Multicore scheduling in automotive ECUs

Aurélien Monot[1,2], Nicolas Navet[3], Françoise Simonot[1], Bernard Bavoux[2]

1: LORIA - Nancy Université, BP 239, 54506 Vandoeuvre, France
2: PSA Peugeot Citroën, Centre technique de Vélizy, Route de Gisy, 78 943 Vélizy-Villacoublay Cedex, France
3: INRIA - RealTime-at-Work, 615 Rue du Jardin Botanique, 54506 Vandoeuvre-lès-Nancy, France

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 eases the respect of the safety requirements introduced by the ISO 26262 and can be taken advantage of in various 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 present the main use cases for multicore ECUs and then 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 subproblems: partitioning the set of runnables and building the schedule on each core. At that point, 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 realistic case-studies.
keywords: multicore ECU, AUTOSAR, static cyclic scheduling, load balancing, offsets.

1 Introduction

Context of the paper. 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. One of the outcomes of the AUTOSAR initiative is indeed to help OEMs shift from the "one function per ECU" paradigm to more centralized architecture designs. As chip manufacturers are reaching the point where they can no longer meet the increasing performance requirements through frequency scaling, multicore ECUs are being gradually introduced in the automotive domain. Those multicore platforms offer also additional benefits such as higher level of parallelism which are needed to meet the upcoming safety requirements, as the ISO 26262 is being introduced. Now, the challenge is to adapt existing design methods to the new multicore constraints. The scheduling of the software components is in one of the key issues in that regard and it has to be revamped.

Existing work. 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.*, partitioning) 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[5]. Scheduling tasks on a multiprocessor systems under the static partitioning approach has been well studied for a long time, see for instance [4] and [14, 13, 9, 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 [6] (mono-processor scheduling of tasks with offsets) and [7] (scheduling of frames with offsets) but it is applied to multicore 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 [15], 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.

Objectives of the paper. Multicore platforms offers two major benefits for the automotive domain: power and parallelism. In this paper, we discuss the main use-cases benefiting from the usage of multicore architectures in the automotive domain. Then, this paper focuses on a particular problem which is the scheduling of numerous elementary pieces of code (called "runnables" in the AU-

TOSAR terminology) with the objective of using the performance of multiple cores to reduce the number of ECUs. In our view, a static cycling scheduling approach is especially suited when, as it is most often the case, there are much more runnables that the maximum number of tasks allowed by the automotive operating system[1]. Thus runnables must be grouped together and scheduled by one or several sequencer tasks (also called dispatcher tasks). Regarding multicore scheduling, we assume a static partitioning scheme which is likely to be adopted in a first step in the automotive domain: it is conceptually simple and provides a better predictability for the ECU designer compared to the global scheduling approach. The aim of this study is to develop practical algorithms to build the partitioning of runnables and to schedule the runnables on each cores so as to respect timing 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.

2 Main use cases for multicore ECU in the automotive domain

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 a while 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.

2.1 Decreasing the complexity of in-vehicle 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.

2.2 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.

2.3 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 [11].
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.

2.4 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

[1] Depending on the conformance class, OSEK/VDX and AUTOSAR OS impose a maximum of only 8 or 16 tasks.

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[2] on a SX12E micro-controller.

3 Partitioned scheduling of tasks on AUTOSAR OS

In this section, we study the first of the use cases we identified for multicore architectures, which is to permit a reduction of the number of ECUs in the E/E architecture by using more powerful ECUs. In this context, we present algorithms to schedule large numbers of runnables on multicore 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 determining the offsets between the runnables allocated on each core so as to balance the load over time.

3.1 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.

3.2 Model description

In this case 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.

3.2.1 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.

3.2.2 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[2], 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.

3.2.3 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 [6]). 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 $C_i \leq 300\mu s$.

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

- 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 [5].

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 §3.3). The second part of the problem consists in building the dispatch table for each core (see §3.4).

3.2.4 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}.

3.3 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 [1]):$\frac{1}{m!}\sum_{i=0}^{m}(-1)^{(m-i)}\binom{m}{i}i^n$. 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.

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 runnables 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.

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.

3.4 Strategies for scheduling tasks

The next stage consists in building the dispatch table for the set of runnables. In a first step, it is assumed that there are no precedence constraints between the runnables and that a single dispatch table is built per core. Those assumptions will be relaxed later in the paper.

3.4.1 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 §3.2.4). 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 which 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 [7] 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 $\mathcal{O}(n(\log n + \max_i\{T_i\}/T_{tic} + T_{cycle}/\min_i\{T_i\}) \leq \mathcal{O}(n(\log n + 2 \cdot T_{cycle}/T_{tic})$.
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.

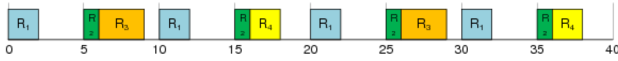

Figure 1: Example of dispatch table.

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 1: 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.

3.4.2 Some theoretical results

Here are present a few useful theoretical results. Extended demonstrations can be found in [12].

Theorem 1: Defining $C_{max} = \max_{i \in \{\mathcal{R}\}_k}\{C_i\}$ and $T_{max} = \max_{i \in \{\mathcal{R}\}_k}\{T_i\}$

$$\rho_k \leq 1 + \frac{C_{max}}{T_{max}} - \frac{C_{max}}{T_{tic}} \quad (1)$$

is a sufficient schedulability condition for a harmonic task set.

Theorem 2: Another way to look at it is to notice that this algorithm guarantees to be able to use $(1 - \frac{C_{max}}{T_{tic}})$ of the capacity of each core.
The worst-case peak load is obtained when allocating a runnable $\mathcal{R} = (C_{max}, T_{max} = T_{tic})$ in a slot allocation with a perfectly balanced 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 has 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. In the following, $(1 - \frac{C_{max}}{T_{tic}})$ will be referred to as the harmonic schedulability bound.

Theorem 3: Considering the problem of scheduling a given harmonic runnable set with as few cores as possible, the previous result gives a bound of the maximum number required by this algorithm. Defining $\mathrm{P} = \sum_i \frac{C_i}{T_i}$ the total load of a runnable set with harmonic periods and m_{min} the minimum number of cores required to schedule it, it follows from theorem 2 that $m_{min} \leq \left\lceil \frac{\mathrm{P}}{1 - C_{max}/T_{tic}} \right\rceil$.

3.4.3 Dealing with non harmonic runnable set

In practice, often, runnable sets do not have strictly harmonic periods. As a consequence, the previous results do not hold anymore. 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 3 shows the dispatch table before the allocation of $\mathcal{R}_4$.

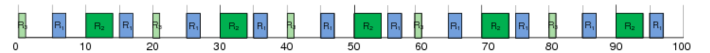

Figure 3: Dispatch table before the insertion of $\mathcal{R}_4$.

The resulting distribution of the load is:

Slot	1	2	3	4	5	6	7	8	9	10	11	12	...
Load	1	2	4	2	1	2	4	2	1	2	4	2	...

Table 2: Load repartition corresponding to the dispatch table in Figure 3.

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

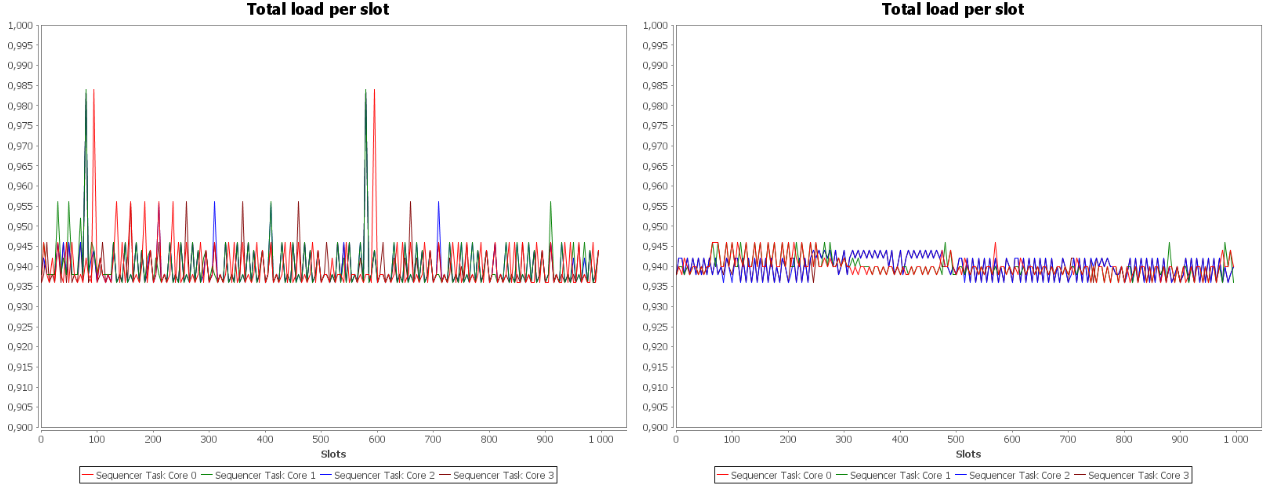

Figure 2: A set of runnables corresponding to slightly less than 94% of CPU load scheduled on a 4-core ECU with LL (left-hand graphic) and G-LL$_{1\sigma}$ (right-hand graphic). Graphics obtained from RTaW NETCAR-ECU.

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 afterward.

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 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 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 $\mathcal{O}(n \cdot T_{window}/T_{tic}) \leq \mathcal{O}(n \cdot T_{cycle}/T_{tic})$ and $\mathcal{O}(n \cdot T_{cycle}/T_i) \leq \mathcal{O}(n \cdot T_{cycle}/T_{tic})$. As a result, the whole algorithm runs in $\mathcal{O}(n \cdot (\log n + 2 \cdot T_{cycle}/T_i + \log T_{cycle}))$.

3.4.4 Improvement: placing outliers first

The algorithms described in sections 3.3 and 3.4 construct the scheduling of runnables with arbitrary periods and possibly with locality and inter-runnable constraints. As shown in the experiments in section 4, 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[3].

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. In the experiments that follows, k is chosen equal to 1.

4 Performances and robustness on automotive ECUs

In this section, we evaluate the performances of the scheduling algorithms on automotive case-studies. The algorithms LL, G-LL, and G-LL$_{k\sigma}$, described respectively in §3.4.1, §3.4.3 and §3.4.4, have been implemented as plug-ins of NETCAR-ECU [15]. The characteristics of the task sets under study (*i.e.*, period, WCET) are drawn at random from distributions derived from an existing body gateway ECU. This ECU has about 200 runnables whose periods are almost harmonic (only about 5% of the runnables have non-harmonic periods). The duration of the slot, T_{tic}, is set to 5ms in all experiments. We distinguish configurations with a single sequencer task per core and configurations with multiple sequencer tasks per core. The latter implementation is required when memory protection is needed among runnables.

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

4.1 Single sequencer task on each core

Here we assess the ability of the algorithms to uniformize the CPU load over time and to keep on providing feasible solutions at very high load level.

4.1.1 Harmonic task sets

The task sets considered here are harmonic with periods in the set $\{10, 50, 100, 500, 1000ms\}$, which is a large subset of the periods used in the real ECU. The WCETs vary from 10μs to 300μs with a probability derived from the real distribution on the body gateway ECU. We choose the average CPU load slightly below 94% so that the feasibility is ensured (see theorem 2). The left-hand graphic of Figure 2 shows the distribution of the load over a LCM of the periods with LL while the right-hand graphic shows the distribution with G-LL$_{1\sigma}$. As it can been seen, the load peaks are much smaller with G-LL$_{1\sigma}$ (peak load is 94.6%) than with LL (peak load is 98.4%). This can be explained because the few largest runnables are placed first and the numerous smaller ones placed afterward fill the gaps in the schedulable table.

4.1.2 Non-Harmonic task sets

The goal is to assess the extent to which the schedulability bound, even if it has been derived in the harmonic case, can provide guidelines for the non-harmonic case. Precisely, we measure the success rate of the algorithms in the non-harmonic case at load levels such that feasibility would be ensured in the harmonic case. In the existing body gateway ECU, the set of task periods is close to be harmonic since withdrawing only a few runnables ensures the harmonic property. To test the algorithms in a more difficult context, we build a “hard” non-harmonic case with more departure from the harmonic property. Precisely the periods are now chosen in the set $\{10, 20, 25, 40, 50, 100, 125, 200, 125, 500, 1000ms\}$. As can be seen in Table 3, when the load is close to the harmonic schedulability bound the algorithms remain efficient, in particular the G-LL which was able to successfully schedule the 1000 random configurations of the test. This suggests to us that the harmonic schedulability bound is a good dimensioning criterion also in the non-harmonic case.

max WCET (μs)	150	300	900
Schedulability bound in the harmonic case	97%	94%	82%
Success % of LL in the “hard” non-harmonic case	96%	96%	92%
Success % of G-LL in the “hard” non-harmonic case	100%	100%	100%

Table 3: Performances of the scheduling algorithms in the non-harmonic case when the load is close to the harmonic schedulability bound. Statistics collected on 1000 random configurations for each max. WCET value. The schedulability bound is derived from theorem 2.

Table 4 presents the results obtained at higher loads, *i.e.* above the harmonic schedulability bound. Precisely sets of runnables with max. WCET equal to 300μs and 900μs and CPU loads equal to 95% and 97% are scheduled with LL, G-LL and G-LL$_{1\sigma}$.

Generated CPU load	95%	97%	95%	97%
Schedulability bound in the harmonic case	94% max WCET=300μs		82% max WCET=900μs	
Success % of LL	64%	18%	12%	1%
Success % of G-LL	94%	94%	30%	5%
Success % of G-LL$_{1\sigma}$	100%	100%	97%	76%

Table 4: Performances of the scheduling algorithms in the non-harmonic case when the load is greater than the harmonic schedulability bound. Statistics collected on 1000 random configurations for each max. WCET value.

As it was expected, the lower the schedulability bound, the harder it is to schedule the runnables (compare for instance the 97% columns). The second lesson is that G-LL$_{1\sigma}$ clearly outperforms all the other contenders especially when the WCETs are large.

4.2 Multiple sequencer tasks on each core

In this section, we suppose that several sequencer tasks need to be scheduled on each core. The case arises when memory protection across runnables is needed. Indeed, memory protection, such as provided by AUTOSAR OS, cannot be ensured at the runnable level but at the task (or ISR and OS-application) level. We assume that the different sequencer tasks are scheduled by a fixed priority scheduler as it is foreseeable in AUTOSAR ECUs. In the next subsections, we distinguish two cases: synchronized sequencer tasks and non synchronized sequencer tasks. In the first case, we assume that the initial offsets between sequencer tasks are known and they have the same T_{tic}. In the second case, the different sequencer tasks may be driven by different clocks. This later case arises, for instance, in engine controllers in which some runnables are driven by the micro-controller clock while others are driven on the basis of the engine RPM which varies over time.

4.2.1 Synchronized sequencer tasks

Synchronized means here that the initial offsets of the different sequencer tasks are known and that they have the same T_{tic}. In this context, the runnable activation times are known and the algorithms presented in the paper can be applied to build the sequencer tasks, which are each assigned a distinct priority level (*i.e.*, FPP scheduling between sequencer tasks).
The first step is to allocate the runnables to the different sequencer tasks, mainly depending on the memory protection needs. Then, the sequencer tasks are built iteratively (by decreasing order of their priority), which consists in allocating the runnables into the slots using LL, G-LL

Figure 4: Incremental scheduling of three synchronized sequencer tasks with respective load of 45%, 35% and 15% for a total of 95% of the core capacity. T_{cycle} is equal to $1000ms$ and T_{tic} is equal to $5ms$ for each sequencer task.

or G-LL$_{k\sigma}$. The least-loaded slot (LL algorithm) or the slot allocation resulting in the smallest peak (G-LL and G-LL$_{k\sigma}$) are determined considering the runnables of higher priority sequencer tasks. This can done for instance by building a global virtual sequencer task with a cycle period equals to the lcm of the sequencer tasks cycle periods.
In Figure 4, three sequencer tasks whose loads respectively correspond to 45%, 35% and 15% of the core capacity are incrementally scheduled. The runnables were randomly generated from distributions derived from an existing body gateway ECU with cycle periods equal to $1000ms$ and slot duration equal to $5ms$. The graphs in Figure 4 show the resulting load of one core after applying G-LL on each sequencer task. Schedulability is then ensured by checking the resulting load of every slot in the virtual global sequencer task. As seen in Figure 4, the load balancing performances of G-LL are very satisfactory in this context too as peaks do not exceed 3% of the core capacity. It should be pointed out that this incremental scheduling approach may prove also very useful when adding new runnables on existing ECU configurations.

4.2.2 Non synchronized sequencer tasks

The previous approach cannot be applied to non synchronized sequencer tasks since their offset and time basis are not known. If the sequencer tasks are scheduled on different time basis (*e.g*, clock or engine RPM), whatever the way a sequencer task is constructed, each of its slot can interfere with any other slots of the other sequencer tasks as all offset configuration between them are possible at run-time. This means that, on the contrary to the synchronized case of §4.2.1, we cannot rely on how higher priority sequencer tasks are defined when building the slot allocation of a sequencer task. The maximum robustness against all possible asynchronisms between different sequencer tasks is achieved by balancing the load of each one of them individually as done in the basic use-case of the algorithms of §4.1.
Because of the possibly varying time basis, the schedulability of the slot allocation cannot be as easily checked as in the previous cases. However, if it is possible to bound the progressing rate of each sequencer task, the multi-frame task model [10] can be used to check the feasibility of the schedule. The transformation of a dispatch table into a multi-frame task is loseless: the slots of a sequencer task become the task instances (with execution times depending on the runnables scheduled in each slot) and the relative deadline of each task instance is given by the duration of a slot. Then, assuming the maximum clock speeds, a multi-frame schedulability test can be applied (see, for instance, [3, 16]).

5 Conclusion

Today's automotive design methodologies need to be adapted to multicore computing 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 in order to reduce the number of ECUs. 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 extending this framework to handle other requirements such as precedence constraints, lockstep redundant executions and distributed timing chains.

References

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

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

[3] S. Baruah, D. Chen, and A. Mok. Static-priority scheduling of multiframe tasks. pages 38–45, 1999.

[4] A. Burchard, J. Liebeherr, Y. Oh, and S. H. Son. New strategies for assigning real-time tasks to multipro-

cessor systems. *IEEE Transactions on Computers*, 44(12):1429–1442, Dec. 1995.

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

[6] J. Goossens. Scheduling of offset free systems. *Real-Time Systems*, 24(2):239–258, March 2003.

[7] 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.

[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)*, Lecture Notes in Computer Science, Berlin, 2009. Springer.

[9] S. Lauzac, R. Melhem, and D. Mossé. An improved rate-monotonic admission control and its applications. *IEEE Transactions on Computers*, 52(3):337–350, 2003.

[10] A. Mok and D. Chen. A multiframe model for real-time tasks. volume 23, pages 635–645, 1996.

[11] 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`.

[12] N. Navet, A. Monot, B. Bavoux, and F. Simonot-Lion. Multi-source and multicore automotive ECUs - OS protection mechanisms and scheduling. In *IEEE International Symposium on Industrial Electronics (ISIE)*, 2010.

[13] Y. Oh and H. S. Son. Tight performance bounds of heuristics for a real-time scheduling problem. 1993.

[14] Y. Oh and S.H. Son. Fixed-priority scheduling of periodic tasks on multiprocessor systems. Technical report, Department of Computer Science, University of Virginia, 1995.

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

[16] A. Zuhily and A. Burns. Exact response time scheduling analysis of accumulatively monotonic multiframe real time tasks. pages 410–424, 2008.