is supported by the cache block size of 32 bytes. With this configuration, the accesses have been categorized to 45% hits, 26% misses and 29% unclassified. To improve the timing, OTAWA provides also an analysis that support locality across conditional branches [10]. One aspect of the cache that has not been explored in this project is the cache effect across different activations of a task.

The MPC5554 provides a 2-bit BTB branch predictor we have disabled for several several reasons: (1) the branch penalty is only of 2 cycles, (2) the shape of the code (mainly composed of conditionals) does not benefit a lot from the predictor and (3) the time cost of branch prediction is usually, and in particular in OTAWA, too much time costly. From a development point of view, the MPC5554 support in OTAWA has only required the adaptation of the flash memory analysis. The computation was a bit longer than for the Star12X (about one hour for the biggest task). Measurements performed on a real board have validated our results, with an overestimation less than 10%.

5. CONCLUSION

In this article, we have described the experimentation and the results of WCET computation for a real automotive application on two processors representing both ends of the embedded processor range. The first conclusion is that it works well for both processors and this application. As most applications generated from models as Simulink, Scade, etc, provide the same properties, we are confident that OTAWA is able to analyze most critical embedded applications. Second, OTAWA lacks an important analysis to get tighter results: the value analysis in order to better support infeasible code or to detect the alignment of addresses. Finally, oRange that only works on sources must be extended to assembly to support hand-written libraries.

We plan to extend OT execution graph approach [7] from a generic processor model deAWA with value analysis and better integrate oRange. As it is almost impossible to support all possible processor features, the philosophy of OTAWA is to provide a sound framework allowing to quickly develop ad-hoc analyses. So we intend to continue in this way (1) supporting new architectures and (2) providing easy access to analysis development.

6. REFERENCES

[1] MasCotTe Project, http://www.projet-mascotte.org/

[2] H. Cassé, P. Sainrat. OTAWA, a framework for experimenting WCET computations. ERTS 2006, http://www.otawa.fr

[3] Y.-T. S. Li, S. Malik. Performance analysis of embedded software using implicit path enumeration. Workshop on languages, compilers, and tools for real-time systems , 1995

[4] T. Ratsiambahotra, H. Cassé, P. Sainrat. A Versatile Generator of Instruction Set Simulators and Disassemblers. IEEE SPECTS 2009.

[5] M. de Michiel, A. Bonenfant, H. Cassé, P. Sainrat. Static loop bound analysis of C programs based on flow analysis and abstract interpretation. IEEE RTCSA, 2008.

[6] J. Engblom, A. Ermedahl. Pipeline Timing Analysis Using a Trace-Driven Simulator. IEEE RTCSA'99.

[7] C. Rochange, P. Sainrat. A Context-Parameterized Model for Static Analysis of Execution Times. Transactions on High-Performance Embedded Architecture and Compilation, 2007.

[8] N. Holsti et al. WCET Tool Challenge 2008: report. WCET'08.

[9] R.T. White, C.A. Healy, D.B. Whalley, F. Mueller, M.G. Harmon. Timing analysis for data caches and set-associative caches , IEEE RTTAS, 1997.

[10] C. Ballabriga, H. Cassé. Improving the First-Miss Computation in Set-Associative Instruction Caches. ECRTS 2008.