Multi-source and multicore automotive ECUs - OS protection mechanisms and scheduling

Nicolas Navet*‡, Aurélien Monot*†, Bernard Bavoux†, Françoise Simonot-Lion*
*LORIA - Nancy Université, BP 239, 54506 Vandoeuvre, France, {firstname.name}@loria.fr
†PSA Peugeot Citroën, Route de Gisy, 78 943 Vélizy-Villacoublay, France, {firstname.name}@mpsa.com
‡RealTime-at-Work (RTaW), 615 rue du Jardin Botanique, 54506 Vandoeuvre, France,{firstname.name}@realtimeatwork.com

***Abstract*—As the demand for computing power is quickly increasing in the automotive domain, car manufacturers and tier-one suppliers are gradually introducing multicore ECUs in their electronic architectures. Additionally, these multicore ECUs offer new features such as higher levels of parallelism which ease the respect of the safety requirements such as the ISO 26262 and the implementation of other automotive use-cases. These new features involve also more complexity in the design, development and verification of the software applications. Hence, OEMs and suppliers will require new tools and methodologies for deployment and validation. In this paper, we review the operating system protection mechanisms (e.g., memory, timing), needed for multi-source software in a safety critical context, with a clear focus on AUTOSAR OS which is the upcoming de-facto standard for automotive ECUs. Then, we identify the main use-cases for automotive multicore ECUs and present solutions for the scheduling in a context where there are hundreds of software components and only a few OS tasks are allowed. Finally, experiments aim to assess the load level that can be reached on realistic case-studies.**

I. Introduction

Multi-source software running on the same ECU (Electronic Control Unit) is becoming increasingly widespread in the automotive industry. One of the main reasons being that OEMs want to reduce the number of ECUs which grew up above 70 for high-end cars. A major outcome of the AUTOSAR initiative, and more specifically of its operating system, is indeed to help OEMs shift from the "one function per ECU" paradigm to more centralized architecture designs by providing appropriate protection mechanisms.

Another crucial evolution in the automotive industry is that chip manufacturers are reaching the point where they can no longer cost-effectively meet the increasing performance requirements through frequency scaling alone. This is one reason why multicore ECUs are being gradually introduced in the automotive domain. Those multicore platforms offer also additional benefits, such as higher level of parallelism allowing for more segregation, which may help to meet the upcoming ISO 26262. Now, the challenge is to adapt existing design methods to the new multicore constraints. The scheduling of the software components is one of the key issues in that regard and it has to be revamped.

Introduction of multi-source and multicore will induce drastic changes in the software architecture of automotive ECUs. The aim of this paper is to provide a review of OS protection mechanisms, needed for multi-source software in a safety critical context, with a clear focus on AUTOSAR OS which is the upcoming de-facto standard for automotive ECUs. We then identify the main use cases for multicore ECUs and eventually focus on one of them. Precisely, we address the problem of scheduling numerous elementary software components, called runnables, on a limited set of identical cores. In the context of an automotive design, we assume the use of the static task partitioning scheme which provides simplicity and better predictability for the ECU designers by comparison with a global scheduling approach. We show how the global scheduling problem can be addressed as two sub-problems: partitioning the set of runnables and building the schedule on each core. Then, we prove that each of the sub-problems cannot be solved optimally due to their algorithmic complexity. We then present low complexity heuristics to partition and build a schedule of the runnable set on each core before discussing schedulability verification methods. Finally, we assess the performance of our approach on a case-study.

II. Automotive OS robustness

It is of major interest for car manufacturers to be able to re-use software components, not to mention the corresponding calibration parameters, in order to save time and reduce costs. On the other hand, car manufacturers, and ECU integrators at large, may want to be able to select software components on the market according to the best performance/cost ratio (e.g., AUTOSAR basic software modules). Both re-use and multi-source software are only possible if the operating system provides protection mechanisms to ensure segregation between software components and fault-confinement when needed (see [1] for a review of the AUTOSAR OS protection mechanisms). In addition, these mechanisms will enable the ECU integrator to accept responsibility because it is ensured that externally supplied software modules will not jeopardize the functioning of the whole ECU.

It has been decided by the automotive industry that OS protection mechanisms, and more generally the whole execution platform, would be outside the scope of competition and thus could be standardized, which is being done in the AUTOSAR consortium. These mechanisms protects against resource confiscation (memory, CPU, drivers, etc) and misuse of OS services (forbidden service calls, or calls issued in

wrong contexts, etc). AUTOSAR OS defines *4 scalability classes*, with different features requirements, to meet a wide range of needs at the best cost.

AUTOSAR introduces the concept of *OS-application* which is a functional unit made of *OS-objects* (tasks, Interrupt Service Routine, schedule table, alarm, etc). OS-applications can be *trusted* or *non-trusted*. In the former case, the code of the OS-application is executed in privileged mode and has unrestricted access to the OS API and hardware resources. Besides, trusted OS-applications can provide so-called trusted functions that can be called in a non-trusted context. Non-trusted applications do not have these capabilities, neither can they be executed without the OS protection mechanisms set at run-time.

The scope of protection is, depending on the situation, the OS-application, a task within an application or an ISR that is under the control of the OS. However, there is no specific protection at the runnable level. When a protection error is detected, the OS calls a system-wide *ProtectionHook* that can terminate the faulty task or the OS-application, then optionally restart the OS-application, reboot the ECU or call an OS-application specific *ErrorHook*.

A. Memory protection

AUTOSAR specifies a basic stack monitoring mechanism that does not require an MPU (Memory Protection Unit). This best-effort scheme is intended to detect an abnormal usage of the stack by a task. However, verifications are performed only at context-switches which might be too late in many cases.

For microcontrollers equipped with an MPU, there are more powerful protection mechanisms that are mandatory in the highest scability classes (3 and 4). The principle is that only write protection is imposed while read and execution protections are optional. The stack of a task/ISR can be protected at the OS-application level or at the task level (i.e., prevent other tasks of the same OS-application to manipulate the stack), the same holds true for data sections. Code sections either belongs to a specific OS-application or are shared between OS-applications.

There are two main strategies for segregating software modules. The first is to use distinct OS-applications but this is not always possible since the standard merely requires the availability of two OS-applications for the time being. The second strategy is to implement the modules as distinct tasks of the same OS-application with the drawback that some OS-objects (such as alarms, peripherals, etc) are not, or weakly, protected.

B. Temporal protection

AUTOSAR, in its highest scalability class, offers advanced timing protection mechanisms that require the availability of a hardware counter. The *execution time protection* guarantees that a task or an ISR will not be executed for more than a statically configured *execution budget*. *Locking time protection* ensures that a task/ISR does not hold resources or disable interrupts for more than its *lock budget*. Finally, *interarrival time protection* guarantees that a task/ISR will not be activated or resumed more often than its *time-frame*.

C. Service protection

Service protection means that there is a protection against incorrect OS system calls. For instance, parameters might be wrong, or the call is issued in an incorrect context (e.g, *terminateTask()* within an ISR) or the issuer may not have enough rights to call a certain service (e.g., *shutdownOS()*) or to manipulate a certain resource.

It should be pointed out that virtualization technologies offer other possibilities, complementary to the ones offered by Autosar, to ensure efficient protection between software components. While virtualization has been used for a long time in the consumer electronics or in the avionics field, the automotive industry is only starting to consider its use. The reader interested in virtualization in an automotive context may consult [2].

III. Scheduling of software components - From monocore ECUs to Multicore ECUs

In this section, the main use cases for multicore ECUs that we can foresee are described, we then discuss the way we envision multicore scheduling in automotive ECUs.

A. Main use cases for multicore ECUs

There exist very distinct hardware and software architectures for multicore ECU platforms. As far as hardware is concerned, suppliers envision various multicore architectures: identical cores, heterogeneous cores with different operating speeds and instruction sets and, possibly, various I/O and memory structures. However, chip manufacturers have been producing multiprocessor cores with identical cores for the PC industry for years which may influence the automotive industry as those architectures are proven in use and are likely to be cheaper thanks to mass production. In this section, we discuss the main use cases for a multicore ECU and implementation solutions that would properly fit them.

1) Decreasing the complexity of the architecture : The higher level of performance provided by multicore architectures allows to simplify in-vehicle architectures by executing on multiple cores the software previously run on multiple ECUs. This possible evolution towards more centralized architectures is also an opportunity for OEMs to decrease the number of network connections and buses. This means that parts of the complexity will be transferred from the E/E architecture to the hardware and software architecture of the ECUs. Furthermore, static cyclic scheduling allows to easily add functions/runnables on an existing ECU. However, in practice, important architectural shifts are hindered by the carry-over of ECUs and existing sub-networks which is widely used by generalist car manufacturers. The extent to which more centralized architectures will be adopted remains thus unsure.

2) Dealing with resource demanding applications: Multicore ECUs bring major improvements for some applications requiring high performance such as high-end engine controllers and real-time image processing applications. This use case does not require any particular hardware feature and identical cores are more likely to be used to meet high performance requirements. In these applications, one takes advantage of the possibility to parallelize jobs on multicore architecture. Typically, the same application can be executed on different cores to process different parts of a same data set in a parallel manner.

3) Improving the safety: Multicore architectures provide efficient ways to implement safety mechanisms. We identify three main methods to improve safety taking advantage of the multicore architecture. The first method consists in segregating trusted code and non trusted code on different cores. For instance, a car manufacturer may consider the software provided by suppliers as non-trusted code, or an ECU integrator may consider the car manufacturer's code as non-trusted for responsibility reasons. This isolation between software components requires strong protection mechanisms for memory, CPU time and the other shared resources, as they are now provided by Autosar OS, or, as they could be provided by virtual machines [2].

The second method consists in executing safety critical software components in a redundant manner, possibly with a system of vote choosing the output given by a majority of the duplicated runnables. It is possible to duplicate the whole set of software components allocated to a core on another core, or only the most critical runnables in order to find a trade-off between safety and computational requirements. To further increase the safety, N-Version Programming (NVP) can be employed: multiple versions of the same runnables, developed by different suppliers, are executed in parallel, instead of executing copies of the same implementation.

Finally, multicore architectures enable easier implementation of function monitoring. In this case, the proper execution of some functions on one core can be monitored from another core. It should be noted that higher levels of safety can be achieved by the usage of several distinct microprocessors instead of several distinct cores on the same microprocessor such as done in the E-Gas framework for engine controllers, though those kind of solutions are more expensive to implement.

4) Dedicated use of cores: Finally, another important use case taking advantage of a multicore ECU consists in using a core to handle specific low-level services. In the context of Autosar OS, a core could serve as a dedicated I/O controller, execute the communication stack or the whole set of basic software modules, while some other core would only take care of applicative level software components. For instance, a core can be used to run the time-triggered application while a second core handles the interruptions as well as the event-triggered runnables such as done in the PharOS project[3] on a SX12E micro-controller.

B. Static cyclic and fixed priority scheduling

Static cyclic scheduling of elementary software components, or runnables, is common because they are usually many more runnables that the maximum number of tasks allowed by automotive operating systems such as OSEK/VDX or AUTOSAR OS. For this reason, runnables must be grouped together and scheduled within a sequencer task (also called dispatcher task). In this paper, we focus on how to schedule large runnable sets on multicore platforms using a static partitioning approach. Indeed, the static task partitioning scheme is very likely to be adopted at least in a first step because it is conceptually simple and provides a better predictability for the ECU designers by comparison with a global scheduling approach. One aims to develop practical algorithms, whose performances can be guaranteed, to build the dispatcher tasks on each core and to schedule the runnables within these dispatcher tasks so as to respect sampling constraints and, as far as possible, uniformize the CPU load over time. This latter objective is of course important to minimize the hardware cost and to facilitate the addition of new functions, as typically done in the incremental design process of OEMs.

C. Partitioning scheduling scheme

In a multicore system, the tasks are either statically allocated to the cores or they can be distributed dynamically at run-time to balance the workload or migrate functions to increase availability. The later approach involves complex task and resource interactions which are difficult to predict and validate. For this reason, approaches relying on static allocation (*i.e.*, partitionning) and deterministic mechanisms such as periodic cyclic scheduling are more likely to be used in the automotive context and this is the option taken within the AUTOSAR consortium. Scheduling tasks on a multi-processor systems under the static partitioning approach has been well studied for a long time, see for instance [4] and [5], [6], [7], [8]. However, the works we are aware of deal with online algorithms such as FPP or EDF, and do not consider the static cyclic scheduling of tasks. The configuration algorithms developed in this paper are closely related to [9] (mono-processor scheduling of tasks with offsets) and [10] (scheduling of frames with offsets) but it is applied to multi-core and goes beyond as we provide lower-bounds on the performances. As the problem is of practical interest in the industry, there are in-house tools at the OEMs as well as commercial tools, such as RTaW NETCAR-ECU [11], that have been developed for configuring the scheduling. However, the proprietary algorithms used in these tools can usually not be disclosed and they are sometimes specialized for some specific usage.

IV. MULTICORE SCHEDULING ALGORITHMS

In this section we present algorithms, and when possible derive lower bounds on their efficiency, to schedule large numbers of runnables on multicore ECUs. These algorithms are especially suited to the first of the use cases we identified for multicore architectures, that is to permit a reduction of

the number of ECUs in the E/E architecture by using more powerful ECUs.

Since automotive OSs can only handle a limited amount of OS-tasks, the scheduling of runnables has to be done within dispatcher tasks. A first step of the approach is to partition the runnable sets onto the different cores. The next and last step consists in determine the offsets between the runnables allocated on each core so as to balance the load over time.

A. Model description

In this study, we consider a large set of *n* periodic elementary software components, also called runnables, that are to be allocated on an ECU consisting in *m* identical cores. In practice, a runnable can be implemented as a function called, whenever appropriate, within the body of an OS task.

1) Runnable characteristics: The ith runnable is denoted by $\mathcal{R}_i = (C_i, T_i, O_i, \{R\}, P_i)$. Quantities C_i, T_i and O_i correspond respectively to the Worst-Case Execution Time (WCET), the period and the offset of the $\mathcal{R}_i$. The offset of a runnable is the release date of the first instance of that runnable, subsequent instances are then released periodically. The choice made for the offset values has a direct influence on the repartition of the workload over time.

A set of inter-runnable dependencies is denoted by $\{R\}$. Indeed, due to specific design requirements, such as shared variables, some runnables may have to be allocated on the same core and the set $\{R\}$ is used to capture those constraints. In addition, some specific features, as I/O ports being located on a given core, may require a runnable to be allocated onto a specific core. This locality constraint is expressed by P_i.

2) Dispatcher task: Runnables are scheduled on their designated core using a dispatcher task, or "sequencer task", that stores the runnable activation times in a table and releases them at the right points in time. A dispatcher task is characterized by the duration of the dispatch table T_{cycle} that is executed in a cyclic manner[1], and by a quantum T_{tic} which is the duration of a slot in the table. For instance, typically, one may have $T_{cycle} = 1000ms$ and $T_{tic} = 5ms$. It should be noted that T_{cycle} must be a multiple of the *gcd* of the runnable periods and the *lcm* of these periods must be a multiple of T_{tic}. As a result, a dispatch table holds T_{cycle}/T_{tic} slots.

3) Assumptions: In this paper, we place a set of working assumptions, which, in our experience, can most often be met in today's automotive applications:

- Each runnable are executed strictly periodically. As a result, the whole trajectory of the system is defined by the first activation times of the runnables (*i.e.*, their offsets).
- The runnables are assumed to be offset-free, in the sense that the offset of a runnable can be freely chosen in the limit of its period (see [9]). Those offsets will be assigned during the construction of the dispatch table with the objective to uniformize the CPU load over a scheduling cycle.
- The worst case execution times of the runnables are assumed to be small compared to T_{tic}. Typical values for the case we consider would be $5ms$ for T_{tic} and $C_i \leq 300\mu s$.
- All cores are identical regarding their processing speed.
- There are no dependencies between runnables allocated on different cores. Therefore, all cores can be scheduled independently. This assumption is in line with the choices made by AUTOSAR regarding multicore architecture [12].

This last assumption allows to divide the overall problem into two independent sub-problems. A first part of the problem consists in allocating all of the *n* runnables onto the *m* cores with respect to their constraints with the aim of balancing the CPU load of the *m* resulting partitions (see §IV-B). The second part of the problem consists in building the dispatch table for each core (see §IV-C).

4) Scheduling condition: In our context, the system is schedulable, and thus can be safely deployed, if and only if on each core:

1) The runnables are executed strictly periodically.
2) The initial offset of each runnable is smaller than its period.
3) The sum of the WCET of the runnables allocated in each slot does not exceed a given threshold, which is typically chosen as the duration of the slot, *i.e.* T_{tic}.

B. Building tasks as a bin-packing problem

It is assumed that the number of cores is fixed. We first distribute all the runnables on the cores without checking the schedulability condition at that stage. Assigning *n* tasks to *m* cores is like subdividing a set of *n* elements into *m* non-empty subsets. By definition, the number of possibilities for this problem is given by the Stirling number of the second kind (see [13]): $\frac{1}{m!}\sum_{i=0}^{m}(-1)^{(m-i)}\binom{m}{i}i^n$. Considering that the runnables may have core allocation constraints, and thus cores should be distinguished, the $m!$ combinations of cores must be considered. As a result, one has at most $\sum_{i=0}^{m}(-1)^{(m-i)}\binom{m}{i}i^n$ different possibilities for the partitioning problem alone. Such a complexity prevents us from an exhaustive search because even for small-sized runnable sets. For instance, with $n = 30$ and $m = 2$, the search space holds more than one billion possibilities.

Considering this complexity, to balance as evenly as possible the utilization of processor cores, we propose a heuristic based on the bin-packing decreasing worst-fit scheme for a fixed number of bins (where "bins" here are processor cores). The heuristic is given in Algorithm 1. Step (1) runs in $\mathcal{O}(n)$. Step (2) runs in $\mathcal{O}(n)$ but all the runnables allocated in (2) will not have to go through the steps (3) and (4) that are algorithmically more complex. Step (3) runs in $\mathcal{O}(n \cdot \log n)$. Finally step (4) runs in $\mathcal{O}(n \cdot m)$. As a conclusion, algorithm 1 runs in $\mathcal{O}(n(m + \log n))$ which does not raise any issue in practical cases. It is worth pointing out that $m \geq \left\lceil \sum_{i=1}^{m} \frac{C_i}{T_i} \right\rceil$ is a necessary schedulability condition which can be used to rule out configurations with too few processor cores.

[1]The total dispatch table is sometimes referred to as the dispatcher round.

Algorithm 1 Partitioning of the runnable set.

input: runnable set $\{\mathcal{R}_i\}$, number of cores m
(1) Group inter-dependent runnables into runnable clusters. Independent runnables become clusters of size 1.
(2) Allocate the runnable clusters which have a locality constraint to the corresponding cores.
(3) Sort runnable clusters by decreasing order of CPU utilization rate $\rho = \sum_i \frac{C_i}{T_i}$.
(4) Iterate over the sorted clusters
 (a) Find the least loaded core,
 (b) Assign the current cluster to this core.

C. Strategies for scheduling tasks

The next stage consists in building the dispatch table for the set of runnables. Here, it is assumed that there are no precedence constraints between the runnables and that a single sequencer table is needed per core (this later assumption can be easily relaxed as done in [14]).

1) Least-loaded algorithm: Considering a runnable R_i of period T_i, there are $\frac{T_i}{T_{tic}}$ possibilities for allocating this runnable (see schedulability condition #2 in §IV-A4). As a result there are $\prod_{i=1}^{n} \frac{T_i}{T_{tic}}$ alternative schedules for the n runnables and, given the cost function, we are not aware of any ways to find the optimal solution with an algorithm does not have an exponential complexity. Considering a realistic case of 50 runnables having their period as least twice as large as T_{tic}, it would be needed to evaluate a minimum of 2^{50} possible solutions. Once again, given the complexity, we have to resort to a heuristic. Here, we adapt to the problem of scheduling runnables the "least-loaded" algorithm proposed by Grenier et al. in [10] for the frame offset allocation on a CAN network.

The intuition behind the heuristic is simple: at each step, we assign the next runnable to the least loaded slot, as described in Algorithm 2. The load of a slot is the sum of the C_i of the runnables $\{\mathcal{R}_i\}$ already assigned to this slot.

Algorithm 2 Assigning runnables to slots: the "least-loaded" heuristic.

input: runnable set $\{\mathcal{R}_i\}$, T_{tic}, T_{cycle}
(1) Sort runnables $\mathcal{R}_i$ such that $T_{tic} \leq T_1 \leq \ldots \leq T_n \leq T_{cycle}$.
(2) For $i = 1 \ldots n$
 (a) Look for the least loaded slot in the $\frac{T_i}{T_{tic}}$ first slots,
 (b) Allocate $\mathcal{R}_i$ in every $\frac{T_i}{T_{tic}}$ slot starting from this slot.

Step (1) runs in $\mathcal{O}(n \cdot \log n)$. Step (2) iterates n times over steps (2a) and (2b) which run respectively in $\frac{T_i}{T_{tic}} \leq \frac{T_{cycle}}{T_{tic}}$ and $\frac{T_{cycle}}{T_i} \leq \frac{T_{cycle}}{T_{tic}}$. As a result, this algorithm runs in $\mathcal{O}(n(\log n + \frac{\max_i\{T_i\}}{T_{tic}} + \frac{T_{cycle}}{\min_i\{T_i\}}) \leq \mathcal{O}(n(\log n + 2\frac{T_{cycle}}{T_{tic}})$.

For practical applications, ties at step (1) are broken using highest WCET first and ties at step (2a) by choosing the central slot of the longest sequence of consecutive slots having the minimum load. While the latter rule for breaking ties does not have any impact on the theoretical results that will be derived next, it helps to separate load peaks, which is important from the ECU designer point of view. As an illustration, the result of applying the least-loaded heuristic to the set of runnables $\mathcal{R}_i(T_i, C_i)$: $\mathcal{R}_1(10, 2)$, $\mathcal{R}_2(10, 1)$, $\mathcal{R}_3(20, 4)$, $\mathcal{R}_4(20, 2)$ leads to the dispatch table shown in Figure 1.

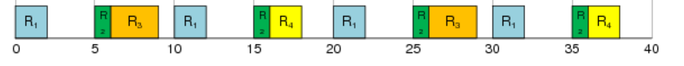

Figure 1. Example of dispatch table.

The resulting distribution of the load is:

Slot	1	2	3	4	5	6	7	8
Load	2	4	2	3	2	4	2	3

Table I
LOAD REPARTITION CORRESPONDING TO THE DISPATCH TABLE IN FIGURE 1.

There are two metrics to evaluate the quality of a dispatch table. The first important criterion is to have the lowest maximum load in the cycle since this will determine the feasibility of the schedule and the possibility to add further functions later in the lifetime of the system. The maximum load over all slots is also referred to as the *peak load*. In a second step, a more fine-grained assessment of the uniformity of the load balancing can be given by the standard deviation of the load distribution over all the slots.

2) Upper bound on the peak load: Here we derive an upper bound on the peak load which holds for runnable sets having harmonic periods. From this bound, we then derive a closed-form sufficient schedulability condition. In this perspective, we first point out that the slots in which a runnable $\mathcal{R}_i$ will be periodically assigned are of equal load.

Lemma 1: Before inserting runnable $\mathcal{R}_i$, the slot allocation induced by the previously allocated runnables repeats with a period $\frac{T_i}{T_{tic}}$.

Proof: This is proved by induction. The property holds for $\mathcal{R}_1$ as all slots are empty. Assuming that the property holds for $\mathcal{R}_i$, this runnable will be periodically allocated in every $\frac{T_i}{T_{tic}}$ slots. Therefore, the slot allocation will still repeat with a period $\frac{T_i}{T_{tic}}$ after its allocation. Since runnables are sorted by increasing periods and that their periods are harmonic, $T_{i+1} = k \cdot T_i$ with $k \in \mathbb{N}^*$ and the slot allocation also repeats with a period $k \cdot \frac{T_i}{T_{tic}} = \frac{T_{i+1}}{T_{tic}}$ before the allocation of $\mathcal{R}_{i+1}$. ■

The least loaded slot in the first $\frac{T_i}{T_{tic}}$ slots is the least loaded over the whole dispatch table and thus one does not need to look farther. As a second step, we show that when the load is equal in every slot, the resulting load after an insertion if the highest.

Lemma 2: The maximum load in the least loaded slot is obtained for a perfect load balancing, which corresponds to a constant load throughout the cycle.

Proof: Any allocation different from a perfectly balanced allocation will result in a load below the average in one or several slots, among which one will eventually be chosen for the runnable under consideration. ■

As a result, the highest peak a runnable can induce occurs in the case of a perfect load balancing. Now, Let us define $\rho_k = \sum_{i \in \{\mathcal{R}\}_k} \frac{C_i}{T_i}$ the total utilization of core k where $\{\mathcal{R}\}_k$ is the set of runnables allocated to core k and $\text{card}\{R\}_k$ the cardinality of $\{R\}_k$.

Theorem 1: On core k, an upper bound on the peak load of a slot allocation is

$$PL_k = \max_{i \in \{\mathcal{R}\}_k} \{C_i + \rho_k T_{tic} - \sum_{j=i}^{\text{card}\{\mathcal{R}\}_k} \frac{C_j}{T_j} T_{tic}\} \tag{1}$$

Proof: In the case of a perfect load balancing, before the allocation of $\mathcal{R}_i$, the load of a slot is given by:

$\sum_{allocated\, runnables} \frac{WCET \cdot number\, of\, allocation\, slots}{total\, number\, of\, slots}$, *i.e.*

$$\sum_{j \in \{\mathcal{R}\}_k}^{i-1} C_j \cdot \frac{T_{cycle}}{T_j} \cdot \frac{T_{tic}}{T_{cycle}}$$

After the allocation of $\mathcal{R}_i$, the load in the corresponding slot is

$$C_i + \sum_{j \in \{\mathcal{R}\}_k}^{i-1} C_j \cdot \frac{T_{tic}}{T_j}$$

Moreover: $\sum_{j \in \{\mathcal{R}\}_k}^{i-1} \frac{C_j}{T_j} = \rho_k - \sum_{j=i}^{\text{card}\{\mathcal{R}\}_k} \frac{C_j}{T_j}$

Consequently, the worst-case peak load on processor core k resulting of the allocation of $\mathcal{R}_i$ in a slot is

$$PL_k^i = C_i + \rho_k T_{tic} - \sum_{j=i}^{\text{card}\{\mathcal{R}\}_k} \frac{C_j}{T_j} T_{tic} \tag{2}$$

Taking the max for all the runnables gives equation 1. ■

If the worst case peak load is below T_{tic} for all runnables, then the solution given by the algorithm is schedulable. Hence the following corollary:

Corollary 1: From theorem 1, we derive the following sufficient schedulability condition:

$$\rho_k \leq 1 + \frac{C_{min}}{T_{max}} - \frac{C_{max}}{T_{tic}} \tag{3}$$

with $C_{min} = \min_{i \in \{\mathcal{R}\}_k} \{C_i\}$, $C_{max} = \max_{i \in \{\mathcal{R}\}_k} \{C_i\}$, $T_{max} = \max_{i \in \{\mathcal{R}\}_k} \{T_i\}$.

Proof: $\forall i$, $C_i \leq C_{max}$ and $\forall i$, $\sum_{j=i}^{\text{card}\{\mathcal{R}\}_k} \frac{C_j}{T_j} \geq \frac{C_{min}}{T_{max}}$ gives :

$$\forall i, PL_k^i \leq C_{max} + \rho_k T_{tic} - \frac{C_{min}}{T_{max}} T_{tic}$$

The scheduling condition 3 in §IV-A4 (*i.e.*, $PL_k^i \leq T_{tic}$) leads to the result. ■

This bound is achievable for $n \cdot k$ identical runnables with period equal to $k \cdot T_{tic}$ and load equal to C and a last runnable $\mathcal{R}_{n \cdot k+1}$ of period T_{max} and load C. With this setup, $C_{min} = C_{max} = C$ and the allocation of $n \cdot k$ first runnable results in a perfect load balancing of constant load $\rho_k \cdot T_{tic} - C \cdot T_{tic}/T_{max}$. As a result, allocating the last runnables induces the load $\rho_k \cdot T_{tic} - C_{min} \cdot T_{tic}/T_{max} + C_{max}$ in some slots.

3) Lower bound on the efficiency: We introduce here a lower bound on the core capacity that algorithm 2 guarantees to be able to use given a harmonic runnable set. This is referred to as the harmonic schedulability bound.

Theorem 2: The harmonic schedulability bound is equal to $(1 - \frac{C_{max}}{T_{tic}})$ of the capacity of the core.

Proof: Reasoning as done for Corollary 1, the worst case peak load is given by allocating a runnable $\mathcal{R} = (C_{max}, T_{max} = T_{tic})$ in a slot allocation with a perfect balance load. In the worst case, the system is still schedulable when this average slot load is equal to $T_{tic} - C_{max}$. In other words, when the system becomes no longer schedulable, every slot has an allocated load greater or equal to $T_{tic} - C_{max}$. As a consequence, at least $(1 - \frac{C_{max}}{T_{tic}})$ of the capacity of the considered core can been used by our algorithm. ■

For example, with $T_{tic} = 5ms$ and $C_{max} = 300\mu s$, at least 94% of the CPU is guaranteed to be usable. In practice, when C_{max} is small, this bound is very useful.

Corollary 2: Considering the problem of scheduling a given harmonic runnable set on a multicore ECU with an infinite number of cores using as few cores as possible, this corollary gives a bound of the maximum number required by this algorithm. Defining $\text{P} = \sum_i \frac{C_i}{T_i}$ the total load of a runnable set with harmonic periods and m_{min} the number of cores required to schedule it, it follows from theorem 2 that

$$m_{min} \leq \left\lceil \frac{\text{P}}{1 - C_{max}/T_{tic}} \right\rceil \tag{4}$$

4) Dealing with non harmonic runnable set: In practice, often, runnable sets do not have strictly harmonic periods. As a consequence, lemma 1 and lemma 2 do not hold anymore and equations 1 and 3 cannot be applied to provide bounds. In particular, placing a runnable in the least loaded slot of the dispatch table could induce peaks because of the runnable periodicity. Take the following runnable set for instance: $\mathcal{R}_1(10, 2)$, $\mathcal{R}_2(20, 3)$, $\mathcal{R}_3(20, 1)$, $\mathcal{R}_4(50, 2)$ with $T_{tic} = 5$ and $T_{cycle} = 100$. Figure 2 shows the dispatch table before the allocation of $\mathcal{R}_4$.

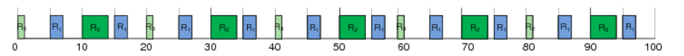

Figure 2. Dispatch table before the insertion of $\mathcal{R}_4$.

The resulting distribution of the load is:

Slot	1	2	3	4	5	6	7	8	9	10	12	12	...
Load	1	2	4	2	1	2	4	2	1	2	4	2	...

Table II
LOAD REPARTITION CORRESPONDING TO THE DISPATCH TABLE IN FIGURE 2.

At that point, choosing one of the least loaded slots in the dispatch table with make the schedule fail because $\mathcal{R}_4$ will also have to be allocated in a slot with the highest load because of its periodicity. For example, if the first instance of $\mathcal{R}_4$ is allocated in slot 1, the next instance will be placed in slot 11

and make the system unschedulable. However, allocating $\mathcal{R}_4$ in any even slot is safe.

In order to deal with non-harmonic runnable sets, we need to go through a larger window of slots for the choice of the offsets. In the following, variable T_{window} is equal to the lcm of the periods of the runnables already scheduled at the current state of the algorithm. Instead of looking for the least loaded slot in the first T_i/T_{tic} slots, we try to create the smallest peak over T_{window}, knowing that the schedule repeats in cycle afterwards.

Algorithm 3 generalized "least-loaded" heuristic.

input: runnable set $\{\mathcal{R}_i\}$, T_{tic}, T_{cycle}
(1) Sort runnables $\mathcal{R}_i$ such that $T_{tic} \leq T_1 \leq \ldots \leq T_n \leq T_{cycle}$.
(2) $T_{window} = T_{tic}$.
(3) For $i = 1 \ldots n$
(a) $T_{window} = lcm(T_{window}, T_i)$,
(b) In the first $\frac{T_i}{T_{tic}}$ slots, look for the slot such that the highest load in the slots where $\mathcal{R}_i$ is periodically allocated in the $\frac{T_{window}}{T_{tic}}$ first slots is the lowest,
(c) Allocate $\mathcal{R}_i$ in every $\frac{T_i}{T_{tic}}$ slot starting from this slot.

Step (1) of algorithm 3 runs in $\mathcal{O}(n \cdot \log n)$. Step (3a) runs in $\mathcal{O}(\log T_{cycle})$. Step (3b) and (3c) respectively run in $\mathcal{O}(n\frac{T_{window}}{T_{tic}}) \leq \mathcal{O}(n\frac{T_{cycle}}{T_{tic}})$ and $\mathcal{O}(n\frac{T_{cycle}}{T_i}) \leq \mathcal{O}(n\frac{T_{cycle}}{T_{tic}})$. As a result, the whole algorithm runs in $\mathcal{O}(n(\log n + 2\frac{T_{cycle}}{T_i} + \log T_{cycle}))$.

5) Improvement: placing outliers first: The algorithms described in sections IV-B and IV-C construct the scheduling of runnables with arbitrary periods and possibly with locality and inter-runnable constraints. Experiments show that these algorithms sometimes do not handle well runnable sets where a few runnables with a low frequency have a very large WCET compared to the other runnables.

In practice, runnables with a large WCET tend to have a large period. As a result, runnables with large WCET are usually processed late in the runnable allocation process which explains the load peaks. In order to reduce those peaks, the scheduling algorithm is improved by processing some runnables with a large WCET first[2].

We define the WCET threshold $C_{critic} = \mu + k \cdot \sigma$ with μ and σ denoting respectively the average and the standard deviation of the distribution of $\{C_i\}$ and k an integer value. The runnables with C_i larger than C_{critic} are allocated first. Then, the rest of the runnables are processed as done in algorithm 3. This new version of the load-balancing algorithm is referred to as Generalized least-loaded sigma, or G-LL$_{k\sigma}$ for short.

V. Load-balancing performances on a case study

To evaluate the performances of the algorithms, we apply them to a set of runnables randomly generated according to realistic distributions of WCET and periods. The largest WCET is $30x$ the smallest and the periods are non-harmonics chosen in $\{10, 20, 25, 40, 50, 100, 200, 250, 500, 1000ms\}$ with distributions derived from an existing body gateway ECU. The resulting average workload is around 85% which is common for automotive ECUs. Random dependencies between runnables are also introduced through the following parameters:

- Interdependency ratio, that is the percentage of runnables that are dependent and thus must be executed on the same core, chosen equal to 25% in the experiments.
- Maximum size of the clusters of dependent runnables is equal to 4.
- Core locality constraint ratio: percentage of runnables that are pre-allocated to a given core, chosen equal to 25%.

The following parameters are used for the simulations: $C_{max} = 1.5ms$, $T_{tic} = 5ms$, $T_{cycle} = 1s$ and there are 600 runnables to schedule on 3 cores. This may correspond to the configuration of a powerful ECU in a few years from now.

The distribution of the load obtained with the generalized least-loaded (LL) and generalized least-loaded sigma (G-LL$_{k\sigma}$) algorithms are shown in figure 3. In these graphics, the X axis is the time-line and the Y axis shows the load of the slots in percentage. The upper graphic shows that LL fails to provide a feasible schedule since the load is above 100% in some slots. The load peaks are due to runnables having a large WCET. For instance, the green peaks are created by runnables with WCET=$1.5ms$. However, G-LL$_{k\sigma}$ is able to sucessfully schedule the runnable sets on the three cores. In addition, about 10% of the capacity of each core remains available almost all the time, which means that some more runnables can be added in future evolution of the ECU.

It is worth mentioning that usually, as in the figure 3, generalized least-loaded sigma outperforms generalized least-loaded for well chosen value of k, although, in some rare cases, generalized least-loaded might produce better solutions. Those experiments also show that, even if the theoretical lower bound does not hold anymore, runnables with non-harmonic periods can be efficiently handled by the heuristics. For instance, here the theoritical lower bound would be 70% for harmonic periods without any locality nor interdependency constraints, whereas G-LL$_{k\sigma}$ is able to produce a schedulable slot allocation for 86% of load, non-harmonic periods and both locality and interdependency constraints. The reader interested in more comprehensive experiments, also in the case where several sequencer tasks are needed for instance for memory protection accross runnables, can refer to [14].

VI. Conclusion

Multi-source and multicore ECUs will drastically change E/E architectures and should enable to conceive more cost-effective and more flexible automotive embedded systems. In our view, the OS protection mechanisms specified by AUTOSAR provide a sound basis for coming up with appropriate safety mechanisms and policies, despite the growing complexity and criticality of software functions.

[2]Allocating the runnables by decreasing order of WCET proves not to be an efficient approach in our experiments.

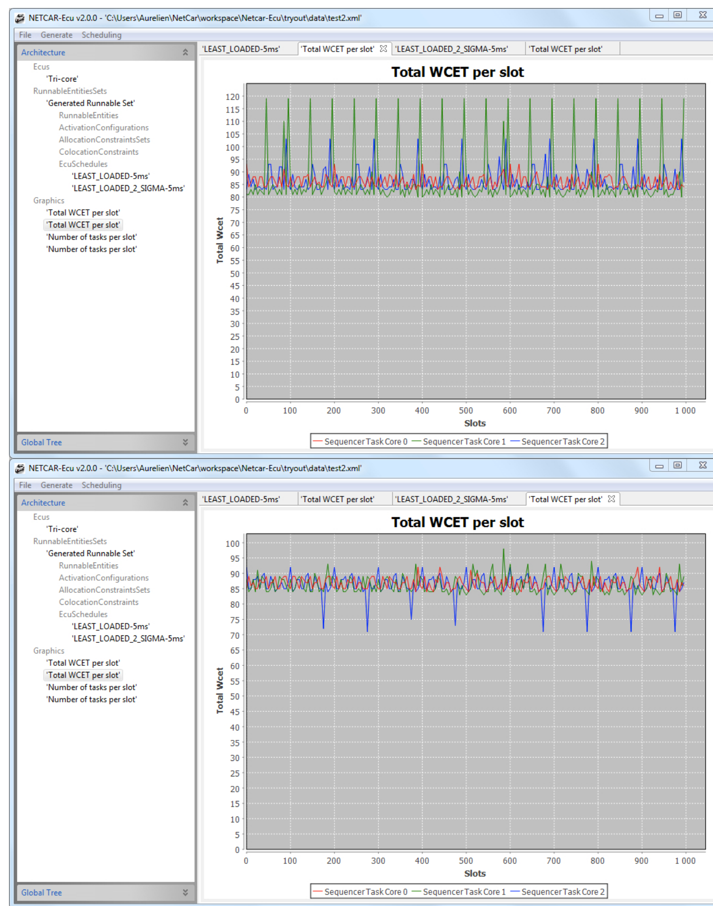

Figure 3. Distribution over time of the load percentage on the 3 cores. The upper graphic shows the result with the generalized least-loaded algorithm while the lower curve is obtained with the generalized least-loaded sigma algorithm for $k = 2$. The algorithms of this study have been implemented as plugins of RTaW's NETCAR-ECU [11].

However, today's design methodologies need to be adapted to this new context and there is a wide range of technical problems to be solved. The design of the software architectures and the scheduling of the software components are among these issues. In this paper, we have presented practical scheduling solutions well suited to the basic use-case which is to execute a large number of software components on the same multicore processor. The set of algorithms described in this paper have shown on realistic case-studies to be versatile and efficient in terms of CPU usage optimization, providing even guaranteed performance levels in some specific contexts. Future work will consist in extanding this framework to handle other requirements such as precedence constraints, lockstep redundant executions and distributed timing chains.

References

[1] N. Navet and H. Perrault, "Mécanismes de protection dans AUTOSAR OS," Seminar at RTS Embedded Systems 2009 (RTS'2009), April 2009, slides available at http://www.realtimeatwork.com/.

[2] N. Navet, B. Delord, and M. Baumeister, "Virtualization in automotive embedded systems: an outlook," Seminar at RTS Embedded Systems 2010 (RTS'2010), March 2010, slides available at http://www.realtimeatwork.com.

[3] C. Aussagues and D. Roux, "An OS for multicore embedded systems compliant with automotive safety standards," in *IAEC'09*, 2009.

[4] A. Burchard, J. Liebeherr, Y. Oh, and S. H. Son, "New strategies for assigning real-time tasks to multiprocessor systems," *IEEE Transactions on Computers*, vol. 44, no. 12, pp. 1429–1442, Dec. 1995.

[5] Y. Oh and S. Son, "Fixed-priority scheduling of periodic tasks on multiprocessor systems," Department of Computer Science, University of Virginia, Tech. Rep. CS-95-16, 1995.

[6] Y. Oh and H. S. Son, "Tight performance bounds of heuristics for a real-time scheduling problem," Department of Computer Science, University of Virginia, Tech. Rep. CS-93-24, 1993.

[7] S. Lauzac, R. Melhem, and D. Mossé, "An improved rate-monotonic admission control and its applications," *IEEE Transactions on Computers*, vol. 52, no. 3, pp. 337–350, 2003.

[8] A. Karrenbauer and T. Rothvoss, "An Average-Case Analysis for Rate-Monotonic Multiprocessor Real-time Scheduling," in *17th Annual European Symposium on Algorithms (ESA)*, 2009.

[9] J. Goossens, "Scheduling of offset free systems," *Real-Time Systems*, vol. 24, no. 2, pp. 239–258, March 2003.

[10] M. Grenier, L. Havet, and N. Navet, "Pushing the limits of CAN - scheduling frames with offsets provides a major performance boost," in *European Congress of Embedded Real-Time Software (ERTS 2008)*, 2008.

[11] RealTime-at-Work, "NETCAR-ECU: a task scheduling configuration tool," Description available at http://www.realtimeatwork.com/, 2009.

[12] AUTOSAR Consortium, "Specification of multi-core OS architecture v1.0," AUTOSAR Release 4.0, 2009.

[13] M. Abramowitz and I. Stegun, *Handbook of Mathematical Functions*. Dover Publications (ISBN 0-486-61272-4), 1970.

[14] A. Monot, N. Navet, B. Bavoux, and F. Simonot-Lion, "Multicore scheduling in automotive ECUs," in *Embedded Real Time Software and Systems (ERTSS'2010)*, May 2010.