

Static Worst-Case Execution Time Analysis Tool Development for Embedded Systems Software

Thomas Jerabek, Martin Horauer

University of Applied Sciences Technikum Wien

Höchstädtplatz 6, A-1200 Vienna, Austria

Email: {thomas.jerabek, martin.horauer}@technikum-wien.at

Abstract—Analyzing the worst-case execution time of embedded systems software is useful for assessing parameters like schedulability, performance (especially with regard to deadlines), etc. A commonly accepted approach to obtain these values is by way of static analysis that uses the software along with a model of the target processors architecture. This paper describes the required steps to construct a tool to assess the worst-case execution time of a given application with the help of an open-source framework. The ensuing evaluation provides a comparison of the results with other approaches. In addition, this paper can be used as guide to implement an instruction set architecture of a target processor in order to enable various static analyses with the aim to estimate the worst-case execution time.

Keywords—architecture description language; instruction set simulator; worst-case execution time analysis.

I. INTRODUCTION

Embedded systems nowadays are ubiquitous in our daily life. One kind of embedded systems are real-time systems where the correct operation of the system depends on the logical correctness of the computations and upon the time at which the result is produced. Hence, knowledge about execution times – and here in particular the worst-case execution time (WCET) – is of relevance, e.g., to assess whether deadlines imposed by application requirements will be met, or to assess the schedulability of an implementation.

The WCET defines the longest time it takes to execute a program on a specific target processor. There are different ways to determine this value.

(1) One method uses static code analysis [1], [2] by way of a model of the processor’s architecture. In fact, various analyses are in use therefore, e.g.:

- Control-flow analysis
- Value analysis
- Cache analysis
- Pipeline analysis
- Path analysis
- WCET estimation

Each of these analyses must be implemented and adapted for every new target architecture.

(2) Another WCET analysis method is measurement-based where the execution time of an application, function or task is recorded during runtime. To that end, the source code must be instrumented to provide suitable triggers for the

measurement and appropriate stimuli are required to stress worst-case behavior. In practice, the latter requires elaborated test-setups [3], [4].

(3) The third approach combines both the static analysis method to evaluate input data and the measurement-based method to estimate the WCET [3].

Independent of the chosen approach, it is essential that the real WCET is never longer than the evaluated value and the result is as close as possible to the reality. These two aspects describe a safe and tight WCET evaluation.

Dependable systems have the ability to avoid service failures, which are unacceptable in terms of frequency and severity. Many aspects need to be considered in order to ensure such a behavior; however, this paper focuses on a specific detail: predictable execution times of dependable software (e.g., a hard real-time system) via WCET analysis. Such an analysis is an inherent part of the safety process during the design and development of automotive and avionics systems to avoid timing issues [5], [6]. For example, unmanned aerial vehicle (UAV) software contains various tasks (e.g., engine control or position sensing) where the knowledge about their WCET is mandatory for safe operation.

The contribution of this paper presents a generic approach of how to enable WCET analysis for a modern processor architecture following approach (1) using static analysis of executable binaries. Besides the architecture module, two analysis tools were implemented using the OTAWA framework’s API. The result is a useful guide to implement WCET analysis for a certain processor architecture. It can be utilized for WCET-aware development to prevent systematic failures in order to increase the reliability of a dependable system.

The structure of the paper is as follows. First, we detail typical design patterns and their effect on WCET using some examples. Next, we present related work in Section III, followed by a description of the implementation in Section IV. Section V provides some benchmarks, and finally, a use case is presented in Section VI before we conclude the paper in Section VII.

II. WCET ESTIMATION OF EMBEDDED SYSTEMS SOFTWARE

Software for embedded systems typically follow either a bare-bone approach or employ some kind of (real-time) operating system. Bare-bone applications in turn either follow a super-loop architecture or a fore-/background approach [7].

When using a preemptive operating system task and/or thread models are in place. Below, we will use these patterns and show how WCET can be estimated by way of examples. This not only gives an insight on the analysis itself, but also encourages the technical background for the further chapters.

In general, static WCET evaluation is based on one of the three present calculation techniques: (1) path-based [8], (2) tree-based [9], and (3) implicit path enumeration technique (IPET) [10]. Since these approaches are usually applied on instructions or basic blocks, they need to be abstracted in order to use them for a higher program representation. The aim of this chapter is to describe the WCET of a program with a formalism using its most basic elements. This high-level formalism is inspired by the path-based approach where the WCET will be determined by first calculating times for different paths within a program and then looking for the path with the longest execution time. It consists of a header T_H and a content/path T_P that is multiplied with its loop-bound LB , as shown in (1). Depending on the programs structure, this basic equation needs to be adapted. All subsequent T variables are already defining the maximum execution time of the corresponding program part as processor clock cycles. This means, that the result is independent of the processors frequency; however, one can convert it into a time via dividing it by the processors clock rate.

$$WCET = T_H + T_P * LB \quad (1)$$

Bare-bone program structures consist of an initialization part T_{IN} and an endless loop T_L . The execution time of the latter equals a cycle time, which is especially relevant for super-loop architectures because they are not using interrupts at all and only detect events via requests (polling). It can be used as maximum response time for a certain event and can be evaluated as shown in (2).

$$WCET_P = T_P = T_L \quad (2)$$

The other kind of bare-bone applications is using an interrupt driven fore-/background architecture. Here, interrupts and the execution of their associated service routines (ISR) need to be considered for WCET evaluation. With its WCET and the execution rate, one can calculate an expected rate (periodicity) relative to the program under analysis. Depending on the referenced program section T_{PS} , the interrupt rate, as estimated by (4), contains only the loop section or the entire application (see (3)). For the latter, interrupts need to be enabled before the program part under analysis. The sum of all interrupt service routines T_{SI} is estimated by adding up the expected timing of every interrupt as shown in (5). The equation assumes that all involved interrupts are activated permanently.

$$T_{PS} = T_L \vee (T_{IN} + T_L) \quad (3)$$

$$R_I = \frac{T_{PS}}{\text{expected ISR Rate}} \quad (4)$$

$$T_{SI} = \sum_{i \in SI} [T_{I_i} * R_{I_i}] \quad (5)$$

The WCET of one loop cycle of interrupt driven applications takes the interrupts into account and can be estimated by (6). The overall WCET is the result for termination after

a certain number of cycles specified by the loop-bound (see (7)).

$$WCET_P = T_P = T_L + T_{SI} \quad (6)$$

$$WCET = T_{IN} + T_P * LB \quad (7)$$

An example is given for a fore-/background structure with 3 interrupt service routines. The WCET of one loop cycle should be estimated. Worst-case time behaviors of each individual part were already evaluated as listed below.

$$T_{IN} = 600 \text{ clock cycles}$$

$$T_P = 9000 \text{ clock cycles}$$

$$T_{I_1} = 250 \text{ clock cycles}$$

$$T_{I_2} = 890 \text{ clock cycles}$$

$$T_{I_3} = 60 \text{ clock cycles}$$

We assume three ISRs with the following shortest possible periodicity:

$$ISR1 : \text{executed every } 5000 \text{ clock cycles}$$

$$ISR2 : \text{executed every } 20000 \text{ clock cycles}$$

$$ISR3 : \text{executed every } 1400 \text{ clock cycles}$$

Thus we get:

$$R_{I_1} = \frac{9000}{5000} = 1.8 \approx 2$$

$$R_{I_2} = \frac{9000}{20000} = 0.45 \approx 1$$

$$R_{I_3} = \frac{9000}{1400} = 6.43 \approx 7$$

$$T_{SI} = 250 * 2 + 890 * 1 + 60 * 7 = 1810 \text{ clock cycles}$$

$$WCET_P = 9000 + 1810 = 10810 \text{ clock cycles}$$

The result shows that an increasing number of interrupts significantly affects the WCET.

RTOS: There are major differences between bare-bone and real-time operating system (RTOS) structures, such as the administrative overhead and the interruption of execution by a higher priority task in real-time operating systems.

When assuming a priority based scheduler, it is a challenge to evaluate a task's WCET because each task, except the highest priority one, can be interrupted by a higher priority task. As a result, evaluation needs to be done by a top-down approach starting with the highest priority one. Equation (8) describes the maximum interruption time of a task by summing up both, all Tasks with a higher priority T_{HPT_i} and their administrative overhead T_{AO} (e.g., context switch, scheduling). The subsequent calculation is identical to bare-bone programs, as shown in (9) and (10).

$$T_{INT} = \sum_{i=1}^n T_{AO_i} + \sum_{i=1}^n T_{HPT_i} \quad (8)$$

$$WCET_P = T_P = T_L + T_{INT} \quad (9)$$

$$WCET = T_{IN} + T_P * LB \quad (10)$$

For tasks using an endless-loop pattern, the worst-case cycle time $WCET_P$ is typically the most relevant. The overall WCET is the result for termination after a specified number of cycles (loop-bound). For tasks using a run-to-completion

pattern there are typically no cycles and, therefore, the loop-bound equals 1 for the overall WCET.

An example calculation follows for a run-to-completion task, which is implemented into a real-time operating system where 2 tasks with a higher priority are existing. The task's entire WCET should be determined with the assumption that the administrative overhead is task independent. Worst-case time behaviors of each individual part were already evaluated as listed below.

$$\begin{aligned} T_{IN} &= 200 \text{ clock cycles} \\ T_L &= 3900 \text{ clock cycles} \\ T_{HPT1} &= 2320 \text{ clock cycles} \\ T_{HPT2} &= 1100 \text{ clock cycles} \\ T_{AO} &= 590 \text{ clock cycles} \end{aligned}$$

This results in:

$$\begin{aligned} T_{INT} &= 590 + 590 + 2320 + 1100 = 4600 \text{ clock cycles} \\ T_P &= 3900 + 4600 = 8500 \text{ clock cycles} \\ WCET &= 200 + 8500 = 8700 \text{ clock cycles} \end{aligned}$$

The outcome of 8700 cycles shows that the RTOS as well as tasks with a higher priority have a major impact on the tasks WCET.

III. RELATED WORK

In the following, we provide a short comparison of (1) existing WCET analysis tools (cf. Table I) and further on (2) describe related architecture description languages (ADLs).

TABLE I. WCET ANALYSIS TOOL COMPARISON.
(*ECLIPSE PLUGIN, **PARTLY, ***NOT REQUIRED)

Name	aiT	Bound-T	RapiTime	SWEET	OTAWA
open source	X	✓	X	✓	✓
static analysis approach	✓	✓	X	✓	✓
measurement-based approach	X	X	✓	X	X
annotations	✓	✓	✓	✓	✓
GUI	✓	X	✓	X	✓*
specify architectures	X	X	✓	X	✓
specify μ C characteristics	✓**	✓**	X***	X	✓
binary file input	✓	✓	✓	✓	✓
ISO 26262	✓	X	✓	X	X
DO-178B	✓	X	✓	X	X

The aiT WCET Analyzer [1] from AbsInt features an easy to use GUI with a straightforward configuration. It is able to compute tight bounds of a programs WCET using static analysis. One can choose between different integer linear programming (ILP) solvers (e.g., CPLEX), analysis options and output reports (e.g., HTML and XML). User defined annotations can be provided using the AIS/AIS2 language in order to define loop bounds and other program information. It fulfills the ISO 26262 as well as the DO-178B level A qualifications.

Bound-T [11] (originally developed by Tidorum Ltd. and now released as open-source) is a command-line tool that uses static analysis for the WCET estimation. Annotations can be provided by the user, although, loop bounds can be

derived automatically. Unfortunately, cached memory or the parallelism of functional units cannot be specified making it hardly unsuitable for many modern processors.

RapiTime [12] uses a measurement-based approach for the WCET evaluation. It derives a structural model of the program and instruments the source code during the build process of a program. Afterwards, it performs all given tests and extracts the timing data via execution traces. To finish up, a prediction of the worst-case path and WCET is carried out by combining the obtained timing information and the structural model of the code. It fulfills both the DO-178B/C and ISO 26262 qualification. Since RapiTime does not rely on a processor model, it is usable for a lot of targets as long as they support a mechanism to extract execution traces.

The SWEDish Execution Time analysis tool (SWEET) [13] is a research prototype that offers best-case execution time (BCET), WCET and flow analysis. It uses a program representation called ALF (Artist Flow Analysis Language) in which either a binary or source code has to be converted for further processing. It implements a flow analysis of a given program to detect infeasible paths and loop bounds. The latter can be exported to aiT or RapiTime flow facts format enabling further analyses. A low-level analysis tool called low-sweet allows evaluating the WCET by its own.

The Open Tool for Adaptive WCET Analysis (OTAWA) [14] is a static analysis framework that allows modifying, extending, or implementing analysis tools by using the OTAWA API. One of the existing tools, called OWCET, evaluates the WCET of a given program by providing the executable binary file, program flow information (flow facts) as well as a processor description (OTAWA script). With this information, it automatically links the corresponding architecture loader module and performs the analyses.

Besides RapiTime with its measurement-based approach, all of the mentioned tools use binary based static program analysis, which requires knowledge of the processor architecture in order to perform a timing analysis for WCET estimation. In fact, implementing support for a new target system requires the implementation/adaption of elaborate analyses to this new architecture.

At the core therefore, are usually architecture description languages (ADLs) that describe the processor model. They have a wide range of application; a majority is the generation of target specific tools (e.g., compiler or simulator). They are also used for the development and rapid prototyping of application-specific instruction-set processors (ASIPs).

There are different ADLs available to create an instruction set simulator (e.g., EXPRESSION [18], LISA [19] and nML [20]), however each of them describe the instruction set architecture of a specific processor family without detailed information of the microcontroller. This makes it possible to use the simulator for every microcontroller with the chosen processor architecture. For timing analyses, there are some details like cache or pipeline behavior missing at this point; therefore, this information is provided via additional description by the developer or end user. Hardware ADLs are likewise known as processor or machine description language and are not only used for simulators but also for processor development [21, p. 2].

These architecture description languages are classified in

three content-based (structural, behavioral and mixed) as well as four objective-based categories (compilation, simulation, synthesis and validation). This classification allows developers to choose an ADL by either content (e.g., instruction-set description) or purpose (e.g., generation of an instruction-set simulator). Due to the fact that not all ADLs can describe the instruction-set behavior with a detailed timing model, only a few are usable for WCET analysis.

Modeling a processor based on the ARMv5 architecture using the ArchC ADL is described in [15]. They divided the implementation into three phases, starting with the choice of a suitable ADL, followed by constructing of a toolchain (e.g., compiler and simulator) for the architecture; and phase three, analyzing the executed instructions by the simulator using program patterns. Their aim was to evaluate often used instruction patterns that can be merged to a new complex instruction in order to increase the performance.

A retargetable software timing analyzer for WCET estimation using the EXPRESSION ADL for processor models is described in the work of [16]. For their case study, a MIPS processor including its instruction set architecture was modeled and evaluated.

An approach for generating instruction set simulators from an enhanced nML architecture description is presented in [17]. The tool generates a simulator usable for static analysis. For demonstration purposes, their tool was applied to an ARMv5 architecture implementation.

For the approach described here, we choose the Sim-nML/nMP formalism in order to create an ARMv7E-M instruction set simulator. Sim-nML is an extension of the nML formalism with the purpose to perform efficient simulations; and the Macro Preprocessor (nMP) extends the Sim-nML syntax (e.g., recursive macros or macro calls within macros) in order to simplify the implementation. OTAWA is the chosen platform for the WCET evaluation because it is expendable, allowing to integrate new architectures and analysis tools. The decision to use a static code analysis approach was based on the requirements of the R&D project, which is intended to extend the analysis environment with other static analyses (e.g., stack usage, control-flow graph) at a later stage. A more detailed description of this choice is given in Section IV.

IV. IMPLEMENTATION

The OTAWA framework features an architecture abstraction layer, enabling to use the analysis framework independently of the actual target platform. This layer applies an architecture plug-in as an interface to the corresponding architecture loader, which contains all relevant details for the instruction set simulation.

Figure 1 shows the structure of the OTAWA framework and highlights the mentioned concept of hardware abstraction by binding modules for the TriCore, PowerPC or ARMv5 architecture towards this layer. The OTAWA core links all modules including a set of analyses (e.g., data flow analysis), graph generation (e.g., control-flow graph), providing an abstract representation of the program as well as accessing an external ILP solver (e.g., `lp_solve`). From a programmer's perspective, the architecture loader together with a processor description in the form of an OTAWA script are necessary to support WCET analyses for a certain processor. These two

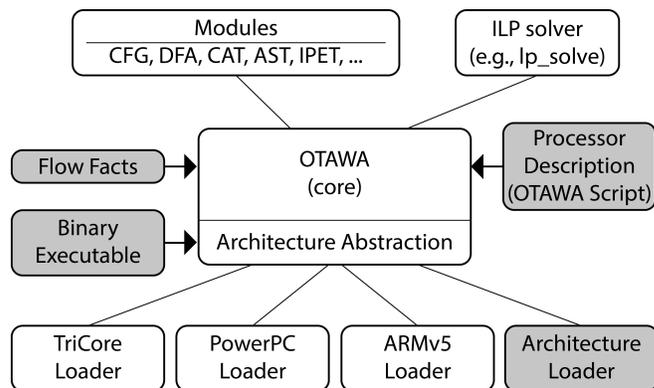


Figure 1. OTAWA Structure

components are highlighted on the right side in Figure 1. One can use the OTAWA framework to perform WCET analyses by providing the binary executable and the flow facts of a program, as shown on the figure's left side. An overview of the necessary implementation steps is given below:

- 1) Description of the instruction set architecture as loader module.
- 2) Implementation of an interface for the architecture abstraction layer in order to link the loader module to the OTAWA framework.
- 3) Creation of a script for the target processor.
- 4) Adaption and extension of the analysis tool if additional features are desired.
- 5) Verification of the implementation.

The next sub-sections describe our implementation of the mentioned steps in a generic manner. Starting with step one and two in Section IV-A, followed by the processor script in Section IV-B as well as two tools (MKFFX and OSWA) in Section IV-C and Section IV-D that are using the OTAWA framework for evaluating a programs WCET, flow facts and basic block statistics. Details of our concrete implementation are given in Section IV-E, and finally, the verification is explained in Section IV-F.

A. Architecture Loader

This module is the core part of the work presented in this paper and includes a description of the processor architecture in form of the instruction set. Most of the code was written in the Sim-nML/nMP language including information of the syntax, the binary representation and the semantics of each instruction from the architecture. The syntax is important for the disassembler output and control-flow graph because it is the representation of the assembler syntax. The image is used for linking the bits of a decoded instruction to the corresponding parameter (e.g., register or immediate value). Within the action part, the parameters from the image are used for describing the instruction's function. This means that calculations (e.g., shift or add), writing and reading registers, as well as updating flags is part of the action.

Beside the instructions themselves, there are registers, conditions, modes and exceptions within the Sim-nML/nMP part. Additionally, macros were defined to decrease the implementation effort and at the same time increase the readability. The rest of the implementation, containing auxiliary functions and algorithms, was written in the C-language. Afterwards, the

Generator of Libraries for Instruction Set Simulators (GLISS) was used to generate a C-library out of both implementation parts. This library along with the ARM module from OTAWA serve as input for the generation of the instruction set simulator, a so-called “architecture loader”. In a final step, it was necessary to define the instructions kind (e.g., ALU or branch), target, semantic and used registers within the OTAWA ARM module in order to interpret each instruction correctly. All relevant architecture information for the work can be found in the corresponding architecture reference manual.

B. Processor Script

This section describes the implementation of processor characteristics as OTAWA script, so that applications targeting a specific microcontroller can be analyzed with the OTAWA framework. The script is written in XML format and consists of several files, each for one component of the processor. The separation is described by the following listing:

- **Main:** This file is usually named after the microcontroller and includes information about the used architecture. It links all parts of the platform description (e.g., memory) and allows to configure items to fine-tune the analysis. In addition, necessary analysis steps can be included, which are accessible through the OTAWA API (e.g., `BB_TIME_FEATURE` ensures that the execution time computation of each basic block has been performed).
- **Memory:** The processor’s different memory banks with their properties are described in this file. A typical description of a memory bank includes a name, the start address and its size, followed by the type (e.g., FLASH or SRAM). Read/write latencies can be defined in order to set a number of cycles for accessing or writing the memory. This is especially relevant for external memories with high access times. Finally, one can specify if a memory is writable or cachable.
- **Pipeline:** This file describes the processor’s pipeline as big picture because its complexity is in many cases not describable. Each stage is described by an ID, a name, a width defining the number of parallel processed instructions, a latency for multi-cycle operations and a type. The type is typical *fetch* for the very first stage (e.g., instruction fetch from memory) and *commit* for the very last stage to declare the exit. In between, there are either lazy stages (e.g., decode) waiting for a defined time as well as execution stages. It is possible to define functional units for the execution stages (e.g., arithmetic logic or floating-point unit), allowing to link certain types of instructions to them.
- **Cache:** The processor’s caches are described in this part. It is possible to state data, instruction or unified caches, whereby the elements are all the same. Each configuration consists of a replacement policy (e.g., LRU or FIFO), the size of a cache block, the number of blocks in each set and the number of sets in the cache. Additionally, different levels of cache can be defined.

It is essential to create a script for any used microcontroller as already small differences can result in a WCET deviation.

C. Flow Facts Evaluation Tool

Since the exact control flow of a program depends on input data, it is impossible to make an estimation without program execution. So-called flow facts, include program flow information like maximum loop iteration counts (loop bounds) or recursion depths and are provided by the user. These details improve the precision of the analysis result; often they are necessary to evaluate a program’s WCET at all. In cases where no explicit limitations (e.g., loop bound depends on input parameter) are given, either the user defines high but safe bounds (e.g., maximum value of the parameter’s data type), which makes the WCET result inaccurate, or the estimation is infeasible. Defining the flow facts by hand is exhausting and imply a risk of incompleteness. Therefore, some analysis tools can automatically detect flow facts and save them into a respective file. The task of filling in missing information (e.g., boundary for a found loop) remains to be done prior to the evaluation of the WCET.

The introduced tool, called `mkffx`, generates flow facts in XML format by combining various input methods. First, it reads possibly existing flow facts from a given file and saves them in an internal representation, followed by analyzing the binary file to detect and record loops and other control information. Next, it will invoke the `oRange` tool [22], which analyzes loop bounds and extracts flow facts from the source code. This is an optional feature, since the source code is not available in every use case. Afterwards, the `mkffx` tool merges all results and outputs the flow facts. In this way, it reduces the necessary effort of describing them by hand because the combination of several inputs increases the rate of automatically detected loop bounds. Our `mkffx` tool extends the features of the `mkff` tool, which already comes with the OTAWA framework.

D. WCET Analysis Tool

Evaluating a programs WCET takes several analysis steps which can vary depending on the processors architecture. Figure 2 shows a typical scenario of a WCET estimation using OTAWA.

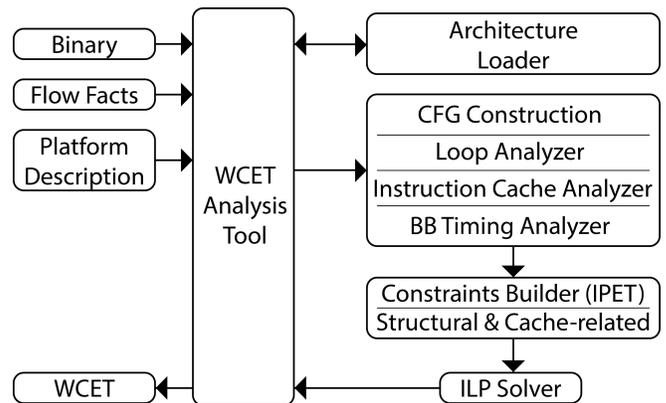


Figure 2. OTAWA Scenario

The very first step is to load the program under analysis in the form of the binary executable as well as its flow facts information and the corresponding processor script (platform description). Next, the program from the binary file is transformed into an internal representation using the architecture

loader and the CFG builder constructs its control-flow graph. Afterwards, analyses are applied starting with the loop analyzer which uses the loop boundaries from the provided flow facts. The platform description is taken into account to analyze the instruction caches behavior. With this information, the execution time of each basic block can be calculated. Sets of graph flow constraints (e.g., program flow and basic-block execution time bounds) are built for the implicit path enumeration technique (IPET) based calculation approach. These constraints are transformed into an integer linear programming problem with a goal function (WCET) and then solved using an external solver (e.g., `lp_solve`). In the end, the analysis tool can output the resulting WCET. The OTAWA framework includes all of the mentioned modules as shown in Figure 1.

The OTAWA Stack and Worst-case execution time Analysis (OSWA) tool combines several features into one application. Beside the two main functions derived from its name: stack usage evaluation and WCET analysis, it can generate a control-flow graph with various output kinds and creates a basic block timing statistic. The latter allows identifying the most time consuming basic blocks within a given function or code snippet. An additional feature is the calculation of a ratio between the time spent inside and outside the function, which can be used to find out how much time is spent in sub-functions. For this paper, the WCET analysis feature is the most important one. OSWA performs the analysis of a specified function from a given binary file by involving flow facts and a processor description (OTAWA script). Our OSWA tool extends the features of the `owcet` tool, which already comes with the OTAWA framework.

E. Specific Implementation for the Use Case

The goal of our work is to enable WCET analysis for software targeting ARM Cortex-M4 processors. For this reason, the implementation in Section IV-A was accomplished for the ARMv7E-M architecture. It features the Thumb-2 technology with both, 16 and 32 bit operations. This architecture loader is based on an existing ARMv5 loader because its 16 bit Thumb instructions are mostly equivalent with the ARMv7 technology.

The Infineon XMC4500-F100K1024 microcontroller features an ARM Cortex-M4 processor core and was chosen for further evaluation (see Section VI). Its characteristics were described in the form of an OTAWA script as presented in Section IV-B.

F. Verification

As described in Section IV-A, the architecture implementation is split in a Sim-nML/nMP description and code written in the C language. As a result of the build process, a C-library that contains both parts is generated. This entire implementation as well as Sim-nML/nMP parts were verified using simulation, code reviews, and disassembler output comparison. The verification of the C parts is completed with the following methods: model checking, static code analysis, and test drivers based on boundary value analysis and equivalence class partitioning. In addition, a plausibility check of the implementation was performed by comparing WCET results of selected test cases with measurements and results from another tool, as shown in the subsequent section.

V. BENCHMARKS

A comparison of WCET results was made between (1) OTAWA with the implemented ARMv7E-M architecture loader as well as the OSWA tool, (2) the Advanced Analyzer for ARM (A³) version 14.04 from AbsInt GmbH, and (3) a measurement-based approach. Although A³ only supports the ARM Cortex-M3 and not the ARM Cortex-M4 processor family, a comparison is possible because both are using the ARMv7 architecture with the Thumb2 instruction set. This behavior is valid as long as no ARM Cortex-M4 specific instructions (e.g., DSP extension) are used, otherwise, the executable would be different from the ARM Cortex-M3 version and not compatible with A³. The measurement-based WCET analysis uses a manually written test driver with input parameters that cause a worst-case scenario. One can identify the worst-case behavior by hand for the chosen benchmarks because of their rather simple program flow; however, it would be a challenge to cause the worst-case behavior for more complex applications. By toggling an I/O pin before and after a certain program code, one can record its execution time using an oscilloscope or logic analyzer. The Infineon XMC4500 is the microcontroller of choice for the measurements, which operates at a frequency of 120 MHz. For the purpose of comparing the measured WCET with the analysis tools, all results are converted into a time unit (based on the processors clock rate) and are recorded in μs rather than in cycles. Table II shows the WCET evaluation results of 4 test cases. The first test case (For-If-Add) is a very basic example, only containing a loop with an if-else construct and some additions. The functions Factorial and Fibonacci are clearly assigned to a known algorithm by their names; however, their results are only valid for the given input scenario (e.g., Fibonacci number and factorial of 50). The fourth and last test case contains a preliminary implementation of the resolution advisory (RA) component from the Traffic Alert and Collision Avoidance System (TCAS). Its purpose is to issue climb or decent directives in case of conflicting aircrafts [23].

TABLE II. WCET RESULTS COMPARISON FOR THE TEST CASES.

Test Case	OTAWA	AbsInt A ³	Measurement
For-If-Add	18.16 μs	18.07 μs	17.98 μs
Factorial	23.33 μs	23.72 μs	22.82 μs
Fibonacci	14.35 μs	8.09 μs	8.07 μs
TCAS	8.71 μs	7.82 μs	6.49 μs

Overall, the results show that the measurement-based approach leads in every case to a lower WCET. This circumstance is very important because otherwise the WCET analysis tools would evaluate a wrong or underestimated result which cannot be used for safety-critical real-time applications or generally for verifying timing constraints as it can lead to software misbehavior that might have a catastrophic impact. In all four test cases, the WCET evaluations by OTAWA and A³ deliver a safe upper bound, meaning it is above the real value and therefore trustworthy. Additionally, the gap between the real WCET and the estimated ones are especially in the first two test cases minor. Since the goal is to get as close as possible to the reality, this tight output is desirable. The For-If-Add test case shows, that A³ delivers a value 0.09 μs above the measured one but also 0.09 μs tighter than OTAWA. At the second function, factorial, OTAWA estimates a 0.39 μs

tighter result than A^3 and $0.51 \mu s$ above the measured WCET. Next, the Fibonacci algorithm shows that OTAWA calculates a weaker WCET bound, whereas A^3 estimates a extremely tight value. Finally, the TCAS test case challenges the analysis tools, as there are many sub-routine calls and thereby, initiating several pipeline refills with a variable duration depending on things like the width of the target instruction. In general, the different results between both tools can be caused by deviating analysis techniques or the usage of other integer linear problem solver.

In summary, it can be stated that the OTAWA Framework with both, the ARMv7E-M loader and OSWA can compete with a commercial tool by delivering safe and mostly tight results.

VI. USE CASE

The use case shows a software part of an unmanned aerial vehicle (UAV), more specific, a quad-copter. Since UAV software contains plenty of software components, it is important to ensure that all of them have enough resources to do their tasks in order to satisfy any deadline, hence, to guarantee safe operation. This section discusses one functionality exemplary, though the entire UAV software needs to be analyzed. The goal is that the quad-copter can remain static in the air at given height between 20 and 150 centimeters. This task consists of a measurement unit to detect the current height and pass on the distance information to the engine task which uses control algorithms for adapting the current height to a given value. Stabilization is performed by the engine control using a triple-axis gyroscope.

This use case describes timing analysis of the distance measuring and evaluation software. An infrared proximity sensor is used to measure the distance between the quad-copter and ground. It delivers an analog output with a nonlinear distance measuring characteristics. The sensors characteristic is approximated and expressed as mathematical equation. As a result, it is necessary to use an analog-digital converter to read in the latest sensor value and calculate the current distance according to its characteristic.

Listing 3 shows the source code of the sensor read function and is described in this paragraph. One can pass the number of measurements as argument to the function. If the parameter is zero or one, only a single value will be measured (see *GetADCValues* function). Otherwise, the given size equals the number of measurements from which the mean value will be generated (see *mean* function). In both cases, the distance is calculated using the *evaluateDistance* function that implements the sensor characteristics as formula. The functions *GetADCValues* and *mean* each contain a loop, which bound depends on the given size. In Addition, the analog-digital conversion executed within the *GetADCValues* function takes several cycles, depending on the ADC configuration (e.g., conversion width of 12 bits or the divider factor for the analog internal clock). For the WCET analysis, this delay is considered and implemented as busy-waiting loop.

Although this software component is a relatively small one, its importance is beyond debate because it delivers information about the current height and wrong or no up-to-date data could lead to an accident. Therefore, it is necessary to estimate the timing behavior of this software component by evaluating its worst-case execution time.

```
uint16_t readSensor(uint16_t size)
{
    uint16_t distance;

    if(size < 2)
    {
        uint16_t adc_value;
        GetADCValues(&adc_value, 1);
        distance = evaluateDistance(adc_value);
    }
    else
    {
        uint16_t adc_values[size];
        uint16_t temp;
        GetADCValues(adc_values, size);
        temp = mean(adc_values, size);
        distance = evaluateDistance(temp);
    }
    return distance;
}
```

Figure 3. Distance measuring source code

The process of analyzing its worst-case timing behavior start with the binary file of the program, shown in Listing 3, by identifying and evaluating all loop bounds using the mkffx tool. In the use case, a mean value from 32 analog-digital conversions is used for the height estimation. Therefore, the loop bounds within the *GetADCValues* and *mean* function need to be 32. After the generated flow facts are checked, the OSWA tool can be executed to estimate the functions WCET.

TABLE III. WCET RESULTS OF THE USE CASE.

Function	OTAWA
readSensor	68.58 μs
GetADCValues	62.22 μs
mean	5.20 μs
evaluateDistance	0.425 μs

Table III shows the worst-case execution time evaluation results of the functions from the use case. All results are estimated for the Infineon XMC4500 microcontroller operating with a clock rate of 120 MHz. The distance calculation, including all sub-routine calls, takes 68.58 μs in the worst-case. A measurement was performed where an execution time of 67.37 μs was recorded, giving the information that the WCET analysis is safe and tight. The results of the sub-routines are revealing where most of the time is spent. In this case, recording the ADC values takes the majority and evaluating the distance the least of the time. This ratio depends on the number of analog-digital conversions taken into account for one distance calculation. Finally, the main statement of the results is emphasizing the necessary time budget of 68.58 μs for the entire task.

VII. CONCLUSION

This paper elaborates on the implementation of a processor model for worst-case execution time analysis. The presented approach integrates with the open-source framework OTAWA and, hence, can serve as guide for similar efforts.

It starts with the architecture implementation, which is split into a Sim-nML/nMP model and an OTAWA script, resulting in a behavioral architecture description with timing information of operations in order to generate a cycle-accurate instruction set simulator. In particular, we choose the ARMv7E-M architecture that is used by ARM Cortex-M4 devices.

The resulting toolset was evaluated by way of a benchmark in order to underline its save and tight WCET calculation.

Our lesson learned is that the seamless and correct description of a processor model is exhaustive; further, the implementation quality depends on the correctness of the architectures datasheet. We experienced that the OTAWA framework is capable of much more than WCET analysis, because the existing analyses can be adapted to fulfill own requirements or purposes.

In summary, the presented approach enables static binary analysis of a program targeting an implemented architecture. This allows to evaluate information of the application like the WCET, which can be used for creating a statement regarding possible violations of deadlines or task scheduling in real-time systems.

ACKNOWLEDGMENT

This work has been conducted in the context of the public funded R&D project Software Analysis Toolbox managed by the Vienna City Council MA23.

REFERENCES

- [1] C. Ferdinand, "Worst case execution time prediction by static program analysis," In 18th International Parallel and Distributed Processing Symposium, IPDPS'04, 2004, pp. 125a.
- [2] H. Cass, H. Ozaktas, and C. Rochange, "A Framework to Quantify the Overestimations of Static WCET Analysis," In 15th International Workshop on Worst-Case Execution Time Analysis, WCET'15, 2015, pp. 1–10.
- [3] R. Kirner, P. Puschner, and I. Wenzel, "Measurement-based worst-case execution time analysis," In IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems, SEUS'05, 2005, pp. 7–10.
- [4] F. Guet, L. Santinelli, and J. Morio, "On the Reliability of the Probabilistic Worst-Case Execution Time Estimates," In Proceedings of the Embedded Real-time Software and Systems, ERTS'16, 2016, pp. 758–767.
- [5] X. Jean, S. Girbal, A. Roger, T. Megel, and V. Brindejonc, "Safety considerations for WCET evaluation methods in avionic equipment," In IEEE/AIAA 34th Digital Avionics Systems Conference, DASC'15, 2015, pp. 7A4–1 to 7A4–15.
- [6] M. Paolieri and R. Mariani, "Towards functional-safe timing-dependable real-time architectures," In IEEE 17th International On-Line Testing Symposium, IOLTS'11, 2011, pp. 31–36.
- [7] S. Fischmeister and I. Lee, "Temporal Control in Real-Time Systems: Languages and Systems," In Handbook of Real-Time and Embedded Systems, 2007, pp. 10–1 to 10–18. CRC Press.
- [8] F. Stappert, A. Ermedahl, and J. Engblom, "Efficient longest executable path search for programs with complex flows and pipeline effects," In Proceedings of the 2001 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, CASES '01, 2001, pp. 132–140.
- [9] A. Colin and G. Bernat, "Scope-tree: a program representation for symbolic worst-case execution time analysis," In 14th Euromicro Conference on Real-Time Systems, ECRTS '02, 2002, pp. 50–59.
- [10] Y. T. S. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," In Design Automation, DAC '95, 1995, pp. 456–461.
- [11] N. Holsti and S. Saarinen, "Status of the Bound-T WCET tool," In 2nd Int. Workshop on Worst-Case Execution Time Analysis, WCET02, 2002, pp. 36–41.
- [12] G. Bernat et al., "Identifying Opportunities for Worst-case Execution Time Reduction in an Avionics System," Ada User Journal, Volume 28, Number 3, 2007, pp. 189–194.
- [13] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper, "Automatic Derivation of Loop Bounds and Infeasible Paths for WCET Analysis Using Abstract Execution," In 27th IEEE International Real-Time Systems Symposium, RTSS'06, 2006, pp. 57–66.
- [14] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat, "Ottawa: An open toolbox for adaptive wcet analysis," In Software Technologies for Embedded and Ubiquitous Systems, SEUS'10, 2010, pp. 35–46.
- [15] H. Arora, A. Gupta, R. Singhai, and D. Purwar, "Design space exploration of risc architectures using retargetability," In VLSI Systems, Architecture, Technology and Applications, VLSI-SATA'15, 2015, pp. 1–3.
- [16] X. Li, A. Roychoudhury, T. Mitra, P. Mishra, and X. Cheng, "A retargetable software timing analyzer using architecture description language," In Design Automation Conference, ASP-DAC'07, 2007, pp. 396–401.
- [17] T. Ratsimbahotra, H. Cassé, and P. Sainrat, "A versatile generator of instruction set simulators and disassemblers," In Symposium on Performance Evaluation of Computer & Telecommunication Systems, SPECTS'09, 2009, pp. 65–72.
- [18] P. Grun et al., "Expression: An ADL for System Level Design Exploration," Technical Report TR 98-29, University of California, Irvine, USA, 1998.
- [19] V. Zivojnovic, S. Pees, and H. Meyr, "Lisa-machine description language and generic machine model for hw/sw co-design," In VLSI Signal Processing, IX, 1996, pp. 127–136.
- [20] A. Fauth, J. Van Praet, and M. Freericks, "Describing instruction set processors using nml," In European Design and Test Conference, ED TC'95, 1995, pp. 503–507.
- [21] P. Mishra and N. Dutt, "Processor Description Languages: Applications and Methodologies," vol. 1, chapter 1, Morgan Kaufmann, 2008, ISBN: 978-0-12-374287-2.
- [22] M. Michiel, A. Bonenfant, C. Ballabriga, and H. Cassé, "Partial Flow Analysis with oRange," In International Symposium on Leveraging Applications, ISoLA'10, Part II, Springer, 2010, pp. 479–482.
- [23] U.S. Department of Transportation, Federal Aviation Administration, "Introduction to TCAS II Version 7.1," 2011.