REFLECTING RTOS MODEL DURING WCET TIMING ANALYSIS: MSP430/FREERTOS CASE STUDY

Josef STRNADEL, Peter RAJNOHA
Brno University of Technology, Faculty of Information Technology, IT4Innovations Centre of Excellence, Bozetechova 2, 61266 Brno, Czech Republic, strnadel@fit.vutbr.cz and http://www.fit.vutbr.cz/ strnadel/index.php.en

ABSTRACT

The determination of the execution time upper bound, commonly called Worst-Case Execution Time (WCET), is a necessary step in the development and validation process for real-time systems. The WCET analysis techniques can be classified as static or dynamic. While a high-level language code suffices for the static techniques, for a precise WCET analysis a target architecture or its authentic simulator able to run the final machine-level code of an analyzed application is needed by the dynamic techniques. In the paper, we have decided not only to present a novel hybrid timing analysis technique, but also to show its practical applicability in the area of WCET analysis over particular embedded architecture (MSP430) and real-time operating system (FreeRTOS). Novelty of the presented method can be seen in the fact the operating system model is reflected during the analysis in order to facilitate the process of derivating schedulability test formulas, create detail task/stack analysis etc. Applicability of the method was tested using the MSPsim simulator of the MSP430 architecture.

Keywords: *analysis, assembly, compiler, FreeRTOS, model, MSPsim, MSP430, operating system, profiler, real time, response time, simulator, worst case, execution time*

1. INTRODUCTION

Many systems exist, which need to satisfy stringent constraints being derived from systems they control. In order to analyze and verify properties of such systems even in early system-life phases, the systems are modeled by means of appropriate modeling techniques abstracting from certain non-essential properties, but based on further properties important from time criticality point of view. The paper is dedicated to systems, whose *operational correctness* is based on both the *correctness* and *timeliness* of the outputs [1, 2, 9]. Such a system, i.e., that is able to produce the right response to given stimuli *on time*, is called a *real-time (RT) system*.

The paper is organized as follows: First, approaches to modeling RT systems are introduced (section 1.1) with special attention payed to the implementation of RT systems by means of tasks running over the *real-time operating system* (RTOS) kernel. At the end of the section, importance of the schedulability tests (section 1.2) and execution time analysis needs are emphasized (section 1.3). In the section 2 common principles and problems related to *timing analysis* (TA) techniques are summarized followed by the motivation and goals of our research 2.3. Lacks of the actual methods can be seen in the fact they do not reflect target RTOS during the analysis. As a result, they make the derivation of mechanisms such as schedulability test more difficult and hard to automate. In the paper, the method reflecting the fact is presented and at the end of the paper it is shown how the results can be utilized to derive the test for given scheduling mechanism and target resources (RTOS, platform and its simulator enriched about RTOS model). In section 3 a principle of the proposed TA method is described. In section 4, particular resources utilized for implementation and experimental verification of the proposed method are summarized. Experimental results w.r.t. the method are summarized in section 5. Section 6 concludes the paper.

1.1. Real-Time System Model Basics

Tradidionally, an RT system is modeled as a set of RT tasks, each having assigned a set of parameters abstracting of an implementation of tasks and containing constraints posed on task execution.

Definition 1.1. *Let* $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ *denote the set of RT tasks (representing partial functionality of the RT system) with particular paramaters and* $\Phi = \{\phi_1, \phi_2, \ldots, \phi_m\}$ *the set of computational resources the task can be executed on.*

Each task is typically asociated with an event. If an event occurs, the task is released in order to react to the event. Because events can be of various priorities and can have various constraints assigned, there must be an arbiter called *a scheduler* making a decision about which tasks will particular system resources be assigned to and in which order. The result of the decisions is called *a schedule*.

Definition 1.2. *Given* Γ *and* Φ*, a schedule is defined as a distribution of executions of tasks from* Γ *among computational resources from* Φ *followed by ordering the executions in time. During the distribution, many criteria can be taken into account, e.g., priorities.*

The scheduler decisions are based on the actual *state* of the RT system, *events* stimulating the RT system, and *parameters* of tasks in the RT system. RT task parameters are given by a given task model and can be divided into the two groups: *primary* and *secondary* [2]. As the primary parameters do not change during the task run time they are also called *static*. List of typical primary parameters follows (rather than described in detail, some are ilustrated in Fig. 1a):

- release (arrival, ready) time of the task, r_i
- worst-case exec. time (WCET) of the task, C
- relative deadline of the task, D
- absolute deadline of the task, $d_i = r_i + D$
- relative laxity time, $L = D - C$

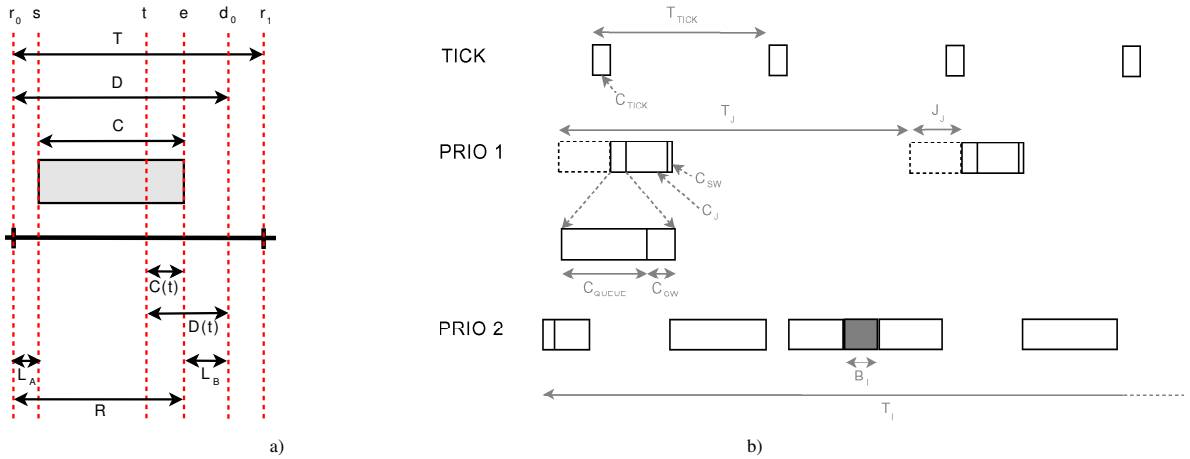


Fig. 1 Illustration to a) RT task parameters and b) extended version of a formula for R_i enumeration

- task invocation period, T; defined for periodic tasks

Contrary to the primary parameters, secondary parameters can change at the run time or their values are known even in the run time in order to reflect behavior of the corresponding tasks, they are often called *dynamic*. Some of the typical parameters are (for their illustration, see Fig. 1a):

- task execution start-time, s
- task execution end-time, e
- response time of the task, $R = e - r$
- time to miss the deadline of the task, measured in time t, $D(t) = d - t$
- remaining execution time of the task, measured in time t, $C(t)$
- actual laxity time of the task, measured in time t, $L(t) = D(t) - C(t)$
- actual CPU load factor of the task, measured in time t, $CH(t) = \frac{C(t)}{D(t)}$

Definition 1.3. *Given* Γ *and* Φ*, a schedule is called feasible* $\Leftrightarrow$ *all the tasks from* Γ *are executed on resources from* Φ *before the tasks miss their deadlines. I.e., in each* t *in the schedule, following condition must be fulfilled for a task* $\tau \in \Gamma$ *running on a resource* $\phi \in \Phi$*:* $C_{\tau,\phi}(t) \leq D_{\tau,\phi}(t)$.

Definition 1.4. Γ *is called schedulable on* $\Phi \Leftrightarrow$ *a feasible schedule exists for* Γ, Φ.

Definition 1.5. *The scheduling problem is defined as the problem of finding a feasible schedule for a given* Γ *and* Φ.

Probably the most important parameter within the RT task model is d, i.e., the time in which the task execution must be completed to prevent from an unpredictable behavior of the system having more or less consequences to the environment. Thus, the scheduler is required to make its decisions with a respect to the parameter. The decisions have to be made in $O(n)$ or $O(c)$ time in the ideal case hereat the *scheduling problem* is NP-complete in general[1].

Definition 1.6. *Let a scheduling mechanism is denoted by* ξ*. An algorithm that is able to say* Γ *is schedulable on* Φ *by means of a given* ξ *is called a schedulability test.*

1.2. Schedulability Tests

The most precise schedulability tests[2] are based on evaluating R_i, i.e., response time, for each $\tau_i \in \Gamma$. To be schedulable on Φ, for each $\tau_i \in \Gamma$ it must hold: $R_i \leq D_i$, i.e., each τ_i must be able to meet its deadline when scheduled by means of given ξ. The big disadvantage of response time analysis based tests is the fact ξ have to be theoretically analyzed in detail together with target platform architecture and operating system before corresponding formulas can be derived. If the derivation is too difficult or imposible, sufficient or necessary conditions can be deduced. But, they do not guarantee Γ is schedulable iff the conditions are met. As an example let sufficient, but not necessary condition for testing schedulability of RM mechanism be presented [10]:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{\frac{1}{n}} - 1)$$

For the condition and RM mechanism (that is optimal over the class of RT task sets fullfilling condition $\forall \tau_i \in \Gamma : D_i = T_i$), Γs can be found which do not met the condition, but are schedulable. Response-time based schedulability tests for RM mechanism can be derived from following recurrent relation [8]:

$$R_i^{k+1} = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i^k}{T_j} \right\rceil C_j \qquad (1)$$

[1] Actually, many *scheduling mechanisms* exists [1, 2, 4, 8, 9], which are able to schedule Γs under certain constraints – e.g.: static priority assignment based *rate monotonic* (RM), *deadline monotonic* (DM) or dynamic priority assignment based *earliest deadline first* (EDF), *least laxity first* (LLF) – which are able to guarantee *schedulability* only for Γs composed of RT tasks with strictly limited parameters ($D \leq T$, Γ's CPU utilization $\leq |\Phi|$ etc.)

[2] i.e., those representing both *sufficient* and *necessary* condition

- where R_i^k is a value of R_i in k^{th} iteration of the relation and $hp(i)$ is a set of tasks with priorities higher than priority of task i. In the initial iteration ($k = 0$), R_i^k is set to C_i. Afterwards, R_i^{k+1} is enumerated until it hols $R_i^{k+1} = R_i^k$ (i.e., the task will met its deadline) or until $R_i^{k+1} > D_i$ (i.e., the task will miss its deadline, so Γ the task belongs to is not schedulable on Φ by means of RM) [18].

Meaning of the formula is as follows: worst-case response time (R_i) of a task (τ_i) is given by task's worst-case execution time (C_i) and and by sum of the worst-case execution times of all tasks (τ_j) which will be executed during the run time of τ_i. Low-priority task τ_i will be preempted by such high-priority tasks and its response time will be delayed about time needed for execution of the higher-priority tasks.

In order to get more precise analysis of response times, the basic formula should be extended about further elements descreasing abstraction level of the formula[3]. The extended formula could look like:

$$R_i^{k+1} = C_i + 2C_{sw} + B_i + \left\lceil \frac{R_i^k}{T_{tick}} \right\rceil C_{tick} + \\ + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i^k + J_j + T_{tick}}{T_j} \right\rceil (C_j + 2C_{sw}) \\ + \sum_{\forall j \in alltasks} \left\lceil \frac{R_i^k + J_j + T_{tick}}{T_j} \right\rceil C_{queue} \quad (2)$$

For detail description of the parameters used to extend the formula see, e.g., [4, 8].

1.3. Reasons for Execution Time Analysis

Although principles of scheduling mechanisms and related schedulabily analysis are different, they all try to schedule tasks with near d before tasks with far d in order to guarantee no task deadline will be missed. It is evident that for parameters of each $\tau \in \Gamma$ to be run in uniprocessor environment, it must hold $C \leq D$, i.e., deadline must be set up in such a way it cannot be exceeded during the task execution. On top of it, for periodical tasks it must also hold $D \leq T$, i.e., period must be set up in such a way that a new instance of a task cannot be called before the deadline for previous instance of the task is over; the whole condition can be written as $C \leq D \leq T$. From the above mentioned, the following can be concluded:

> Before the crucial RT task parameters such as D and T can be set up, the C (i.e., WCET) value must be precised for each of the RT tasks.

The shortest execution time is called the *best-case execution time* (BCET) and the longest time is called the *worst-case execution time* (WCET). In most cases the state space is too large to exhaustively explore all possible executions and thereby determine the exact BCETs and WCETs. So, it is not possible to obtain bounds on execution times for all programs in general. Otherwise, one could solve the *halting problem*. However, real-time systems only use a restricted form of programming, which guarantees that programs always terminate; recursion is not allowed or is explicitly bounded. A reliable guarantee based on the worst-case execution time of a task could easily be given if the worst-case input for the task were known [20].

2. TIMING ANALYSIS

Today, in most parts of industry a common method utilized to estimate execution-time bounds is to measure the end-to-end execution time of the task for a subset of the possible executions (*test cases*). This determines the *minimal observed* and *maximal observed* execution times. In general, they overestimate the BCET and underestimate the WCET, so they are not safe for hard RT systems. This method is often called a *dynamic timing analysis*. Newer *measurement-based* approaches make more detailed measurements of execution times of several task portions first to combine them later to estimate BCET/WCET values with a smaller error.

Surely, the bounds can be precisely computed only by methods that consider all possible task executions. Those methods use a task abstraction to make the TA process feasible. But, the abstraction loses information, so the computed WCET bound usually overestimates the exact WCET and vice versa for the BCET. The criteria for evaluating TA methods are [20]:

- *safety* – does it produce bounds or estimates?
- *precision* – are the bounds or estimates close to the exact values?

Before basics of TA techniques will be presented, it should be emphasized that TA results could be precise only if the analysis is done over *particular data/control paths* allowed during execution of *particular machine code* of the task generated for *particular hardware*. In relation to that, many problems must be dealt with – some of the most important are mentioned below:

- **Problems related to data dependency in a program flow:** A task to be analyzed attains its WCET on one or more of its valid execution paths. If the input and the initial state leading to the worst-case execution path were known, the problem would be easy to solve. But, the worst-case input and initial state are not known in general because they are hard to determine. A superset of the set of all task execution paths is usually described by means of *control-flow graph*, CFG [19]. However, CFG can contain input-data dependent paths that would never be executed because of input-data limitations or existence of contradictory conditions. The other problem can appear if bounds

[3] Deviation of analyzed response times from real response times can be minimized if the formula is extended, e.g., about following elements: B_i (*blocking time* of τ_i, i.e., maximal time for which τ_i can be in not-ready state), J_i (*jitter time* of τ_i, i.e., maximal difference between two release times of τ_i), C_{sw} (*context-switch time*), T_{tick} (*time resolution of a scheduler tick*), C_{tick} (*time needed to service a scheduller tick*) or C_{queue} (*time needed to change state of a task to ready*)

on the iterations of the loops or on the depth of recursive functions are unknown.

- **Problems related to the impact implying from modern architecture principles:** Even the above-mentioned problems are solved, accuracy of the results is questionable if tasks are to be executed on a platform equipped with pipelines, branch predictors, cache memories, speculative execution units etc. Those modern components can bring into being several problems such as execution instructions in a different order (due to prefetching, loop optimizations, speculative executions etc.) or context-dependency of the executions (e.g., due to various cache or pipeline contents). On top of this, it is intuitively supposed the latter executions of the same task would always lead to the same or shorter execution time, e.g., due to cache hits. But, it was observed that it can also happen the latter executions may, in fact, lead to a longer execution time, e.g., because of mispredictions during the speculative execution. This phenomenon is called a *timing anomaly* [11]. In the mentioned case, it is necessary to invalidate all results achieved during execution of wrong-directed instructions, which leads to clean-up of pipeline, cache etc. an extended execution of right-directed instructions. The important conclusion can be made at this point: the assumption that the global WCET of the task equals to sum of the local WCETs is incorrect in general, but it can be correct for the particular platform.

2.1. Static Timing analysis Techniques

This class of methods does not rely on executing a task code on a real hardware or on a simulator, but rather it takes the code itself, combines it with some more or less abstract model of the system with the goal to obtain upper bounds based in the combination [7,12]. Typically, the methods are composed of the following steps:

- *control-flow graph* (CFG) creation [19]: during the step (also called a *frontend*), the task source code is transformed into the CFG describing a relation between vertices called *basic blocks*, each of them representing maximal sequence of non-loop/non-branch instructions. The relation is expressed by means of edges representing loops or branches in the code. Both the vertices and the edges can be evaluated by an extra information such as maximal number of loop iterations, value boundaries etc.

- *control-flow analysis* (CFA) [20]: in the step, CFG is analyzed in order to remove subgraphs, which do not represent valid execution path of the code. (The effort is to eliminate so much invalid paths as possible, because each such a path can potentially contribute to underestimation of BCET or to overestimation of WCET,

- *low-level analysis* [20]: in the step, both compiler and platform attributes are analyzed (compiler options, platform pipelines, memory access mechanisms, branch predictors, caches etc.); the platform behavior is typically aproximated rather than modeled precisely,

- *evaluation of bounds*: on basis of *facts* about a code flow and a platform model, one of the following methods is typically utilized to evaluate the bounds [3,15,16]: *syntax-tree based*, *path-based* or *implicit path enumeration*

- *visualization and statistics*.

Detail description of the above-mentioned steps and related methods is out of scope of the paper. Instead of, dynamic TA techniques are briefly described in the next.

2.2. Dynamic Timing analysis Techniques

These methods attack some parts of the TA problem by executing a task on a given hardware/simulator for a set of inputs to measure the execution time of the task [20,21]. It should be noted that if the subset do not contain the worst case then the end-to-end measurements of a subset of all possible executions are able to produce estimates/distributions, but not exact execution time bounds. On contrary, even one execution would be enough if the worst-case input were known.

Other approaches measure the execution times of code segments, typically of CFG basic blocks [19, 20]. The measured execution times are then combined and analyzed, usually by some form of bound calculation, to produce WCET/BCET estimates. Thus, measurement replaces the processor-behaviour analysis common for the static methods. This solution would include all possible paths, but would still produce unsafe results if the measured basic block times were unsafe or if only a subset of input states (contexts) is considered. Dynamic techniques can measure execution-time bounds for processors with simple timing behaviour and and collect and analyze multiple measurements to provide a picture of the variability of execution times. There are multiple ways in which measurement can be performed [20]. The simplest approach is by extra instrumentation code that collects a timestamp or CPU cycle counter (available in most processors). Mixed HW/SW instrumentation techniques require external hardware to collect timings of lightweight instrumentation code. Fully transparent (non-intrusive) measurement mechanisms are possible using logic analyzers. Also hardware tracing mechanisms like the NEXUS standard, ETM tracing mechanism in ARM or BDM interface in Freescale products are non-intrusive, but don't necessarily produce exact timings. Measurements can also be performed from the output of processor simulators or even VHDL simulators. Let the following dynamic techniques be described in detail [14]:

- *code tracing*: special trace instructions are put to the code to capture the state of a system in the context of the actual code flow. Big advantage of the approach can be seen in the simplicity: trace functions can be put to the code being analyzed; so, no

extra requirements are posed on the target simulator/platform (for ilustration, see Fig. 2a with tracing instructions `INSTR 02_A` to `INSTR 02_D`). The analysis can be done over the simulated platform (if a simulator is available) or over the real platform. But, the main disadvantage of the approach implies from the fact each trace instruction becomes a part of the code and thus, it surely impacts the execution.

- *code simulation*: a simulator with integrated tracing or with a support of trace points is required to utilize the technique (see Fig. 2b). Main advantage of the technique can be seen in a full control over the dynamic analysis – e.g., simulation can be stopped for certain amount of time in order to change parameters of the simulation, trace program flow, read from or write to memory places (e.g., to change a context), let the simulator perform defined actions (the run time value or context is stored into a file etc.) if certain conditions are met (e.g., if an address is accessed) etc. before the simulation continues. Rather than the real run time, the logical run time (evaluated by the simulator according to instruction flow, peripheral access times, interrupt related latencies etc.) is taken as a basis for all measurements. The big disadvantage of the technique can be seen in the fact it is practically applicable only to systems, which can be simulated in an authentic way. During experiments related to the paper, such a simulator was available, so it was utilized for the purpose.

2.3. Our Research: Motivation and Goals

On basis of the above-mentioned information about actual TA techniques, especially following can be concluded:

- TA technique cannot be independent of the target platform,
- static TA techniques are suitable to preprocess the high-level source-code, to detect basic programming constructs in the code, to analyze the program flow and to produce bound estimates,
- dynamic TA techniques are suitable for measurement of bounds in the run/simulation time over the target platform, so the anomalies can be eliminated and both safe and precise bounds can be produced,
- existing TA techniques abstract from an RTOS model, so their results are not directly applicable for analysis of high-level RTOS parts such as queue management, interrupt system, resources utilization, context-switch overhead, stack utilization etc. However, the creation of the schedulability test formulas depends just on the high-level analysis – if the RTOS model is reflected, the creation can be automated. Otherwise, the formulas must be created manually.

Because existing TA techniques dealing with analysis of RT systems have focused to enumeration of task's *C* parameter only, we have decided to design a method, which can be utilized also for detail analysis of an RTOS tasks are supposed to run on. Such kind of analysis is needed especially if schedulability tests are to be derived for given scheduling mechanisms and target platforms. Because the kind of analysis is not present in any of actual solutions, we have decided to present that the method can be constructed if implementational details related to particular RTOS are involved in the analysis. To take advantages of existing static and dynamic approaches, it was decided proposed TA method will be hybrid and will be composed of a static part used for code preprocessing and of a dynamic part used to measure bounds. For the purpose of dynamic analysis, code simulation method was selected because it offers the most information about the analyzed system and full control over the analysis.

3. PROPOSED METHOD

In the section, principle of proposed TA method built over RTOS model extension of a target architecture simulator will be described.

3.1. Inputs and Outputs

There are several inputs to the method:

- *source-codes* usually written in a high-level language (e.g., C) of an application to be analyzed. The code is necessary especially for an automated detection of basic programming constructs (if-then-else, switch-case, for, while etc.)
- *annotations*, i.e., information about subjects of analysis and about types of analysis to be performed over the subjects, variable intervals, loop bounds etc. The information can be automatically extracted from the source-codes or adjusted manually by a user.
- *executable binary file*, i.e., a machine-level code that is supposed to be run on a target platform; typically, the code is completed with extra debugging information in order to offer more information for further analysis purposes. On a basis of the code, behavior of the system can be simulated together with the TA done according to informations extracted from source codes and annotations.

After analysis is completed, requested outputs (related to execution statistics related to observed pieces of code) are produced by the `RangeProfiler` module (able to perform TA of a code stored within particular memory space) and the `MultiStackMonitor` module (able to perform analysis related to stack utilization during run of the application) – see section 4.4, page 26.

3.2. Principle

The below-mentioned steps desctibe principle of the method in an illustrative way rather than presenting the method in an exhaustive form. Overall complexity of the method can be easily derived from the description of the steps.

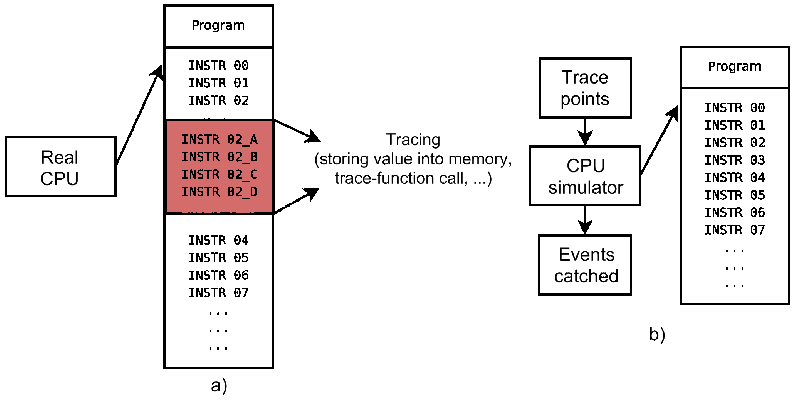
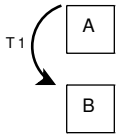


Fig. 2 Typical approaches to dynamic timing analysis: a) code tracing, b) code simulation

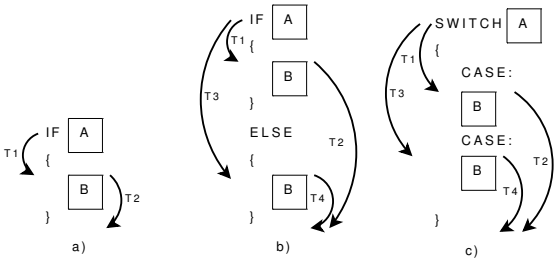
Step 1 – Constructs detection: Before TA can start, *basic programming construct* (e.g., `if-then-else`, `switch-case`, `for`, `while`, function calls in C language) must be detected together with *basic blocks* (the longest continual sequences of instructions with no branches and cycles included). The detection can be done, e.g., on basis of a syntax-tree analysis.

Step 2 – Range profiling: After the constructs are detected, TA can be performed over the constructs. In the next, principle of the analysis is outlined by means of typical representatives of the constructs (let it be mentioned that the range profiling of the constructs is done in a recursive manner, so the following list of steps is to be get unordered rather than ordered – the steps are performed if it is applicable during the recursion process):

2A – Basic blocks: A range profile is created for each basic block in a following way: *Start of the profile* is specified by an address of the first instruction within the block (profiling starts if PC is loaded with the address) and *end of the profile* is specified by an address following after the address of the last instruction in the block (profiling ends if PC is loaded with the address, i.e., the instruction placed on the address is not involved in the analysis process). Below, the situation is ilustrated so: `A` is a basic block being profiled, `B` is a basic block following `A`, `T1` is `A`'s execution-time being measured.



2B – `if` and `switch` constructs: Many variants of the constructs exists, so creation of profiles will be illustrated only over elementary `if` variant in order to show basic principle. For `if` variant of the construct, at least two range profiles are created (supposing both the condition and conditioned block are formed by a basic blocks): the first one (used to measure execution time `T1` of the condition itself) starts at an address the conditional command starts (profiling starts if PC is loaded with the address) and ends at an address the conditioned block starts (profiling ends if PC is loaded with the address). The second one (used to measure execution time `T2` of the conditioned block) starts at an address the conditioned block starts and ends at an address following after the address of the last instruction in the block. If the condition or the conditioned block are not formed of basic blocks, corresponding profile (used for `T1` or `T2` evaluation) must be created by recurrent application of rules forming step 2 of the method. Alike, profiles for more complex `if` constructs (e.g., `if-then-else`) or `switch` constructs can be created – for illustration, see the below-mentioned figure.

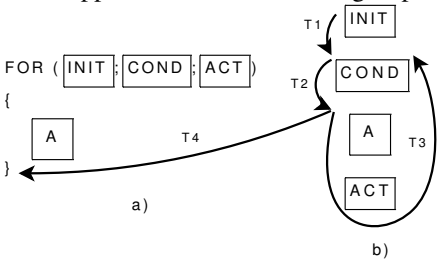


2C – `for` construct: The construct consists of an *initial* part, *condition* part, *action* part and of a *block to be executed* during each iteration of the cycle. Because the contruct is more complex than the yet mentioned, let a creation of range profiles (at least 4 profiles are needed) for a `for` cycle be illustrated by means of an example of a particular cycle implemented in `vListInsert` function within FreeRTOS interface [6]: The initial part (starting at `54be` address) is executed just once, before the condition is enumerated first time. The condition (enumerated by `cmp` instruction) starts at address `54cc` and is followed by a jump to the conditioned block (`instruction jmp` at `54d2` address) starting at `54d4`, followed by an action (if the condition was met) or to the end of cycle (i.e., after the address following the last instruction of the cycle) - i.e., to `54e0` (if the condition was not met). `Jmp` placed at `54de` makes a jump to next iteration of

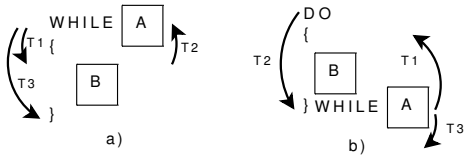
the cycle. All instructions etc. are related to MSP430 family of microcontrollers [17].

```
;address |              instruction
;(hex)   | opcode  name    operands    operand addr.
;-------------------------------------------------------
;init
54be: 94 44 02 00  mov     2(r4), 4(r4)   ;0x0002(r4),
                                          ;0x0004(r4)
54c2: 04 00
54c4: 1f 44 04 00  mov     4(r4), r15     ;0x0004(r4)
54c8: 1f 4f 02 00  mov     2(r15), r15    ;0x0002(r15)
;cond
54cc: a4 9f 06 00  cmp     @r15, 6(r4)    ;0x0006(r4)
54d0: 01 24        jz      $+4            ;abs 0x54d4
54d2: 06 3c        jmp     $+14           ;abs 0x54e0
;conditioned block (body)
;act
54d4: 1f 44 04 00  mov     4(r4), r15     ;0x0004(r4)
54d8: 94 4f 02 00  mov     2(r15), 4(r4)  ;0x0002(r15),
                                          ;0x0004(r4)
54dc: 04 00
54de: f2 3f        jmp     $-26           ;abs 0x54c4
54e0: ...
```

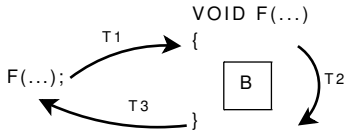
Following the above-mentioned example illustrating structure of particular `for` cycle, range profile related to `for` cycle can be intuitively constructed as depicted in the following figure (in the figure, both C-syntax representation of the cycle (a) and compiler-generated code of the cycle (b) are presented): `T1` represents execution time needed for initialization (involved only once), `T2` time for condition enumeration (involved before new iteration starts, i.e., enumerated at least once), `T3` time needed for conditioned block and action execution (involved each times condition is met). `T4` time needed for `for` exit if the condition is not met (involved after the last iteration is executed, i.e., at least once). The final execution-time of `for` construct equals to $\texttt{T1}+\texttt{T2}+k\times(\texttt{T3}+\texttt{T2})+\texttt{T4}$, where k is an upper bound of cycle iterations. Problem with enumerating k can arise, e.g., if k depends on a value to be received in the run time from external sources or if the loop is infinite. The first problem can be solved by setting k to the upper bound of an iteration-variable data type (which can result to overestimation of WCET and to problems related to selection of proper target platform and scheduling mechanisms), while the second problem can solved by setting k to a special value (e.g., -1) used to indicate infinity of the loop. Alike in case of other types of constructs, if non-basic blocks are involved in the `for`-cycle declaration, range profiles must be created by recurrent application of rules forming step 2 of the method.



2D – `while` constructs: Range profiles for `while` constructs are created in similar way as those for `for` constructs, with dame problems related to the profiles. Following times are avaluated by (at least 3) profiles: `T1` condition enumeration time, `T2` conditioned block execution time, `T3` time needed to exit the loop. The final execution-time of `while` construct equals to $\texttt{T1}+k\times\texttt{T2}+\texttt{T3}$, where k is an upper bound of cycle iterations. Alike in case of other types of constructs, the number of profiles can increase if non-basic blocks are involved in the cycle declaration. In such a case, range profiles must be created by recurrent application of rules forming step 2 of the method.



2E – function calls: As the last example, principles related to creation of range profiles for functions and their calls will be presented. Each function typically consists of 3 parts: a *prologue*, a *body* and an *epilogue*. Prologue and epilogue parts reflect the fact that certain overhead is needed if a function is called (context storage during function call, argument-passing overhead, local-stack preparation overhead, jump to function body etc.) and after the function is finished (stack-cleanup overhead, return-value storage, context recovery etc.). Thus, execution-time related to function call and execution of its body is composed of at least following 3 times: `T1`, (`T2`, `T3`), i.e., time needed for prologue (function body, epilogue) execution. The final execution-time related to fuction call and execution equals to $\texttt{T1}+\texttt{T2}+\texttt{T3}$. But, alike in case of other types of constructs, the number of profiles can increase if non-basic blocks are involved in the prologue, epilogue or prologue of the function. In such a case, range profiles must be created by recurrent application of rules forming step 2 of the method.



The above-mentioned rules for creation of range profiles can be applied to any part of a code. As it can be induced from the examples, times being enumerated for one branch of the code are summarized in order to get execution length of the branch. WCET of branched code equals to execution time that is maximal over the set of code branches. In Fig. 3, CFG related to `vListInsert` function implemented in FreeRTOS is presented, completed with execution times analyzed for particular constructs used in the function. In the CFG, following symbols are utilized: `START` (CFG start), `BB` (basic block), `COND` (enumeration of a condition statement), `FOR INIT` (`for` initialization), `FOR COND`, `WHILE COND`, `DOWHILE COND` (enumeration of a condition for cycle constructs) `END` (CFG end). On the basis of the CFG and evalution of its nodes by means of execution times, the longest path (representing WCET of the function) through the CFG can be found. The path is represented by edges drawn by continuous lines while shorter paths are represented by dash lines.

4. CASE STUDY

In order to demonstrate practical applicability of proposed method, we have decided to realize a set of experiments over the real resources being utilized in practice: particular RTOS (FreeRTOS [6]) running on particular microcontroller (MSP430F168 [17]). In the next, basic information about the resources is presented. Although the experiments are related only to MSP430F168, the method is genarally applicable to various implementations based on MSP430 family members. Also, the method is not limited only to FreeRTOS – if the profiler is completed with a model related to a different RTOS, the results of the method can be utilized to derive schedulability tests for the RTOS.

4.1. FreeRTOS

FreeRTOS [6] is simple, open-source RTOS designed for implementation of embedded applications. It is written mainly in C, with minimum low-level code written in assembly. Its kernel has modular structure and is configurable by means of simple plain-text configuration file named `FreeRTOSConfig.h`. In the file, parameters (preemption on/off, system tick frequency, number of priority levels, minimal stack size for tasks etc.) of the kernel required for target application can be setup in order to ensure required performance and availability of required OS services and to prevent from wasting of computational resources by functionalities unused in the application. FreeRTOS can be easily ported to many target platforms and its ports can be utilized for implementation of commercial (*OpenRTOS*) and safety-critical applications (*SafeRTOS*). Detail description of FreeRTOS is out of scope this paper and can be found in [6] – let only following information related to FreeRTOS be presented:

- Following schedulable entities are supported: *tasks* and *co-routines*. The application can be designed using tasks, co-routines, or a mixture of both. Tasks and co-routines differ maily in a way they work with their context: while a task executes within its own context with no coincidental dependency on other tasks within the system, co-routines share a single stack, so there is no context-switch overhead related to co-routines.

- Multiple tasks can exists with the same priority assigned. Tasks of the same priority are organized in a double-linked list. Co-routines are organized into a double-linked list common to all co-routines.

- FreeRTOS scheduler is able to schedule tasks and/or co-routines in a preemptive or non-preemptive way. But, it holds that CPU is always assigned to a task/co-routine that has the highest priority among all ready tasks and co-routines. If preemptive mechanism is turned on, entities stored in the same ready list are scheduled using a *round-robin* manner.

4.2. Target Platform: MSP430F168

Microcontroller MSP430F168 is a product of *Texas Instruments* company [17] and is designed for construction of extremely low-power applications. MSP430 is a low-endian architecture and involves 16-bit RISC CPU, 16 registers, 2 KB RAM and 48 KB FLASH memory and many peripherals typical for embedded applications (1 watchdog, 2 timers, analog comparator, DMA controller, 8-channel 12-bit ADC and DAC modules, 2 USART modules, JTAG interface 6 general-purpose 8-bit input/output ports etc.). On top of the common modules, there is one 16x16-bit hardware multiplier implemented in MSP430, which operates in parallel with the CPU.

Because instruction set of MSP430 consists of 27 fixed-execution time instructions, there are no advanced components – e.g., cache, branch predictors – present in MSP430's architecture and an authentic MSP430 simulator with integrated tracing and profiling support is available, MSP430 was evaluated as a suitable platform for testing our method.

4.3. Simulator: MSPsim

MSPsim [13] is an open-source instruction-set level MSP430 simulator written in Java. As an input, binary file in *executable and linkable format* (ELF) – produced, e.g., by a C compiler – is taken. MSPsim can be easily extended to simulate various peripherals, so it can be utilized to simulate behavior of the whole platform based on MSP430. In MSPsim distribution package, following modules are simulated: CPU, basic clock module, timers, USART, GPIOs, hardware multiplier, AD modules, watchdog. The most important part of the simulator is a *simulator core* implemented in `MSP430Core` class. There are several goals related to the core: it makes a connection among basic components within MSP430 architecture (memory, modules, peripherals etc.), it is used to interpret instructions and to computing logical time. Simulation can be stopped, performed in single-step or continuous mode. Speed of the simulation can be set up by means of speed-up factor.

There are many other functionalities offered by original MSPsim interface (e.g., `CPUMonitor`, `Profiler`, `SimEventListener`). From TA point of view, especially following is implemented: instruments for reading symbols and debugging information from ELF, access to CPU registers, access to data stored in memory cells, *watchpoints* used for detect an access to a particular symbol, address or register, *stack trace* used to access the stack, *profiler* used to work with profiling data, *breakpoints* used for debugging or instruments used to get MSP430's state after an instruction is simulated.

However, for the purpose of RTOS timing analysis, some important functionality is missing in the original MSPsim[4], e.g., support for multitask systems and for modeling particular operating systems. Missing the functionalities makes analysis of certain system properties (time-analysis related to task context switches, stacks, interrupt-

[4] of course, it is because the simulator was developped for different purpose – for simulation of embedded sensor network ESB/Sky devices based on MSP430 running no operating system

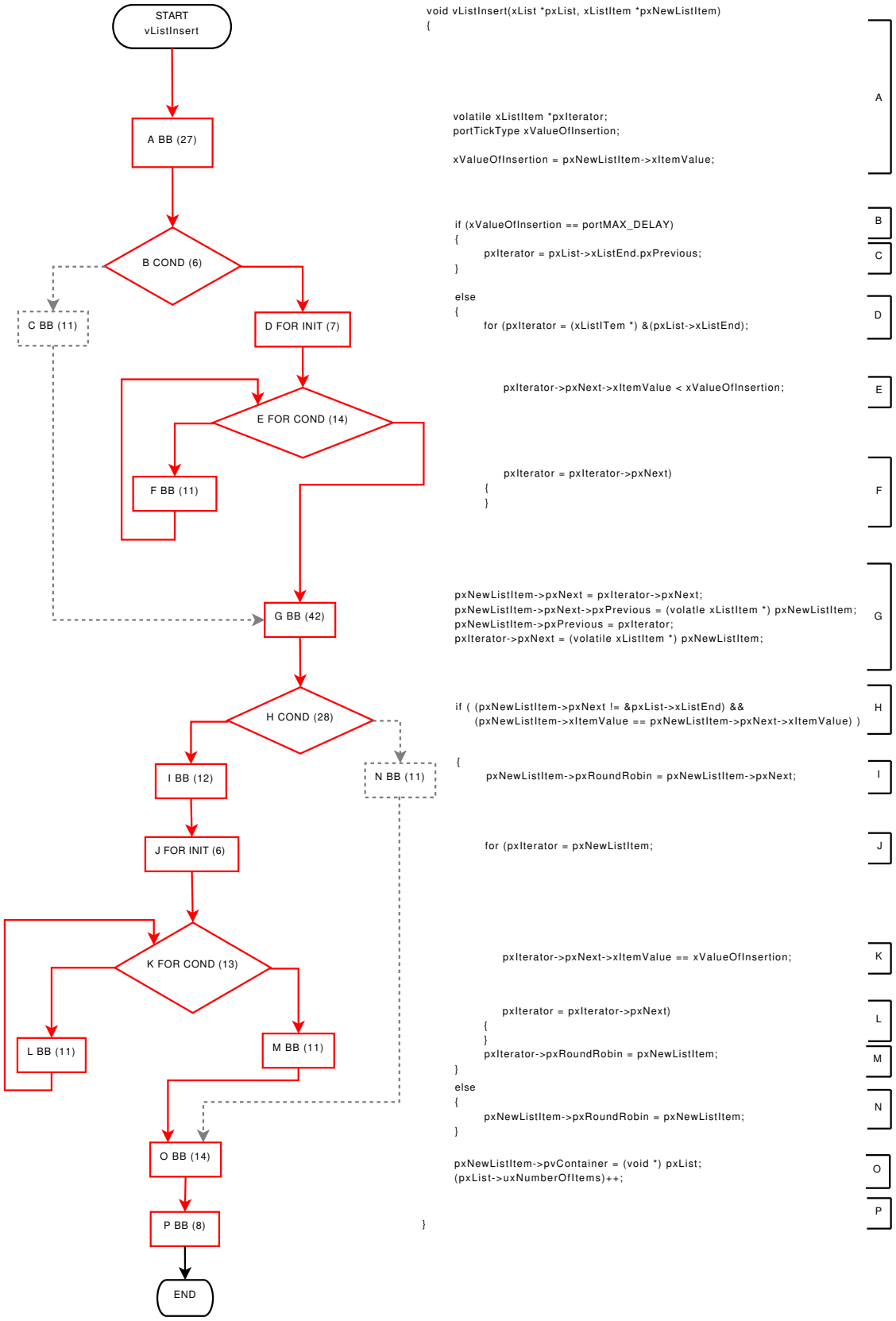


Fig. 3 CFG of `vListInsert` function

routine services, queue management, shared resources etc.) difficult. Thus, it was necessary to extend MSPsim first in order to prepare it to experimental part of the work. On top of it, the simulator was extended about necessary model of FITkit platform [5] – equipped with MSP430 MCU, Xilinx Spartan3 FPGA, 8x8Mbit DRAM etc. – selected for final implementation. In order to be correct, in Fig. 4 it is

ISSN 1335-8243 (print) © 2012 FEI TUKE
www.aei.tuke.sk

ISSN 1338-3957 (online)
www.versita.com/aei

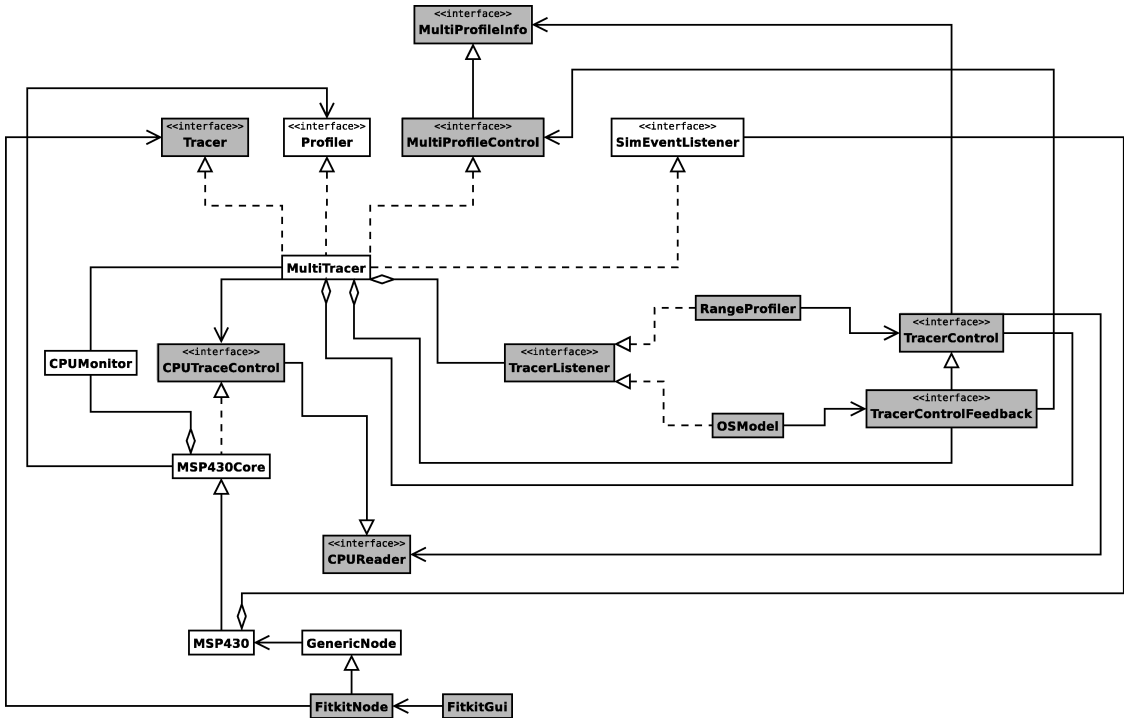


Fig. 4 MSPsim extension – original (white boxes) and added parts (grey boxes)

depicted which parts within extended MSPsim architecture are original and which ones were added to the architecture for our purpose. Thus, it can be said MSPsim was significantly extended to be utilized for the TA of RTOSes.

Detail description of the extensions is out of scope of the paper, but let at least the following be mentioned: `MultiProfileControl` and `MultiProfile` classes implement profiling functionalities related to task creation, deletion, actual state, context-switch etc. – i.e., to various function calls in multitask environment. `Tracer`, `TracerControl` and `TracerControlFeedback` classes extend the function-call related functionalities about those related to generation, redirection and capture of events (associated with PC value, access to memory places, enabling/disabling interrupts, starting/exiting interrupt service routines etc.) and their time-stamps. `Tracer`, `Profiler` and `MultiProfiler` interfaces are implemented by `MultiTracer` class, which also collect information about catched events and distribute the information to corresponding `TraceListener` objects.

4.4. RTOS model over MSPsim model

In our case, the information can be send to an `OSModel` object – implementing model of particular RTOS consisting of a task model and a scheduling-policy model; `OSCommand` is used to implement a reaction to an event – and/or to a `RangeProfiler` object implementing module for collecting statistics about previously specified piece of a code[5]. For the code, especially following information is collected automatically by the object: a) how many times the code was executed, b) last (L), average (A) and the worst (W) execution time of the code including all latencies like context switches etc., c) same as b, but with latencies excluded, d) L/A/W number of interrupts occured during execution of the code etc.

4.5. Stack Monitoring in Multitask Environment

Besides TA, we have decided to enrich the MSPsim about dynamic monitoring of local stacks belonging to particular tasks running in multitask environment. This functionality is implemented in `MultiStackMonitor` – for each task within the system, following information is collected: a) stack start address, b) stack size, c) actual and d) actual, minimal and maximal stack capacity occupied during execution of the task. The information can contribute to more precise analysis of memory requirements leading to safe-stack design preventing stack to overflow/underflow in the run time.

5. RESULTS SUMMARY

In the section, it will be shown how results produced by proposed TA method can be utilized.

5.1. WCET Results

There are many functions implemented in FreeRTOS interface, but presentation of results related to all of them would occupy a lot of space of the paper. So, we have decided to present only subset of the results. Because the most frequent operations of an RTOS are those related to

[5]The piece of code is specified by means of addresses specifying beginning and end of the piece of code. The addresses can be got, e.g., from dissassembled executable or by means of `lineaddr` command of an extended MSPsim run over the source codes of the application

context-switching and scheduling policy as well as operations over lists of tasks and operations over tasks, it was decided subset of results related to representatives of the operations will be presented only. Because the operations can be easily deduced from their names, descriptions of the operations is skipped in the following text. Time-complexity of the operations is introduced as a function of n (number of tasks within an RT system) and m (number of ticks all tasks were in suspended state, i.e., time measured between `vTaskSuspendAll` and `vTaskResumeAll`). In Tab. 1 the results are presented. In the tables, n represents the number of tasks in the system while m represents the number of OS time ticks during which all tasks were suspended.

5.2. Derivation of Response-Time Based Schedulability Test

On page 19, a formula (2) was presented illustrating how basic formula (1) can be extended in order to achieve more precise results during response time (R_i) analysis of tasks (τ_i) from given Γ to be scheduled on Φ by means of ξ, i.e., RM mechanism.

5.2.1. Formula Modification According to FreeRTOS

Because the presented formulas abstract from the implementation of particular RTOS, it is necessary to modify them if response times are to be enumerated precisely according to a real implementation. E.g., if FreeRTOS is supposed as an implementational RTOS then C_{sw} related terms can be counted in C_{tick} because each tick of FreeRTOS-timer leads to a context-switch. So, the formula (2) can be simplified to the form:

$$R_i^{k+1} = C_i + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i^k + J_j + T_{tick}}{T_j} \right\rceil C_j + \sum_{\forall j \in alltasks} \left\lceil \frac{R_i^k + J_j + T_{tick}}{T_j} \right\rceil C_{queue} + \left\lceil \frac{R_i^k}{T_{tick}} \right\rceil C_{tick} \quad (3)$$

Furthermore – in FreeRTOS – task periods (T_is) can be set-up as multiples of timer-tick periods (T_{tick}). If the condition is met for all $\tau_i \in \Gamma$, the formula can be yet simplified to the form:

$$R_i^{k+1} = C_i + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i^k}{T_j} \right\rceil C_j + \sum_{\forall j \in alltasks} \left\lceil \frac{R_i^k}{T_j} \right\rceil C_{queue} + \left\lceil \frac{R_i^k}{T_{tick}} \right\rceil C_{tick} + C_{tick} + C_{yield} \quad (4)$$

After the removal of corresponding terms, the end of the formula (4) was completed by term $C_{tick} + C_{yield}$, which reflects overhead needed for context-switch before task start and overhead needed for context-switch after task end – for illustration, see Fig. 5[6].

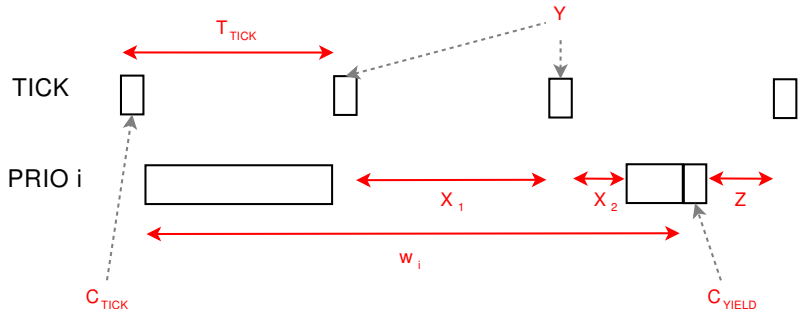


Fig. 5 Illustration to parameters for R_i enumeration of FreeRTOS tasks

5.2.2. Enumeration of Terms involved in the Formula

If the formula (4) is to be utilized in practice, particular terms (i.e., C_i, C_j, T_j, B_i, C_{queue}, C_{tick}, T_{tick}, C_{yield}) involved in the formula must be enumerated first. Some of the terms (C_i, C_j, T_j) are related to particular tasks τ_i, τ_j or reflect preemption overhead of a high-priority task τ_i by lower-priority tasks (B_i) while remaining (C_{queue}, C_{tick}, T_{tick}, C_{yield}) represent Γ-independent overheads given by RTOS kernel implementation. Below, it will be outlined how the values can be get:

C_i (C_j) – the values equal to WCET values of particular tasks τ_i (τ_j). In section 3, it was shown how the values can be get by means of range profiles created over programming constructs within tasks' codes. After the values are set, D_i (D_j) can be adjusted in such a way it holds $C_i \leq D_i$ ($C_j \leq D_j$),

T_j – the value is set in such a way that it must hold $D_j \leq T_j$,

B_i – to be able to set up the value, information about all lower-priority tasks, i.e., $lp(i) = \{\tau_j |\ \tau_j$ is of lower priority than $\tau_i \wedge \tau_i, \tau_j$ access the same critical section$\}$ must be available together with information about time for which each τ_j access the section (for each such a resource and τ_j, e.g., ex-

[6] in the figure, following symbols are utilized: X represents time needed for execution of higher-priority tasks, Y represents overheads related to timer-tick, i.e., context-switch overhead and overhead related to placement of tasks into ready-queue, Z represents time needed for execution of tasks having no impact to execution of a task being observed because the task is in a not-ready state

Table 1 WCETs for selected FreeRTOS calls

FreeRTOS function	WCET (in CPU cycles)
vListInitialise	71
vListInitialiseItem	36
vListInsertEnd	103
vListInsert	$49n+139$
vListRemove	$21n+102$
vTaskDelay	$91mn^2+91n^2+425mn+99m+486n+921$
vTaskDelayUntil	$91mn^2+91n^2+425mn+99m+486n+976$
prvCheckDelayedTasks	$91n^2+425n+30$
vTaskSuspend	$63n+789$
vTaskResume	$70n+662$
vTaskSuspendAll	30
xTaskResumeAll	$91mn^2+91n^2+425mn+99m+416n+325$
xTaskIsTaskSuspended	100
portENTER_CRITICAL	5
portEXIT_CRITICAL	15
vPortYield	230
prvAddTaskToReadyQueue	$49n+151$
vTaskSwitchContext	131
vTaskIncrementTick	$91n^2+425n+82$
portSAVE_CONTEXT	49
portRESTORE_CONTEXT	41

ecution time between `portENTER_CRITICAL` and `portEXIT_CRITICAL` calls must be measured – execution times related to the calls can be found in Tab. 1 are to be involved in C of a calling task). If the information is available, B_i is set to the maximal access time to the resources measured over $lp(i)$,

C_{queue} – the value (49n + 151) equals to WCET of `prvAddTaskToReadyQueue()` function call, which can be found in Tab. 1

C_{tick} – the value is given by implementation of FreeRTOS tick-service routine. By default, the code of the routine is as follows:

```
interrupt (TIMERA0_VECTOR) prvTickISR( void )
{
  portSAVE_CONTEXT();
  vTaskIncrementTick();
  vTaskSwitchContext();
  portRESTORE_CONTEXT();
}
```

So, C_{tick} equals to the sum of the WCETs of partial function calls forming body of the routine. In Tab. 1, it can be found partial values are as follows: 49 (`portSAVE_CONTEXT()`), $91n^2 + 425n + 82$ (`vTaskIncrementTick()`), 131 (`vTaskSwitchContext()`), 41 (`portRESTORE_CONTEXT()`). I.e., the sum is dependent on total number of tasks running in FreeRTOS that is $49 + 91n^2 + 425n + 82 + + 131 + 41 = 91n^2 + 425n + 303$,

T_{tick} – the value represents FreeRTOS timer period specified in the number of CPU cycles. So, the value can be enumerated by `configCPU_CLOCK_HZ` (by default, set to 7372800) and `configTICK_RATE_HZ` (by default, set to 100) parameters defined in `FreeRTOSConfig.h`. I.e., one FreeRTOS tick takes $\frac{configCPU\_CLOCK\_HZ}{configTICK\_RATE\_HZ} = \frac{7372800}{100} = 73728$ CPU cycles,

C_{yield} – the value is constant (230) for given FreeRTOS implementation and it can be found in Tab. 1 in `vPortYield` row. It was produced after analysis of corresponding routine:

```
void vPortYield( void )
{
  asm volatile ( "push r2" );
  _DINT();
  portSAVE_CONTEXT();
  vTaskSwitchContext();
  portRESTORE_CONTEXT();
}
```

6. CONCLUSION

In the paper, novel hybrid method for timing analysis of RT systems controlled by an RTOS was presented, with descriptions mainly related, but not limited to WCET analysis. The same principles can be utilized for BCET analysis if required – however, usually the kind of analysis is not necessary for RT systems, so the paper was focused only to the foremost. Novelty of the method lies in the fact OS model is reflected during the analysis. By the reflection, more detail results about analyzed system can be produced than it is typical for actual methods dealing only with techniques leading to safe and precise enumeration of WCET/BCET values. But, because they abstract from RTOS the system is supposed to run on, they cannot produce certain results valuable especially from practical point of view, e.g., results related to interrupt processing overheads, context-switch overheads, management of task queues, utilization of shared resources and stack etc. It was shown how the results can be utilized for derivation of a schedulability test for RM scheduling mechanism. Practical applicability of the method was demonstrated using MSP430 architecture running RT system controlled by FreeRTOS.

ACKNOWLEDGEMENT

This work has been partially the Research Plan No. MSM 0021630528 (Security-Oriented Research in Information Technology), the BUT FIT-S-11-1 and the IT4Innovations Centre of Excellence CZ.1.05/1.1.00/02.0070 projects.

REFERENCES

[1] CHENG, A. M. K.: *Real-Time Systems, Scheduling, Analysis, and Verification*. John Wiley & Sons, 2002.

[2] COTTET, F. — DELACROIX, J. — KAISER, C. — MAMMERI, Z.: *Scheduling in Real-Time Systems*. John Wiley & Sons, 2002.

[3] ERMEDAHL, A. — STAPPERT, F. — ENGBLOM, J.: Clustered worst-case execution-time calculation. *IEEE Transactions on Computers*, 54(9):1104–1122, 2005.

[4] FIDGE, C. J.: Real-Time Schedulability Tests for Preemptive Multitasking. *Real-Time Systems*, 14:61–93, 1998.

[5] FITkit: hardware/software co-design based educational platform. Availaible on: `http://merlin.fit.vutbr.cz/FITkit/`. Accessed on March 29, 2012.

[6] FreeRTOS.org project. Availaible on: `http://www.freertos.org`. Accessed on June 24, 2011.

[7] HARDY, D. — LESAGE, B. — PUAUT, I.: Scalable Fixed-Point Free Instruction Cache Analysis. In: *Proceedings of the 32nd IEEE Real-Time Systems Symposium (RTSS)*, IEEE CS, 204–213, 2011.

[8] JOSEPH, M.: *Real-time Systems Specification, Verification and Analysis*. Prentice Hall, 2001.

[9] LAPLANTE, P. A.: *Real-Time Systems Design and Analysis*. John Wiley & Sons, 2004.

[10] LIU, C. L. — LAYLAND, J.: Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[11] LUNDQVIST, T. — STENSTROM, P.: Timing Anomalies in Dynamically Scheduled Microprocessors. In Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS). pp. 12–21, 1999.

[12] LV, M. — GUAN, N. — DENG, Q. — YU, G. — YI, W.: Static worst-case execution time analysis of the uC/OS-II real-time kernel. *Front. Comput. Sci. China*, 4(1): 17–27, 2010.

[13] MSPsim - a Java-based simulator of MSP430 sensor network platforms. Availaible on: `http://www.sics.se/project/mspsim`. Accessed on June 5, 2011.

[14] RAJNOHA, P.: *Analysis of real-time operating system kernels running on FITkit.* Master's thesis, Faculty of Information Technology, Brno University of Technology, Brno, 2009.

[15] SANDELL, D.: *Evaluating static worst-case execution-time analysis for a commercial real-time operating system.* Master's thesis, Mälardalen University, Sweden, July 2004.

[16] SANDELL, S. — ERMEDAHL, A. — GUSTAFSSON, J. et al.: Static timing analysis of real-time operating system code. In *1st International Symposium on Leveraging Applications of Formal Methods*, 2004.

[17] Texas Instruments. Availaible on: `http://www.ti.com/`. Accessed on June 10, 2011.

[18] TINDELL, K. — HANSSON, H.: Real time systems and fixed priority scheduling. Department of Computer Systems, Uppsala University, 1995.

[19] WENZEL, I. — RIEDER, B. — KIRNER, R. — PUSCHNER, P.: Automatic Timing Model Generation by CFG Partitioning and Model Checking. In Proceedings of Design, Automation and Test in Europe (DATE). Munich, pp. 606 – 611, 2005.

[20] WILHELM, R. — ENGBLOM, J. — ERMEDAHL, A. et al.: The worst-case execution time problem – overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 5(3):1–53, 2008.

[21] ZOLDA, M. — KIRNER, R.: Compiler Support for Measurement-based Timing Analysis. *Proceedings of the 11th International Workshop on Worst-Case Execution Time Analysis (WCET'11)*. OCG, 10 p., 2011.

Received April 24, 2012, accepted December 17, 2012

BIOGRAPHIES

Josef Strnadel received the MSc. degree in computer science and electrical engineering at the Faculty of Electrical Engineering and Computer Science (FEE), Brno University of Technology (BUT), Czech Republic, in 2000 and the PhD. degree in information technology at the Faculty of Information Technology (FIT) of BUT, in 2004. Now he works as an assistant professor at FIT BUT. His main research interests are related to dependability of embedded and real-time systems.

Peter Rajnoha received the MSc. degree in information technology at the Faculty of Information Technology (FIT), Brno University of Technology (BUT), Czech Republic, in 2009. His main research interests are related to implementation and analysis of operating systems. Since 2008, he works in a development department of Redhat company and solves problems related to Linux device-mapper and corresponding subsystems.