Real World Automotive Benchmarks For Free

Simon Kramer, Dirk Ziegenbein, Arne Hamann
Corporate Research
Robert Bosch GmbH
Renningen, Germany
{simon.kramer2|dirk.ziegenbein|arne.hamann}@de.bosch.com

The progress and comparability of real-time analysis methods that are applicable to real-world is slowed by the absence of realistic benchmarks, mainly due to intellectual property (IP) concerns. We propose a method that supports the generation of realistic but IP free benchmark sets. Further, we provide the application characteristics of a specific real-world automotive software system.

Keywords—benchmarks, timing analysis, automotive software

I. Introduction

A large quantity of innovative functionalities in modern automotive systems are realized using significant amounts of software technologies. As a consequence the job of integrating many different applications onto the same target platform has grown to a more and more complex and time consuming task. One important tool for guaranteeing the correctness of the dynamic behavior of the integrated system, especially for the cyber physical parts, is timing analysis.

There exists a large body of work in the domain of worst-case timing (or real-time) analysis. Each method assumes specific application and platform models addressing a subset of existing real-world timing effects. For cases where the application and platform models are simple enough, maximum utilization bounds can be derived to decide whether or not a given system adheres to some predefined real-time constraints. Prominent examples for this kind of analyses are, for instance, the work of Liu and Layland for independent periodic task sets under rate-monotonic fixed priority scheduling on a single core platform [1], or the work of Dertouzos on earliest deadline first (EDF) scheduling [2]. An overview of extensions on those basic works can be found under [3]. More complex application and platform models are addressed by approached based on the so-called busy window analysis combined with reasoning about the critical instant as proposed by Lehoczky [4]. As of today there exist proprietary industry strength tools based on that approach, such as SymTA/S, that are capable of analyzing most timing effects in current automotive systems. However, the detailed analysis techniques in those tools are not published, and thus, not accessible for the real-time research community.

As a result, there currently exist only few directly applicable tools and approaches that can cope with the complexity of the dynamic behavior in modern automotive systems. Therefore, simulation based methods are very popular albeit time consuming and inherently unsafe, which is unsatisfactory given the potential for front loading with formal analysis techniques.

Due to the introduction of multi-core execution platforms, the risk of divergence between academic research and industrial practice is currently increasing. The reason is the strongly increased problem space for timing analysis induced by multi-core systems.

Extending existing approaches is very challenging since the system structure and the dynamic system behavior of automotive systems is very complex. The reasons are manifold:

- Control of many physical processes with strongly varying dynamics
- Co-existence of sampled and reactive system parts
- Different time domains (e.g. crank angle, timers, incoming network traffic) leading to complex scheduling situations
- Numerous complex communication dependencies between functional entities in different time domains due to high coupling which is due to physical dependencies
- Sophisticated platform mechanisms influencing the dynamic behavior (cooperative tasks for saving stack space, automated copy mechanisms for data consistency, etc.)

There exist some tools that address the generation of synthetic applications for benchmark purposes that are worth being mentioned here. For instance, the *Task Graphs for Free (TGFF)* tool [8] generates a set of random independent task trees. Thereby, the structure of the generated task trees is very general, and could most likely be tweaked to fit the structure of automotive applications. This paper can help to do the mapping of the TGFF model to automotive application model including a sensible parameterization. However, TGFF lacks a model for memory accesses that can contribute (depending on the mapping decisions) massively to the execution times especially in multi-core systems.

Another tool for generating synthetic applications models is called *System Models for Free (SMFF)* [9]. The SMFF tool focuses on generating models that are “ready for scheduling analysis”. For that purpose, it generates (in contrast to TGFF) not only an application graph, but also a platform graph consisting of computational and communication resources.

Additionally, a mapping of task and communication links to the platform graph along with scheduling parameters is generated. Overall, it might be hard to use SMFF to generate benchmarks representing typical automotive system. The main reason is that important structural application and platform elements are not represented in the application and platform models.

The goal of this paper is to give an insight into the structure of typical automotive real-time software systems, along with challenges for research into relevant novel analysis techniques. To that end, the paper presents characteristics of an application with real-world complexity which is applicable to many control dominated application domains at Bosch. Based on these characteristics, real-time research groups are capable of generating expressive benchmarks for their real-time research without IP limitations.

II. Automotive Applications

A. *Structure*

In automotive systems software is structured into components, e.g. AUTOSAR [7] Software Components. Components which can’t be decomposed further are called atomic software components. Atomic software components finally contain runnables that are subject to scheduling, i.e. they have an associated activation pattern. Runnables with the same activation pattern are typically mapped to the same task. Thus task and activation scheme are used interchangeably in this paper. However, please note that multiple tasks with the same activation pattern may exist in one system. Tasks are finally scheduled by the OSEK [6] or AUTOSAR [7] operating system with fixed priorities. Both operating systems support so-called basic and extended tasks. Once started, basic tasks run to completion. During their execution they can only be pre-empted by higher priority tasks. Extended tasks are additionally allowed to wait upon events and can pass the thread of control to tasks with lower priorities. In engine control applications only basic tasks are used. Runnables communicate with each other either by sender-receiver communication or by client-server calls. In AUTOSAR these communications are generated by the Runtime Environment (RTE).

B. *Task Activation*

The activation pattern of tasks and thus of the corresponding runnables are manifold. For periodic recurring jobs, tasks are triggered time-synchronously according to the given period. Additionally, tasks can also be activated by asynchronous events. A special case of these events in engine management systems is the angle-synchronous activation according to the rotation of the crankshaft. Here the period of the tasks depend on the revolutions per minute (rpm) and the number of cylinders (#cyl) of the engine as given in (1).

$$\textit{period} = 120 \,/\, (\textit{rpm} * \#\textit{cyl}) \qquad (1)$$

Scheduling of tasks is based on fixed priorities. Furthermore, tasks are scheduled either in a fully preemptive or cooperative manner. Tasks that participate in preemptive scheduling can preempt every other task at any time, whereas tasks that are scheduled cooperatively interrupt each other only at runnable borders.

Runnables with the same activation scheme can also be grouped into multiple tasks, e.g. for separation or distribution purposes. For still guaranteeing a predefined order, a task can chain, i.e. activate, another task. Here the chaining task is terminated and the thread of control is handed over to the newly activated task. With this mechanism also inter-core activations are possible.

C. *Communication*

Communication is based on so-called *labels* that are read and written by runnables. Depending on the access specification a runnable can either read or write a label or do both. If a label is used for data exchange between runnables it is called a message.

For message access two different mechanisms are configurable. The first possibility, which is not the most common, is the so-called direct access (called explicit access in AUTOSAR) where the runnable directly reads the message from memory. More frequently the so-called implicit access is used, where a task-local copy for data access is created. The copying is performed at the beginning of the task and modified data is written back at the task’s termination. Using this mechanism the value of a message does not change during runtime of a task and all runnables operate on consistent data. Obviously, this mechanism influences the execution time of the task. For instance, frequent time consuming accesses to messages stored in the global memory are replaced by faster accesses to the copies stored in local memories. However, this increases the overall memory requirements, and in cases where copied messages are accessed only a few times, the overall task execution time might increase.

D. *Timing Requirements*

The basic timing requirement for all tasks in automotive systems is to finish execution before their deadline. For time-synchronous tasks with fixed periods, no overlapping executions and no backlog is allowed. This requirement translates to implicit task deadlines which are equal to the task periods. Angle-synchronous tasks have to finish before the injection or ignition take place. This requirement can be translated into a deadline which is equal to half the period (see Formula 1 in Section II.B).

Additional timing requirements are provided as end-to-end latency constraints for cause-effect chains that are critical for the functional behaviour. Examples for such constraints are 1) sensor to actuator latency constraints, where an actuator must be set within a maximum delay after a sensor value was read, or 2) fault reaction delays, where a certain action must be triggered within a maximum delay after a fault in the system is detected.

A cause-effect chain consists of multiple segments, each represented by two runnables that are connected by a read/write dependency over a label. The first runnable is called the stimulus, and the second the response of the segment. Please note, that for two consecutive segments it is required, that the response of the first segment is equal to the stimulus of the second segment.

E. Cost Model

For mapping of the software to a target platform, costs have to be added to the model. The level of detail that is necessary to describe costs is influenced by the target platform.

For single-core architectures the given execution time for runnables typically covers the complete execution including label accesses, but excludes scheduling effects like pre-emption. These have to be considered by analysis. In multi-core architectures this approach is not feasible as the execution time of a runnable also depends on the placement of the labels and the corresponding access time. So the execution time of a runnable is decomposed into code execution time and label access time.

The level of detail can be increased further e.g. with the following elements, but is not in scope of this paper:

- Code access times: The provided execution times assume that code is executed directly from flash without contention. Enhancements could consider contention or different placements, i.e. to a local scratch-pad memory.
- Fine granular interleaved modeling of instructions and label accesses: Enables a more detailed analysis of delays due to memory accesses.
- Modes and variants: Enables a more precise timing analysis, giving tighter real-time bounds.
- Cache related preemption delay (CRPD): The provided execution times already contain penalties due to caching effects. Since in current automotive microcontrollers caches are only used for flash memories that contain code and constant data, the increase in precision by performing a detailed CRPD analysis are expected to be rather small. Nevertheless, for future execution platforms it might be necessary to enhance the presented benchmark.
- Locking and synchronization methods: Additional delays due to locking and synchronization, such as spin-locks across cores, are not explicitly modeled and contained in the execution time.

To sum up, the benchmark presented in this paper makes some simplification to cope with complexity. Nevertheless, the level of detail is sufficient to achieve high accuracy for current execution platforms. Users of this benchmark are encouraged to extend the level of detail to fit their needs.

III. Benchmark Generation Method

The dominant reason for the lack of real-world application examples is intellectual property (IP) protection. While the exact functionality that is performed by an application does not need to be given for a timing benchmark, even the provision of the required information as detailed in Section II reveals enough insight on critical IP such that industry is not willing to provide it.

The idea we propose in this paper, is to define a set of application characteristics which is abstract enough to not reveal IP but nevertheless allow creating realistic application benchmarks from it. In Section IV, we propose such a set of application characteristics.

We have implemented an extraction tool that computes the concrete application characteristics for an AMALTHEA model of a real-world application. AMALTHEA [5] is a system model which has been developed in an ITEA project. At Bosch, we use AMALTHEA for timing analysis and optimization algorithms as well as for timing model exchange with customers.

Based on concrete application characteristics, we are able to generate benchmark applications which we can easily hand out to external partners without IP protection concerns. The overall flow is shown in Figure 1.

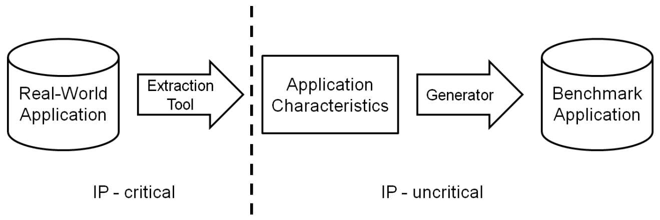

Figure 1: Flow and Elements of Benchmark Generation

An added benefit of this approach is that it supports the generation of a suite of several benchmarks with similar structure and elements. Furthermore, scaling benchmarks to reflect future growth of applications can be easily achieved by scaling the application characteristics. This reduces the risk that scheduling or optimization algorithms are fine-tuned to a single application.

IV. Application Characteristics

This section explains the various typical application characteristics in automotive systems and gives concrete value ranges derived from a real-world engine control application.

A. Labels (Number, Size, Mode Of Access)

Labels in automotive applications can be divided into three categories: atomic data types, arrays and structures, as well as interpolation curves and maps. Most labels have an atomic data type, i.e. 1, 2 and, 4 bytes. This group already covers over 90% of all labels. The most complex application in terms of labels is combustion engine control requiring between 10000 and 50000 labels, depending on the implemented features. The distribution of the label sizes for an exemplary engine control application is shown in TABLE I.

Additionally to the label size, also the access types need to be specified. A label can either be only read (parameters), only written (measurement points), or both (communication between runnables).

TABLE I. DISTRIBUTION OF LABEL SIZES

Size (byte)	Share
1	35 %
2	49 %
4	13 %
5 - 8	0,8 %
9 - 16	1,3 %
17 - 32	0,5 %
33 - 64	0,2 %
> 64	0,2 %

Labels that are only read are constants that are usually stored in non-volatile memory such as flash. Specialties of constants are interpolation curves and maps. Those items are not read as a whole, but only the relevant range is accessed, depending on the value to be interpolated.

Labels which are only written are called measurement points. A measurement point is gathering data, which can be read by external tools during development to show internal states.

The partitioning of all accesses is as follows:

- Read-only : 40%
- Write-only: 10%
- Read-Write: 50%

B. Intra-/Inter Task Communication

Read/write labels are used to realize the communication between runnables. Here, three different cases can be distinguished:

- Forward intra-task communication
- Backward intra-task communication
- Inter-task communication

Intra-task communication implies that both communicating runnables are mapped to the same task. Forward communication is the subset of those communications, where the writer runs prior the reader so the flow of information is direct, without a huge delay. Backward communication is the opposite. Here the information always has a delay of one task instance. Inter-task communication is, when both communicating runnables are mapped to different tasks. Here the delay depends on multiple factors such as scheduling, employed communication scheme, and memory mapping.

The amount of communications per task heavily varies. But the percentage of each type is similar over all tasks. Forward and backward intra-task communication occurs in 25% and 35% of all cases, respectively. Inter-task communication is the most frequent type with a share around 40%.

The characteristic of Inter-Task communication is shown in TABLE II. The sending tasks are listed in the rows. The receiving tasks are listed in columns. The colour codes the amount of communications with the following legend:

I	II	III	IV	V	VI
<10	10-50	51-100	100-500	501-1000	>1000

TABLE II. INTER-TASK COMMUNICATION

Period	1 ms	2 ms	5 ms	10 ms	20 ms	50 ms	100 ms	200 ms	1000 ms	sync
1 ms				I	I		I			I
2 ms				I	I		I			
5 ms		I	IV	IV	II	II	I			
10 ms	II	II	II	VI	IV	II	IV	II	III	IV
20 ms	I	I	I	IV	VI	II	IV	I	II	IV
50 ms			II	II	II	III	I			
100 ms		I	I	V	IV	II	VI	II	III	IV
200 ms				I	I		I	I	I	
1000 ms				III	II		III	I	IV	I
Angle-sync	I	I	I	IV	IV	I	III	I	I	V

C. Runnables (Number, Activations)

Runnables that are mapped to tasks as explained in section II.B share the same activation. Most Runnables are triggered time-synchronously in predefined periods. Special cases in engine control are runnables which are scheduled synchronously to the rotation angle of the crankshaft. Furthermore, some runnables are triggered sporadically by external events (e.g. interrupts) whose activation pattern can be modelled as arbitrary arrival curves. TABLE III lists all common periods used in engine management systems and gives their percentage of the share of all runnables.

The total number of runnables depends, similar to the number of labels, on the implemented features. The typical range is between 1000 and 1500.

TABLE III. RUNNABLE DISTRIBUTION AMONG PERIODS

Period	Share
1 ms	3 %
2 ms	2 %
5 ms	2 %
10 ms	25 %
20 ms	25 %
50 ms	3 %
100 ms	20 %

Period	Share
200 ms	1 %
1000 ms	4 %
angle-synchronous	15 %

D. *Runnable Execution Times*

The execution times of runnables are specified with minimum (best-case), average, and maximum (worst-case) values. The distribution of the execution times can be approximated with a Weibull distribution. Please note that the given execution times exclude the mapping-dependent label access delay, but include delays due to a specific code placement. The former represents one of the analysis challenges, especially for multi-core systems.

TABLE IV shows the distribution of the average execution times (ACET) of runnables in each task. The Min-value denotes the shortest ACET and the Max-value denotes the longest ACET of a runnable within the given task. During benchmark generation the average over all generated ACET has to match the Avg. ACET of the chosen task.

TABLE IV. RUNNABLE AVERAGE EXECUTION TIMES

Period	Average Execution Times in µs		
	Min.	Avg.	Max.
1 ms	0,34	5,00	30,11
2 ms	0,32	4,20	40,69
5 ms	0,36	11,04	83,38
10 ms	0,21	10,09	309,87
20 ms	0,25	8,74	291,42
50 ms	0,29	17,56	92,98
100 ms	0,21	10,53	420,43
200 ms	0,22	2,56	21,95
1000 ms	0,37	0,43	0,46
angle-synchronous	0,45	6,52	88,58
Interrupts	0,18	5,42	12,59

TABLE V specifies factors that describe the relationship of the chosen base ACET to the minimum (f_{min}) and maximum (f_{max}) best-case (BCET) and worst-case execution times (WCET) of runnables in each task as depicted in Figure 3. By choosing random factors between f_{min} and f_{max}, the BCET and WCET for each runnable in a task can be calculated.

As mentioned in II.B, there exist also runnables that are sporadically triggered by interrupts. Obviously, the load induced by those runnables has to be considered in system design and analysis. In total, interrupts add about 30 percent load to the overall system utilization. The occurrence of these interrupts can be modeled by arbitrary arrival curves.

TABLE V. FACTORS FOR DETERMINING RUNNABLE BEST- AND WORST-CASE EXECUTION TIMES

Period	Best		Worst	
	f_{min}	f_{max}	f_{min}	f_{max}
1 ms	0,19	0,92	1,30	29,11
2 ms	0,12	0,89	1,54	19,04
5 ms	0,17	0,94	1,13	18,44
10 ms	0,05	0,99	1,06	30,03
20 ms	0,11	0,98	1,06	15,61
50 ms	0,32	0,95	1,13	7,76
100 ms	0,09	0,99	1,02	8,88
200 ms	0,45	0,98	1,03	4,90
1000 ms	0,68	0,80	1,84	4,75
angle-synchronous	0,13	0,92	1,20	28,17
Interrupts	0,12	0,94	1,15	4,54

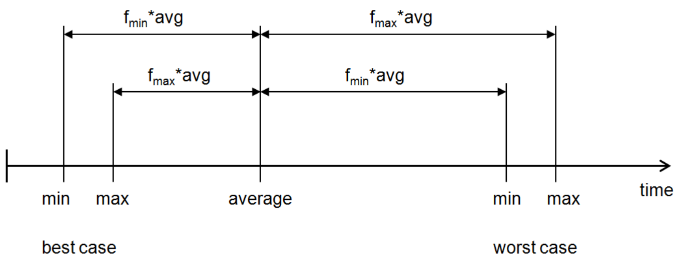

Figure 3: Runnable Execution Time calculation

Please note that the given values already assume a multicore architecture. Adaptations to smaller and bigger platforms can be achieved by scaling the given values. This can be done either by scaling the total number of runnables, or by scaling the execution times of the runnables.

E. *Cause-Effect Chains*

As explained in section II.D, end-to-end latency constraints for critical cause-effect chains are additional timing requirements imposed on an engine control application. Most cause-effect chains only contain runnables with the same activation pattern. On a single-core platform, those are usually grouped into the same task. However, there are also cause-effect chains spanning multiple activation patterns. These chains usually involve transitions from the time-synchronous to the angle-synchronous domain and vice-versa, i.e. for conversion of requested torque from the driver (sampled time-synchronously) into injection mass and time for the engine (angle-synchronous).

In typical engine control applications there are between 30 and 60 cause-effect chains that are critical for the functional behavior. The shares for the number of involved activation patterns per cause-effect chain can be found in Table VI. The number of runnables in each of those activation patterns is given in Table VII.

TABLE VI. INVOLVED ACTIVATION PATTERNS PER CAUSE-EFFECT CHAIN

Involved Activation Patterns	**Share**
1	70 %
2	20 %
3	10 %

TABLE VII. RUNNABLES PER ACTIVATION PATTERN

Number of Runnables	**Share**
2	30 %
3	40 %
4	20 %
5	10 %

The end-to-end latency constraint for a cause-effect chain can be obtained by adding the periods of the involved activation patterns. For instance, if a cause-effect chain consists of 3 runnables with a *10 ms* period and *2* runnables with a *20 ms* period, the resulting constraint is equal to *30 ms*. Please note, that there are no cyclic transitions between two activation patterns (e.g. *10 ms* → *20 ms* → *10 ms*) within a cause-effect chain.

V. CHALLENGES

In order to be applicable in the design process of automotive software systems, future real-time analysis techniques must address the following analysis challenges:

1. *Precise analysis of worst-case end-to-end latencies along complex cause-effect chains*: the analysis of task response times in not sufficient for automotive applications since relevant cause-effect chains that are subject to timing constraints usually span over different tasks and time domains.

2. *Interleaved WCET and WCRT analysis for memory accesses*: with the advent of multi-core execution platforms, the classical share of work between WCET and WCRT analysis is not valid any longer. Since automotive applications extensively communicate over (different) shared memories, the access times including arbitration have to be considered in a combined WCET and WCRT analysis.

3. *Automatic optimized application mapping:* as can be derived from challenge 2, the mapping of runnables to tasks to cores along with the mapping of labels to memory locations have a huge influence on whether or not a given system adheres to real-time requirements. Since the design space cannot be handled manually based on experience, tool support for guiding or (partly) automating the mapping is needed.

4. *Evaluation of digital (multi-core) execution platforms:* the application characteristics presented in this paper can be used to evaluate the suitability of digital execution platforms. Especially the memory layout seems an interesting object of investigation.

VI. CONCLUSION & OUTLOOK

The application characteristics proposed in this paper aim at providing realistic but IP-free benchmarks to the real-time community. We are aware that especially in the area of execution time modeling the characteristics are rather simplistic but should suffice as a starting point.

We certainly admit that the presented benchmark is not really “for free” but requires work for creating the benchmark from the presented characteristics. Thus,we currently evaluate whether we can make tools mentioned in Section II available as open source in context of the AMALTHEA project.

References

[1] C. L. Liu and J. W. Layland, “Scheduling algorithms for multiprogramming in a hard real-time environment,” in Journal of the ACM, vol. 20(1), 1973, pp. 46–61.

[2] J. M. L. Dertouzos, “Control robotics: the procedural control of physical processes,” in Proceedings of the IFIP Congress, 1974, pp. 807–813.

[3] L. Sha, T. Abdelzaher, K.-E. Årzén, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. K. Mok, “Real time scheduling theory: A historical perspective,” Real-Time Systems, vol. 28, no. 2-3, pp. 101–155, Nov. 2004.

[4] J. P. Lehoczky, “Fixed priority scheduling of periodic task sets with arbitrary deadlines,” in Proceedings of IEEE Real-Time Symposium (RTSS), 1990, pp. 201–209.

[5] H. Mackamul, “AMALTHEA - An Open Tool Platform for Embedded Multicore Systems”, EclipseCon Europe 2013, Ludwigsburg, Germany, Oct 2013.

[6] OSEK VDX, “Open systems and the corresponding interfaces for automotive electronics,” http://www.osek-vdx.org.

[7] AUTOSAR, “Automotive Open System Architecture”, http:// www.autosar.org.

[8] R.P. Dick and W. Wolf. “TGFF: Task Graphs for Free”, International Workshop on Hardware/Software Codesign (CODES/CASHE), 1998, pp. 97-101.

[9] M. Neukirchner, S. Stein, and R. Ernst. „SMFF: System Models for Free”, 2nd International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS), 2011.