Experiences from Adjusting Industrial Software for Worst-Case Execution Time Analysis

Patrick Denzler*, Thomas Frühwirth*†, Andreas Kirchberger*, Martin Schoeberl‡ and Wolfgang Kastner*
*Institute of Computer Engineering, TU Wien, Vienna, Austria
Email: patrick.denzler@tuwien.ac.at, a.kirchberger@kbit.pro, wolfgang.kastner@tuwien.ac.at
†Research Department, Austrian Center for Digital Production, Vienna, Austria
Email: thomas.fruehwirth@acdp.at
‡Department of Applied Mathematics and Computer Science, DTU, Lyngby, Denmark
Email: masca@dtu.dk

***Abstract*—Worst-case execution time (WCET) analysis is a prevalent way to ensure the timely execution of programs in time-critical systems. With the advent of new technologies such as fog computing and time-sensitive networking (TSN), the interest in timing analysis has increased in industrial communication. This paper highlights experiences made while adjusting the publisher of the open62541 OPC UA stack to enable WCET analysis, following a simple process combined with the open-source platform T-CREST. The main challenges are the required knowledge about the code and the specific communication software characteristics like variable message sizes. Other findings indicate the need for other types of annotation for indirect recursion or callback functions. The paper provides the foundation for further research on adjusting the implementation of existing industrial communication protocols for WCET analysis.**

***Index Terms*—worst-case execution time, industrial software, transformation rules, real-time communication, OPC UA**

I. Introduction

Program timing analysis aims to determine the program's execution-time characteristics and is mainly used for hard real-time systems with strict timing requirements [1]. A fundamental problem of this type of analysis is that the execution time varies. The cause for those variations is, amongst other things, changes in input data, as well as the specifics of the software and the used hardware (i.e., processor, platform).

A well-known timing measure is the worst-case execution time (WCET) of a program. Commonly used in schedulability analysis, WCET is studied to ensure that interrupt reaction and periodic processing times or operating system calls occur within a specified time-bound. In essence, WCET analysis determines if a piece of code will execute within its allocated time budget [1], [2]. Accurate timing and WCET estimates are vital in the design and verification of real-time and embedded systems, primarily if used in safety-critical products such as aircraft, vehicles, or industrial plants [3], [4].

This work has been partially supported and funded by the Austrian Research Promotion Agency (FFG) via the "Austrian Competence Center for Digital Production" (CDP) under the contract number 854187. Moreover, the research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 764785, FORA—Fog Computing for Robotics and Industrial Automation.

WCET and timing analysis are tool-based and follow different methodologies such as measurements and static analyses [5]. Measurement-based tools are suitable for less time-critical software, as the focus lies on the average timing behavior. For time-critical software with tight WCET bounds, static analysis or some hybrid method is preferable. Most tools consider the used processor and hardware specifics to obtain realistic WCET results [6].

The interest in timing analysis has also reached industrial automation, as it currently changes towards higher integration and seamless communication. While most industrial systems do not require exact timing estimations, they still can benefit the overall system's performance [7]. Another emerging topic is end-to-end real-time machine-to-machine communication. While technological advances such as fog computing, time-sensitive networking (TSN) [8] and communication protocols such as OPC unified architecture (OPC UA) [9] provide the technological means to achieve this goal, the required predictability of the software parts remains an open issue. The complexity of the communication protocols and limited experience with WCET analysis in the automation industry hinder obtaining reliable timing measures necessary for end-to-end real-time machine-to-machine communication.

This paper addresses the feasibility of static WCET analyses of industrial non-real-time communication protocols to support timing analysis in industrial automation. The outcomes unveil that manual annotation's effort strongly depends on the programmer's familiarity with the code and used WCET tool. Exact code and comment guidelines would further ease the adjustment. Other results show that message sizes in communication protocols strongly influence the WCET. The time-predictable platform T-CREST provided the necessary WCET tools and hardware [6]. Further contributions are a WCET adjusted publisher of the open-source OPC UA project open62541 [10], and WCET estimates relevant for ongoing research in end-to-end real-time machine-to-machine communication. The evaluation measurements confirm the calculated WCET results. The introduced adjustment process for WCET program analysis provides an introduction for programmers not familiar with the topic.

This paper contains seven sections: the following two sections present background and related work relevant to the topic. Section IV introduces the applied process and the transformation rules followed by a proof of concept and evaluation in Section V. Section VI discusses the findings and results, and Section VII concludes the article.

II. Background

This section provides background information on WCET analysis and tools. Interested readers can refer to [1] for a detailed survey about WCET analysis.

A widely used industry method to determine program timing is by measurements [5]. This method executes a program several times with different inputs to measure the execution time. The results represent a statistical description of the timing behavior or an approximate WCET value.

A limitation is that each run only follows one program path; therefore, in most cases where the execution paths are too numerous, this method is not applicable. In such cases, the measurements will underestimate the WCET and require adding safety margins, to try to ensure that the actual WCET value is less than that bound. However, adding margins always carries the risk of over and underestimation and causes either waste of resources or schedulability issues. Often measurement-based methods use oscilloscopes, logic analyzers, and in-circuit emulators on the actual hardware to obtain measurements.

Static WCET analysis is a technique to determine WCET estimates. This type of method does not execute the program but statically analyses the timing properties [11]. Such tools tend to give larger WCET estimates (upper bounds) than the actual execution time without the need for additional margins. A typical WCET analysis contains three phases: a flow analysis to identify the possible program execution paths, a low-level analysis to estimate times for atomic parts of the code (e.g., instructions, basic code blocks), and the calculation phase combines the two previous phases into a WCET estimation.

Flow analysis focuses mainly on loop bound analysis [12], [13] since the amount of iterations of a loop affects the WCET estimates. Modern tools contain methods to determine loop bounds; however, in most cases, manually adding the loop bounds is still required. Other features of the flow analysis are the possibility to identify infeasible paths, i.e., paths which are feasible in the control-flow graph (CFG), but impossible when examining the input data values and the semantics of the program [12].

The low-level analysis takes care of modern hardware's combined behavior, with features like pipelines, caches, and out-of-order execution [6]. Models, e.g., simulators, of the hardware, are a common way to approach these issues and make the analysis without the actual hardware possible. However, accurate models of a processor and hardware can become complex, sometimes too complex to be usable. Therefore, safe simplifications of the processor models are needed, leading to higher WCET bounds.

Overall the combination of flow and low-level analysis allows the calculation of a WCET bound. As the complexity of current software and hardware continually increases, there is a wide range of research activities. The areas span from integer linear programming (ILP) [14], model checking [15], and tree-based calculation [12]. Other research directions are concerned with code conversion to WCET-analyzable single-path code to improve the execution time [16] or specialized programming languages [17].

There are various commercially available timing analysis tools, including *aiT (static)* [18] from AbsInt or *RapiTime (hybrid)* [19] from Rapita Systems and several academic open-source prototypes, such as *T-CREST* [6], or *SWEET* [12].

III. Related work

The body of knowledge concerned with timing analysis is considerably large; however, concrete studies on analyzing industrial software are not widespread. There are studies on analyzing code for space applications [3], [20], [21], avionics [22], and the automotive industry [4].

Specific in the industrial context, the authors in [23] conducted a case study to find upper time bounds for time-critical industrial code with static WCET analysis. In their study, the authors identified practical difficulties when applying current WCET analysis methods and determined how labor-intensive the analysis becomes. Similarly, Gustafson et al. [7] summarized their experiences from five different industrial case-studies using both static and measurement-based tools. They investigated if current timing analysis methods are suitable for industrial code and their accuracy. The WCET results show very high accuracy but required a sound knowledge of the tool and the code. Also, the effort increases when adding annotations to create better estimates, especially for complex code. In [24], the authors tested an automatic flow analysis method on industrial real-time system code and achieved high accuracy results. However, all these case studies concern the analysis of binary code only. A common problem is the need to provide cumbersome and error-prone manual annotations for program flow properties at the binary level. Lisper et al. [25], approached the annotation issue by executing the program flow analysis on the source level and found that this method can resolve some program flow constraints on the binary level.

Other authors focused on general WCET challenges. For example, Sehlberg et al. [26] found that industries in this field oppose complexities with strict guidelines that exclude language constructs that make programs not WCET-analyzable. Such guidelines are most important for general-purpose languages such as C. In C, structures such as pointers, recursive data structures, dynamic memory allocation, assignments with side effects, recursive functions, and variable-length loops are known to be an obstacle for WCET [27]. While modern tools can handle most of such language constructs, some require reprogramming. Sehlberg et al. [26] concluded that it is relatively easy to obtain loose WCET bounds using analysis tools. However, the effort increases when more accurate WCET estimates are required [28].

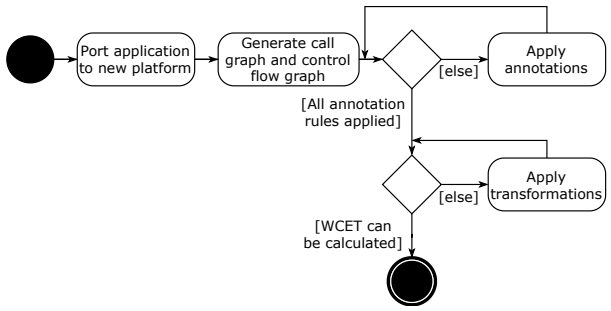

Fig. 1. Process for adjusting existing software for WCET analysis

IV. Code transformation process

A considerable effort in the WCET community is directed towards automating WCET analysis. The work resulted in commercial tools like Absint *aiT* and Rapita Systems *RapiTime* but also open-source tools like *SWEET* and the *T-CREST* platform. Although these tools offer significant support for all necessary steps, determining the WCET cannot be fully automated. In practice, static WCET analysis of existing code that has not been written for real-time applications often requires additional manual work to calculate the WCET. Furthermore, finding reasonable tight bounds to make the code applicable in real-time applications may require additional effort.

The essential steps to prepare an existing program for WCET analysis have been derived from WCET analysis tools and combined into the process illustrated in Figure 1. It is intended to serve as an abstract guide and a helpful starting point for static WCET analysis. The process consists of *(A)* porting the existing code to the new platform, *(B)* examining the code structure via the call graph and the CFG, *(C)* applying code annotations, and *(D)* code transformations. The following paragraphs provide more details on each process step, focusing specifically on challenges that may arise from conducting static WCET analysis of existing software.

A. Port application to new platform

Many industrial applications, in particular, if they employ real-time aspects, are executed on specialized computing platforms. However, most existing computer programs are developed for general-purpose processors. Therefore, porting the application to the target platform is often required as a preliminary step. Examples for platform-specific functionality include:

- Network communication
- Hardware access
- Timing and interrupts
- File access
- User interaction via input/output devices

This step of the process results in a program that can be compiled and executed on the target platform.

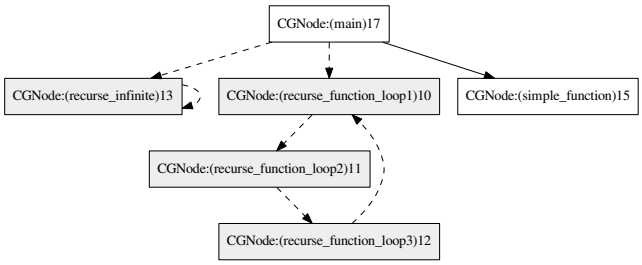

Fig. 2. Example of a call graph with direct and indirect recursions

B. Generate call graph and control flow graph

The call graph and the CFG provide a structured overview of software programs. A call graph visualizes the interdependencies between functions, as illustrated in Figure 2. Nodes in the graph represent functions of the program, and directed edges indicate function calls. A directed edge from node f to node g shows that f calls g, i.e., g is a sub-routine of f.

On the other hand, the CFG contains program statements (one or several lines of code) represented as nodes. Directed edges visualize the control flow between these program statements, as depicted in Figure 3. The CFG is called intraprocedural if it covers a single function, or interprocedural if it spans across multiple functions.

In the code transformation process, the call graph and the CFG fulfill two purposes. First, they provide graphical means to examine the structure of possibly large and complex programs. This also includes identifying all program parts relevant for the WCET analysis, as in many industrial applications only specific functionalities are time-critical.

Second, the call graph and the CFG allow determining programming constructs that pose a hindrance for WCET analysis. Thereby, the call graph allows detecting direct and indirect recursions, which are visible as cycles, as shown in Figure 2. Likewise, the CFG allows identifying loops, conditional, and jump statements. They are visible as non-sequential paths in the CFG, as illustrated in Figure 3.

The programming constructs that do not allow for fully automated calculation of the WCET need to be addressed either by adding code annotations or applying code transformation rules. Whether a specific programming construct requires annotation or transformation depends mainly on the features of the WCET analysis tool. For example, annotations

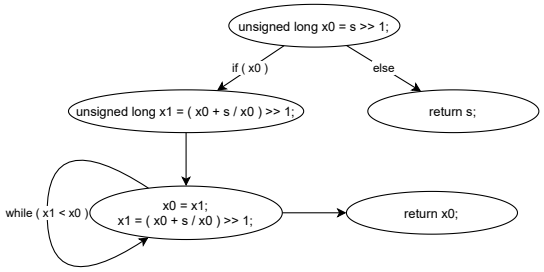

Fig. 3. Example of a control flow graph with condition and loop

to define the minimal and maximal number of loop iterations are supported by virtually any WCET analysis tool. However, this is not necessarily the case for recursions. Specifically for indirect recursions, applying a code transformation rule, e.g., replacing the recursion by iterations, may be required.

Programming constructs that pose an obstacle for WCET analysis were identified by examining the WCET benchmarks TACLeBench [29], PolyBench [30], and the benchmarks provided by Mälardalen University [31]. Additional constructs motivated by practical experience have also been added.

C. Apply annotations

This section contains programming constructs analyzable by most WCET analysis tools if appropriate code annotations are added. Depending on the tool, these annotations are added directly in the source code or in a separate file. In the following pseudocode examples, the syntax suggested by the TACLeBench is used.

1) While loop: Repetition (loop) is a fundamental part of a program's control structure yet presents a challenge for WCET analysis. The reason is that the loop condition might depend on unavailable information during compile time, e.g., a configuration file, user input, or sensor data. Therefore, it is often impossible to automatically calculate a valid and tight upper bound for the number of loop iterations.

Algorithm 1 (top) shows the annotation that needs to be applied to enable WCET analysis of a while loop. The idea is to define the minimum and maximum number of loop iterations via the code annotation (*_Pragma*). An assert statement may be added within the loop to ensure that the upper bound is not exceeded during runtime. This may be required if the loop bound cannot be determined with absolute certainty by static value analysis or similar techniques. Violating this assertion constitutes a WCET violation error. The programmer needs to implement appropriate error handling measures in such a case.

2) Do-while loop: In do-while loops, the same rules apply as for *while loops* with one exception. The minimum loop bound in Algorithm 1 (middle) must be at least 1 ($X \geq 1$) as a do-while loop is executed at least once. Again, an assert statement may be required.

3) For loop: Algorithm 1 (bottom) shows the annotation of a for loop. It is semantically equivalent to the while loop.

4) Recursion: A function that calls itself is said to be directly recursive. The call graph shows a direct recursion as a self-referencing node. Algorithm 2 shows the annotation of a direct recursion.

In contrast, an example of an indirect recursion is a function *f1* calling another function *f2*, which calls *f1*. In the call graph, indirect recursions manifest as a cycle spanning across multiple nodes. Some WCET analysis tools do not provide annotations for indirect recursions. Instead, code transformation needs to be applied.

D. Apply transformations

The last step in the process is iteratively applying the transformation rules until the program is WCET-analyzable. This section presents these transformation rules.

Algorithm 1 Annotations: loops

While loop

```
▷ Original code:
while loop condition do
    execute loop content
end while

▷ Code annotated for WCET analysis:
_Pragma("loopbound min X max Y")
while loop condition do
    assert loopbound defined by _Pragma is not violated
    execute loop content
end while
```

Do-while loop

```
▷ Original code:
do
    execute loop content
while loop condition

▷ Code annotated for WCET analysis:
_Pragma("loopbound min X max Y")
do
    assert loopbound defined by _Pragma is not violated
    execute loop content
while loop condition
```

For loop

```
▷ Original code:
for initialization; loop condition; iteration statement do
    execute loop content
end for

▷ Code annotated for WCET analysis:
_Pragma("loopbound min X max Y")
for initialization; loop condition; iteration statement do
    assert loopbound defined by _Pragma is not violated
    execute loop content
end for
```

1) Recursion: If the WCET analysis tool does not support the annotation of a recursion, the recursion needs to be replaced by iterations. As shown in Algorithm 3, a directly recursive function may perform calculations on the state/variable F. The next step executes the recursive call. Upon completing the recursive call, calculations on the resulting state/variable R might be executed.

One general solution to transform a directly recursive function for WCET analysis is using a stack. The resulting iterative function consists of two loops; the first adds data to the stack, and the second retrieves the data back from the stack in a last-in first-out manner. By determining the maximum stack size, the loop bound can be defined for the two loops and the WCET can be calculated. Again, an additional assertion statements ensure that the loop bounds are not exceeded.

Indirect recursions can be transformed to direct recursions by replacing the function call with the body of the called function. This process is called inlining. Considering the example

where *f1* calls *f2*, and *f2* calls *f1*, the function call to *f2* can be replaced by the contents of *f2*, and vice versa. Algorithm 3 can then be applied to the resulting direct recursions.

Algorithm 2 Annotation: direct recursion

```
▷ Original code:
function REC(F)
    if termination condition then
        return F // base case
    else
        F := calculations before recursive call
        R := REC(F) // solve subproblem
        R := calculations after recursive call
        return R
    end if
end function

▷ Code annotated for WCET analysis:
if loopbound defined by _Pragma is violated then
    handle WCET violation error
else
    if termination condition then
        return F // base case
    else
        assert recursion depth defined by _Pragma is not violated
        F := calculations before recursive call
        _Pragma("marker recursivecall")
        R := REC(F) // solve subproblem
        _Pragma( "flowrestriction X*Rec≤Y*recursivecall" )
        R := calculations after recursive call
        return R
    end if
end if
```

Algorithm 3 Transformation rule: direct recursion

```
▷ Original code: cf. Algorithm 2

▷ Code transformed for WCET analysis:
function RECURSION(F)
    Stack S
    _Pragma("loopbound min X max Y")
    while base case not reached do
        assert loopbound defined by _Pragma is not violated
        F := calculations before recursive call
        S.push(F)
    end while
    _Pragma("loopbound min X max Y")
    while S not empty do
        assert loopbound defined by _Pragma is not violated
        F := S.pop()
        R := calculations after recursive call
    end while
end function
```

2) Function pointer / callback function: Function pointers and callback functions are very commonly used in C programming. A pointer is a variable that holds the reference of another variable; function pointers are similar but with the difference that the reference points to a function. Callback functions are passed as an argument (function pointer) to other functions for execution at a given time. In WCET analysis, function pointers and callback functions are difficult to analyze. The reason is that it is unknown at compile-time which callback function will be called. To enable WCET analysis, the programmer needs to explicitly state the possible callback functions, as shown in Algorithm 4.

Algorithm 4 Transformation rule: callback function

```
▷ Original code:
FunctionPointer cb
function REGISTERCALLBACK(FunctionPtr)
    cb = FunctionPtr
end function
function CALLCALLBACK
    cb()
end function

▷ Code transformed for WCET analysis:
FunctionPointer cb
function REGISTERCALLBACK(FunctionPtr)
    cb = FunctionPtr
end function
function CALLCALLBACK
    assert cb is one of Function0 .. FunctionX
    switch cb
        case Function0 do
            Function0()
        case Function1 do
            Function1()
        case FunctionX do
            FunctionX()
    end switch
end function
```

3) Jump table: A jump table is an array of function pointers. Thereby, the function to be called is selected during runtime via the array index. The jump table needs to be replaced by a switch statement or an equivalent if-else-if chain to enable WCET analysis. Algorithm 5 shows the corresponding transformation rule.

Algorithm 5 Transformation rule: jumptable

```
▷ Original code:
Array JumpTable[] = {Function0, Function1, FunctionX}
function FUNCTION(I)
    JumpTable[I]()
end function

▷ Code transformed for WCET analysis:
function FUNCTION(I)
    assert I is a valid index for the JumpTable
    switch I
        case 0 do
            Function0()
        case 1 do
            Function1()
        case X do
            FunctionX()
    end switch
end function
```

4) Non-WCET-analyzable code: Finally, existing software may contain additional non-analyzable code not covered by the previous annotation and transformation rules. For example,

if the program uses pre-compiled libraries, the source code is not necessarily fully available. Relevant parts of such libraries may or may not be WCET-analyzable, depending on whether they make use of any non-analyzable programming constructs. Therefore, it may be necessary to replace pre-compiled libraries with WCET-analyzable alternatives or re-implement the required functionality.

Another aspect is that the program may use functions that are not WCET-analyzable, are infeasible to be transformed because of their complexity, or result in a WCET bound that is impractical for the intended application. Examples are output functions such as printf, blocking input function such as scanf, and functions regarding the dynamic allocation and de-allocation of memory such as malloc and free. Such functions are usually not WCET-analyzable and need to be removed or replaced. This completes the last step of the process, and the resulting program should be WCET analyzable.

V. Proof of concept

An industry-relevant use case is chosen as a proof of concept to demonstrate the process steps and address their practical difficulties. This section covers the technical background, additional information, and main findings. Furthermore, it presents the evaluation setup and WCET results.

A. OPC UA

OPC UA is the successor of the open platform communications (OPC) protocol and a major standard in industrial communication. According to Mahnke et al. [9], OPC UA builds upon two pillars. Firstly, the Meta Model enables information modeling. Secondly, the Transport Mechanisms handle the encoding of data and the exchange of messages between devices. OPC UA offers support for Server-Client and Publish-Subscribe (OPC UA PubSub) communication patterns. The Server-Client mechanism is used for invoking complex services like browsing the information model and calling methods. The OPC UA PubSub mechanism minimizes the communication overhead and is primarily intended for exchanging process data.

The transmission of time-critical messages over the network can be handled by existing real-time Ethernet protocols like TSN [32]. However, providing a real-time capable OPC UA software stack is yet an open challenge. A first step to address this issue is performing a WCET analysis of existing implementations of the OPC UA publisher.

OPC UA software stacks are available in a variety of programming languages. The open62541 open-source stack [10] is chosen for this proof of concept because it is one of the most-sophisticated stacks in terms of supported features, implements the OPC UA PubSub specification, and is designed to be easily ported to other hardware platforms. Furthermore, it is implemented in C, which is well-supported by the T-CREST project and other WCET analysis tools.

B. Patmos and T-CREST

The Patmos processor [6] was designed to simplify the determination of the WCET bounds of tasks compared to general-purpose processors. It builds upon two main mechanisms to enable static WCET analysis: a predictable pipeline design and a predictable memory model. Its in-order execution dual-issue pipeline requires resolving potential hazards with the compiler instead of delaying the pipeline entirely at runtime. Furthermore, its memory architecture allows controlling the timing of memory accesses due to its predictable function cache and a software-managed scratchpad area.

Being a non-standard processor, Patmos requires a set of additional, specialized software tools, which are provided by the T-CREST open-source project [33]. The T-CREST project includes a time-predictable multi-core platform, Patmos-specific compilers, and a WCET analysis toolchain, including coding guidelines [34].

C. Port OPC UA to Patmos

According to the proposed process (Figure 1), the first step is porting the open62541 stack to the Patmos processor. The stack is built in a very modular way. All platform-specific functionality is separated from the rest of the software stack and defined in so-called open62541 architectures. Therefore, porting the stack to the Patmos processor can easily be achieved by defining a new open62541 architecture. The only platform-specific functionality that needs to be implemented is sending and receiving Ethernet frames via the Patmos-specific Ethernet library. The open62541 stack can now be compiled and executed on the Patmos processor. The modified version of the open62541 stack is available at [35].

The following steps cover the actual WCET analysis of the real-time-critical parts of the open62541 stack. For the reasons outlined above, only the parts of the stack handling publishing and subscribing to messages need to be WCET analyzed to enable real-time end-to-end data transmission over OPC UA.

D. Generate call graph

The next step of the process is generating the call graph and the CFG of the time-critical parts of the application. At the publisher, the `UA_WriterGroup_publishCallback` method is the entry point for the WCET analysis. It handles encoding and sending the network messages. Therefore, the call graph for this function is generated. However, the resulting call graph contains 344 nodes and 698 edges in its original form and is too extensive to be included in this paper. To provide an overview of the functions that need to be WCET-analyzed, Figure 4 depicts the call graph resulting after conducting the remaining steps of the process.

The CFGs for each of the involved functions can also be generated using the T-CREST project's tools. However, they are too numerous and too complex to be included in this paper.

E. Apply annotations

The T-CREST project supports manual loop bound annotations. Therefore, the functions illustrated in the call graph are systematically examined and loop bounds are added for each while, do-while, and for loop.

In some cases, determining the loop bound is relatively straightforward. However, as the open62541 stack was not

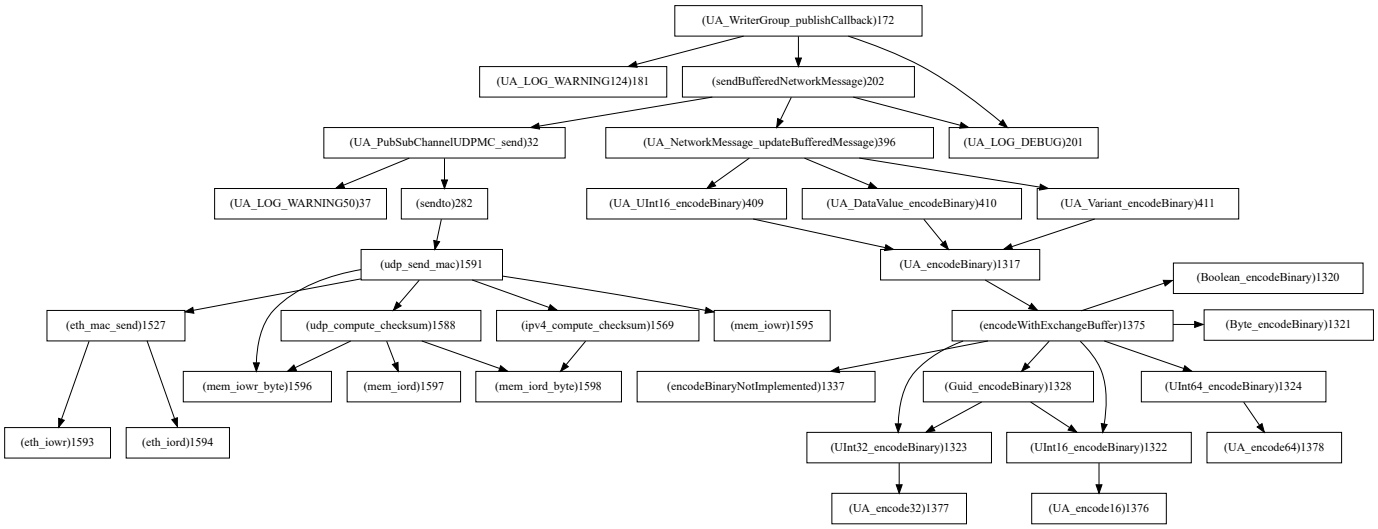

Fig. 4. Call graph of function UA_WriterGroup_publishCallback including WCETs

developed for WCET analysis, most loops do not have predefined bounds. For example, an OPC UA message can contain arbitrarily many data fields. Encoding them in a message requires iterating over the data structure that holds the corresponding values. For such iterations, reasonable limitations need to be defined (e.g., a message may contain at most two data fields). Following the specified rules (cf. Algorithm 1) the _Pragma defining the loop bounds and the corresponding check are added. Table I summarizes how often each loop annotation had to be applied for the open6541 publisher.

TABLE I
PROGRAMMING CONSTRUCTS AND NUMBER OF OCCURRENCES FOR THE UA_WRITERGROUP_PUBLISHCALLBACK FUNCTION

Programming construct	Number of occurrences
While loop	1
Do-While loop	0
For loop	1
Indirect recursion	1
Jumptable	1
Other, non-WCET-analyzable code	6

F. Apply transformation rules

T-CREST does not support annotations for recursions. Therefore, they need to be replaced with iterations as suggested in Algorithm 3.

The next programming constructs that need to be addressed by their corresponding transformation rule are jumptables (cf. Algorithm 5). The open62541 stack utilizes these quite heavily, e.g., to call the corresponding function for encoding each of the data fields depending on their data types.

Finally, there are two more programming constructs that fall into the category of non-WCET-analyzable code (cf. Section IV-D). Firstly, the open62541 logging mechanism is disabled. Per default, it outputs messages to the console via printf, which is not WCET analyzable. Secondly, the open62541 stack contains an implementation to encode floating-point values following the IEEE 754 standard. Although a value for the WCET can be calculated by applying the annotation rules for this function, the number of nested loops results in a theoretical WCET that is too high to be applicable in practical applications. However, the Patmos processor (as many other processors) already uses the IEEE 754 encoding. Therefore, the function can be replaced by copying the data value's in-memory representation to the message's corresponding data field. Table I summarizes how often each of the transformation rules had to be applied for the open62541 publisher.

G. Evaluation

The WCET analysis tool *platin* included in the T-CREST project can now be used to determine the WCET of the `UA_WriterGroup_publishCallback` method. The WCET bound for encoding and publishing a single data value on the Patmos processor is 18,550 processor cycles. At a clock rate of 80 MHz, this corresponds to a WCET of 231.875 µs.

To verify these values, a single-core Patmos processor is instantiated on an Altera DE2-115 development board featuring a Cyclon IV FPGA. The theoretical WCET is empirically verified using two different techniques.

Firstly, the Patmos processor provides programmatic access to its clock cycle counter via the corresponding register. Thus, the difference between the clock cycle counter before and after executing the `UA_WriterGroup_publishCallback` is utilized to determine the execution time. Secondly, to exclude possible problems with the hardware and the configuration (e.g., a mismatch of the expected clock speed), the development board is connected to a basic logic analyzer for external timing measurements. The measurement setup is depicted in Figure 5. It performs the following steps:

1) Store the clock cycle counter value (start timestamp)
2) Set a general purpose input/output (GPIO) pin to high (set GPIO pin)
3) Execute the method (UA_WriterGroup_publishCallback)

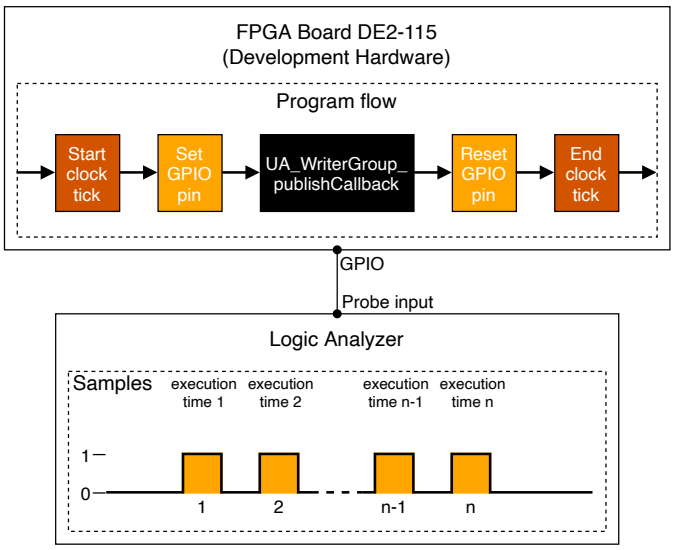

Fig. 5. Evaluation setup used to obtain execution time measurements

4) Set a GPIO pin to low (reset GPIO pin)
5) Store the clock cycle counter value (end timestamp)

A logic analyzer samples the pin at a frequency of 24 MHz. Furthermore, the difference between the end clock tick and the start clock tick is recorded to the console. This is repeated $n = 1000$ times. Figure 6 depicts the distribution of the execution times for publishing a 32 bit integer value measured via the logic analyzer and the clock cycle counter. The average measured execution times are about 138.65 µs and 139 µs, respectively. The execution times measured by the logic analyzer are about 0.35 µs lower than the values obtained via the clock cycle counter. This offset is expected and caused by the additional time required to set and reset the GPIO pin.

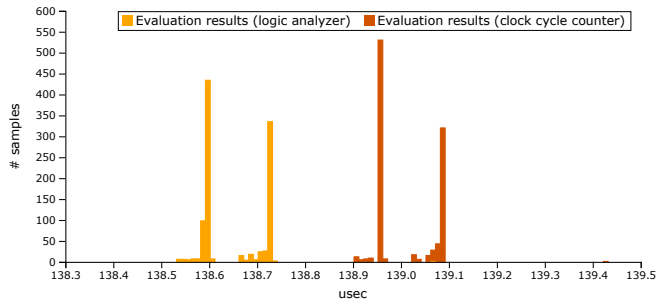

Fig. 6. Histogram of the evaluation results obtained via the logic analyzer (orange) and via the clock cycle counter (red)

The highest value measured by the logic analyzer in this test is 138.74 µs. Thus, the theoretical WCET is approximately 67 % higher than the execution time obtained by the measurements. This is well in line with the results of other studies analyzing industrial software, e.g. [7]. Nevertheless, it depends on the specific application if these values are acceptable or if additional effort needs to be spent to obtain a tighter WCET bound.

VI. Discussion

The proof of concept demonstrates the feasibility of adjusting an industrial communication protocol to enable time analysis based on the introduced structured process and the open-source T-CREST toolchain. However, one limitation is that the Patmos is a non-standard processor specifically designed to simplify the determination of WCET bounds. The timing behavior of the program on an off-the-shelf processor might differ from the obtained results. Nevertheless, the measured execution times validate the calculated WCET value and provide a starting point for non WCET specialists to continue analyzing other protocols. Based on the publicly available modified publisher [35] of the open62541 stack, other parts of the stack can be adjusted to get closer to an end-to-end real-time machine-to-machine communication.

Additionally, the proof of concept confirmed the main finding of previous research that a significant adjustment effort lies in the manual annotation [7], [28]. However, the effort also depends on how well the programmer is familiar with the code and the WCET toolchain, also mentioned in [23]. In the case of the OPC UA publisher, understanding the different contributors' programming styles and the general code complexity increased the effort significantly. Adjusting the publisher took around eight working weeks, including getting to know the T-CREST environment. The actual time spent on applying the annotations and the code transformations was only a small part. Clear guidelines for programming WCET-analyzable code would undoubtedly reduce the effort substantially, as indicated by Sehlberg et al. [26].

Specifically, for the OPC UA publisher, there is nearly no dependency on global data structures, contrarily as reported in [24] for embedded systems. The reason is that there are no global values that external program parts can change, and loops do not contain any break statements, nor are variables involved in the loop condition that are manipulated within the loop. However, as message sizes in communication protocols vary, the loop bounds depend primarily on the input data. Additionally, the open62541 stack uses jump tables to encode data fields depending on their type and special encodings for floating points values. Other authors did not report such constructs as a significant issue [7], as in the WCET community such code constructs are generally not recommended. In contrast, the publisher does not use dynamic memory allocation, quite common in other software types.

A practical outcome of the proof of concept is that it makes sense not to adjust the code permanently, instead use macros to enable and disable the WCET adjustments or take advantage of WCET tools that provide code annotations in separate files. Such an approach enables using existing programs for real-time applications while minimizing the effects on productive code. Moreover, whenever possible, the use of annotations over transformations should be preferred. There is undoubtedly a need for additional control flow restrictions (pragmas) for all programming constructs like indirect recursion and callback functions in most WCET toolchains.

VII. Conclusion

The paper presents obtained experiences when adjusting a part of an open-source industrial middleware for WCET analysis. With a simple process and the open-source T-CREST platform, it was possible to achieve representative WCET values, confirmed by actual measurements. While the annotation and transformation of the OPC UA publisher were time-consuming, the knowledge about the code and the platform became as important. The obtained results are the foundation for further studies on industrial communication software to allow time-critical communication.

References

[1] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The Worst-Case Execution-Time Problem—Overview of Methods and Survey of Tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, 5 2008.

[2] P. Puschner and A. Burns, "A Review of Worst-Case Execution-Time Analysis," *Real-Time Systems*, vol. 18, no. 2/3, pp. 115–128, 2000.

[3] N. Holsti, T. Langbacka, and S. Saarinen, "Using a worst-case execution time tool for real-time verification of the DEBIE software," *Proceedings of DASIA 2000 Conference (Data Systems in Aero- space 2000, ESA SP-457)*, vol. 457, pp. 307–312, 2000.

[4] P. Montag, S. Görzig, and P. Levi, "Challenges of Timing Verification Tools in the Automotive Domain," in *Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (isola 2006)*, 2006, pp. 227–232.

[5] M. Lv, N. Guan, Y. Zhang, Q. Deng, G. Yu, and J. Zhang, "A Survey of WCET Analysis of Real-Time Operating Systems," in *2009 International Conference on Embedded Software and Systems*, 2009, pp. 65–72.

[6] M. Schoeberl, W. Puffitsch, S. Hepp, B. Huber, and D. Prokesch, "Patmos: a time-predictable microprocessor," *Real-Time Systems*, vol. 54, no. 2, pp. 389–423, 2018.

[7] J. Gustafsson and A. Ermedahl, "Experiences from Applying WCET Analysis in Industrial Settings," in *10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'07)*, 2007, pp. 382–392.

[8] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, ser. MCC '12. New York, NY, USA: Association for Computing Machinery, 2012, pp. 13–16.

[9] W. Mahnke, S.-H. Leitner, and M. Damm, *OPC unified architecture*. Springer Science & Business Media, 2009.

[10] F. Palm, S. Grüner, J. Pfrommer, M. Graube, and L. Urbas, "open62541-der offene OPC UA-Stack," *5. Jahreskolloquium "Kommunikation in der Automation"(KommA 2014)*, 2014.

[11] P. Puschner and C. Koza, "Calculating the maximum execution time of real-time programs," *Real-Time Syst.*, vol. 1, no. 2, pp. 159–176, 1989.

[12] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper, "Automatic Derivation of Loop Bounds and Infeasible Paths for WCET Analysis Using Abstract Execution," in *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, 2006, pp. 57–66.

[13] M. de Michiel, A. Bonenfant, H. Casse, and P. Sainrat, "Static Loop Bound Analysis of C Programs Based on Flow Analysis and Abstract Interpretation," in *2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2008, pp. 161–166.

[14] B. Lisper, "Fully Automatic, Parametric Worst-Case Execution Time Analysis." *WCET*, vol. 3, pp. 77–80, 2003.

[15] S. Wilhelm, "Efficient analysis of pipeline models for WCET computation," in *5th International Workshop on Worst-Case Execution Time Analysis (WCET'05)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007.

[16] D. Prokesch, P. Puschner, and S. Hepp, "A Generator for Time-Predictable Code," in *Proceedings - 2015 IEEE 18th International Symposium on Real-Time Distributed Computing, ISORC*, 2015, pp. 27–34.

[17] B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, and E. A. Lee, "Predictable Programming on a Precision Timed Architecture," in *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. New York, USA: Association for Computing Machinery, 2008, pp. 137–146.

[18] AbsInt, "ait," Available at https://www.absint.com/ait/, 2021.

[19] Rapita Systems, "Rapitime," Available at https://www.rapitasystems.com/products/rapitime, 2021.

[20] N. Holsti, T. Långbacka, and S. Saarinen, "Worst-case execution time analysis for digital signal processors," in *2000 10th European Signal Processing Conference*. IEEE, 2000, pp. 1–4.

[21] M. Rodríguez, 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 *Proceedings of 3rd International Workshop on Worst-Case Execution Time Analysis (WCET'2003)*, 2003.

[22] 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 *Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN)*, 2003, pp. 625–632.

[23] S. Byhlin, A. Ermedahl, J. Gustafsson, and B. Lisper, "Applying static WCET analysis to automotive communication software," in *17th Euromicro Conference on Real-Time Systems (ECRTS'05)*, 2005, pp. 249–258.

[24] D. Barkah, A. Ermedahl, J. Gustafsson, B. Lisper, and C. Sandberg, "Evaluation of Automatic Flow Analysis for WCET Calculation on Industrial Real-Time System Code," in *2008 Euromicro Conference on Real-Time Systems*, 2008, pp. 331–340.

[25] B. Lisper, A. Ermedahl, D. Schreiner, J. Knoop, and P. Gliwa, "Practical experiences of applying source-level WCET flow analysis to industrial code," *International Journal on Software Tools for Technology Transfer*, vol. 15, no. 1, pp. 53–63, 2013. [Online]. Available: https://doi.org/10.1007/s10009-012-0255-9

[26] D. Sehlberg, A. Ermedahl, J. Gustafsson, B. Lisper, and S. Wiegratz, "Static WCET Analysis of Real-Time Task-Oriented Code in Vehicle Control Systems," in *Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (isola 2006)*, 2006, pp. 212–219.

[27] P. Axer, R. Ernst, H. Falk, A. Girault, D. Grund, N. Guan, B. Jonsson, P. Marwedel, J. Reineke, C. Rochange, M. Sebastian, R. V. Hanxleden, R. Wilhelm, and W. Yi, "Building Timing Predictable Embedded Systems," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 4, 03 2014.

[28] M. Platzer and P. Puschner, "A Real-Time Application with Fully Predictable Task Timing," in *Proceedings - 2020 IEEE 23rd Int. Symposium on Real-Time Distributed Computing, ISORC*, 2020, pp. 43–46.

[29] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sørensen, P. Wägemann, and S. Wegener, "TACLeBench: A benchmark collection to support worst-case execution time research," in *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, ser. OpenAccess Series in Informatics (OASIcs), M. Schoeberl, Ed., vol. 55. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016, pp. 2:1–2:10.

[30] L.-N. Pouchet, "PolyBench/C," *Available at https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/*, 2021.

[31] Mälardalen Real-Time Research Center, "WCET Benchmarks," *http://www.mrtc.mdh.se/projects/wcet/benchmarks.html*, 2021.

[32] D. Bruckner, M. Stănică, R. Blair, S. Schriegel, S. Kehrer, M. Seewald, and T. Sauter, "An Introduction to OPC UA TSN for Industrial Communication Systems," *Proceedings of the IEEE*, vol. 107, no. 6, pp. 1121–1131, 2019.

[33] M. Schoeberl, S. Abbaspour, B. Akesson, N. Audsley, R. Capasso, J. Garside, K. Goossens, S. Goossens, S. Hansen, R. Heckmann, S. Hepp, B. Huber, A. Jordan, E. Kasapaki, J. Knoop, Y. Li, D. Prokesch, W. Puffitsch, P. Puschner, A. Rocha, C. Silva, and J. Sparsø, "T-CREST: Time-predictable multi-core architecture for embedded systems," *Journal of Systems Architecture*, vol. 61, no. 9, pp. 449–471, 2015.

[34] S. Hepp, B. Huber, J. Knoop, D. Prokesch, and P. P. Puschner, "The platin tool kit - the T-CREST approach for compiler and WCET integration," in *Proceedings 18th Kolloquium Programmiersprachen und Grundlagen der Programmierung, KPS 2015, Pörtschach, Austria, October 5-7, 2015*, 2015.

[35] A. Kirchberger and M. Schoeberl, "Readme [Source code]," *https://github.com/t-crest/rt-ua*, 2021.