EMSBench: Benchmark and Testbed for Reactive Real-Time Systems[*]

Florian Kluge[†1], Christine Rochange[2], and Theo Ungerer[3]

1 **Department of Computer Science, University of Augsburg, Augsburg, Germany**
`fkuau@gmx.net`

2 **IRIT, Université de Toulouse, CNRS, France**
`http://orcid.org/0000-0001-7257-7114`
`christine.rochange@irit.fr`

3 **Department of Computer Science, University of Augsburg, Augsburg, Germany**
`ungerer@informatik.uni-augsburg.de`

Abstract

Benchmark suites for real-time embedded systems (RTES) usually contain only pure computations that are often used in this domain. They allow to evaluate computing performance, but do not reproduce the complexity and behaviour that is typical for such systems. Actual RTES have to interact with the physical environment, which is often reflected by code that is executed concurrently. In this article, we present the software package EMSBench that mimics such complex behaviour, and highlight some of its use cases. The benchmark code `ems` of EMSBench is based on the open-source engine management system (EMS) FreeEMS. Additionally, EMSBench contains a trace generator (`tg`) that provides input signals for `ems` and enables to execute `ems` close to reality. We provide detailed descriptions of the `ems`'s execution behaviour and of trace generation. EMSBench can be used as test or benchmark program to compare different hardware platforms, e.g. in terms of schedulability. Also, we use EMSBench as a benchmark for static worst-case execution time (WCET) analysis and compare these results to measurements performed on existing hardware. Our results based on the OTAWA WCET estimation tool show WCET over-estimations by the static analysis from 11.9% to 41.1% depending on the complexity of the analysed functions.

2012 ACM Subject Classification Computer systems organization → Real-time systems, Software and its engineering → Real-time systems software

Keywords and Phrases Real-time benchmark, WCET Analysis, Engine Management System

Digital Object Identifier 10.4230/LITES-v004-i002-a002

Received 2016-07-29 **Accepted** 2017-05-08 **Published** 2017-07-07

1 Motivation

Benchmark programs are widely used to assess the performance of execution platforms and development tools. In hard real-time computing domains, they also play an important role when comparing tools for WCET analysis. Widely used, for example, is the *Mälardalen Benchmark Suite* [4]. A drawback of this and similar suites is that the contained programs do not reproduce the complexity of actual real-time systems. Usually, each program is a closed system that implements only a single algorithm. In contrast, actual real-time software mostly consists of multiple interacting modules. The modules are executed concurrently and may even interfere with each other, thus mutually affecting each other's timing behaviour. Also, real-time software usually is an open system that interacts with processes in the physical world. It must react to physical events and

[*] Parts of this article have been published before in [10].
[†] Florian Kluge is now with Elektronische Fahrwerkssysteme GmbH.

its timing behaviour is heavily depending on the physical processes. Thus, there is also a need for *system benchmarks* that enable a more global assessment of real-time platforms. However, only few works exist that tackle this need.

In our view, a system benchmark could be employed in multiple manners: (1) During platform (e.g. processor + operating system) development, it may act as a test program to investigate functional aspects of a platform. When development is finished, the system benchmark will be used for (2) an experimental evaluation of important aspects of the system. This could be, e.g., the evaluation of operating system mechanisms, schedulability analysis, or response-time analysis (RTA). Finally, the program can be used as a (3) benchmark for the evaluation of WCET analysis tools. Thus, it would be integrated in the whole development process.

This work is guided by the following requirements:

Complexity. The overall behaviour of the program shall arise from the interaction of multiple modules. These modules should be scheduled independently from each other.

Reactivity. The program shall react upon external events. Reaction times should be constrained by deadlines.

Ease of Use. The program should be as easy to use as possible. Thus its potential for a widespread use would be increased.

When considering actual RTESs, it becomes obvious that there must be a tradeoff between the first two requirements and the last one. Real RTESs may employ a large variety of sensors to monitor the physical world. The software can consist of hundreds of tasks. So while being both complex and reactive, we guess such a software would never be easy to use. For the purpose of this work, we set our focus differently: We aim to have a program that requires as few as possible sensor inputs, but still exhibits as much as possible of its original dynamic behaviour.

The software package EMSBench is based on the open source EMS FreeEMS[1]. We have stripped down the FreeEMS software such that it requires only positional signals of the crankshaft as inputs. Additionally, we developed a testbed for EMSBench that generates these input signals and thus allows to execute the benchmark program. EMSBench is available for download [2] at GitHub under the conditions of the GNU GPL. The benchmark code derived from FreeEMS consists mainly of multiple interrupt service routines (ISRs) that interact among each other to control fuel injection and ignition in a spark ignition engine. Thus, EMSBench exhibits some of the complexity and reactivity of a real use-case application, even though not on an industrial scale. We balance these properties against an easy porting to and employment on other hardware platforms.

In this article, we present the software package EMSBench and examine some of its use-cases. Therefore, we provide a detailed characterisation of the code's structure and execution behaviour. This information is used to ease flow analysis in static WCET estimation which we demonstrate using the OTAWA toolset [1]. Additional use-cases are execution time measurements and the schedulability/response-time analysis for tasks. Execution time measurements are derived from realistic execution traces on two hardware platforms.

We proceed as follows: We describe FreeEMS in Section 2, and the EMSBench software package in Section 3. Possible uses of the benchmark (execution time measurements, static analysis of the worst-case execution time, analysis of task interferences) are discussed and experimented in Section 4. In Section 5, we review existing benchmarks and discuss how they compare to EMSBench. We conclude this article in Section 6.

[1] `http://freeems.org/`

2 FreeEMS

FreeEMS is an open source engine management system for four-stroke spark-ignition engines. It is designed for execution on a 16-bit microcontroller from the Freescale HCS12X family. Hitherto, it was deployed successfully to over 20 different engines. We use version 0.1.1 of FreeEMS as base for this work. Although newer versions are available, the dependencies between the individual modules are recognisable more clearly in version 0.1.1. Furthermore, the newer version is split in many more modules to be applicable more universally. For the purpose of our work using the newer version thus would only have increased the required effort, but would not have changed the outcome. FreeEMS is designed such that it can be used with different types of rotary encoders. For this work, the implementation for a 24/2 camshaft encoder from Denso for engines with intake-manifold fuel injection was chosen. In the following discussions, the term FreeEMS shall refer to this specific version of the FreeEMS software.

2.1 Spark Ignition Engine and Engine Management

Before we explain the structure of FreeEMS, let us recall the operation of a spark ignition engine. Each combustion chamber (cylinder) of the engine is terminated downwards by a movable piston. Connecting rods join all pistons to the crankshaft. Thus, the vertical movement of the pistons is converted to an axial movement of the crankshaft. The cylinder housing has at least two valve openings, one as intake for air and fuel, the other as outlet for exhaust. The valves are controlled mechanically by two camshafts. These move synchronously with the crankshaft, but only at half its speed. A spark plug is placed in each cylinder head. The inlet valve discharges into the intake system.

One vertical movement of a piston resembles one stroke, during which the crankshaft moves by 180°. A full engine cycle consists of four strokes which corresponds to a movement of the crankshaft by 720°. During the first stroke, the piston moves downwards and the inlet valve is open. The cylinder ingests air that is enriched with fuel through an injection valve. During the following upward movement of the piston during the second stroke, the mixture of air and fuel is compressed. All valves are closed now. The third stroke is initiated by spark at the spark plug. The resulting combustion leads to a downward movement of the piston. During the fourth stroke, exhaust is diverted from the cylinder through the outlet valve by the upward movement of the piston.

The opening times of the injection valves and the ignition times are controlled by the EMS. The calculations of the EMS are based on the positions of crank- and camshafts which are captured with encoders. The duration of injection is mainly influenced by the state of the throttle position, and additionally by air pressure and temperature. The ignition time, i.e. when the spark is produced, depends on the position of the crankshaft and the current speed. It is calculated such that the piston of the cylinder is near its top dead point when the air/fuel mixture ignites. Additional sensors are used to capture, e.g., temperature and air pressure in some components of the engine.

2.2 Interfacing with the Physical World

Like any other EMS [6, 23], FreeEMS utilises sensors and actuators to interact with the engine and the car's driver. The driver's command is recognised through the throttle position. Several temperature and pressure sensors provide information about the current state of the engine and the environment, e.g. monitoring of exhaust gas oxygen allows to draw conclusions about the current combustion behaviour. The data collected from these sensors mainly influences the calculation of injection and ignition parameters, e.g. the amount of fuel that is injected in each cycle. Most sensors are connected to A/D converter (ADC) channels of the microcontroller on which FreeEMS is executed. In total, FreeEMS currently uses 11 ADC channels.

For further monitoring, and also to set outputs in time, FreeEMS uses the enhanced capture timer (ECT) of the HCS12X microcontroller. The ECT has a global counter and 8 channels that can either act as input capture (IC) or output compare (OC). The counter is incremented continuously. If a channel is configured for input capture mode, it monitors an input pin. On the configured event, e.g. if the input level on the pin toggles, the current value of the global counter is stored in a register, and an interrupt request (IRQ) is generated. If a channel is used in output compare mode, a timestamp calculated by software is stored in a register. When the global counter equals the timestamp, an output action (set high/low, toggle) is performed on a pin and also an IRQ is generated.

The dynamic behaviour of the engine is monitored with at least one rotary encoder mounted to the crank- or camshaft. Certain angular positions trigger reactions of the EMS that control actual fuel injection and ignition. The 24/2 camshaft sensor possesses 24 equally spaced primary teeth and 2 equally spaced secondary teeth that generate corresponding signals during each camshaft revolution. The movements of crank- and camshaft are coupled together. The camshaft revolves with half the speed of the crankshaft. Insofar, the 24/2 camshaft sensor is equivalent to a 12/1 crankshaft sensor. To achieve a low latency and a high accuracy of the EMS reactions, the signal lines of the encoder are connected to two IC channels of the microcontroller's ECT. If a tooth is detected, the IC channel automatically saves the current time stamp from the ECT's global counter and raises an IRQ that is handled by one of the FreeEMS ISRs.

The main actuators that are controlled by the EMS are the fuel injection valves, and the spark coil and plugs. FreeEMS performs fuel injection in a semi-sequential manner, i.e. the injection valve for any cylinder opens twice per engine cycle and injects fuel into the cylinder's intake system. The amount of fuel that is injected is regulated through the opening times of the valves. The injection valves are controlled through OC channels of the ECT. Opening and closing is performed automatically by the OC channels at times that are set by FreeEMS. To trigger fuel combustion, FreeEMS uses wasted-spark ignition: In any cylinder, two sparks are produced during each engine cycle, but actually only one triggers a combustion. This approach allows to simplify the software and hardware for ignition distribution. The ignition channels are connected to regular I/O pins of the microcontroller. The pins are controlled by ISRs that are triggered by periodic interrupt timer (PIT) units. The PITs are set anew for each new ignition. FreeEMS can generate a tachometer signal to display the engine's current revolution speed. Finally, FreeEMS provides a serial (UART) interface for monitoring and tuning of the EMS.

The minimum hardware requirements to execute FreeEMS on a microcontroller can be summed up as follows: The controller must possess at least 8 capture/compare (C/C) channels that can access a global counter. Two periodic interrupt timers are needed for further control of I/O operations. Additionally, FreeEMS uses another timer to control the execution of periodic tasks. The original implementation uses the real-time interrupt functionality of the HCS12X microcontroller. Concerning I/O, at least 25 pins are required in total. 11 pins must be accessible by an ADC unit. Each of the 8 C/C channels must be connected to a separate I/O pin, 2 for input from the rotary encoder, and 6 for control of the injection valves. Another 6 output pins are needed for driving the ignition channels.

2.3 Operation of FreeEMS

The most important relationships between FreeEMS and the underlying hardware are shown in Figure 1. Two ECT channels (0 and 1 in the figure) are configured as IC. The remaining ECT

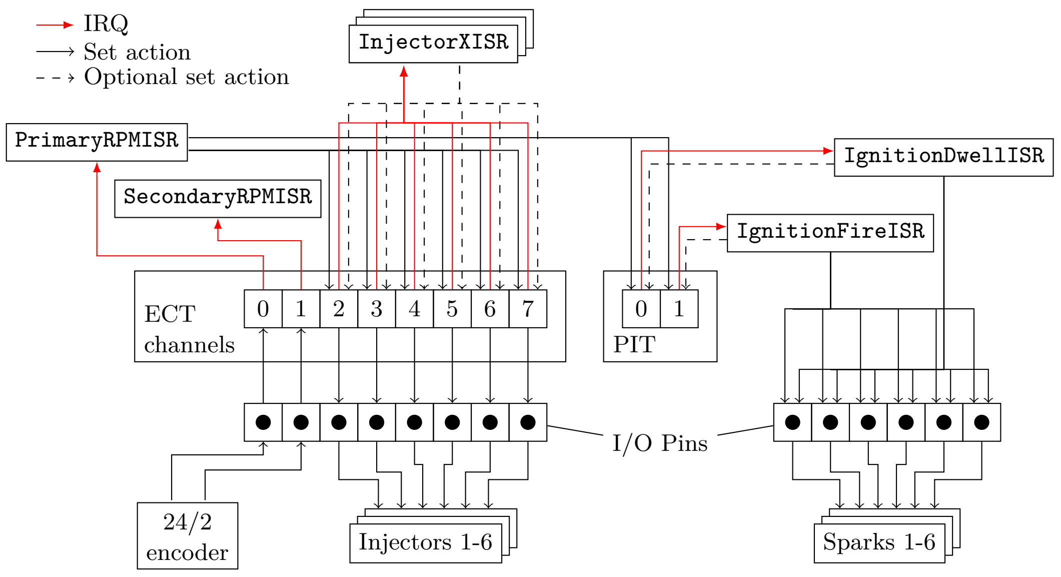

Figure 1 Important FreeEMS IRQs and their interaction with with the µC peripherals and external sensors/actuators.

channels are configured as output compare to trigger up to 6 injection valves. The IC channels monitor 24/2 encoder mounted to the camshaft. The interrupts generated by these channels are used by FreeEMS to determine the speed and position of the camshaft. Based on this and further data (derived from the sensors connected to the ADCs), times for opening the injection valves are calculated and set in the respective channels. Once an injection valve is opened through its channel's timer expiring, it triggers an ISR that reconfigures the channel for closing of the valve. Similar actions are performed for ignition, i.e. dwelling and firing of the spark plugs. Here, the PIT of the microcontroller is used. The ignition pins are completely controlled by software (`IgnitionDwellISR`, `IgnitionFireISR`).

The timing requirements for the single ISRs can be derived from their chain of effects: the `PrimaryRPMISR` and the `InjectionXISR`s are responsible for (re-)activating timers, based on calculations they perform. Obviously, these calculations must be finished before the calculated timer values expire. Similar requirements can be found in the `IgnitionXISR`s, which may prepare their next activation. Numbers about the execution times and the timer values will be presented in Section 3.4.2, where we discuss the execution behaviour, and in Section 4.1, where we present execution time measurements.

2.4 Code Structure

The code of FreeEMS can roughly be divided into functions that are performed inside a loop in the `main` function, and a number of ISRs. The modules communicate via global variables, critical sections are protected by disabling interrupts. Figure 2 gives an overview of the most important modules and their dependencies. First, it shows which modules access global data (reading/writing). Second, trigger dependencies are indicated in the figure. These can be either the setting of an interrupt timer, or the use of global flags, if functions in the main loop are concerned.

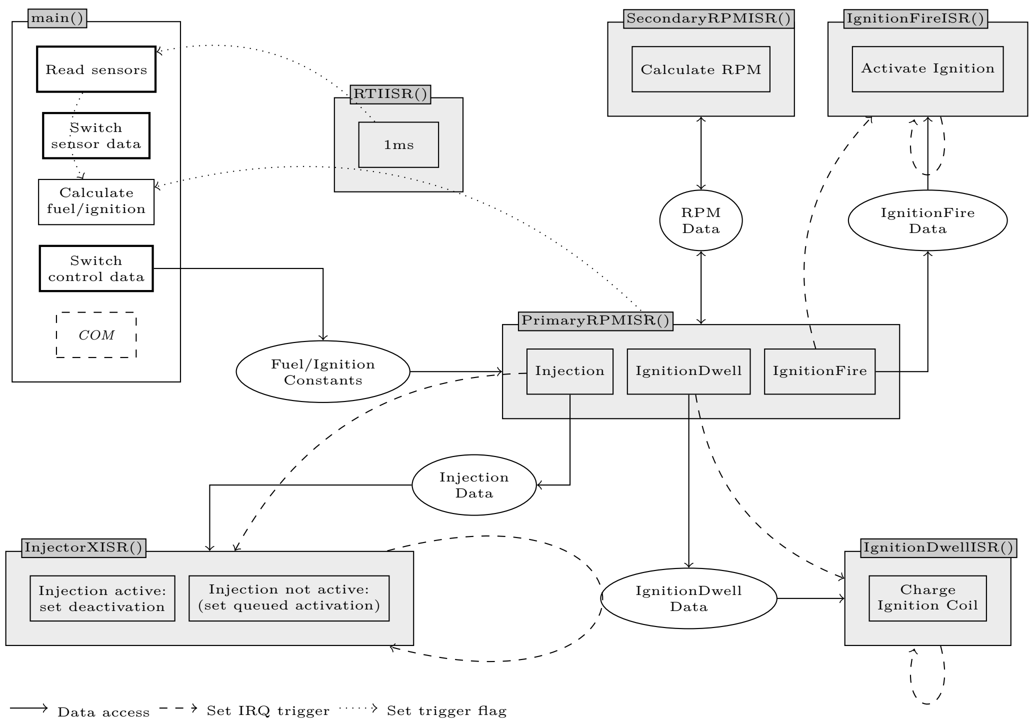

Figure 2 Structure and dependencies between ISRs and main function tasks in FreeEMS; red frames indicate critical sections during which IRQs are disabled.

2.4.1 `main`

After having initialised the whole EMS, the `main` function executes an infinite loop. Inside the loop, three tasks are performed:

1. If demanded from other modules, sensors are read. The demand is signalled by a flag in a global variable. The snapshot of all sensors is stored in a data structure. To ensure consistency of the structure and a low latency between the single readings, this task is a critical section during which interrupts are disabled. Reading of new sensor data automatically triggers the second task.
2. If new sensor data has been read, the sensor data set is switched inside a critical section with interrupts disabled.
3. Fuel and ignition parameters are calculated based on new sensor data. Both input and output data structures are allocated twice to ensure that always one structure with consistent data is available. The input data structure is filled by the previous task in the `main` function. Switching between the input (resp. output) structures is performed at the beginning (resp. end) of this task. Both operations are critical sections that are protected by disabling interrupts.
4. If new parameters have been calculated, the parameter data set is switched inside a critical section with interrupts disabled.
5. Requests that were received over the serial interface are handled. The requests are used for debugging, monitoring and tuning of the system. This task can be interrupted any time. It is not covered by the work at hand and will be ignored in the following considerations.

The code in the main function specifies low-priority tasks, as these can be interrupted any time (except during critical sections) by an ISR.

2.4.2 `PrimaryRPMISR`

The `PrimaryRPMISR` is bound to an IC channel of the platform's ECT. It is triggered by each primary pulse of the rotary encoder, i.e. it is executed 24 times each camshaft revolution. It counts the number of primary pulses since the last secondary pulse. The counter is used together with the `SecondaryRPMISR` to ensure synchronism between engine and EMS (see Section 2.4.3). If a loss of synchronism due to losses of primary or secondary pulses is detected, the ISR terminates immediately. Else, each second pulse, several control tasks are performed: (1) Injection times are calculated and the OC timer of the relevant injection channel is set. If another injection is already pending for the channel, the event is queued for evaluation by the `InjectorXISR` (see 2.4.4). (2) Times for charging of the ignition coil (`IgnitionDwellISR`) and triggering of the ignition (`IgnitionFireISR`) are calculated. If no other ignition events are pending, PITs for both events are set directly. Else, the times are put into queues that are handled by the `IgnitionDwellISR` resp. `IgnitionFireISR`.

2.4.3 `SecondaryRPMISR`

A second IC channel of the ECT activates the `SecondaryRPMISR` any time a secondary pulse from the rotary encoder is captured. The ISR's main task is to ensure synchronism between the engine and the EMS. This is achieved by checking whether the correct number of primary pulses has arrived since the last secondary pulse. For the 24/2 rotary encoder, this means that between any two secondary pulses 12 primary pulses must occur. If loss of synchronism is detected, a flag is set to signal this to the `PrimaryRPMISR`. Additionally, the `SecondaryRPMISR` calculates the current revolution speed of the engine.

2.4.4 `InjectorXISR`

FreeEMS supports up to 6 injection channels. Each injection channel `X` is handled by a separate `InjectorXISR` (with $\texttt{X} = 1, 2, \ldots, 6$), which in turn is bound to a separate OC channel of the ECT. Activation and deactivation of the injection valve is performed automatically when the associated interrupt is triggered. If injection was activated when the ISR is released, the time for deactivation is determined and the channel is configured appropriately. Upon deactivation, the ISR checks whether another injection event is queued for this channel. If necessary, it sets the timer anew.

2.4.5 `IgnitionDwellISR`

The `IgnitionDwellISR` is responsible for charging of the ignition coil. This is done by activating the power supply of the relevant ignition channel. If further *IgnitionDwell* events are queued, the associated PIT channel is restarted with a new offset, else the channel is deactivated.

2.4.6 `IgnitionFireISR`

Actual ignition is triggered through the `IgnitionFireISR`. It deactivates the power supply of the ignition coil which leads to an immediate discharge. The discharge results in a spark at the associated spark plug. If further *IgnitionFire* events are queued, the associated PIT channel is restarted with a new offset, else it is deactivated.

2.4.7 `RTIISR`

The `RTIISR` manages the execution of tasks that must be performed periodically. It implements intervals of 1 ms, 100 ms, 1 s, and 60 s. The `RTIISR` is released each 1/8 ms to accommodate also for this interval if need should arise. Depending on internal counters, it decides whether a task of one of the implemented intervals must be executed. Task execution can either take place within the `RTIISR`, or the ISR sets corresponding flags to trigger the execution inside the `main` loop. In the current implementation, only each 500 ms (via the 1 ms part of the `RTIISR`) a flag is set to trigger sampling of the ADC channels in the `main` loop.

2.4.8 Further ISRs

The `TimerOverflow` ISR extends the maximum time span that can be measured with the ECT. The 16-bit counter of the ECT is configured to be incremented each 0.8 µs. An overflow occurs after ≈ 52 ms. On each release, the `TimerOverflow` ISR increments an additional 16-bit counter, thus extending the counter effectively to 32 bits. An overflow of the thus available time span of ≈ 57 min is handled appropriately.

The `LowVoltageISR` is currently only used for diagnostics and counts the frequency of low voltage events. The `ModDownCtrISR` generates a tachometer signal. These three ISRs yield only a minor contribution to the overall behaviour and therefore are ignored in the following analysis.

2.5 Interaction between ISRs

The interaction between the important ISRs and hardware units is depicted in Figure 1. The `PrimaryRPMISR` is the most important ISR in FreeEMS. Supported by the `SecondaryRPMISR`, it has exact knowledge about the crankshaft's position and speed, and thus can calculate the times for fuel injection and ignition. By setting the relevant timers, it controls the execution of the injection and ignition ISRs (continuous lines in Figure 1). If there are still pending events for the timers, the events are queued instead. In this case, the timers are set by the corresponding ISRs themselves as soon as the pending event occurs (dashed lines). Furthermore, the `PrimaryRPMISR` can trigger a recalculation of the injection and ignition parameters that is performed in the `main` loop. Reading of sensor data inside the `main` loop is triggered only by the `RTIISR` each 500 milliseconds, also leading to a recalculation of the injection and ignition parameters.

3 EMSBench

In this section, we present the software package EMSBench in detail. First, we describe the changes we made to the FreeEMS code. To execute the code successfully, signal traces must be generated, which we will discuss in Section 3.2. Furthermore, we explain how EMSBench can be ported to other hardware platforms, and discuss the timing behaviour of the single modules.

3.1 Code Changes

The FreeEMS code was adjusted to provide a preferably simple program that still exhibits a behaviour that is as close to the original one as possible. The resulting implementation will be termed `ems` in the following. Most accesses to input devices (see Section 2.2) were replaced by initialised constants. Only the input signals of the rotary encoder were kept as they influence code execution significantly. By triggering the `PrimaryRPMISR`, they also trigger the ISRs for injection and ignition indirectly. The corresponding output signals are produced and can be tapped from the associated pins. The input signals from the rotary encoder can be provided by a trace generator that emulates arbitrary driving cycles (see Section 3.2). Due to all other input values being constant, we expect no variations in the injection times. Ignition times should vary with the speed of the engine. Thus, EMSBench can only reproduce an abstract variant of an EMS's actual behaviour.

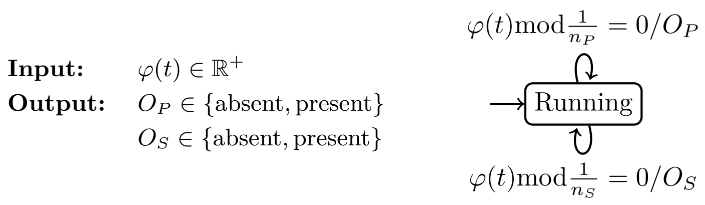
Input: $\varphi(t) \in \mathbb{R}^+$
Output: $O_P \in \{\text{absent}, \text{present}\}$
$O_S \in \{\text{absent}, \text{present}\}$
$\varphi(t) \bmod \frac{1}{n_P} = 0 / O_P$
Running
$\varphi(t) \bmod \frac{1}{n_S} = 0 / O_S$

Figure 3 Behaviour of the crankshaft rotary encoder.

Portability of the `ems` to arbitrary platforms is enabled by the definition of a hardware abstraction layer (HAL). The HAL defines interfaces that are used by `ems` to control the various hardware timers and capture/compare channels. It also declares the `ems` functions that must be called by platform-specific ISRs that are part of the HAL. Currently, HAL implementations for the STM32F4-Discovery platform using an ARM Cortex-M4, and a custom FPGA-based Nios II platform are available.

3.2 Trace Generation

To run the EMSBench benchmark program in a meaningful manner, it needs signal traces that emulate the behaviour of the 24/2 camshaft sensor. We provide a trace generator that generates these signals based on driving cycles. Such driving cycles are used to perform reproducible and comparable experiments with cars. For example, the *New European Driving Cycle* [13] is widely used to estimate cars' fuel consumption. A driving cycle consists of multiple phases, which can, in turn, consist of one or multiple operations. Acceleration, initial and terminal velocity, duration, and used gear are given for each operation.

Signal generation in EMSBench is divided in two components: A preprocessor (`tgpp`) converts the driving cycle data into a crank shaft cycle. It was introduced because car movement cannot be directly related to crankshaft movement in all cases. For example, if the clutch is open, car and crankshaft move independently from each other. Actual signal generation (`tg`) executes the crankshaft cycle and generates the corresponding signals on which the `ems` must react. In the following, we first describe the model underlying the signal generation, and then its implementation.

3.2.1 Model

The relevant control signals are generated by a rotary encoder that monitors the crankshaft. The variant of FreeEMS used in this work uses a 24/2 rotary encoder that monitors the camshaft. This is equivalent to a 12/1 rotary encoder monitoring the crankshaft, which runs with twice the speed of the camshaft (see Section 2.1). Figure 3 shows a model of the rotary encoder as a real-time automaton. The current angle of the crankshaft $\varphi(t)$, measured in revolutions of the crankshaft, is used as input. Depending on the number of primary and secondary teeth, n_P and n_S, appropriate primary and secondary signals O_P and O_S are generated at certain times.

The behaviour of the crankshaft is modeled in Figure 4. This model evolves current angle $\varphi(t)$ and angular speed $\omega(t)$. If a new angular acceleration is set via the input α_N, the values for angular position and speed evolved so far are stored. The current time is used as new time offset t_0. To simplify the model, we assume that the crankshaft initially rotates with idle speed ω_I.

When combined, both models describe the signal generation by crankshaft and rotary encoder. To emulate the signal generation, we have to calculate the concrete signal times from the models and a given driving cycle. Therefore, angular position $\varphi(t)$ and speed $\omega(t)$ must be evolved based on the current angular acceleration α.

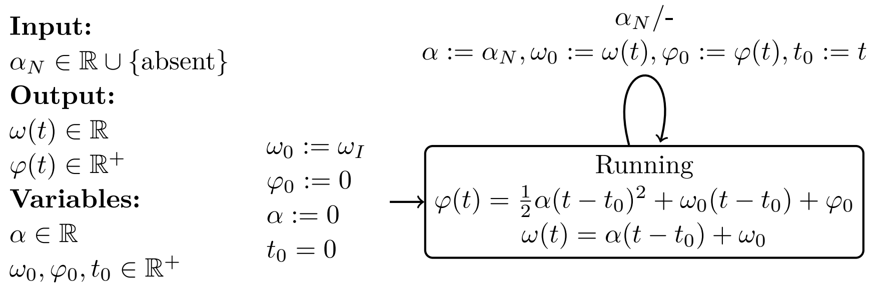
Input:
$\alpha_N \in \mathbb{R} \cup \{\text{absent}\}$
Output:
$\omega(t) \in \mathbb{R}$
$\varphi(t) \in \mathbb{R}^+$
Variables:
$\alpha \in \mathbb{R}$
$\omega_0, \varphi_0, t_0 \in \mathbb{R}^+$
$\omega_0 := \omega_I$
$\varphi_0 := 0$
$\alpha := 0$
$t_0 = 0$
α_N/-
$\alpha := \alpha_N, \omega_0 := \omega(t), \varphi_0 := \varphi(t), t_0 := t$
Running
$\varphi(t) = \frac{1}{2}\alpha(t-t_0)^2 + \omega_0(t-t_0) + \varphi_0$
$\omega(t) = \alpha(t-t_0) + \omega_0$

Figure 4 Behaviour of the crankshaft.

3.2.2 Preprocessor

The preprocessor `tgpp` requires two files as input. The first file contains the driving cycle that shall be emulated. The second file describes the car parameters that are needed to translate from car speed to angular speed of the crankshaft. These are the dimensions of the tyres, and the transmission ratios of the gearbox, axles and drive shaft. For idle phases, the idle speed of the engine and the acceleration with which the engine assumes this speed are given. Additionally, the file contains data that is directly passed on to signal generation. These are the number of primary teeth of the rotary encoder, and the angular distance between the secondary tooth and the preceding primary tooth. We assume that only one secondary tooth exists.

`tgpp` creates one or multiple crankshaft phases for each operation of the driving cycle. A crankshaft phase is described by its duration and the angular acceleration that acts on the crankshaft. We assume that the angular acceleration is constant during a phase. The translation of one operation into a single crankshaft phase is only possible, if the engine is idle, the car drives with constant speed, or accelerates or decelerates with closed clutch and set gear. The following operations are split into multiple crankshaft phases:

Driveaway from standstill is performed by slowly engaging the clutch. For simplification, we assume that the engine runs with its idle speed until the clutch is fully closed (first phase). The time of the full closure is calculated from the acceleration of the operation such that the car speed resembles the idle speed of the engine. In a second phase, the engine is accelerated as required by the end speed of the operation.

On a gear change the clutch is first opened, and engine speed converges to the idle speed. Then the gear is changed, and the clutch is closed again. For simplification, we assume that the opening of the clutch happens instantaneously at the beginning of the operation, and that during the operation no car speed is lost. So, concerning the crankshaft we can identify two intervals initially. For further simplification, we assume that each of these takes exactly one half of the duration of the operation. At the start of the first interval, the clutch is opened, and we assume that throttle control is free. The crankshaft speed converges to the idle speed following the idle acceleration. If the idle speed is reached before the end of the interval, we add another phase during which the angular acceleration $\alpha = 0$. During the second interval, the clutch is slowly being closed. We translate the interval to a phase where α is set such that the angular speed of the crankshaft at the end of the phase resembles the car speed of the operation, assuming that the throttle is being pressed by the driver.

Deceleration with open clutch is translated to one or two phases, depending on the initial angular speed ω_0. During the first phase, the idle acceleration acts on the crankshaft. If the crankshaft reaches its idle speed before the end of the operation, we add another phase with $\alpha = 0$ and appropriate duration to span the remaining time.

The single phases are stored as an array in a `C` source file. This file also contains additional constants that are important for signal generation, e.g. information about the rotary encoder, or idle speed. The file is then compiled, and linked with the code of the actual signal generator `tg`.

3.2.3 Signal Generation

The aim of the signal generator `tg` is to generate primary and secondary signals as they would be generated by an actual crankshaft sensor when a driving cycle is performed. `tg` is executed on an embedded platform. Its task is to evolve angular speed $\omega(t)$ and position $\varphi(t)$ according to the model presented previously, and to generate the primary and secondary signals at the appropriate times. For the calculations involved, please refer to the `tg` documentation [20]. We assume a perfect car driver who follows the operation cycle exactly. However, the throttle position is currently disregarded in EMSBench and assumed to be constant. Signal generation itself uses two OC channels of the embedded platform. The channels and associated ISRs are configured such that at the activation of the channel the pin is activated (logic 1). Simultaneously, the channel is reconfigured such that the pin is deactivated (logic 0) after a short time. Setting of the activation times for all channels is exclusively performed by the ISR for the primary channel (see Algorithm 1) when the channel is deactivated. The sole task of the ISR of the secondary channel is to set the channel's deactivation time. Additionally, the ISR for the primary channel has the following tasks:

- After each full revolution, $\varphi(t)$ is re-normalised to 0. Thus, we can keep the value of the variables in a range with high accuracy. Simultaneously, the time counter is reset and the current angular speed $\omega(t)$ is stored in ω_0.
- *Phase changes* are only performed after full revolutions, i.e. when the primary tooth at $\varphi(0)$ was released. This leads to small deviations between model and implementation (less than 1 revolution per phase change), which can affect only very short phases significantly. During a phase change, some parameters are recalculated which are used for the calculations of the succeeding activation times.
- The secondary tooth is placed between the third and the fourth primary tooth at $\phi_S \in [2\Delta_P, 3\Delta_P)$. Thus, we achieve good dispersal of the computing load of the primary ISR. This is illustrated in Figure 5, where the secondary tooth is released somewhere in the shaded area between P2 and P3. The ISR calculates on each second call (when the channel is deactivated, downward arrows) the next activation time. When it handles the first primary tooth P0, it performs additionally the re-normalisation, and, if necessary, the phase change. Furthermore, we require that for the calculation of the secondary tooth as much time is available as possible. This time is bounded by the distance between two primary teeth. Thus, the secondary calculation must be finished before the primary tooth preceding the secondary tooth is activated. Actually, the secondary tooth may be placed also between later primary teeth, but it should be avoided that the corresponding calculation coincides with renormalisation and phase change.

Similar to `ems`, the implementation of `tg` consists of two parts: A platform-specific abstraction layer provides a generic interface for managing the hardware units. All calculations are performed in a platform-independent application layer.

3.3 Adopting EMSBench

To execute `ems`, the target platform must have a timer device with at least 8 capture/compare channels that can access a common counter register. Additionally, three timers are required with a freely configurable activation interval. At least two pins must be connectable to capture/compare

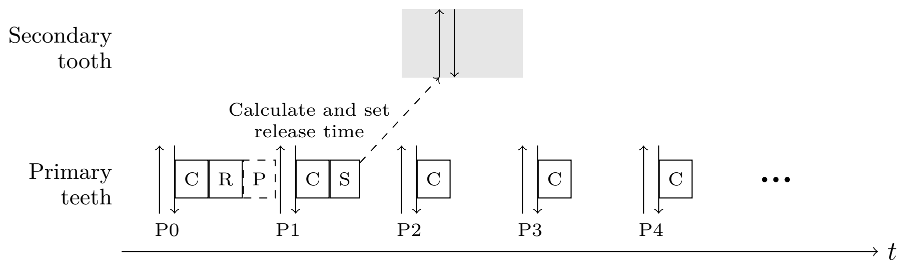

Figure 5 Timing for teeth calculations; ↑= activation of output pin and first call to ISR; ↓= deactivation of output pin and second call to ISR; C = Calculation for next primary tooth; R = Renormalisation; P = Phase change (optional); S = Calculation for secondary tooth.

channels, such that the signals of the trace generator can be routed to the correct device. For the execution of the trace generator, a timer with at least two compare channels and a common counter is required.

For all hardware-related functions we have defined a HAL that provides a generic interface for `ems` and `tg`. When porting EMSBench to a certain platform, only the relevant HAL functions have to be implemented. Due to the widespread use of 32-bit architectures, we have chosen such one for the implementation of our prototype. In a first step we have implemented EMSBench on the STM32F4-Disovery platform from ST Microelectronics. This cheap board contains a STM32F407VGT6 microcontroller, which is based on a ARM Cortex-M4 [19]. The Cortex-M4 implements the ARMv7 instruction set architecture (ISA). The microcontroller has several C/C timers with each providing up to four C/C channels. To accommodate the requirements of EMSBench, several C/C timers and their counters are configured to run synchronously with a common clock. Our second implementation of the HAL is aimed at a self-designed FPGA-based microcontroller. It uses the Nios II IP-Core from Altera and features a capture/compare timer with 8 channels which was developed in our group [9]. The main aim of this implementation was the validation of the HAL.

Porting EMSBench to new platforms requires the implementation of all HAL interface functions. Detailed instructions on how to proceed with this task can be found in the EMSBench code repository at GitHub.

3.4 Timing properties

3.4.1 Execution Scenario

`ems` is executed using the new european driving cycle [13] for trace generation. The whole cycle consists of an urban and an extra-urban driving cycle. The urban cycle takes 195 s and is repeated four times, while the extra-urban cycle takes 400 s and is performed once. In total, the cycle takes 1,180 s (≈ 20 minutes). During the cycle, the revolution speed of the crankshaft ranges from $11.67\,\mathrm{s}^{-1}$ to $63.64\,\mathrm{s}^{-1}$ (700 rpm to ≈ 3820 rpm).

The counter frequencies of the C/C timers in both implementations were set such as to approximate the time base of the original FreeEMS implementation as closely as possible. In FreeEMS, the counter of the ECT is incremented each 0.8 µs. The same time base is also used for signal generation by the trace generator.

Algorithm 1 ISR for primary channel.

```
$k \leftarrow 0$                                          ▷ Global counter for primary teeth
procedure PrimaryISR
    if pin active then
        set deactivation time
        return
    else
        if $k == 0$ then                                  ▷ re-normalise
            $\omega_0 \leftarrow \omega(t)$
            $\varphi_0 \leftarrow 0$
            $t \leftarrow 0$
            if phase change pending then                  ▷ execute phase change
                $\alpha \leftarrow \alpha_N$
            end if
        end if
        calculate primary release time $t_P$
        set primary release time
        if $k == 1$ then                                  ▷ also prepare secondary channel
            calculate secondary release time $t_S$
            set secondary release time
        end if
        $k \leftarrow (k + 1) \mod n_p$
    end if
end procedure
```

3.4.2 Execution Behaviour

Most ISRs are executed recurringly according to different time bases. Based on the physical time, the `RTIISR()` is called each 125 µs, and performs actual calculations each 1 ms. All ISRs that deal directly with the engine are coupled to the time base that is generated by the rotary sensor of the crank- or camshaft. Their release frequencies during one revolution of the crankshaft are specified in Table 1. The `PrimaryRPMISR()` is released on each primary tooth, i.e. 12 times per crankshaft revolution. It performs actual work only on each second call, based on an internal counter. On odd teeth, only internal variables are advanced and debug outputs are set. On even teeth, also calculations for fuel injection and ignition are performed. The `SecondaryRPMISR()` is called once each crank revolution. Each `InjectionXISR()` is released twice, once for opening and once for closing of the injection valve. The `IgnitionDwellISR()` and `IgnitionFireISR()` are released six times per revolution, each pair activating a different ignition channel. The frequency of the crank-angle-triggered ISRs with respect to physical time depends on the current revolution speed of the crank shaft.

When the adjusted implementation is executed using the standardised driving cycle [13], no queuing of injection or ignition events can be observed. Instead, all timers are set and activated directly. Furthermore, all times are aligned such that injection and ignition is finished before the occurrence of the next `PrimaryRPMISR()`.

Figure 6 shows an exemplary execution sequence of ISRs for one fuel channel. Start times are derived from ticks of the global clock in the STM32F4-Discovery implementation. Execution on a different platform will yield similar results, if timer settings are kept similar. Due to abstraction from many inputs, most times are fixed. Only the closing of the injection valve (`Inj-L`) varies in

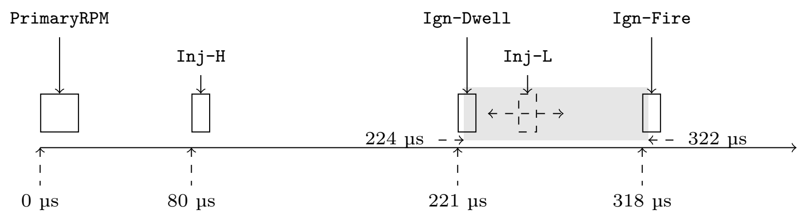

Figure 6 Exemplary sequence for one channel; Injection-Low start times vary inside the shaded interval (box widths are symbolic).

Table 1 Release frequency of crank-angle-triggered ISRs; CR frequency = based on one crankshaft revolution.

ISR	IRQ Source	Frequency	
		CR	s^{-1}
`PrimaryRPMISR()`	ECT-IC	12	144–768
`SecondaryRPMISR()`	ECT-IC	1	12–64
`InjectionXISR()`	ECT-OC	2 (×6)	24–128 (×6)
`IgnitionDwellISR()`	PIT	6	60–384
`IgnitionFireISR()`	PIT	6	60–384

time. Here, the behaviour of EMSBench deviates from a real EMS: Assuming the adjustments described in Section 3.1, one would rather expect the injection times to be constant and the ignition times to vary, as the latter clearly must depend on the engine's current speed. We could trace the depicted behaviour back to a fault in the original implementation of FreeEMS that we were not able to correct, and to some simplifying assumptions in our adjustment. As the goal of this work is to provide a benchmark that behaves *similar* to a real EMS, but not to really control an engine, we neglect this deviation.

Furthermore, the injection may overlap with the ignition event. Although this may seem strange in the first place, this behaviour is valid as injection is performed not directly into the cylinder but into the intake manifold. When an actual ignition occurs, fuel is already injected for the next cycle of the channel. From the diagram, one can observe that overlaps of ISRs probably may occur between the Injection-Low ISR and one of the ignition ISRs. This may pose a problem, as the ignition pins are managed by software inside the ISR and thus dwelling and firing could be deferred. A deeper discussion of this and similar problems can be found in Section 4.3.

4 Use of EMSBench

4.1 Execution-Time Measurements

In the following, we present the execution-time measurements that we have performed on the STM32F4-Discovery and our custom Nios II platforms. For each series of measurements, we used two identical hardware boards, one for trace generation and another to run `ems`. The counters of the C/C timers on both boards were configured to run with identical frequencies. For the `PrimaryRPMISR()`, two classes of measurements are shown: On *even teeth*, calculations for fuel and ignition are performed, while on *odd teeth* the ISR only basic management functions are executed. Similarly, the measurements for the `InjectorXISRs` distinguish the instances for opening and closing of the injection valves. The different instances of these ISRs are merged in these two classes.

Table 2 Measured execution times of ISRs and critical sections on the STM32F4-Discovery platform (clock cycles).

ISR	min	max	avg	med
`PrimaryRPMISR` (even teeth)	1403	1438	1415	1415
`PrimaryRPMISR` (odd teeth)	361	384	364	364
`SecondaryRMPISR`	275	291	275	275
`InjectorXISR` (open)	553	594	561	560
`InjectorXISR` (close)	508	537	518	516
`IgnitionDwellISR`	158	169	165	162
`IgnitionFireISR`	143	153	149	149
`RTIISR`	112	301	121	112
`main` (sample)	238	238	238	238
`main` (switch sensor data)	65	67	66	66
`main` (switch control data)	53	73	63	63

4.1.1 STM32F4-Discovery

The STM32F407 microcontroller (µC) on this platforms runs at a frequency of 168 MHz. The common clock for the timers has a period of 0.8 µs (125 kHz). Code is executed directly from on-chip Flash memory. Instruction prefetching and caching are disabled to ease the comparison of the measurements with static WCET analysis. Volatile data is stored in on-chip SRAM, and data caching is also disabled. The memory footprint of the `ems` is about 52 kB for code and about 49 kB for data.

Table 2 shows minimum, maximum, average and median execution times that were observed during one driving cycle on the STM32F4-Discovery platform. As code and data are loaded directly from Flash resp. SRAM memories, there is only a low variance in the execution times of the different ISRs. Compared to the other ISRs, the `PrimaryRPMISR` for even teeth has a very high execution time, as it performs a large number of calculations. Also, each `InjectorXISR` has to perform several calculations. The other ISRs execute only few calculations or, like the ISRs related to ignition, only set output pins, and thus have lower execution times. The observed execution time of the `SecondaryRPMISR` is mostly 275 cycles. The maximum value of 291 cycles was observed only once during the driving cycle. It represents a corner case due to error conditions like lost synchronisation between primary and secondary teeth. Concerning the `main` function, only the execution times of the critical sections during which IRQs are disabled are shown. These numbers stand for the release delay any ISR might experience.

The seemingly high variance in the execution times of the `RTIISR` is due to the different branches that can be taken during the ISR. For example, during most calls (7 out of 8), only a counter is increased. Each 8th call, i.e. each millisecond, the ISR additionally can perform periodic tasks (see Section 2.4.7). Depending on the number of periods that must be checked and the amount of work to be performed accordingly, the execution time increases. Table 3 shows the measured execution times for the different periods that are handled in `RTIISR`. Tasks with higher period include the work for lower-period tasks. As the numbers show, the seemingly high variance in the `RTIISR` execution times in Table 2 can be attributed to the different branches that are taken. Inside a single task class in `RTIISR`, the execution times are quite stable.

We must note that the execution times of the 1 s and 1 min paths differ only slightly by one cycle. This low difference is based on the structure of the assembler code that is generated by the compiler and the fetch and execution behaviour of the STM32F407 µC, and also on the fact that

```
                   // 1 s code ...
80063ca:           cbnz r6, 80063f2 ; branch if 1 min task should NOT be executed
                   // Now execute 1 min task ...
80063cc:           ldrh r0, [r2, #8]
80063ce:           strh r4, [r2, #14]
80063d0:           adds r3, r0, #1
80063d2:           strh r3, [r2, #8]
80063d4:           movs r4, #102 ; Execution path indicator (1 min)
80063d6:           b.n 80062e6 ; Jump to return block of function
                   ...
80063f2:           movs r4, #101 ; Execution path indicator (1 s)
80063f4:           b.n 80062e6 ; Jump to return block of function
80063f6:           nop
```

Figure 7 Assembler code of 1 min part in `RTIISR`.

Table 3 Measured execution times of `RTIISR` depending on execution path (clock cycles).

Period	min	max	avg	med
125 µs	112	112	112	112
1 ms	183	183	183	183
500 ms (via 1 ms)	203	203	203	203
100 ms	243	263	243	243
1 s	300	300	300	300
1 min	301	301	301	301

no actual work is performed in the 1 min path. Concerning fetching of instruction, the processor always loads a full line of 16 bytes from Flash memory. Thus, fetching from a new line incurs a longer latency (in our case 4 cycles), while all further instructions from the same line can be fetched immediately. Any time the 1 min path is taken, the 1 s part is also executed. The relevant part of the assembler code is depicted in Figure 7. The branch instruction at address `80063ca` decides whether the 1 min part is executed (branch not taken) or not (branch taken). On the one hand, if the branch is not taken, the first two instructions are already loaded from Flash, and can directly be fetched. Only at `80063d0` another long-latency fetch from Flash is necessary. On the other hand, if the branch is taken, i.e. the 1 min part is not executed, the processor incurs a branch penalty, and also must wait for the new line (containing instructions at `80063f2` and following) being loaded from Flash. Thus, the overheads of the different paths are lying in balance.

4.1.2 Nios II

The Nios II platform is deployed to a Cyclone II FPGA on a Terasic DE2-70 development board. The processor runs with a clock frequency of 50 MHz. It comprises 32 kB of L1 data and instruction cache each. We employ the *simple capture/compare timer (SCCT)* [9] which provides a global global counter and 8 C/C channels. The global counter of the SCCT is configured to run with 125 kHz. Code and data are both stored in off-chip SDRAM. On this platform, the memory footprint is about 36 kB for code and 52 kB for data sections. The large difference of the code size compared to the STM32F4-Discovery stems mainly from the use of a different board support package.

The execution times measured on the Nios II platform can be found in Table 4. Additionally, the table also shows the measured execution times of the first execution of each ISR/function

Table 4 Measured execution times of ISRs and critical sections on the Nios II platform (clock cycles).

ISR	first	min	max	avg	med
`PrimaryRPMISR()` (even teeth)	1316	732	1422	959	886
`PrimaryRPMISR()` (odd teeth)	322	290	809	419	306
`SecondaryRMPISR()`	238	180	352	239	238
`InjectorXISR()` (open)	462	315	462	347	345
`InjectorXISR()` (close)	290	255	337	274	273
`IgnitionDwellISR()`	112	65	112	68	67
`IgnitionFireISR()`	84	64	114	66	66
`RTIISR()`	170	66	280	89	82
`main()` (sample)	174	174	242	213	210
`main()` (switch sensor data)	35	35	93	37	38
`main()` (switch control data)	37	34	73	36	37

block, when it was executed with a cold cache. The first execution of each function is not always the one with the highest execution time, as in some cases a shorter path through the function may be taken. All execution times exhibit a greater variation due to the caches used on this platform. In terms of median or average execution time, the results are comparable to those from the STM32F4-Discovery platform, even though the ratio between any two ISRs may vary. In average, the execution times are lower due to the usage of SDRAM to store the code and caches.

4.2 Static WCET Analysis

Another suggested use of EMSBench is exercising static WCET analysis techniques and tools. In this section, we briefly review the main principles of static WCET analysis, then show how it might be applied to EMSBench. This is illustrated with some preliminary results.

4.2.1 Principles of Static WCET Analysis

The building of a valid scheduling of tasks in a real-time system relies on the knowledge of each task's worst-case execution time. In a system that runs tasks in isolation (i.e. where a task executes without being delayed due to resource sharing with any other piece of software), the execution time of a task only depends on (a) the initial state of the system (e.g. the contents of cache memories), and (b) the input data set.

Measurement-based timing analysis techniques require the selection of relevant input data sets: unfortunately, when the longest possible execution time is searched, it might be difficult to determine the worst-case input data, or to show that a given input data set leads to an execution time that is close to the WCET. In addition, initializing the hardware to any possible state before performing measurements is usually infeasible, while identifying the worst-case state might be complex. Static WCET analysis techniques instead abstract input data and derive an upper bound of the execution time that is valid for any input data within a domain that might be restricted by user-provided annotations (e.g. to express the range of a sensor outputs) and for any initial hardware state. The usual method to derive this upper bound is the Implicit Path Enumeration Technique [11] that considers short segments of code (basic blocks). It maximizes the execution time of the program defined as the sum of the individual execution times of basic blocks weighted by their respective execution counts, under some constraints on the possible execution flow (e.g. loop bounds, infeasible paths). Flow constraints can be provided as user annotations [26] and/or extracted automatically from the source or binary code [5, 12, 7]. The individual WCETs of basic blocks are derived from a model of the target hardware (this phase is often referred to as *low-level analysis*) [21, 17, 16]. The difficulty of getting detailed and reliable information on commercial platforms and to design accurate and safe models, is clearly the weak point of static WCET analysis and we were faced with this issue for the experiments we report in this paper.

Various WCET analysis tools, either commercial or academic, exist [24]. In this paper, we use the OTAWA toolset [1].

4.2.2 Methodology

In this section, we estimate WCETs considering the STM32F4-Discovery platform that features a single-core processor. If we assume a non preemptive scheduling scheme, only interrupts can interfere with tasks and impact their WCETs. Although the Cortex M4 processor does not include any standard instruction nor data cache, it features a mechanism designed to hide the latency of accesses to the Flash memory, the ART accelerator. It is based on specific small-size memory that behaves similarly to a cache memory. An interrupt routine might alter the contents of this memory and thus degrade the WCET of a task. However, interrupts are disabled during the execution of interrupt service routines, as well as during the execution of the three critical sections in the main function. As a consequence, we can safely consider that all the tasks and ISRs under analysis run in isolation.

The first step when performing the timing analysis of a task is to determine flow facts, primarily loop bounds and targets of indirect branches (used in switch-like statements). Since OTAWA does not support value analysis for the ARMv7 ISA, we had to annotate indirect branches with their possible targets. The code contains few loops which are easy to bound. The only additional flow facts that had to be specified are the direction of the two conditional branches that distinguish the even/odd case for `PrimaryRPMISR()` and the open/close case for `InjectorXISR()`.

The second step of static WCET analysis is to determine the local WCET of sequential pieces of code, i.e. basic blocks. OTAWA extracts the control flow graph (CFG) of the task under analysis from the binary code of the application using the indirect branch targets provided as flow fact annotations. Then, based on a model of the hardware architecture, it determines the worst-case execution cost of each basic block whatever the execution path before it. This model must reflect the pipeline architecture and the instruction latencies of the real hardware so that a valid abstract state of the processor after each basic block can be computed from the initial abstract state (when the fetching of the block into the pipeline starts), using the technique described in [18].

For the purpose of this paper, we have designed a model of the Cortex M4 processor featured by the STM32F4-Discovery board. This model was validated against measurements using a micro-benchmark that we designed to observe specific instruction latencies (taken branches, load and store accesses to the Flash and SDRAM memories, etc.) as well as the overhead due to the measurement process (enabling a timer, then reading it after the function under analysis has been executed). However, we did not model the ART Accelerator device used to hide part of the latency to the embedded Flash memory. Instead, we have considered the full Flash latency for each access to a new Flash line. Specific latencies for accesses to the registers of I/O devices and timers have been considered, based on their address ranges.

4.2.3 Static WCET Estimations

The WCET estimations, as well as the overestimation against the highest observed watermark (i.e. numbers given in Table 2), are reported in Table 5. It appears that the overestimation is reasonable (ranging from +11.9% to +41.1%) with respect to what is usually expected from static

Table 5 Estimated WCETs of ISRs and critical sections considering the STM32F4-Discovery platform (clock cycles).

ISR	WCET	overestimation
`PrimaryRPMISR()` (even teeth)	1695	17.9%
`PrimaryRPMISR()` (odd teeth)	542	41.1%
`SecondaryRMPISR()`	343	17.9%
`InjectorXISR()` (open)	668	12.5%
`InjectorXISR()` (close)	601	11.9%
`IgnitionDwellISR()`	228	34.9%
`IgnitionFireISR()`	199	30.1%
`RTIISR()`	343	14.0%
`main()` (sample)	304	27.7%
`main()` (switch sensor data)	88	31,3%
`main()` (switch control data)	97	32.9%

Table 6 Estimated WCETs of ISRs and critical sections considering the STM32F4-Discovery platform without ART vs. with a perfect ART (clock cycles).

ISR	without ART	with a perfect ART
`PrimaryRPMISR()` (even teeth)	1695	915
`PrimaryRPMISR()` (odd teeth)	542	254
`SecondaryRMPISR()`	343	192
`InjectorXISR()` (open)	668	375
`InjectorXISR()` (close)	601	325
`IgnitionDwellISR()`	228	118
`IgnitionFireISR()`	199	110
`RTIISR()`	343	216
`main()` (sample)	304	161
`main()` (switch sensor data)	88	54
`main()` (switch control data)	97	51

WCET analysis. However, it can still be considered as a bit high given that the target processor is very simple and time-predictable. However, not modelling the ART accelerator and assuming maximum latency for each access to the memory (either Flash or SRAM) is pessimistic and has a noticeable impact on estimated WCETs. Table 6 gives an insight into this impact by showing WCETs computed with a perfect (always hit) vs. without ART.

4.3 Interferences

An important aspect in real-time systems is a RTA, which helps to ensure that reactions happen in time and supports schedulability analysis. The response times of tasks or ISRs must not only take their execution times into account, but also possible interferences from other tasks/ISRs. In the following, we discuss the timing interferences that can occur in `ems`, and how they can influence reactions.

In the EMS implementation, the `main()` loop may interfere with the ISRs, and ISRs may interfere among each other (in terms of delaying each others execution). These interferences can delay the execution of an ISR. In the current implementation, the following interferences can occur (all numbers refer to the STM32F4-Discovery platform):

- Any ISR can be delayed through one of the three critical sections in the `main()` loop. In the first critical section, input signals from the ADCs are sampled. In our implementation, these I/O accesses are replaced by reading constants from memory. In the other two critical sections, only buffers for input resp. injection/ignition data are switched.
- The `RTIISR()` may delay the start of any other ISR and vice versa. The `RTIISR()` is triggered according to physical time each 125 µs. All important other ISRs are bound to the crankshaft rotation. Their trigger times and intervals change over time in accordance with the current rotation speed of the crankshaft. Occasionally, their activation/execution times may overlap, resulting in the ISR triggered later being delayed.
- As already noted in Section 3.4.2, the activation time of each second instance of any `InjectorX-ISR()` (which pulls the output pin back to low) varies inside a certain interval (see Figure 6). If the associated timer interrupt is triggered very early in the cycle, the execution of the `InjectorXISR()` can be delayed by the `IgnitionDwellISR()`. If it is activated very late, its execution may overlap with the activation of the `IgnitionFireISR()` and thus delay actual ignition. It may even happen that the `InjectorXISR()` is activated after the `IgnitionFireISR()`, in which case the `InjectorXISR()` is delayed.

Not all delays mentioned above have the same criticality. ISRs that are bound to a C/C timer channel can cope better with an execution delay: On an IC channel, the timestamp of the relevant event was already stored by the hardware. On an OC channel, the relevant pin output was already set by the hardware in time. Here, a delay may be critical, if the corresponding ISR has to set the channel's timer anew. The `IgnitionDwellISR()` and `IgnitionFireISR()` have a higher sensitivity to execution delays, as the output pins for ignition control are set by software. This means that e.g. delaying the execution of the `IgnitionFireISR()` will result in an ignition happening later. Here, it mainly depends on the actual system (EMS+engine), whether a certain delay is acceptable. Delays that are incurred by the `RTIISR()` can only have minor effects on the behaviour of the EMS. As already specified in the original FreeEMS implementation, tasks that must be performed periodically should be included in `main()` loop, while the `RTIISR` should only be used to set activation flags. A delay of the `RTIISR()` can lead to later execution of the relevant tasks, leading to a certain jitter. However, these tasks actually have to accept that they might be interrupted any time if they execute outside a critical section, and thus possible delays must be heeded during design.

To quantify the possible delay of an ISR more clearly, consider the `PrimaryRPMISR` which has the highest WCET of all tasks. The STM32F407 µC core runs at 168 MHz. Thus, the WCET of 1695 cycles corresponds to a time of $\frac{1695}{168\,\text{MHz}} \approx 10.09\,\mu\text{s}$. Assuming the maximum observed revolution speed of the crankshaft ($63.64\,\text{s}^{-1} \approx 3820\,\text{rpm}$), this time corresponds to an angle $\alpha = 10.09\,\mu\text{s} \cdot 63.64\,\text{s}^{-1} \approx 0.16°$. For the other critical sections, this angle is even smaller. In [25], ignition timing is varied in the range of -41° to 10° relative to the top dead point of the piston. Thus, we infer that the above deviation α is tolerable.

5 Existing Benchmarks and Related Work

Since the initial works on WCET analysis, a growing number of programs have been used as benchmarks. Earlier works have been evaluated considering short C programs inspired by algorithms described in [15] (Fast Fourier Transform, FIR filter, array sort, Fibonacci computation, etc.) and developed at the Singapore National University. Later, these benchmarks have been included in a larger collection at Mälardalen University [4]. This collection extends the former one with programs that exhibit more complex flow patterns, so that flow analysis techniques can be exercised. More recently, the TACLeBench collection has been released [3]. It gathers 55 re-formatted and versioned benchmarks.

The first report of using an industrial application as a WCET benchmark can be found in [8]. The application is on-board software for the Debie satellite instrument that measures impacts of small space debris or micro-meteoroids. The application consists of six tasks, including three interrupt service routines, that record information on debris hits and handle the reception of telecommands as well as the transmission of telemetry. This application is now available as open-source software[2] and has been considered in the last editions of the WCET Tool Challenge[3] for which it has been ported to Java. However, no input data generator is publicly available, which prevents from executing the benchmark to compare measured execution times to statically estimated WCETs, or from using measurement-based timing analyses.

PapaBench [14] is a benchmark built from Paparazzi, an open-source drone hardware and software project[4]. This application consists of two parts, *fbw* (fly-by-wire) that controls the drone in flight (engines and flap control, radio link with ground, IR sensor support, stabilization) and *autopilot* that controls the GPS and executes a flight plan. It is composed of about 20 tasks (including ISRs) that are statically scheduled. In contrast, the tasks (ISRs) in EMSBench are mostly triggered through external events or events that depend on the physical state of the system.

In [22], the authors argue that real-world applications might be too complex for existing academic WCET tools, mainly because of their complex flow structure. They introduce GenE, a benchmark generator, that provides flow fact annotations together with the generated code. Benchmarks are generated from code patterns that are commonly found in real-time applications. The idea is to focus on these patterns and to get rid of specific/unpredictable flow structures.

6 Conclusions

The work presented in this article was motivated by the fact that only few free benchmark programs for real-time systems exist that exhibit a behaviour similar to real applications. Widely spread benchmark suites consist usually of rather small, self-contained programs. More complex programs have so far only been reported from the aerospace domain [8, 14]. With the software package EMSBench we are undertaking a step beyond existing benchmarks to close this gap between actual real-time software and benchmark programs. In this article, we have described EMSBench and examined several of its use cases.

EMSBench is based on FreeEMS, an open source software for engine management, and was developed as a system benchmark for embedded real-time systems [10]. It consists of several ISRs and periodic tasks that are executed concurrently. Thus, it exhibits a behaviour that is significantly more *complex* than that of simple linear programs. Especially, some ISRs may interfere with other ones and delay their execution. The ISRs cannot be scheduled statically, as they must *react* upon events occurring in the phyical world with low latency. To *ease the use* of FreeEMS as a system benchmark, we have applied adjustments to the code and provide additional programs. We have removed most of the input dependencies in the `ems` part of EMSBench, and kept only the use of the crankshaft decoder. To allow for a realistic execution of `ems`, we provide a trace generator (`tg`) that emulates the behaviour of the crankshaft encoder according to arbitrary driving cycles. A HAL allows to adapt EMSBench to other hardware platforms.

[2] `http://www.tidorum.fi/debie1/`
[3] `http://www.mrtc.mdh.se/projects/WTC/`
[4] `https://wiki.paparazziuav.org`

We demonstrated the application of EMSBench with several use cases. Timing measurements were performed on two hardware platforms, the ARM Cortex-M4-based STM32F4-Discovery, and a self-designed Nios II-based FPGA microcontroller. A static WCET analysis of important parts of the EMS code from EMSBench was performed using the OTAWA toolset [1]. Reported WCET estimations considering the STM32F4-Discovery platform are above the longest observed execution times. Modelling the behaviour of hardware components that have been ignored in this first study should improve accuracy.

The results of the WCET analysis were used for an analysis of interferences that may occur between ISRs and periodic functions. Due to the low utilisation generated by the EMS on the STM32F4-Discovery platform, only minor interferences were identified. However, on platforms with less performance, the EMS will generate higher utilisation, and thus possibly more serious interferences might be found. It appears that EMSBench is a valuable benchmark for static WCET analysis tools. First of all, its structure is very similar to industrial applications that we could see in projects. Several modules (ISRs or tasks invoked in the main loop) can be analysed separately. Several of them exhibit different behaviours, depending on when they are executed (e.g. `PrimaryRPMISR()` is triggered either by an even or an odd tooth), which suggests that several scenario-related WCET values can be derived. In addition, the code contains a lot of indirect branches and a specific WCET could be computed for each possible target (e.g. for the `PrimaryRPMISR()` routine related to each injection channel, which should improve the accuracy of the overall WCET).

Due to its complexity, the use of EMSBench is not restricted to WCET benchmarking. If used with the accompanying trace generator, it can also act as a test program to evaluate other aspects of an execution platform. For example, schedulability aspects of an underlying operating system could be examined based on the results of a WCET analysis. To mimic the higher processor utilisation of industrial EMSs, some ISRs in EMSBench might be extended by code that synthetically increases the load. Such code could be based on signal processing algorithms like the Fast Fourier Transform, to, e.g. imitate software for knocking detection. Thus, higher interferences could be generated, allowing for a more challenging schedulability analysis. To allow for a more realistic execution of EMSBench, it would also be interesting to extend signal generation by a throttle signal. Thus, a higher degree of variance in the `ems`'s behaviour could be generated as the injection times would no longer be constant. Further works on EMSBench could include such extensions, but also fixing bugs and shortcomings that exist in the current version. The source code of EMSBench is available at `https://github.com/unia-sik/emsbench`. We encourage the research community to submit their own HAL implementations for EMSBench via GitHub to extend the useability of EMSBench.

References

1 Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. OTAWA: An Open Toolbox for Adaptive WCET Analysis. In Sang Lyul Min, Robert G. Pettit IV, Peter P. Puschner, and Theo Ungerer, editors, *Software Technologies for Embedded and Ubiquitous Systems – 8th IFIP WG 10.2 Int'l Workshop, SEUS 2010, Waidhofen/Ybbs, Austria, October 13-15, 2010. Proceedings*, volume 6399 of *Lecture Notes in Computer Science*, pages 35–46. Springer, 2010. `doi:10.1007/978-3-642-16256-5_6`.

2 Benchmark program and test bed for reactive embedded systems. GitHub repository. URL: `https://github.com/unia-sik/emsbench`.

3 Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sorensen, Peter Wägemann, and Simon Wegener. TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research. In Martin Schoeberl, editor, *16th Int'l Workshop on Worst-Case Execution Time Analysis, WCET 2016, July 5, 2016, Toulouse, France*, volume 55 of *OASIcs*, pages 2:1–2:10. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2016. `doi:10.4230/OASIcs.WCET.2016.2`.

4 Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET Benchmarks: Past, Present And Future. In Björn Lisper, editor, *10th Int'l Workshop on Worst-Case Execution Time Analysis, WCET 2010, July 6, 2010, Brussels, Belgium*, volume 15 of *OASIcs*, pages

136–146. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, Germany, 2010. doi:10.4230/OASIcs.WCET.2010.136.

5 Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Björn Lisper. Automatic Derivation of Loop Bounds and Infeasible Paths for WCET Analysis Using Abstract Execution. In *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS 2006), 5-8 December 2006, Rio de Janeiro, Brazil*, pages 57–66. IEEE Computer Society, 2006. doi:10.1109/RTSS.2006.12.

6 Jeff Hartman. *How to Tune and Modify Engine Management Systems*. Motorbooks Workshop. MBI Publishing Company, 2004.

7 Niklas Holsti, Jan Gustafsson, Linus Källberg, and Björn Lisper. Analysing Switch-Case Code with Abstract Execution. In Francisco J. Cazorla, editor, *15th Int'l Workshop on Worst-Case Execution Time Analysis, WCET 2015, July 7, 2015, Lund, Sweden*, volume 47 of *OASIcs*, pages 85–94. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2015. doi:10.4230/OASIcs.WCET.2015.85.

8 Niklas Holsti, Thomas Langbacka, and Sami Saarinen. Using a worst-case execution time tool for real-time verification of the DEBIE software. *European Space Agency Publications – ESA SP*, 457:307–312, 2000.

9 Florian Kluge. A Simple Capture/Compare Timer. Technical Report 2015-01, Department of Computer Science, University of Augsburg, June 2015. doi:10.13140/2.1.1251.2321.

10 Florian Kluge and Theo Ungerer. EMSBench: Benchmark und Testumgebung für reaktive Systeme. In Wolfgang A. Halang and Olaf Spinczyk, editors, *Echtzeit 2015*, Informatik Aktuell, pages 11–20. Springer, 2015. doi:10.1007/978-3-662-48611-5_2.

11 Yau-Tsun Steven Li and Sharad Malik. Performance Analysis of Embedded Software Using Implicit Path Enumeration. In Bryan Preas, editor, *Proceedings of the 32st Conference on Design Automation, San Francisco, California, USA, Moscone Center, June 12-16, 1995.*, pages 456–461. ACM Press, 1995. doi:10.1145/217474.217570.

12 Marianne De Michiel, Armelle Bonenfant, Hugues Cassé, and Pascal Sainrat. Static Loop Bound Analysis of C Programs Based on Flow Analysis and Abstract Interpretation. In *The Fourteenth IEEE Internationl Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2008, Kaohisung, Taiwan, 25-27 August 2008, Proceedings*, pages 161–166. IEEE Computer Society, 2008. doi:10.1109/RTCSA.2008.53.

13 COUNCIL DIRECTIVE of 20 March 1970 on the approximation of the laws of the Member States on measures to be taken against air pollution by emissions from motor vehicles. Version from 01.01.2007.

14 Fadia Nemer, Hugues Cassé, Pascal Sainrat, Jean Paul Bahsoun, and Marianne De Michiel. PapaBench: a Free Real-Time Benchmark. In Frank Mueller, editor, *6th Int'l Workshop on Worst-Case Execution Time (WCET) Analysis, July 4, 2006, Dresden, Germany*, volume 4 of *OASIcs*. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2006. doi:10.4230/OASIcs.WCET.2006.678.

15 William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 1992. 2nd edition.

16 Wolfgang Puffitsch. Efficient Worst-Case Execution Time Analysis of Dynamic Branch Prediction. In *28th Euromicro Conference on Real-Time Systems, ECRTS 2016, Toulouse, France, July 5-8, 2016*, pages 152–162. IEEE Computer Society, 2016. doi:10.1109/ECRTS.2016.23.

17 Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2):99–122, 2007. doi:10.1007/s11241-007-9032-3.

18 Christine Rochange and Pascal Sainrat. A Context-Parameterized Model for Static Analysis of Execution Times. *Trans. HiPEAC*, 2:222–241, 2009. doi:10.1007/978-3-642-00904-4_12.

19 STMicroelectronics. *UM1472 User Manual – Discovery kit for STM32f407/417 lines*. STMicroelectronics, November 2013.

20 Trace Generation in EMSBench. URL: https://github.com/unia-sik/emsbench/blob/master/doc/tg/tg.pdf.

21 Stephan Thesing. *Safe and precise WCET determination by abstract interpretation of pipeline models*. PhD thesis, Saarland University, Saarbrücken, Germany, 2004. URL: http://scidok.sulb.uni-saarland.de/volltexte/2005/466/index.html.

22 Peter Wägemann, Tobias Distler, Timo Hönig, Volkmar Sieh, and Wolfgang Schröder-Preikschat. GenE: A Benchmark Generator for WCET Analysis. In Francisco J. Cazorla, editor, *15th Int'l Workshop on Worst-Case Execution Time Analysis, WCET 2015, July 7, 2015, Lund, Sweden*, volume 47 of *OASIcs*, pages 33–43. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2015. doi:10.4230/OASIcs.WCET.2015.33.

23 Henning Wallentowitz and Konrad Reif, editors. *Handbuch Kraftfahrzeugelektronik: Grundlagen, Komponenten, Systeme, Anwendungen*. Vieweg, Wiesbaden, 2006.

24 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem – overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3):36:1–36:53, 2008. doi:10.1145/1347375.1347389.

25 Javad Zareei and Amir H. Kakaee. Study and the effects of ignition timing on gasoline engine performance and emissions. *European Transport Research Review*, 5(2):109–116, 2013. doi:10.1007/s12544-013-0099-8.

26 Jakob Zwirchmayr, Pascal Sotin, Armelle Bonenfant, Denis Claraz, and Philippe Cuenot. Identifying Relevant Parameters to Improve WCET Analysis. In Heiko Falk, editor, *14th Int'l Workshop on Worst-Case Execution Time Analysis, WCET 2014, July 8, 2014, Ulm, Germany*, volume 39 of *OASIcs*, pages 93–102. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2014. doi:10.4230/OASIcs.WCET.2014.93.