

RunPar: An Allocation Algorithm for Automotive Applications Exploiting Runnable Parallelism in Multicores

Miloš Panić^{*,†}, Sebastian Kehr[±], Eduardo Quiñones[†],
Bert Boddecker[±], Jaume Abella[†], Francisco J. Cazorla^{†,‡}

^{*} Universitat Politècnica de Catalunya (UPC)

[†] Barcelona Supercomputing Center (BSC)

[±] DENSO AUTOMOTIVE Deutschland GmbH

[‡] Spanish National Research Council (IIIA-CSIC)

{milos.panic,eduardo.quinones,jaume.abella,francisco.cazorla}@bsc.es {s.kehr,b.boeddecker}@denso-auto.de

ABSTRACT

Automotive applications increasingly rely on AUTOSAR for their design and execution. AUTOSAR applications comprise functions, called *runnables*, that are grouped into *AUTOSAR tasks*. Tasks are the unit of scheduling (UoS) of the AUTOSAR operating system as the legacy of the single-core platforms. However, on multi-core platforms using tasks as UoS considerably reduces the available parallelism due to communication dependencies, which in turn reduces the potential average and guaranteed performance obtainable with multi-cores. Furthermore, running tasks in parallel requires re-validating the functional correctness of the application, since current AUTOSAR applications are designed following a sequential execution model of tasks.

In this paper, we propose a new allocation algorithm, *RunPar*, that considers runnables and not tasks as the UoS and assigns runnables that form tasks to different cores. *RunPar* improves the application performance, while keeping the sequential execution of tasks, hence not requiring any extra validation effort when migrating AUTOSAR applications from single-core to the multi-core platforms. We evaluate *RunPar* with a real automotive application, an Engine Management System (EMS) for which we observe an average WCET reduction on EMS' tasks of 26% and 30% in a two-core and four-core ECU.

1. INTRODUCTION

Modern road vehicles carry up to 70 single-core Electronic Control Units (ECUs) performing various functions, from opening a window to controlling the engine. This makes automotive industry to pay special attention to minimize size, weight and power costs, while increasing the services delivered per ECU. Multi-core processor architectures, which are nowadays a reality in the automotive domain [9] as well as in other embedded domains [6, 7], are considered as one promising solution to cope with such performance and cost constraints.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ESWEEK'14, October 12 - 17 2014, New Delhi, India
Copyright 2014 ACM 978-1-4503-3051-0/14/10 ...\$15.00
<http://dx.doi.org/10.1145/2656075.2656096>.

Multi-core ECUs aim at providing the performance required to run high number, complex functions by: (i) integrating distributed applications into a single ECU; (ii) parallelizing the computation of complex systems, such as the Engine Management System (EMS) or Advance Driver Assistant Systems (ADAS); or (iii) a combination of both. This paper focuses on parallelization of complex applications, i.e. improving the performance of an automotive legacy application by effectively parallelizing it over several cores, considering an EMS as a case study. In this respect, it is important to remark the relevance of this problem given that a significant part of automotive software is composed of legacy code.

Automotive applications increasingly rely on the AUTomotive Open System ARchitecture (AUTOSAR) [2], a standardized system software architecture upon which applications are built and executed. In AUTOSAR, applications are composed of a set of functions, named *runnables*, that are executed periodically or triggered by an interrupt. When developing an AUTOSAR application, runnables are grouped into *AUTOSAR tasks*¹, which are the unit of scheduling (UoS) of the AUTOSAR Operating System (AROS) [2]. The runnable-to-task mapping and the single-core task scheduling of an application is known as *application configuration* and it is static and known at system integration time. Application configuration is done only once due to the expensive cost of validate its functional and timing correctness [10].

The current strategy of using tasks as UoS works well on applications running on single-core ECUs, because it facilitates scheduling runnables with the same timing properties by grouping runnables with the same release period or interrupt into the same task.

When moving to multi-core ECUs, current approaches also consider tasks as UoS [13][5], allocating them to different cores. To do so, all dependent runnables are grouped into a single task, minimizing or even removing all inter-task communications, and so scheduling independently the different tasks to the processor cores. This approach, which is in-line with the latest AUTOSAR guide for developing and configuring AUTOSAR-compliant software for multi-core systems [1], works well for integrating multiple applications into a single ECU or for parallelizing applications with little inter-runnable communication. However, the use of tasks as UoS on multi-core processors to extract parallelism of applications with highly-connected runnables is inefficient, because most runnables are allocated to a single task and so executed serially in one core. This is the case for the EMS application considered in this pa-

¹In this paper, we refer to an AUTOSAR task simply as task.

per, in which almost all runnables depend on each other. Figure 3 provides an intuition on the level of communication existing in the EMS, showing the inter-runnable dependencies of three of the twelve tasks that compose the EMS (concretely tasks 1, 4 and 8 ms; see Section 4.1 for further details).

Moreover, current approaches require changing the runnable-to-tasks mapping and/or execution of tasks in parallel as means to improve application performance. This, in turn, implies changing the application configuration, resulting in extra effort to validate the new configuration [21]. The reason is, the sequential execution model abstracts and may hide mutual exclusion constraints when accessing shared resources, critical sections, etc. The parallel execution of tasks can then break this mutual exclusion relations present in applications configured for execution in single-core processors [21].

In this paper, we propose exploiting the performance opportunities of multi-core ECUs by proposing a new allocation strategy in which *legacy automotive applications* with runnables highly connected are parallelized while maintaining the single-core application configuration. We present *RunPar*, a new allocation algorithm that considers runnables, and not tasks, as the UoS. *RunPar* assigns runnables of the same task to different cores respecting inter-runnable dependencies and does not allow runnables from different tasks to be executed in parallel. As a result, tasks are forced to execute sequentially, following the task ordering of the application’s single-core task scheduling. Hence, the composition of tasks and the order in which they are executed in the single-core and the multi-core platforms remains the same, and so the same functional behavior is guaranteed in both platforms. Maintaining the sequential execution of tasks guarantees that the mutual exclusion relations between the critical sections defined among tasks are respected.

We evaluate the benefits of *RunPar* on an EMS, a real automotive application that controls the injection time and amount of fuel in a diesel engine and composed of more than one thousand highly connected runnables grouped into twelve tasks. Our results confirm that *RunPar* effectively increases the performance of EMS’ tasks by providing WCET speed-ups² to 1.35x and 1.43x on average, representing a reduction of WCET estimations of 26% and 30% on average, when allocating it on a two-core and four-core processor architecture respectively, with respect to WCET estimates of sequential EMS’ tasks obtained in isolation.

Such a WCET speed-up translates into an increment of the CPU capacity of 31% and 42% for the two-core and four-core ECUs respectively. This extra capacity can be then exploited for executing new application functionality or other automotive applications, which ultimately contributes allocating more functionality per ECU, reducing size, weight and power costs.

We consider *RunPar* a necessary step towards porting current legacy software on multi-core ECUs, for exploiting the multi-core performance potential and containing validation effort. The use of runnables as UoS implies minimum modifications at AR-OS level: The scheduling tables used in the AR-OS to execute tasks are extended to incorporate the core, the order and the time in which runnables are executed, so inter-runnable dependencies are respected. How to better exploit multi-core ECUs capabilities for new AUTOSAR applications to minimize inter-runnable communi-

²In parallel execution, *speed-up* determines how faster a parallel program runs with respect to its corresponding sequential version. The maximum speed-up a program can obtain is determined by the number of cores, so in a four-core processor, the maximum speed-up expected is four. Such a speed-up is referred to as *ideal speed-up*.

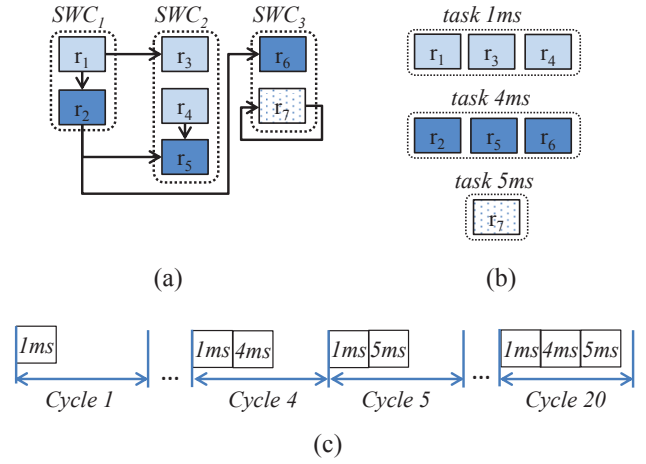


Figure 1: Part of the runnable flow-graph of an automotive application composed of 3 SWC, 7 runnables and 3 tasks executed in a single-core processor. (a) Structure of the application; (b) application configuration from an AR-OS point of view; (c) a possible single-core task scheduling of the three tasks.

cations, and so increase parallelism, is a challenging problem that is out of the scope of this paper and part of our future work.

The rest of the paper is organized as follows. Section 2 summarizes the structure of AUTOSAR applications and introduces the multi-core processor architecture and the timing analysis tool considered in this paper. Section 3 presents our allocation algorithm. Section 4 describes the experimental setup, including the case study upon which *RunPar* allocation algorithm is evaluated and the timing analysis parameters used to compute the WCET estimates, and evaluates *RunPar* allocation algorithm. Finally, Sections 5 and 6 presents the related work and conclusions respectively.

2. BACKGROUND

2.1 AUTOSAR Applications

The structural elements of an AUTOSAR application are software-components (SWC), each containing a set of *runnable entities* (which we call *runnables* for short) that implement the functionality of the SWC. AUTOSAR provides two inter-runnable communication methods: *sender-receiver* and *client-server* ports. The former uses a global shared memory for communication; the latter allows runnables to invoke services from other runnables. In case of sender-receiver, runnables read all input data before starting the execution and results are written back after finishing the execution. No limitations on the number of ports or complexity of components are imposed by the model. All SWC, ports and runnables are known at application configuration time.

During the application configuration phase, runnables are assigned to *tasks*, which are the UoS of the AR-OS [15]. The execution of runnables follows a model in which they are periodically executed in a recurring *cycle* or triggered by an interrupt. To that end, in legacy applications running on single-core systems, runnables with the same period or interrupt invocation are grouped into the same task, so each task is executed based on either a fixed period or an interrupt. As a result, from an AR-OS point of view, an automotive application can also be defined as a set of tasks with a period or interrupt invocation equal to the period or interrupt invocation of the runnables that compose those tasks.

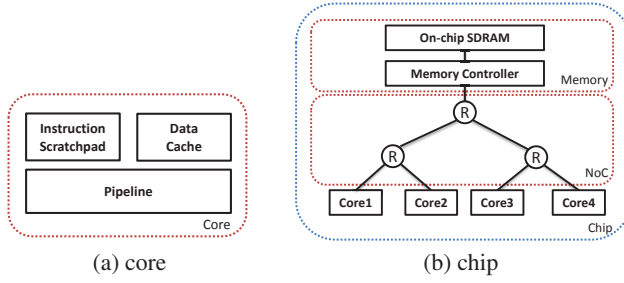


Figure 2: Block diagram of our reference architecture.

Figure 1 shows the relationship between SWC, runnables and cyclic tasks for an automotive application composed of three SWC and seven runnables with a period associated to each runnable: r_1, r_3 and r_4 have a 1 ms period, r_2, r_5 and r_6 a 4 ms period and r_7 a 5 ms period. The arrows represent the communication ports, and so *run-after dependencies*, among them. Figure 1(b) shows the structure of the application from the AR-OS point of view after the configuration phase. The runnables are grouped into three tasks based on their periods: Tasks *1ms*, *4ms* and *5ms* contain runnables whose period is 1, 4 and 5 ms respectively. Moreover, the order in which runnables are executed within tasks must respect run-after dependencies. In this case runnable r_1 executes before r_3 in task *1ms*. Finally, Figure 1(c) shows a possible single-core scheduling of the three tasks. Again, the order in which runnables and tasks are executed must respect the run-after dependencies. Hence, task *1ms* must execute before task *4ms* to respect the dependencies between runnables r_4 and r_5 executed in tasks *1ms* and *4ms* respectively. Note that a different sequence of execution of runnables and tasks could be defined as well, still respecting dependencies among runnables.

When a runnable consumes data produced by the runnables from other tasks or by past instances of the same task it belongs to, the sequential execution of tasks guarantees that, when a new task instance starts, all previous tasks instances (including itself) have already finished, and so the data dependence is not violated. This is the case of runnable r_7 in Figure 1.

Although the structure of the application in Figure 1(a) is relatively simple, real automotive applications are composed of a high number of highly connected runnables. This is the case for the EMS application considered in this paper, which is composed of **more than one thousand runnables highly connected among them**. Figure 3 shows a *part of* the inter-runnable dependencies existing in three of the twelve tasks that compose the EMS (concretely tasks 1, 4 and 8 ms; see Section 4.1 for further details).

2.2 Multi-cores and WCET estimation

This paper considers an analyzable multi-core processor similar to the one described in [23, 18, 17] as the target processor in which runnables are allocated. In this processor the maximum delay a request can suffer when accessing hardware shared resources is bounded by a pre-computed *Upper Bound Delay (UBD)*. Architectures based on UBD have been shown to provide competitive results in terms of average and guaranteed-performance (i.e. WCET estimates) with respect to other time-predictable approaches such as TDMA [11].

We consider a two- and a four-core processor, in which each core has a private instruction scratchpad and a data cache, see Figure 2(a). The core is assumed to exhibit no timing anomalies [24]

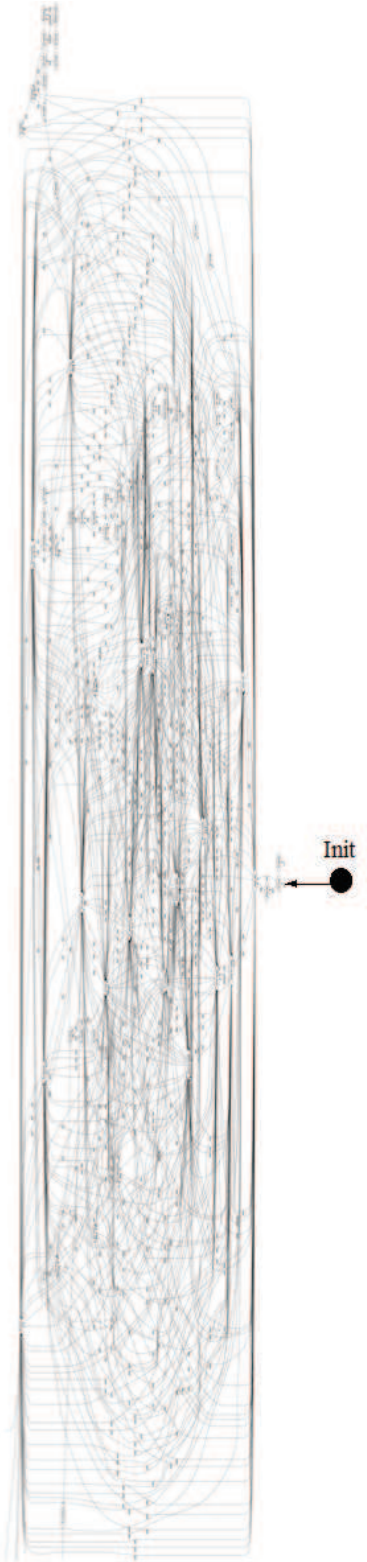


Figure 3: Inter-runnable dependencies existing among three of the twelve tasks that compose the EMS (tasks 1, 4 and 8 ms). Nodes represent runnables and lines the dependencies among them

and it is connected to an on-chip SDRAM memory device through a tree network-on-chip (NoC), see Figure 2(b). The tree is a wormhole-based topology implementing 3 simple pipelined 2-to-1 routers, so each core requires 2 hops to reach the memory [20]. Such an architecture is similar to the one used in current multi-core ECUs [9].

Under this architecture, there are two sources of interferences that can increase the WCET estimate of runnables, and so tasks: *NoC* and *memory interferences*. The maximum delay a request to both resources can suffer due to interferences is shown in Equation (1). L_{tree} is the tree traversal time. For each router we consider a round-robin arbitration policy, which makes L_{tree} independent of the number of cores, $ncores$. L_{mem} is the memory latency. More details on how L_{tree} and L_{mem} are computed for the setups considered in this paper are provided in Section 4.1.2. Note that, in the worst-case, a memory request is stalled by the rest of cores when accessing the memory.

$$UBD = L_{tree} + (ncores - 1) * L_{mem} \quad (1)$$

Note that in Equation 1, the NoC latency, i.e. L_{tree} , does not depend on the number of cores. The reason is because, the delay introduced by the NoC is overlapped with the latency of the memory access. This is so because the latency of the memory is higher than the latency of the NoC.

In order to estimate the WCET for runnables, we use OTAWA static timing analysis tool set [3]. Static timing analysis techniques rely on the construction of a cycle-accurate model of the system (in our case the multi-core processor architecture presented above) and a mathematical representation of the analyzed code (in our case the runnables) which makes possible to determine the timing behavior of runnables on that model. The mathematical representation is then processed with a linear programming solver to determine a trustworthy upper-bound of the WCET.

OTAWA is compatible with multi-core processor environments and parallel execution [16]. To do so, it considers that the maximum delay a request to both NoC and memory resources can suffer due to interferences is bounded by UBD defined above, so runnables are subject to the worst-case delay they can suffer due to interferences. As a result, the WCET estimates of runnables computed considering UBD are time-composable, i.e. their timing behaviour is independent of the runnables executed simultaneously on other cores and insensitive to allocation to the core and independent of the sharing the state of caches with other runnables.

It is important to remark that the approach presented in this paper is independent of the processor architecture and timing analysis tool considered. Therefore, any particular core count, NoC topology and timing analysis tool can be used to estimate the WCET for runnables.

3. RUNPAR ALLOCATION ALGORITHM

This section presents the main contribution of this paper: RunPar allocation algorithm that allows parallelizing AUTOSAR legacy applications, while maintaining the single-core application configuration. This allows reducing the effort of validating applications when migrating from single-core to multi-core ECUs.

In this paper, we consider a partitioning multi-core scheduling approach as it better fits current AUTOSAR standard prescriptions [15, 2]. That is, the use of static cyclic scheduling of runnables is common in AUTOSAR, making the static partition approach very likely to be adopted in a first step when moving towards multi-core ECUs.

3.1 Problem Definition

AUTOSAR allows describing a wide range of applications to cover most of the functionalities required within a car. In this paper we focus on applications with the following properties:

- Runnables exchange data through sender-receiver ports using a shared global memory.
- Client-Server communication is always synchronous, i.e. the execution of the server blocks the client until the server finishes.
- Each runnable is assigned to exactly one task.
- Tasks are executed based on either a fixed period or an interrupt. In case of interrupt-driven tasks, the period for scheduling purposes is defined as the maximum frequency at which the interrupt can occur.
- Tasks are not preempted, so once a task starts executing it runs until completion.

These application properties cover a wide range of AUTOSAR applications, including the one considered in this paper, the EMS. It remains as future work to extend *RunPar* to applications with other properties, specially task preemption.

Overall, this paper focuses on the *allocation problem* of runnables of an AUTOSAR application in *homogeneous multi-core processors*, considering a *partitioned* scheduling approach in which once a runnable has been assigned to one core it is not allowed to migrate.

We define an AUTOSAR automotive application as a set of tasks $\mathcal{T} = \{\tau_1, \dots, \tau_k\}$ executed sequentially as defined by the single-core task scheduling. Each task τ_p is represented by a direct acyclic graph $\delta_p = (\mathcal{R}_p, \mathcal{E}_p)$. The nodes in $\mathcal{R}_p = \{r_1, \dots, r_n\}$ represent the runnables that comprise the task. Each runnable r_i is characterized by a WCET estimate C_i , a period P (the same for all runnables in a task) and a deadline D , assuming *implicit deadlines*, i.e. $D = P$. The utilization u_i of runnable r_i is defined as $\frac{C_i}{P}$, where $0 \leq u_i \leq 1$. The edges in $\mathcal{E}_p$ represent communications between runnables: An edge $e_{i \rightarrow j} \in \mathcal{E}_p$ represents any communication method implemented from runnable r_i to runnable r_j , so r_j cannot start until r_i finishes. The period of τ_p is equal to the period of all of its runnables in $\mathcal{R}_p$. The *application configuration* Ψ is defined as the single-core task scheduling $\mathcal{T}$ and the runnable-to-task mapping per each task $\tau_i = (\mathcal{R}_i, \mathcal{E}_i)$.

RunPar assigns the n runnables in $\mathcal{R}_p$ to a set of m identical cores $sc = (c_1, \dots, c_m)$ respecting the run-after dependencies defined in $\mathcal{E}_p$ and the application configuration Ψ . Concretely, the allocation algorithm generates a static partition $\Phi_p = (\varphi_1, \dots, \varphi_m)$ in which a subset of $\mathcal{R}_p$ is assigned to each core. Thus, $\varphi_k = \mathcal{R}_p^k \subseteq \mathcal{R}_p$ assigns $\mathcal{R}_p^k$ to core c_k with a *cumulative utilization* defined as $u_{sum}^k = \sum_{r_i \in \mathcal{R}_p^k} u_i \leq 1$. Each runnable can be assigned to only one core. Moreover, given any two runnables $r_i \in \mathcal{R}_p$ and $r_j \in \mathcal{R}_p$, *RunPar* guarantees that r_j is allocated after r_i finishes if exists $e_{i \rightarrow j} \in \mathcal{E}_p$. Furthermore, in case a runnable consumes data from two or more runnables, it must be allocated after all of them are guaranteed to complete.

The utilization of a task τ_p , given by the longest dependent runnable chain, is the maximum cumulative utilization of all cores in Φ_p , and it is defined as $u_{\tau_p} = \max(\forall \varphi_k \in \Phi_p \mid u_{sum}^k)$. Similarly, the WCET estimate of a task τ_p is the maximum sum of C_i allocated to a core in Φ_p , and it is defined as $WCET_{\tau_p} = \max(\forall \varphi_k \in \Phi_p \mid \sum_{r_i \in \mathcal{R}_p^k} C_i)$.

In a partitioned scheduling scheme, once runnables are allocated to cores, an on-line *uniprocessor* scheduling algorithm is used. In our case, the on-line scheduler must guarantee that, when a runnable starts executing all its predecessor runnables upon which it depends have already finished.

RunPar is compatible with static time-triggered schedulers, like in AR-OS. AR-OS implements *scheduling tables* [2] that define the starting point and the order in which tasks are activated. Section 3.3 explains the changes required at AR-OS level to schedule, not only tasks but runnables as defined by *RunPar* in Φ so the starting point of runnables guarantees the fulfillment of inter-runnables dependencies. Extension of RunPar to support dynamic priority-based schedulers (e.g. rate monotonic) remains as future work.

3.2 Mapping Runnables to Cores

RunPar considers the set of tasks $\mathcal{T} = (\tau_1, \dots, \tau_k)$ that form the application, and allocates the runnables of each task τ_p into m cores. Runnables from different tasks are not allowed to be executed in parallel, forcing tasks to be executed sequentially and so respecting the single-core task scheduling. This section presents our runnable-to-core allocation algorithm, which is called for each of the tasks in $\mathcal{T}$.

Figure 4 shows the pseudo-code implementation. The algorithm takes as input the task $\tau_p = (\mathcal{R}_p, \mathcal{E}_p)$ (i.e. Figure 3) and a set of m cores $sc = (c_1, \dots, c_m)$. As output it provides a valid allocation Φ_p .

RunPar starts classifying runnables into two different types: *dependent runnables* (dR) and *independent runnables* (iR), lines 2 and 3 in the algorithm. A runnable r_i in $\mathcal{R}_p$ is dependent if there exists a runnable r_j in $\mathcal{R}_p$ and the edge $e_{i \rightarrow j}$ or $e_{j \rightarrow i}$ in $\mathcal{E}_p$. In other words, the runnable r_i produces (or consumes) data that is consumed (or produced) by r_j , creating a run-after dependence among them. Similarly, runnable r_i is independent if for all runnables in $\mathcal{R}_p$, neither exist edges $e_{i \rightarrow j}$ nor $e_{j \rightarrow i}$ in $\mathcal{E}_p$. In Figure 1, runnables r_4 and r_7 from tasks *1ms* and *5ms* respectively, are classified as independent runnables. The rest is classified as dependent.

It is important to recall that runnables can also consume data produced by other tasks or by past instances of the same task it belongs to. Such a dependence is not taken into account because the sequential execution of tasks guarantees that when a new task instance starts, all previous tasks instances (including itself) have already finished, and so the data dependence is not violated. This is the case of runnable r_7 in Figure 1.

After the classification of runnables, dR and iR are sorted, creating Sorted dependent Runnable (SdR) and Sorted independent Runnable (SiR) sets and *RunPar* spawns into two different sub-phases: allocation of dependent and independent runnables.

3.2.1 Dependent Runnables

The allocation of dependent runnables (lines 7 to 28) is based on a variant of the worst-fit decreasing heuristic [12, 4, 14] in which runnables are allocated to the least-occupied cores, i.e. cores in which u_{sum}^k is smaller. Moreover, runnables are sorted in decreasing order by its utilization, which prioritizes runnables with higher utilization to be allocated first.

The use of worst-fit heuristic guarantees a better runnable-to-core load balance, which in turn provides more parallelism among runnables with respect to other heuristics such as first-fit decreasing heuristics. This is so because data dependencies among runnables reduce considerably the possibilities in which runnables can be allocated, and so worst-fit allows selecting the less loaded core.

Moreover, *RunPar* considers the *combined utilization* (cU) of runnables instead of using their utilization (u). Section 3.1 shows

RunPar Allocation Algorithm

Input $\tau_p = (\mathcal{R}_p, \mathcal{E}_p)$: A task of the application,

$sc = (c_1, \dots, c_m)$: total number of cores

Output $\Phi_p = (\varphi_1, \dots, \varphi_m)$: A valid allocation of $\mathcal{R}_p$ into sc

```

1  $\Phi_p = \emptyset$ 
2  $dR = \forall r_i \in \mathcal{R}_p \mid \exists r_j \in \mathcal{R}_p, \exists e_{i \rightarrow j} \vee e_{j \rightarrow i} \in \mathcal{E}_p$ 
3  $iR = \forall r_i \in \mathcal{R}_p \mid \forall r_j \in \mathcal{R}_p, \nexists e_{i \rightarrow j} \wedge e_{j \rightarrow i} \in \mathcal{E}_p$ 
4
5  $\langle sdR, siR \rangle = \text{sort\_runnables\_cU}(dR, iR)$ 
6
7 // First-phase: Allocation of dependent
8 // runnables
9 forall ( $r_i \in sdR \mid r_i \notin \Phi_p$  and  $\nexists r_j \in sdR, e_{j \rightarrow i} \in \mathcal{E}_p$ ) do
10    $c = \text{worstfit}(r_i)$ 
11    $\Phi_p += \text{allocate}(c, r_i)$ 
12 endfor
13 forall ( $r_i \in sdR \mid r_i \notin \Phi_p$  and  $\exists r_j \in \Phi_p, e_{j \rightarrow i} \in \mathcal{E}_p$ ) do
14    $\text{maxdep} = 0$ 
15   forall  $c_k \in sc$  do
16     if ( $\text{maxdep} < u_{sum}^k$  and  $r_j \in \varphi_k$ ) then
17        $\text{maxdep} = u_{sum}^k$ 
18        $r_{largest} = r_j$ 
19     endif
20   endfor
21    $c = \text{worstfit\_startdef}(r_i, \text{maxdep})$ 
22    $n = \text{core}(r_{largest})$ 
23   if ( $u_{sum}^n - u_{sum}^c > 0$ ) then
24      $u_{idle} = u_{sum}^n - u_{sum}^c$ 
25      $\Phi_p += \text{allocate}(c, r_{idle})$ 
26   endif
27    $\Phi_p += \text{allocate}(c, r_i)$ 
28 endfor
29
30 // Second-phase: Allocation of independent
31 // runnables
32 forall ( $r_i \in siR \mid r_i \notin \Phi_p$ ) do
33   if ( $\exists r_{idle} \in \Phi_p$  and  $u_i \leq u_{idle}$ ) then
34      $c = \text{core}(r_{idle})$ 
35      $\Phi_p += \text{allocate}(c, r_i)$ 
36     if ( $u_{idle} - u_i = 0$ ) then
37        $\text{remove}(r_{idle}, \Phi_p)$ 
38     else
39        $u_{idle} = u_{idle} - u_i$ 
40     endif
41   else
42      $c = \text{worstfit}(r_i)$ 
43      $\Phi_p += \text{allocate}(c, r_i)$ 
44   endif
45 endfor
46
47 return  $\Phi_p$ ;
```

Figure 4: Pseudo-code implementation of the allocation algorithm.

that the utilization u_i of runnable r_i is computed based on its WCET estimate and period (C_i/P). The cU_i instead, considers the maximum sum of the utilizations of all chains of dependent runnables of r_i . For example, in Figure 1 the cU_1 of runnable r_1 from task *1ms*, is $u_1 + u_3$. Similarly, cU_3 of runnable r_3 is u_3 , and cU_2 of runnable r_2 from task *4ms* is $u_2 + \max < u_5, u_6 >$. The use

of cU guarantees that the longest chain of dependent runnables is allocated first. The function in line 5, sorts dR and iR runnables based on cU . Note that for independent runnables cU equals to u .

The algorithm works as follows. The first runnables to be allocated are those that produce data, but do not consume data from any other runnable from the same task (line 9). Runnables are allocated to cores using the worst-fit heuristic (lines 10 and 11). In Figure 1, the first allocated runnables when processing tasks $1ms$ and $2ms$ are r_1 and r_2 respectively.

In a subsequent step, the rest of consumer/producer runnables are allocated respecting run-after dependencies, i.e. a runnable is not allocated until all its precedence dependent runnables are allocated (line 13). Moreover, the time at which a runnable starts executing must be after all runnables producing its input data finish. Therefore, given a runnable r_i and its producer runnable finishing the latest r_j , r_i is allocated after r_j finishes its execution, considering its WCET estimate (C_j) to guarantee that the data will be available. To do so, the algorithm searches for the largest cumulative utilization among the cores in which the dependent runnables are allocated (lines 14-20). Then, function *worstfit_startdef* (line 21) returns the core in which r_i fits better, starting its execution after $r_{largest}$ finishes, i.e. at $maxdep$ (lines 17 and 18).

In case the cumulative utilization of the core c in which r_i will be allocated (u_{sum}^c), is smaller than the time at which r_i must at least start, i.e. after the cumulative utilization of the core n in which $r_{largest}$ is allocated (line 22) (u_{sum}^n), an *idle runnable* (r_{idle}) is inserted (lines 23-26) with its corresponding utilization (u_{idle}) (line 24). In other words, idle runnables delay the start of runnables until all their input data is guaranteed to be available. Idle regions will then be used to allocate independent runnables as explained in the next section. Note that u_{sum}^c can be considered when allocating dependent runnables because they are scheduled contiguously, introducing an r_{idle} if required.

3.2.2 Independent Runnables

The allocation of independent runnables occurs once all dependent runnables have been allocated (lines 30 to 45). The reason is that independent runnables have the freedom to be allocated at any point in time within its corresponding period. As already pointed above, it is important to remark that independent runnables are in fact dependent runnables that consume data produced by other tasks or by past instances of the same task it belongs to. The EMS does not contain any purely independent runnable.

RunPar is based on a worst-fit decreasing heuristic that prioritizes runnables by sorting them based on their utilization, so independent runnables with higher utilization are allocated first. The algorithm first checks if a runnable fits within any idle region (line 33). If it fits, the runnable is allocated within the same core and time slot assigned to the idle runnable (lines 34 and 35), and the utilization of the idle runnable is reduced (line 39) or even eliminated (line 37). If it does not fit, the runnable is allocated using a worst-fit decreasing heuristic (lines 42 and 43).

3.3 Allocation Algorithm Solution: Φ

If a valid solution is found, i.e. all dependent and independent runnables are allocated, and so the algorithm returns a valid allocation Φ_p of task τ_p , in which each runnable (including idle regions) is assigned to a core (line 47). A valid solution means a task will respect its deadline i.e. all runnables comprising the task will execute within their period.

Figure 5 shows the resultant Φ_{1ms} , Φ_{4ms} and Φ_{5ms} of $1ms$, $4ms$ and $5ms$ tasks respectively, when applying *RunPar* to the application presented in Figure 1, executing in a two-core processor. C_i is

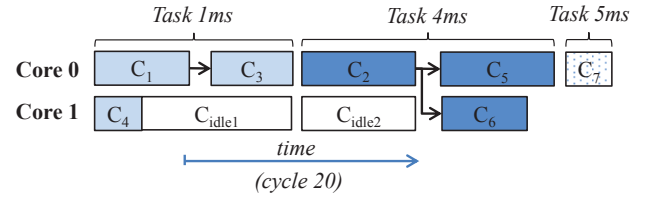


Figure 5: Valid allocation (Φ) of the automotive application presented in Figure 1 in a two-core processor, executing cycle 20. C_i is the WCET estimate of runnable r_i .

the WCET estimate of runnable r_i ; C_{idle} is the time slot in which the core executes an idle runnable.

In order to support *RunPar*, AR-OS scheduling tables has to be extended to schedule the runnables from task τ_p as defined by Φ_p . To that end, each task entry in the scheduling table is extended with a new *runnable scheduling table* that defines the starting point, the order and the core of each of the runnables that form the task. It is important to remark that runnables are scheduled based on their WCET estimates (C), guaranteeing that no race conditions can occur, i.e. consumer runnables do not start executing until all its dependent producer runnables finish, even if they execute for their WCET estimates.

3.4 Validating the Single-core Task Scheduling

Our allocation strategy reduces the WCET estimate of tasks, by exploiting runnable-level parallelism of application's task, but does not necessarily reduce the response time of the overall application. The reason is because our strategy maintains the single-core task scheduling, and so the starting point and order in which tasks are executed remains the same. However, the overall time the processor is used by the application is reduced because tasks execute faster, providing extra computational space in the task scheduling to allocate new application functionality (or even to allocate other tasks from different applications).

The single-core task scheduling remains valid in the multi-core ECU if the resultant task utilization after parallelizing runnables (u_{τ_p}), is smaller than or equal to the task utilization when executing runnables sequentially ($useq_{\tau_p}$).

It could be the case, however, that the WCET estimate reduction obtained by executing runnables of a task in parallel is not enough to hide the overhead introduced due to interferences when accessing the hardware shared resources [18, 17], making u_{τ_p} be higher than $useq_{\tau_p}$. In this case, it is required to validate that the single-core task scheduling is still valid, and the increment of the utilization of a task can be compensated by the utilization reduction of the other tasks.

If u_{τ_p} is higher than $useq_{\tau_p}$, the system can emulate the single core execution. All runnables of the tasks are executed sequentially in one core while other cores are idle and do not execute anything. Since we know that there will be no interference from other cores, we can safely use WCET estimates obtained for single core execution (assuming $UBD = 0$) and by doing so decrease the WCET estimate of the task.

4. RESULTS

4.1 Experimental setup

4.1.1 EMS application

Our allocation algorithm has been evaluated with a real automotive application, an Engine Management System (EMS). An EMS is a typical automotive embedded real-time system in which the amount of fuel and the injection time are fundamental for smooth revolutions of the engine. The injection time and fuel amount depend on the state and the rotation speed of the engine, which changes continuously during operation. Hence the EMS requires an update either from cyclic executed tasks, whose period ranges from one millisecond to one second, or a crank-angle interrupt triggered task. Overall, the examined engine control contains eleven cyclic tasks, with periods of 1, 4, 5, 8, 16, 20, 32, 64, 96, 128 and 1024 ms, and one crank-angle triggered task, with a minimum period of 1 ms. The crank-angle task has a period that varies depending on the revolutions of the engine. In order to guarantee the correct execution of the task, the smallest possible period is considered for scheduling purposes. The execution is non-preemptive, i.e. the execution of a task cannot be interrupted by any other task.

4.1.2 WCET analysis tool and Processor Setup

In order to compute the WCET estimates (C) of runnables, we use a static timing analysis tool OTAWA. OTAWA models the multi-core processor architecture presented in Section 2.2 in which every request that accesses the NoC and the on-chip memory is delayed by UBD , so the runnable is subject to the worst-case delay that it can suffer due to interferences. We consider 2-core and 4-core processors, with private per-core scratchpads for instructions and write-through data caches. For both processor configurations cores are connected through a tree NoC to the on-chip RAM memory. We consider $L_{tree} = 1$ cycles for a 2-core processor, $L_{tree} = 2$ for a 4-core processor (i.e. each core has to traverse 1 and 2 routers for 2- and 4-core processors respectively) and 1-cycle routers. The memory latency is set to $L_{mem} = 10$ cycles. This configuration provides a $UBD = 11$ cycles for the 2-core architecture and $UBD = 32$ cycles for the 4-core architecture (see Section 2.2 for further details).

The approach presented in this paper is independent of the processor architecture and the timing analysis method, so other architectures and tools can be used to compute the WCET estimates of runnables.

4.1.3 Metrics

In order to evaluate our allocation algorithm, we consider the $WCET_{\tau}$ speed-up metric, defined as $\frac{seqWCET_{\tau}}{parWCET_{\tau}}$, where $seqWCET_{\tau}$ is the WCET estimate of task τ executing runnables sequentially, and $parWCET_{\tau}$ is the WCET estimate of task τ executing runnables in parallel as defined in Φ by our allocation algorithm.

In case of $seqWCET_{\tau}$, the C of runnables is computed assuming $UBD = 0$, i.e. runnables do not suffer any delay due to interferences. In case of $parWCET_{\tau}$, C is computed assuming $UBD = 11$ and $UBD = 32$ for the 2-core and 4-core processor architecture configurations as defined before.

Moreover, in order to evaluate the maximum runnable-level parallelism that our algorithm is able to exploit, we use the *no interference* $WCET_{\tau}$ speed-up metric, in which the effect of interferences is assumed 0. To do so, the $parWCET_{\tau}$ of the 2-core and the 4-core processor architectures are computed assuming no interferences, i.e. $UBD = 0$. Considering WCET estimates in which interferences are not accounted, allows discarding the pessimism

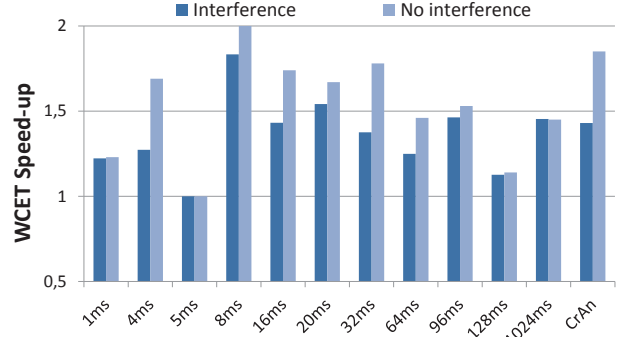


Figure 6: WCET speed-ups of EMS' tasks in a 2-core processor architecture, in which the WCET estimation accounts and discards the impact of interferences (labeled as *interferences* and *no interferences* respectively).

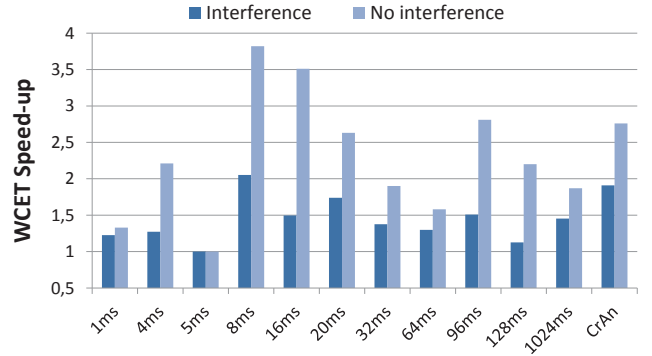


Figure 7: WCET speed-ups of EMS' tasks in a 4-core processor architecture, in which the WCET estimation accounts and discards the impact of interferences (labeled as *interferences* and *no interferences* respectively).

introduced by timing analysis tools because of sharing processor resources such as the NoC and the memory as well as the actual contention in shared resources.

4.2 WCET Speed-up of EMS tasks

Figures 6 and 7 show the WCET speed-ups of EMS' tasks obtained with RunPar in a 2-core and 4-core processor architectures respectively (the higher the better). In both cases, RunPar is used with (1) WCET estimates that consider the effect of interferences (labeled as *interferences*) and (2) WCET estimates assuming no interferences (labeled as *no interferences*).

In the 2-core processor architecture, RunPar is capable of exploiting the performance opportunities of the multi-core ECU, improving the WCET performance of almost all tasks with respect to single-core processor architectures. The speed-up of the 8 ms task is significant, 1.8x, being close to the ideal speed-up in a 2-core processor architecture, i.e. 2x. Tasks 16, 20, 96 and 1024 ms, and the CrAn task also exhibit high speed-ups, around 1.5x. Such a WCET speed-up represents a WCET reduction of 45% in case of 8ms task, and around 33% for tasks 16, 20, 96 and 1024 ms.

When we compare the speed-ups obtained with and without the effect of interferences in the computation of the WCET estimates, we observe the performance degradation of multi-core execution due to sharing the NoC and the memory processor resources. Such

a degradation is further augmented by the pessimism introduced by the timing analysis tools. In our case, OTAWA assumes that each memory requests is delayed by UBD cycles. Hence, if no interference is assumed, the 8 ms task achieves the maximum speed-up, i.e. 2x, and CrAn and 4, 16 and 32 ms tasks increase the speed-up close to 1.8X for the 2-core processor.

However, achieving maximum levels of parallelism not only depends on the effect of interferences, but also on the characteristics of the task, e.g. number of runnables, inter-runnable dependencies. This is the case of tasks 1, 20, 64, 96, 128 and 1024 ms tasks for the 2-core processor, in which the WCET estimates are not affected much by interferences.

The 5 ms task is composed of one single runnable which is executed in a single core, and so having no benefit at all of parallel execution.

In the 4-core processor architecture, the impact of interferences on the WCET estimates increases significantly (OTAWA assumes UBD equal to 32 cycles), which makes the performance benefits brought by multi-core execution being higher than those for the 2-core processor but relatively worsen as the WCET degradation due to considering interferences grows noticeably. Thus, the speed-up of EMS tasks slightly increases when moving from a 2-core to a 4-core processor architecture, achieving speed-ups of 2.1x in case of 8 ms and CrAn tasks, and 1.7x in case of 20 ms task. Such a WCET speed-up represents a WCET reduction of 53% in case of 8 ms and CrAn tasks, and 41% for 20 ms task.

When discarding the effect of interferences in WCET estimates, the speed-up of 8, 16, 96 ms and CrAn tasks increases significantly, with speed-ups of 3.8x, 3.5x, 2.8x and 2.7x respectively. In the case of 1, 64 and 1024 ms tasks the WCET speed-up is not affected much by interferences. In fact, if we compare the WCET speed-up of these three tasks obtained in the 2-core and the 4-core processor architectures, we observe the exact same performance.

We conclude that RunPar effectively exploits the maximum runnable-level parallelism exposed in EMS' tasks in a multi-core ECU. Unfortunately, the impact of interferences due to shared resources reduces the benefits brought by multi-core execution. Such negative effect is increased as the number of ECU core increases. Results could be improved by using a timing analysis tool delivering less pessimistic WCET estimates or a processor architecture where the maximum effect of interferences is lower so UBD is also lower. Investigating new timing analysis techniques and processor architectures to reduce the impact of interferences is out of the scope of this paper, though part of our current work.

4.3 Increasing Overall Available CPU Capacity

The parallel execution of runnables reduces the WCET estimates of tasks, which in turn reduces the task utilization (u_τ) and so reduces the CPU capacity required by the EMS application with respect to the single-core execution. This extra capacity can be then exploited for executing new application functionality or other automotive applications.

Figure 8 shows the utilization of EMS' tasks when being allocated on a single core ECU (labeled as *sequential*) and on a 2-core and 4-core ECU using the RunPar. Note that in all processor configurations, the task scheduling is the same as our allocation strategy maintains the single-core task scheduling.

The utilization (u_τ) of all EMS' tasks (except 5ms task because it contains one single runnable) is reduced. It is of special interest the tasks with highest utilization i.e. those whose WCET estimation is higher with respect to its period. This is the case of tasks 16 and 32 ms and CrAn with utilization reductions of 8, 6 and 6 percentage

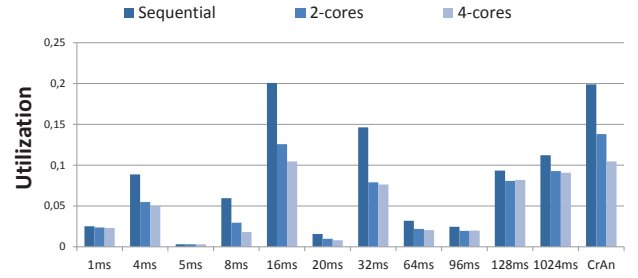


Figure 8: Utilization of EMS' tasks being allocated on a single-core, 2-core and 4-core ECU (labeled as *sequential*, *2-core* and *4-core* respectively).

points (pp) respectively in case of the 2-core ECU (a reduction of the utilization of, for instance, 8 percentage points in case of 16 ms task means that the utilization goes from 0.20 to 0.12). In the case of 4-core ECU, these 3 tasks have utilization reduced by 10, 7 and 9 percentage points respectively. These three tasks are, in fact, the longest tasks of the EMS application with respect to their period, and so are the tasks that benefits the most of a reduction on its WCET estimation. This is not the case of 8 ms task, which despite being the task that achieves the highest WCET speed-up (see Figures 6 and 7), this is translated in an utilization reduction of only 3 percentage points.

Overall, the reduction on EMS' tasks utilization represents an increment of the CPU capacity of 31% and 42% for the two-core and four-core ECUs respectively. Such an extra capacity can be then re-used for executing new application functionality or scheduling other applications using AUTOSAR recommendations for executing in a multi-core ECU environment [1]. Alternatively, it can be used for applying voltage and frequency techniques, so the energy consumed by ECU can be reduced [19].

4.4 The Cost of Maintaining Tasks' Ordering

Our allocation strategy maintains the application configuration by parallelizing only runnables that belong to the same task, and so forcing tasks to be executed sequentially. In order to evaluate the potential performance lost due to maintaining the sequential execution order of tasks, we perform the following experiment:

We first group all EMS' runnables into a unique task representing the scenario that happens when all the tasks execute one after the other, so runnables from different tasks can now execute in parallel, and use RunPar to allocate them to a four-core ECU. This scenario, which occurs around every 15 seconds, has the greatest opportunity to exploit the parallelism among all runnables. This new task includes the new set of communication dependency edges between runnables from different tasks (e.g. in Figure 1, a task grouping the three tasks 1ms, 4ms and 5ms includes $e_{1 \rightarrow 4}$ and $e_{4 \rightarrow 5}$ edges). Then, we compare the WCET estimation of the new task ($WCET_{new-task}$) with the sum of WCET estimates of all EMS' task ($WCET_{EMS-tasks}$) applying RunPar to each task as done in Section 4.2.

The difference between $WCET_{EMS-tasks}$ and $WCET_{new-task}$ shows the performance opportunities lost due to sequential execution of tasks. It is important to remark that the purpose of this experiment is only to evaluate the parallelism lost due to maintaining sequential execution of tasks. Using a unique task to group all runnables would change the application configuration, requiring to re-analyze the complete application, which is contrary to the purpose of our allocation strategy.

$WCET_{new-task}$ is around 5% lower than $WCET_{EMS-tasks}$. As a result, the small WCET estimation reduction benefits due to exploiting runnable-level parallelism among tasks does not compensate the cost of re-validating and re-calibrating the application.

5. RELATED WORK

In partitioned approaches, finding an optimal allocation is an NP-hard problem in the strong sense [8] and so non-optimal solutions derived from the use of bin-packing heuristics are typically used [12, 4, 14]. Most of scheduling works proposed in the critical real-time embedded system domain consider independent processes that do not communicate among them.

Along this line, Monot et al. [13] presented recently a scheduling algorithm for multi-source AUTOSAR applications on multi-core ECUs. Their approach groups all runnables with inter-runnable dependencies into a single task, allowing tasks to be scheduled independently. In the next step, a runnable scheduling is build on each core independently of other cores. Overall, one of the main objectives of the approach is obtaining uniform utilization of the cores during the application execution. The scheduling algorithm presented in [13] has two main differences with our allocation strategy. First, it requires changing the application configuration, i.e. it reassigns runnables to tasks, creating a new runnable and task scheduling, which causes the need for re-validating legacy applications. Second, it assumes little dependencies among runnables so a sufficient number of tasks can be created to exploit parallelism in multi-core ECUs. This is not the case of the EMS in which almost all of runnables have inter-runnable dependencies (see Figure 3). In fact, applying this approach to the EMS would lead to schedule all runnables into a single core.

Faragardi et al. [5] presented a scheduler for AUTOSAR applications on multi-core ECUs having as scheduling criteria to minimize the worst-case communication delays among runnables scheduled in different cores. This solution assumes that the runnable-to-task mapping that minimizes the communication among cores is already given, and so it focuses only on the task scheduling. This approach requires changing the application configuration, causing the need for re-validating legacy applications.

An interesting work to mention is the one presented by Sinnen et al. [22] targeting general purpose processor architectures. In [22] a task scheduling approach is proposed to make the scheduler aware of the cost of inter-processor communication in a general purpose architecture. The scheduler is a variant of the list scheduling in which tasks are allocated in two phases following a similar approach to the one used by *RunPar*, i.e. allocating first dependent tasks and then independent tasks. The main difference with *RunPar* is that the purpose of [22] is to improve the execution time of a parallel application in a general purpose by minimizing the communication delay.

To the best of our knowledge, this is the first allocation algorithm that exploits runnable-level parallelism instead of task-level parallelism and maintains the application configuration. Schneider et al. [21] pointed that the parallel execution of tasks can break the mutual exclusion relations between the critical sections present in applications configured to be executed in single-core processors, leading to race conditions. Therefore, considering runnables as the UoS is the best approach to avoid race conditions.

6. CONCLUSIONS

This paper presents a new allocation strategy in which the single-core configuration of AUTOSAR applications, i.e. its runnable-to-task mapping and single-core task scheduling, is maintained when

migrating from single-core to multi-core ECUs, so the effort of re-validating the applications is minimized.

To do so, we present *RunPar*, a new allocation algorithm for AUTOSAR automotive applications with runnables highly connected among them, which considers runnables, and not tasks, as the UoS. *RunPar* is based on a variant of the worst-fit decreasing heuristic which prioritizes runnables with higher *combined utilization*, so the longest chain of dependent runnables is allocated first. Moreover, in order to maintain the single-core task scheduling, only runnables from the same task are allowed to be executed in parallel. By doing so, tasks are forced to execute sequentially, allowing to maintain the task ordering of the application's single-core task scheduling. This approach requires minimum modifications at AR-OS level.

The allocation has been evaluated with a real automotive application, an Engine Management System (EMS), which controls the injection time and amount of fuel in a diesel engine and is composed of more than one thousand highly connected runnables, grouped into twelve tasks.

Results show that *RunPar* algorithm reduces on average WCET of the EMS' tasks by approximately 26% and 30% in the case of 2-core and 4-core architectures respectively, representing a WCET speed-up of 1.35x and 1.43x. Such a WCET speed-up translates into an increment of the CPU capacity of 31% and 42% for the two-core and four-core ECUs respectively.

Therefore, it stands as a necessary step for porting AUTOSAR automotive applications from single core and exploiting the potential performance of multi-core ECUs.

Acknowledgments

The authors would like to thank Jorge Alberto Becerril Sandoval for his assistance in analysing the EMS during his internship and his Master's thesis. The research leading to these results has been funded by the European Union Seventh Framework Programme under grant agreement no. 287519 (parMERASA). This work has also been funded by the Ministry of Science and Technology of Spain under contract TIN2012-34557. Miloš Panić is funded by the Spanish Ministry of Education under the FPU grant FPU12/05966.

7. REFERENCES

- [1] Guide to multi-core systems.
<http://www.autosar.org>, 2014.
- [2] AUTOSAR GbR. AUTomotive Open System ARchitecture (AUTOSAR) Operating System.
<http://www.autosar.org>, February 2013.
- [3] C. Ballabriga, H. Casse, C. Rochange, and P. Sainrat. Ottawa: an open toolbox for adaptive wcet analysis. In *Workshop on Software technologies for Embedded and Ubiquitous Systems (SEUS)*, 2011.
- [4] S.K. Dhall and C. L. Liu. On a real-time scheduling problem. In *Operation Research*, pages 127–140, 1978.
- [5] Hamid Reza Faragardi, Björn Lisper, and Thomas Nolte. Towards a communication-efficient mapping of AUTOSAR runnables on multi-cores. In *Emerging Technologies and Factory Automation (ETFA)*, pages 1–5, 2013.
- [6] Freescale. *P4080 QorIQ Integrated Processor Hardware Specifications, Rev. 0, 02/2011, Doc. Num. P4080EC*, 2011.
- [7] Aeroflex Gaisler. *Quad Core (LEON4 SPARC V8) Processor - (LEON4-NGMP-DRAFT) - Data Sheet and Users Manual*, 2011.
- [8] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*.

1990.

- [9] Infineon. *AURIX - TC27x B-Step, 32-bit Single-Chip Micro-controller, User's Manual, v14.1*, 2014-02.
- [10] ISO. *Road vehicles – Functional safety – Part 6: Product development at the software level, Ref. num. ISO 26262-6:2011(E)*, 2011.
- [11] Javier Jalle, Jaume Abella, Eduardo Quinones, Luca Fossati, Marco Zulianello, and Francisco J. Cazorla. Deconstructing bus access control policies for real-time multicores. In *International Symposium on Industrial Embedded Systems (SIES)*, pages 31–38, 2013.
- [12] Jörg Liebeherr, Almut Burchard, Yingfeng Oh, and Sang H. Son. New strategies for assigning real-time tasks to multiprocessor systems. *IEEE Transactions on Computers*, 44(12):1429–1442, 1995.
- [13] Aurelien Monot, Nicolas Navet, Bernard Bavoux, and Françoise Simonot-Lion. Multisource software on multicores automotive ecus - combining runnable sequencing with task scheduling. *IEEE Transactions on Industrial Electronics*, 59(10):3934–3942, 2012.
- [14] Yingfeng Oh and Sang H. Son. Allocating fixed-priority periodic tasks on multiprocessor systems. *Real-Time Systems*, 9(3):207–239, 1995.
- [15] OSEK/VDX Group. Offene Systeme und deren Schnittstellen fuer die Elektronik im Kraftfahrzeug (OSEK) Operating System Specification. <http://www.osek-vdx.org>, February 2005.
- [16] Haluk Ozaktas, Christine Rochange, and Pascal Sainrat. Automatic wcet analysis of real-time parallel applications. In *13th International Workshop on Worst-Case Execution Time Analysis WCET, Satellite Workshop of ECRTS*, 2013.
- [17] M. Paolieri, E. Quinones, and F. J. Cazorla. Timing effects of the memory system in real-time multicores integrated architectures: Problems and solutions. In *Transactions on Embedded Computing Systems*, 2012.
- [18] M. Paolieri, E. Quinones, F.J. Cazorla, G. Bernat, and M. Valero. Hardware support for WCET analysis of hard real-time multicores systems. In *International Symposium on Computer Architecture (ISCA)*, 2009.
- [19] Padmanabhan Pillai and Kang G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *SOSP*, pages 89–102, 2001.
- [20] A. Roca, C. Hernandez, J. Flich, F. Silla, and J. Duato. Enabling high-performance crossbars through a floorplan-aware design. In *Intl. Conf. Parallel Processing (ICPP)*, pages 269–278, 2012.
- [21] Jorn Schneider, Michael Bohn, and Robert Robger. Migration of automotive real-time software to multicores systems: First steps towards an automated solution. In *22nd EUROMICRO Conference on Real-Time Systems, WIP Session*, 2010.
- [22] O. Sinnen, L.A. Sousa, and F.-E. Sandnes. Toward a realistic task scheduling model. *Parallel and Distributed Systems, IEEE Transactions on*, 17(3):263–275, 2006.
- [23] Theo Ungerer, Francisco Cazorla, Pascal Sainrat, Guillem Bernat, Zlatko Petrov, Christine Rochange, Eduardo Quinones, Mike Gerdes, Marco Paolieri, Julian Wolf, Hugues Casse, Sascha Uhrig, Irakli Guliashvili, Michael Houston, Floria Kluge, Stefan Metzloff, and Jorg Mische. Mera: Multicores execution of hard real-time applications supporting analyzability. *IEEE Micro*, 30(5):66–75, 2010.
- [24] Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 28(7):966–978, 2009.