A Resource Efficient Framework to Run Automotive Embedded Software on Multi-core ECUs

Hamid Reza faragardi ^a, Björn Lisper ^a, Kristian Sandström ^b, Thomas Nolte ^a

^a*MRTC/Mälardalen University
P.O. Box 883, SE-721 23 Västerås, Sweden
Email: {hamid.faragardi, bjorn.lisper, thomas.nolte}@mdh.se*
^b*RISE SICS,
SE-721 78 Västerås, Sweden
Email: kristian.sandstrom@ri.se*

Abstract

The increasing functionality and complexity of automotive applications requires not only the use of more powerful hardware, e.g., multi-core processors, but also efficient methods and tools to support design decisions. Component-based software engineering proved to be a promising solution for managing software complexity and allowing for reuse. However, there are several challenges inherent in the intersection of resource efficiency and predictability of multi-core processors when it comes to running component-based embedded software. In this paper, we present a software design framework addressing these challenges. The framework includes both mapping of software components onto executable tasks, and the partitioning of the generated task set onto the cores of a multi-core processor. This paper aims at enhancing resource efficiency by optimizing the software design with respect to: 1) the inter-software-components communication cost, 2) the cost of synchronization among dependent transactions of software components, and 3) the interaction of software components with the basic software services. An engine management system, one of the most complex automotive sub-systems, is considered as a use case, and the experimental results show a reduction of up to 11.2% total CPU usage on a quad-core processor, in comparison with the common framework in the literature.

1. Introduction

The increased use of embedded systems in automotive applications has led to a situation where a considerable share of the total production cost of a car or truck is now being allocated to electronic equipment. In a modern car more than one hundred Embedded Control Units (ECUs) are used to execute a large number of software functions. The development and management of such a large software system requires the use of a standard software architecture. AUTOSAR (Automotive Open System Architecture) [1] is the standard software architecture developed for automotive systems. The basic idea of AUTOSAR is derived from a component-based software design model where a system is divided into a set of loosely-coupled software components. AUTOSAR not only provides a methodology and standardized work-flows for development of software components, but it also provides a set of basic software services along with an abstraction of hardware devices [2].

The automotive industry is migrating from traditional single-core processors to parallel multi-core processors, i.e., single-core ECUs are being replaced by multi-core ECUs. The reason behind this shift is mainly due to the tremendous growth in size, number and computational complexity of software features, emerging as a result of using modern automotive facilities, such as driver assistance technologies (e.g., smart cruise control system), active safety systems (e.g., driver inattention monitoring systems), and interactive entertainment platforms [3]. If more powerful processors are considered as the solution to this growth in demand, then due to longer pipelines and higher clock frequencies, both energy consumption and the number of non-deterministic run-time behaviors (timing anomalies) will increase. Therefore, multi-cores are being widely touted as an effective solution, offering a better performance per watt along with a high potential for scalability [4]. Moreover, multi-core processors are efficiently able to co-host applications with different criticality levels, allowing for the co-hosting of non-safety and safety-critical applications on the same ECU. Even though multi-core processors offer several benefits to embedded systems, it is more complicated to efficiently develop an embedded software on a multi-core ECU than on a traditional single-core ECU. In the recent versions of AUTOSAR (since version 4.0)

a few considerations are interleaved to support multi-core ECUs, nevertheless the AUTOSAR standard still requires extensions and modifications in order to become mature enough to fully utilize the potential advantages of multi-core ECUs.

An AUTOSAR-based software application consists of a set of Software Components (SWCs). The interaction of SWCs creates a variety of timing dependencies due to scheduling, communication and synchronization effects that are not adequately addressed by AUTOSAR [5]. Each software component comprises a set of runnable entities (runnables for short); a runnable is a small piece of executable code. Similar to other component-based software, an AUTOSAR-based software is constructed using interaction among runnables, sending a lot of messages among the runnables. The runnables should be mapped onto a set of Operating System (OS) tasks. We call the process of assigning runnables to tasks, mapping. The mapping directly affects the schedulability of the created task set. The output of the mapping process is stipulated in the task specifications, indicating how many tasks that are required to be in the system, and which subset of runnables that are assigned to each task respectively. AUTOSAR only provides a few basic rules for mapping runnable entities, while the rules are regardless of: the communication structure between the runnables; the shared resources and synchronization requirements between the runnables; hardware architecture details such as whether the hardware architecture of the system is a single-core or a multi-core. Therefore, relying on such naive rules does not necessarily result in a desirable mapping in terms of resource efficiency of the system.

The generated task set should be allocated to the cores of a multi-core processor. AUTOSAR uses partitioned fixed-priority scheduling to schedule tasks, where each task is statically allocated to a core (i.e., core binding cannot be changed at run time), and each core is scheduled using a single-core fixed-priority policy. Therefore, the priority of each task must be specified as well as the core to which a task should be allocated (partitioning).

The problem becomes more challenging whenever there are runnables being shared between several transactions; a transaction is formed by a sequence of runnables. Basically, a transaction corresponds to a mission in the system, for example, the transactions in a car could be: the anti-lock braking system; an engine control process; or

temperature control. Therefore, the system can conceptually be considered as a set of transactions. In several applications there are dependent transactions, for example, the cruise control transaction calls the speed control function (runnable) to increase or decrease the current speed of the car, while this function also is invoked by the collision avoidance system. When dependent transactions are considered, more details should be taken into account, such as providing synchronization for dependent transactions, and attempting to minimize the cost of synchronization.

Both the mapping of runnables to tasks and the partitioning of tasks among the cores should be performed in such a way that in addition to the guarantee of schedulability of the system, the resource efficiency of the system is optimized. We particularly concentrate on the total CPU utilization of the cores as one of the main metrics of resource efficiency in a multi-core computing system. In other words, minimizing the processing load of a given workload can lead to enhancement of the resource efficiency of the system. Then the remaining processing capacity can be used for other proposes, such as: execution of non-real-time applications, fault recovery and checkpointing for reliability considerations. In order to minimize the total load of the processors of a multi-core ECU, we attempt to reduce both the inter-runnable communication cost and the waiting time caused by the synchronization between dependent transactions. We distinguish between different communication cost depending on if the runnables that are communicating with each other are allocated within the same task, or if they are allocated to different tasks on the same core or on different cores.

Contribution: In this paper, we propose a solution framework to optimize the resource efficiency of software executing on multi-core processors in the automotive domain. The solution framework includes three different methods for both mapping of runnables to tasks and for partitioning of tasks among cores. In order to evaluate the proposed solution framework, we have developed a simulation platform on which a set of experiments are conducted, derived by a real-world automotive benchmark. Several effective parameters such as the number of cores, the number of runnables and transactions, the size of data communication, and the dependency ratio of transactions are considered in the experiments. Three alternative approaches for the problem are also discussed and implemented, and these approaches are compared with the proposed

solution framework to demonstrate their respective efficiency. The main contributions of this paper are the following:

- 95 1. We propose a feedback-based solution framework for execution of component-based embedded software on a multi-core processor, subject to the minimization of the processing load. In this framework not only the allocation of tasks to cores is performed according to both the communication of tasks and synchronization among them, but the tasks configuration is also refined with respect to the task
100 allocation.
2. Dependent transactions (shared runnables) are addressed.
3. Alternative approaches are discussed and compared to the proposed solution framework, indicating directions for future work.

Outline: The rest of this paper is organized as follows. In Section 2 a brief survey on related work is presented. The problem is described in detail and assumptions are defined in Section 3. The solution framework is introduced in Section 4. In Section 5 the performance of the solution framework is assessed in comparison with other alternative approaches. Finally, concluding remarks and future work is discussed in Section 6.

110 2. Related Work

The problem of optimal partitioning of real-time periodic tasks on a multi-processor where each processor executes a fixed priority scheduling algorithm, in particular the Rate Monotonic (RM) algorithm, was shown to be NP-hard [6]. As a result, research efforts have focused on the development of suitable heuristic algorithms [7], mostly
115 bin packing variations [8, 9], which are able to find a near optimal partitioning in a reasonable execution time. Different criteria are considered as the optimality of a partitioning problem, such as: minimizing the required number of processors [10], improving load balancing to increase parallelism [11], minimizing the inter-tasks communication time [12, 13] and minimizing energy consumption [14].

120 A large number of studies have been conducted to solve the challenges related to static allocation of communicating real-time tasks to a set of processors [15, 16, 17]. In such studies, the task set is often described as an acyclic directed graph where the tasks indicate nodes, and the edges between the tasks display either data dependency [18] or triggering [19]. In [20] a holistic solution containing task allocation, processor schedul-
125 ing and network scheduling, was presented. They applied Simulated Annealing (SA) to find an optimal task allocation in a heterogeneous distributed system. In [16] two algorithms based on the Branch and Bound (BB) technique were proposed for the static allocation of communicating periodic tasks in distributed real-time systems. The first technique assigns tasks to the processors and the latter schedules the assigned tasks
130 on each processor. Due to the exponential nature of BB, it fails in finding a solution for most real-world sized problems. In [21] this problem is solved for a more general communication model where the tasks can send data to each other at any point of their execution – not necessarily at the end point of their life time. In [22] a mixed linear integer programming formulation is proposed to model the problem where, in
135 addition to find a proper allocation of tasks to the processing nodes, task priorities and the minimum cost of underlying hardware are taken into account. Similarly, [18] investigated the problem of task allocation, priority assignment and signal to message mapping however, with an overall goal to minimize end-to-end latencies.

The above-mentioned studies have mainly been dedicated to the task allocation
140 problem on a single-core distributed system. Although the proposed solutions for distributed single-core systems provide us useful ideas, concentration on inter-core communication properties in multi-core systems leads to higher efficiency and predictability. There are several works considering the problem on a multi-core system [23, 24, 25], however they ignore that a prominent complexity exists in the problem
145 for automotive systems, namely, mapping of runnables to tasks. An improper task set, even with an optimal task allocation, may not result in a reasonable system performance (this statement will be verified by our experiments).

In the context of automotive systems, most of the papers assume that the task set has been created either in advance [11] (e.g., the case for legacy software), or according
150 to the *Common Mapping Approach* in which the runnables with the same activation

pattern are mapped to the same task [26]. They then apply one of the bin packing variations as the partitioning algorithm, which is quite common not only in academia but also in industry. In practice most automotive applications are assigned to cores according to one of the bin packing variations, because they are easy to implement and
155 fairly efficient.

In [27], to deploy a set of AUTOSAR software components onto a set of distributed ECUs connected by a CAN network [28], a heuristic algorithm is introduced, inspired from a clustering algorithm [29]. They propose a bi-objective optimization problem where the former objective is to minimize the load on the network, and the latter is to
160 uniformly distribute SWCs among the ECUs.

Panic et al. [11] presented a framework to migrate legacy AUTOSAR tasks from single-core to multi-core ECUs, in which the task set is provided in advance, and they do not aim to change the tasks' configurations. They apply a variant of the worst fit heuristic (namely, decreasing worst fit) to allocate dependent runnables to cores where
165 the target is to provide a load-balanced solution. The load balancing is also the goal in [30], where at the beginning they ignore the dependencies between the runnables and propose a heuristic based on the decreasing worst fit. They then use the release jitter to cover dependency between the runnables. Nevertheless, the worst fit algorithm apparently is not suitable to deal with our problem since it attempts to allocate the task
170 set onto the cores in a balanced manner, resulting in using a high number of cores, whereas in our problem we aim to consolidate tasks onto a minimum number of cores to reduce both the communication cost and remote blocking time.

The common mapping approach to create a task set (i.e., mapping runnables with the same period to the same task) can reduce parallelism, since it restricts us to allocate
175 all runnables with the same period to one core which may not be efficient. Apart from that, in some applications where most of the runnables have the same period, this approach may not even be feasible and more considerations are required. On the other hand, when most of the runnables have different periods, the number of tasks can go as high as above the maximum number of tasks allowed by AUTOSAR OS. To deal with
180 this problem, task merging approaches have emerged to diminish the number of tasks, such as [31, 32]. Nonetheless, the merging approaches presented in these papers are

not directly applicable to our problem, because they attempt to minimize the number of tasks even if it results in a higher processor utilization, as long as the task set on each core stays schedulable with respect to the scheduling policy in use on the processor.

185 However, decreasing the number of tasks is desirable for us as long as it does not increase the CPU utilization, and just decreasing the number of tasks itself is not our goal.

Recently, in [33] the problem of assigning a set of runnables into tasks was discussed where the target objective is to minimize the end-to-end latencies, however the requirements on support of mutually exclusive resources was not taken into account.

190 The authors apply a Genetic Algorithm (GA) to cope with the problem. Nevertheless, we will show that it is often not possible to obtain an optimal solution (or even a feasible solution) whenever mapping of runnables to tasks and partitioning are not interleaved. Moreover, in [34, 35], maximizing the parallelism of runnables across the

195 cores of a multi-core ECU is targeted to speedup the system, while the same data-flow existing in a single-core ECU should be preserved on multi-core ECUs to guarantee the same functional behavior without introducing exhaustive additional validation and testing efforts.

3. Problem Modeling

200 This section starts with a short overview of the AUTOSAR architecture, followed by the formal definition of the problem addressed in this paper. Then, a communication time analysis is suggested, considering the architecture of the target ECU. Finally, the problem is formulated as an optimization problem.

3.1. The AUTOSAR Architecture

205 An AUTOSAR-based application consists of a set of interconnected SWCs. Each SWC specifies its input and output ports, and the SWC can only communicate through these ports. AUTOSAR provides an abstract communication mechanism called the Virtual Functional Bus (VFB). The VFB allows for a strict separation not only between applications and infrastructure, but also inter-SWC communications are performed by

Figure 1: AUTOSAR architecture according to AUTOSAR version 4.2.

210 this mechanism in a standard way. It conceptually makes a SWC independent of the underlying hardware architecture of the ECU. All services demanded by the SWCs are provided by the AUTOSAR Run-Time Environment (RTE). Application SWCs are conceptually located on top of the RTE. The RTE is generated and customized for each ECU. The RTE itself uses the AUTOSAR OS and Basic Software (BSW). The VFB
215 functionality is also implemented by the RTE for each ECU. Figure 1 depicts this architecture. The BSW provides a standardized, highly-configurable set of services, such as communication over various physical interfaces, NVRAM1 access, and management of run-time errors. The BSW forms the biggest part of the standardized AUTOSAR environment [36].

220 AUTOSAR has currently reached to its 4.2.2 version, and multi-core support for the system is still optional in this version. Making a copy or moving some basic software components on to other processing cores is suggested in AUTOSAR 4.2.2, to increase the parallelization of the system. Although in edition 4.2.2, a set of general guidelines are mentioned to move (or copy) a sub set of basic software components on other cores
225 to avoid such a bottleneck, it is not sufficiently discussed in detail. In other words, it is not clearly specified which BSW component that can move on to other cores, and which

type of BSW components that can be copied on to other cores, and how many copies of each BSW component that can be generated to run on these cores. The configuration of the basic software on different cores is out of the scope of this paper, and it remains
 230 as one of the concerns of the RTE designers. However, design decisions affect the cost of allocation of runnables onto cores, and thus providing allocation solutions where the runnables that require a specific BSW are preferably allocated to the core on which the BSW is located.

3.2. Problem Description

235 Let us suppose a set of SWCs, each of which comprises a set of runnables (at least one runnable). Here the problem to be solved can be considered as a set of loosely-coupled runnables that are subject to scheduling and can be concurrently executed on different cores of a multi-core ECU. Let $R = \{R_i : i = 1, 2, \dots, m\}$ be the set of $m \geq 2$ runnables to be allocated among a set of $N \geq 2$ identical processing
 240 cores $\rho = \{\rho_j : j = 1, 2, \dots, N\}$ of a homogeneous multi-core ECU. The runnable R_i has a Worst Case Execution Time (WCET) denoted by e_i . Runnables have inter-communication relationships that are assumed to be based on non-blocking read/write semantics [37]. To exchange data among the runnables located on the same task, Inter-Runnable Variables (also called local labels) are used that are read and written by the
 245 runnables. Runnables located on different tasks have to use RTE mechanisms, e.g., the Sender-Receiver mechanism, to transfer data. Indeed, reading and writing inter-task labels are managed by the RTE. Let us also suppose that runnables located on different tasks have read-execute-write semantic, a common semantic in AUTOSAR that is also called *implicit access* [26], where a local copy of the inter-task label for data access is
 250 created and the modified data is written back at the termination of the task execution.

In our model, three types of communication are taken into account where the first and second type indicate the inter-runnable communications while the third type shows the interaction between the runnables and BSWs. The first type covers data dependency between the runnables, where they have to start to run with the fresh data generated by
 255 the predecessors to fulfill the dependency. In other words, there is a precedence among their execution order. The second communication type is when a pair of runnables

can transfer data in between each other while the freshness of data does not matter or at least as long as all runnables are completed within their periods the *maximum age of data* [38] is acceptable, in other words, there is no precedence among their execution order. The third type represents communication cost originating from runnables communicating with BSW modules.

The first communication type is modeled by a set of transactions $\{\Gamma_i : i = 1, 2, \dots, M\}$, where each of which represents an end-to-end function implemented by a sequence of runnables. Indeed, each transaction is a directed acyclic graph where each node is a runnable and each link represents data dependency between the corresponding two nodes. Note that the dependency between the runnables in a transaction does not imply triggering, in the sense that a successor can start with obsolete data generated by its predecessor. However, to fulfill the mission of a transaction, fresh data generated by the latest instance of the predecessors should be provided. Figure 2 shows a sample of a transaction. Without loss of generality we can assume that all runnables are covered by at least one transaction; if a runnable is not included in any transaction, then we assume a new dummy transaction covering only this runnable.

Figure 2: A sample of a transaction.

The transaction Γ_i has a relative end-to-end deadline denoted by D_i before which all runnables of the transaction must finish their execution. The transaction deadline corresponds to either:

- The deadline of the mission associated to that transaction. For example, the mission could be the braking system in a car where the whole transaction must complete before a specific end-to-end deadline.
- Only a portion of a mission is covered by this transaction running on a single multi-core ECU. In other words, the other parts of the mission are executed by other ECUs in the system. In this case, let us assume that the system designers have defined a partial deadline for the transaction within this ECU (deadline

decomposition [39]), meaning that if this ECU completes the transaction before the partial deadline, and provides the output data for transmission to other ECUs on time, then the whole mission is able to meet its deadline.

285

There are three approaches to handle the scheduling of such transactions: time triggering, event triggering and mixed time/event triggering. In this paper, we adopt the time triggering approach, in the sense that a transaction arrives periodically or sporadically, but with a known minimum inter-arrival time denoted by P_i . In the time triggering approach, determining the optimal period of transactions strongly affects the system performance [37], because finding a maximum period in which a transaction meets its deadline reduces the processing load. Nevertheless, specifying the optimal periods is not included in the scope of this paper. Instead, a conservative approach is to consider the period of a transaction to be equal to its given relative deadline. It is worth noting that after finding an optimal mapping of runnables to tasks and the allocation of the tasks to cores by our proposed solution, the proposed method by [37] can be applied to find the optimal periods to improve the CPU utilization further than what we provide here. We also assume that all runnables in a given transaction share the same period, which is equal to the transaction period; that is the case in several automotive applications [26]. Additionally, in the following it is discussed what the period would be of a runnable that is shared between multiple transactions when there are dependent transactions.

290

300

The second communication type is modeled by a directed graph, where there could be self-loops and cycles. We name this graph Runnable Interaction Graph (RIG). Each node of the RIG represents a runnable, and the arcs between the runnables show transferring of data from the sender to a receiver. When a node of the RIG has a self-loop this means that each instance of the runnable sends data for the next instance of the same runnable. Furthermore, there is a label on each arc indicating the amount of data that is sent from the sender to the receiver per *hyper-period*. The hyper-period is the Least Common Multiple (LCM) of the periods of all the transactions, denoted by H . When a runnable has a short period (recall that we assumed the period of a runnable to be equal to the period of the associated transaction), it generates data more frequently. Hence, to

310

compare the amount of data sent across various runnables irrespective of their periods, we consider data transfer rate per hyper-period. Figure 3 illustrates a RIG instance. For more information regarding RIG design, a real-world example of RIG is presented in Figure 2 in [40] where the events triggering the runnables and labels (shared variables used for inter-runnable communications) are also illustrated.

Figure 3: A sample of a RIG in an AUTOSAR system.

In the third communication type the interaction between runnables and BSWs are taken into consideration. In the AUTOSAR standard, the BSWs are introduced to provide services for the application runnables running on top of them. Each runnable can invoke BSW(s) to complete its execution. Although the impact of the system calls has usually been reflected in the WCET of the runnable, allocation of runnables regardless of their interaction with BSW(s) may not provide us with an optimal solution in terms of resource efficiency. In other words, if a runnable allocated to the core i has a lot of interaction with a BSW located on the core $j \neq i$, it imposes a considerable inter-core communication overhead aggravating the efficiency of the system. The interaction between the runnables and BSW components can be represented in the RIG while data transfer could be in both directions¹. Figure 4 shows a RIG including the interactions with BSW components where BSW_i denotes the i th BSW component.

3.2.1. Dependent Transactions

Another challenge that should be covered by our model is the dependency between transactions, meaning that multiple transactions may share the same runnable(s). In such cases, the problem can be categorized into two types. In the first type, there is no

¹The simple directed graph is not sufficient as we can have multiple arrows between a pair of nodes.

Figure 4: A sample of a RIG including the interaction with basic software components.

internal state within the shared runnable(s). In other words, multiple transactions are
 335 allowed to execute a shared runnable at the same time. An example of the first type is:
 the speed measurement function (runnable) which can be invoked by multiple transac-
 tions at the same time. In the second type the shared runnables contain internal states,
 for example, as long as the execution of the speed control function is not completed
 by one of the transactions, this function should not be executed by other transactions.
 340 Dealing with the first type is much easier than the second type. It is worth noting that
 most automotive applications contain both types of dependencies.

For the first type we can simply make a copy from the shared runnable(s), which
 in this case results in that each transaction will have its own copy, therefore, we have
 made independent transactions. Although this approach might generate extra overhead
 345 because of copying the shared runnable(s), it significantly decreases the problem com-
 plexity (it will be further discussed by an example in Section 4.4.2). Furthermore, each
 copy of the shared runnable has its own period, equal to the period of the transaction to
 which this copy belongs. Figure 5(a) shows an illustrative example, in which we make
 a copy from R_2 to provide two independent transactions represented in Figure 5(b).

350 In the second type we also make multiple copies of the shared runnables, thus

Figure 5: Two transactions with a common runnable.

each transaction will have its own copy. However, due to the requirement of having to preserve the internal states within the shared runnable(s), multiple copies of a shared runnable are not allowed to run at the same time. In other words, a kind of mutual exclusion is required. Hence, the shared runnables can be considered as a sort of critical section of a transaction. In other words, we expect if the transaction Γ_i and Γ_j have a shared runnable R_k , then when Γ_i is running R_k , Γ_j cannot run R_k until the execution of R_k within Γ_i is completed. When both the transaction Γ_i and Γ_j are located on the same task, we do not need to be concerned, but whenever they are located on different tasks, we need to deal with this issue. We can use the well-known task synchronization mechanisms while here, the shared resources are the runnables (a part of the tasks themselves). Therefore, the length of a critical section is equal to the execution time of the shared runnable associated to the critical section. With this definition of the critical section we implicitly assume that the whole part of a shared runnable must be mutually exclusive; opposite of the alternative assumption that only a part of a runnable needs to be mutually exclusive.

Let us have a look at the AUTOSAR task synchronization mechanisms. AUTOSAR recommends to use the Priority Ceiling Protocol (PCP) [41] for intra-core task synchronization. PCP ensures that the priority inversion is no more than the length of a single resource-holding by a lower priority task.

AUTOSAR provides a *SpinlockType* mechanism [3] for inter-core task synchro-

nization. It is a busy-waiting mechanism that polls a lock variable until it becomes available. This mechanism properly works for multi-core processors, where shared memory between the cores can be used to implement the lock variables. Basically, the SpinlockType mechanism increases the execution time of a task due to the busy-
 375 waiting time to access a global shared resource. The busy-waiting time is also called spinlock time. Now it is time to calculate the spinlock time [42]. It should be noted that PCP is used for all requests to local resources. The spinlock time that every task located on core k has to wait to access the global resource j is calculated by

$$spin(Res_j^{global}, \rho_k) = \sum_{\forall \rho_l \in \{\rho - \rho_k\}} \max_{\forall \tau_i \text{ assigned to } \rho_l, \forall h} \chi_{ih}^j \quad (1)$$

where χ_{ih}^j denotes the duration of global critical section for the h th access to the re-
 380 source Res_j^{global} by τ_i . As a result, in our problem, the spinlock time of τ_i to run the global shared runnable R_j^{global} is derived by

$$spin(R_j^{global}, \rho_k) = \eta_j e_j \quad (2)$$

where η_j is the number of cores on which at least one dependent task sharing R_j^{global} is located, excluding the core hosting τ_i .

Accordingly, the total spinlock time of a task located on the processing core k is
 385 given by Eq. 3.

$$t_{spinlock}(i) = \sum_{\forall Res_j^{global} \text{ accessed by } \tau_i} spin(Res_j^{global}, \rho_k) \quad (3)$$

The blocking time of task τ_i can be calculated by adding the blocking time due to local and global resources.

$$B_i = B_i^{local} + B_i^{global} \quad (4)$$

where B_i^{local} according to the PCP mechanism is achieved by

$$B_i^{local} = \max_{\forall \tau_j \text{ on the same core, } \forall h} \{\chi_{jh}^k \mid p_i > p_j \wedge p_i \leq ceil(Res_k^{local})\} \quad (5)$$

and B_i^{global} can be computed by Eq. 6 [42] when the task τ_i is assigned to the processing core ρ_l .

$$B_i^{global} = \max_{\forall \tau_j \text{ located on } \rho_l, \forall h, \forall Res_k^{global}} \{\chi_{jh}^k + spin(Res_j^{global}, \rho_l) \mid \lambda_i > \lambda_j\} \quad (6)$$

where λ_i is the static preemption level of τ_i , meaning that the task τ_i is not allowed to preempt task τ_j unless $\lambda_i > \lambda_j$. The concept of static preemption level is introduced in the definition of the stack resource protocol [42]. In fact, the SpinlockType mechanism used by AUTOSAR is developed according to that. Whenever static fixed-priority scheduling (e.g., fixed-priority scheduling with Rate Monotonic priority assignment) is used, we can assume that the preemption level of a task is equal to the priority of the task.

Global resources potentially result in a longer blocking duration (also, a longer spinlock time), thus tasks sharing the same resources are preferably assigned to the same processing core as far as possible [43]. However, it is not always possible (or at least efficient) to place dependent transactions on the same core. In our problem where there exist a lot of dependencies between transactions, the restriction of always placing dependent transactions on the same core may lead us to find no schedulable solution.

3.3. Communication Time Analysis

The communication cost among runnables and BSWs is one of the factors imposing a considerable overhead on the resource efficiency in a multi-core platform. To investigate the effect of this overhead on the CPU utilization, we need to formulate and analyze the communication cost. While we intend to present a general solution not limited to a specific hardware architecture, we need to know at least an abstract level of the hardware architecture of the target ECUs. Taking a closer look at the target multi-core architecture, we assume a multi-core processor with a common three-level cache architecture. The shared-cache architecture has become increasingly popular in a wide variety of embedded real-time applications [44]. In such architectures each core has its own private $L1$ cache while a second-level ($L2$) cache is shared across each pair of cores, and finally a third-level cache ($L3$) is shared among all cores. It is difficult to characterize the latency values with precise numbers, but in general the $L2$ cache

latency is almost two to three times larger than the L1 cache latency, the L3 cache latency is roughly ten times larger than the L1 cache latency, and the RAM latency is two orders of magnitude larger than the latency of the L1 cache [45]. Figure 6 represents a sample of such an architecture with 4 processing cores.

Figure 6: A three-level shared-cache quad core architecture.

When a pair of runnables are located in two different tasks, the Sender-Receiver mechanism is provided by the RTE to enable data transfer in between the runnables. The same mechanism is also used for interaction between a runnable and the BSW. In the Sender-Receiver mechanism a shared buffer accessible by both sender and receiver tasks is employed. At each activation of the sender task, after completing the execution of the sender task, it writes on the shared buffer, and at each activation of the receiver task, before beginning its execution, it reads the shared buffer. The process works according to the producer and consumer semantic. We presume that the read-time and write-time are identical and equal to one access to the memory on which the shared buffer is located; either within the main memory or within different levels of the cache.

In this abstract model, four scenarios to communicate between the runnables are considered. The last three scenarios also cover the interaction between runnables and BSWs.

1. If the runnables R_i and R_j are allocated within the same task, then the runnables

share the same address space and communicate with each other by means of local variables through the local cache (L1). It should be noted that in the same task, since the time interval between writing and reading data is quite short, the chance of preemption and removal of data from the L1 cache during this period is negligible. Hence, suppose that the latency to access variables in this scenario, for x units of data, is $\alpha(x)$.

2. When runnables are allocated in different tasks on the same core, the RTE (and underlying OS) is responsible to perform the data transmission, resulting in more overhead compared to the first scenario. The amount of extra overhead strongly depends on the memory structure of the system. Additionally, in this case, a first-level cache miss is also more likely to happen, in comparison to the first scenario. The reason is that after finishing the execution of the writer task, the scheduler may select another task to run instead of the reader task. The longer the duration between producing and consuming a particular piece of data, the more likely it is that other data will occupy the L1 cache. In such cases, even intra-core communication will have to go through the shared memory (or L2 or L3 cache), thus reducing the communication time gain from allocating communicating tasks to the same core. Suppose that the average latency to access the shared buffer in this scenario, for x units of data, is $\beta(x)$. The second scenario could also imply the interaction between a runnable and a BSW component located on the same core.

3. When runnables are executed on separate cores sharing an $L2$ cache, the RTE can potentially utilize the second-level cache to conduct the communication. In this scenario, the average communication delay to access the shared buffer, for x units of data, is $\theta(x)$. This could also be the case when a runnable interacts with a BSW component located on a separate core sharing the $L2$ cache.

4. Finally, when runnables are located on different cores without a shared $L2$ cache, communication has to go through the shared memory (or $L3$ cache). As the shared memory has a significantly larger latency than the local cache, a considerable difference is expected between this scenario and the third scenario [46].

$\gamma(x)$ is chosen to represent the memory access time, for x units of data, in this case.

This model can easily be generalized to cope with other common types of shared-cache processors. For example, if, in a given processor architecture, the L2 cache is also private for each core and it is not shared among a pair of cores (e.g., Intel Core i7), then it is sufficient to set $\theta(x)$ equal to $\gamma(x)$. In this case, we expect a lower value for both $\alpha(x)$ and $\beta(x)$. Formally, Eq. 7 formulates the above mentioned communication time delays.

$$C_{R_{ij}} = \begin{cases} \alpha(cr_{ij}) & \text{if } I \\ \beta(cr_{ij}) & \text{else if } II \\ \theta(cr_{ij}) & \text{else if } III \\ \gamma(cr_{ij}) & \text{else} \end{cases} \quad (7)$$

465 where $C_{R_{ij}}$ denotes the delay of accessing the local variable or the shared buffer for cr_{ij} units of data in between either R_i and R_j or BSW_i and R_j , and I denotes a condition in which R_i and R_j belong to the same task (the first scenario) whereas, II denotes a condition in which either the corresponding tasks of R_i and R_j are located on the same core, or BSW_i and R_j are located on the same core (the second scenario), and III is
470 corresponding to the third scenario.

The two following points are important to be emphasized:

1. In the worst case (does not happen in most cases), the last three scenarios are performed with the same maximum latency, which is the case whenever a cache miss happens when accessing data. As a result, we only use the scenarios to investigate and boost the resource efficiency of the system, whilst to be used in the calculation of the worst-case execution time of tasks for a schedulability test, we provide Eq. 8.

$$C_{R_{ij}}^{worst} = \begin{cases} \alpha(cr_{ij}) & \text{if } I \\ \gamma(cr_{ij}) & \text{else} \end{cases} \quad (8)$$

2. If we want to include more details, then several other factors impact on the communication time analysis, such as: the size of the first, second and third-level

cache; the cache replacement mechanism; the hit rate of cache levels (L1, L2
475 and L3). Nonetheless, in this paper an abstract communication model is applied
to address the problem in a general manner without confining the model to a
limited range of shared-cache multi-core processors.

3.4. Formal Definition of the Problem

This subsection describes the problem in a more formal way. The problem boils
480 down to (i) allocate a set of m communicating runnables to a given set of N homogenous
processing cores connected through a shared memory architecture, and (ii) mapping of
the runnables into a set of tasks. Since the size of the task set is unknown, it is also one
of the parameters of the problem. Nevertheless, we know that the size of a task set is
an integer value in the range of $[1, m]$. The allocation of runnables to cores, mapping
485 of runnables to tasks, and specifying the size of the task set should be performed such
that the total processing load of the N processing cores is minimized, subject to (i)
dependency constraints, and (ii) end-to-end deadlines. Minimizing the total processing
load is fulfilled by considering the following parameters:

1. Inter-runnable communication costs: we translate the inter-runnable communi-
490 cation time into execution time of the tasks hosting the runnables (will be ex-
plained further in Section 3.5), thus, the reduction of inter-runnable communica-
tion times results in a lower utilization of the processors.
2. The communication costs between runnables and BSWs: since the communica-
tion time to exchange data between runnables and basic software components is
495 affecting the execution time of the tasks hosting the runnables, the reduction of
this type of communication cost leads to a lower utilization of the processors.
3. The blocking-times for the synchronization of dependent transactions: as is men-
tioned, a spinlock time is added to the execution time of tasks waiting for shared
resources, thereby decreasing the blocking-time and resulting in a lower utiliza-
500 tion of the processors.

3.5. Optimization Problem

In this subsection, we introduce an optimization problem. There are two variable vectors in the optimization problem. The first vector shows mapping of runnables to tasks and the second vector is used for allocation of tasks to cores. Suppose that the assignment SR comprises both of these vectors. The straightforward way to model an optimization problem is to define a total cost function reflecting the goal function along with constraint functions. In this case, minimizing the total cost function is equal to minimizing the goal function while there is no constraint violation. The total cost function for our problem can be computed by Eq. 9, which returns a real value for the assignment SR_z , and this value is used to evaluate the quality of the given assignment.

$$TC(SR_z) = U(SR_z) + \sigma \times P(SR_z) \quad (9)$$

where $U(SR_z)$ is the total CPU utilization of the given workload by the assignment SR_z , which can be calculated by Eq. 10, and $P(SR_z)$ is the penalty function being applied to measure satisfiability of a given assignment. It means that if the value of the penalty function is zero, then the assignment SR_z satisfies both the end-to-end timing constraints and dependency constraints. Otherwise, some of the deadlines are missed. σ is the penalty coefficient used to guide the search towards valid solutions. This coefficient tunes the weight of the penalty function with regards to both the range of the cost function and the importance of the constraint violation. For example, in a soft real-time system, where missing a small number of deadlines may be tolerable, the coefficient should be set to a lower value. However, since we focus on hard real-time systems, the penalty coefficient should be high enough to avoid the search algorithm to converge to an infeasible solution. In Section 5.2, the proper value of the penalty coefficient is discussed in more details.

$$U(SR_z) = \sum_{\forall \text{ task } k} \frac{E_k(SR_z)}{T_k} \quad (10)$$

where T_k denotes the period of the task τ_k , and $E_k(SR_z)$ indicates the WCET of the k th task for the assignment SR_z , meaning that in our model, the task execution time is dependent on both the assignment of runnables to tasks and the tasks to cores. Task

execution time can be calculated by

$$E_k(SR_z) = t_{comput}(k) + t_{commun}(k) + t_{spinlock}(k) \quad (11)$$

where $t_{spinlock}(k)$ is already calculated by Eq. 3 and $t_{comput}(k)$ implies the computation time of τ_k which is independent from the assignment of tasks to cores and it can be calculated by

$$t_{comput}(k) = \sum_{\forall R_l \in \tau_k} e_l \quad (12)$$

$t_{commun}(k)$ represents the communication time between τ_k and other tasks (including the inter-runnable communications within τ_k itself) according to the assignment SR_z , which is derived by

$$t_{commun}(k) = \frac{T_k}{H} \sum_{\forall R_i \in \tau_k} \sum_{\forall R_j \in R} C_{R_{ij}}(SR_z) + \frac{T_k}{H} \sum_{\forall BSW_i} \sum_{\forall R_j \in \tau_k} C_{R_{ij}}(SR_z) \quad (13)$$

515 Hence, we translate the communication times into task execution times in order to integrate both the inter-runnable communication times and the runnable interactions with BSW components in our model. The mechanism for this translation is reflected by Eq. 13. The first term says that when a pair of runnables are located in the same task, the communication time will be added to the execution time of that task, and when a
520 runnable interacts with another runnable belonging to another task (does not matter on the same core or on a different core), then the communication time will be added to the execution time of both tasks hosting these runnables, because both the sender and the receiver tasks are engaged in the data transmission process. Recall that we assume that the worst-case time suffered by the reader and the writer tasks to access the shared
525 buffer is identical. The second term reflects the interaction of runnables with BSW components. Furthermore, since $C_{R_{ij}}$ denotes the cost of communication between R_i and R_j per hyper-period, in order to calculate the communication time per one period of the task τ_k , we multiply $C_{R_{ij}}(SR_z)$ by T_k/H .

The penalty function should be defined according to the processor scheduler. As is mentioned AUTOSAR uses partitioned fixed-priority scheduling to schedule tasks, and each core is scheduled using a single-core fixed-priority policy. It should be mentioned that other scheduling policies, including dynamic priority scheduling such as

Earliest Deadline First (EDF), or global scheduling, while quite popular in the research community, are not supported by the standard. Moreover, utilization bounds available from the theory of real-time global scheduling policies are quite pessimistic. Because of requirements on resource efficiency, most automotive systems are designed based on a static priority-based scheduling [18]. We assume a priority-driven OS scheduler with task priorities assigned according to Rate Monotonic, i.e., tasks with shorter periods (shorter relative deadlines) have higher scheduling priorities [47]. For simplicity of the presentation, we also assume that all tasks have unique priorities. The penalty function is given by

$$P(SR_z) = \sum_{i=1}^N \sum_{\forall \tau_k, \text{ allocated to the } p_i} \max\{0, r_k - T_k\} \quad (14)$$

$$r_k = E_k^{worst}(SR_z) + \sum_{j \in hp(k)} \lceil \frac{r_k}{T_j} \rceil E_j^{worst}(SR_z) + B_k(SR_z)$$

where $hp(k)$ implies the set of tasks with higher priority than that of τ_k , B_k is already
530 computed by Eq. 4, and $E_k^{worst}(SR_z)$ is the worst-case execution time of the k th task for the assignment SR_z and it occurs whenever we encounter with a cache miss while accessing the shared data, calculated by Eq. 11 while instead using the worst-case communication time of the task. To compute the worst-case communication time of a task we only replace $C_{R_{ij}}$ with $C_{R_{ij}}^{worst}$ in Eq. 13.

535 4. Solution Framework

Our solution framework for running AUTOSAR runnables on a multi-core ECU suggests first to make an initial task set with respect to the given set of runnables, irrespective of the location of runnables. Then the allocation algorithm is executed to find the partitions of the tasks on the cores. Finally, a refinement algorithm attempts to
540 enhance the task set on each core to generate a more efficient task set. In fact, using the refinement step, the task set configuration is modified with respect to the allocation of the tasks to cores. The framework is illustrated in Figure 7.

In this paper three different methods are developed to implement the mentioned solution framework. These methods are subsequently evolved in the sense that each
545 method attempts to resolve potential disadvantages of the previous one.

The solution framework

Figure 7: The solution framework for execution of AUTOSAR runnables on a multi-core ECU.

4.1. Method 1: Simple Mapping-based Approach

In the first method, generating an initial task set is simply carried out by considering each transaction (either actual or dummy) as one task. Let us assume that the transactions are not so long such that they cannot be schedulable on one processor, which is the case in several automotive applications. Consequently, the task period is therefore equal to the transaction period. Since the whole transaction is mapped to a single task, if a task meets its deadline, then the corresponding transaction meets its end-to-end deadline as well. If a transaction would have been split into multiple tasks, the arrows between the runnables of the transaction would have been generating the execution order (precedence) requirements among the tasks, and since the cost of synchronization is considerable, it can have a negative impact on the CPU utilization. Moreover, the no-split assumption leads us to stay in line with the choice made by AUTOSAR; there should be no precedence between the runnables located on different cores [48]. The simple mapping method is therefore prohibited from splitting up the tasks.

After creating an initial task set, the allocation phase is executed to assign the generated task set among the cores. In fact, the allocation algorithm generates a set of partitions of tasks which then per partition will be allocated to a single core. An evolutionary algorithm called Systematic Memory Based Simulated Annealing (SMSA) [49] is applied as an allocation algorithm. The experimental results in [50] demonstrated that SMSA outperforms both Simulated Annealing (SA) and Genetic Algorithms (GA) for the task allocation problem in multi-processor systems. The pseudo code of the SMSA algorithm is provided in Algorithm 1. In SMSA, a set of items need to be clarified.

Algorithm 1 SMSA

```
1: Inputs: task set  $\tau$  generated based on the transaction set, and the RIG
2: Initialize the cooling parameters  $\Psi_s, \Psi_f, \mu$ 
3: Initialize a queue with size Q
4: Generate the initial solution  $SR_0$  randomly
5:  $TCV_0 = TC(SR_0)$  {Compute the total cost function for  $SR_0$  according to Eq. 9}
6:  $SR_b = SR_c = SR_0$  {assign the initial solution to both of the current solution and the best solution}
7:  $TCV_b = TCV_c = TCV_0$  {assign cost of initial solution to both cost of current and cost of best solution}
8: Add  $SR_0$  to the queue of recently visited solutions
9:  $\Psi_c = \Psi_s$  {assign the start temperature to the current temperature}
10: repeat
11:    $SR_n = \text{Select one of the neighbors based on the stochastic-systematic selection}$ 
12:   if  $SR_n$  is not visited recently then
13:      $TCV_n = TC(SR_n)$ 
14:      $\Delta = TCV_n - TCV_c$ 
15:     if  $\Delta \leq 0$  then
16:        $SR_c = SR_n$ 
17:        $TCV_c = TCV_n$ 
18:       Add  $SR_n$  to the queue
19:       if  $TCV_n \leq TCV_b$  then
20:          $SR_b = SR_n$ 
21:          $TCV_b = TCV_n$ 
22:       end if
23:     else
24:       Generate a uniform random value x in the range (0, 1)
25:       if  $x < e^{\frac{-\Delta}{\Psi_c}}$  then
26:          $SR_c = SR_n$ 
27:          $TCV_c = TCV_n$ 
28:         Add  $SR_n$  to the queue
29:       end if
30:     end if
31:   else
32:     Update position of  $SR_n$  in the queue
33:   end if
34:    $\Psi_c = \Psi_c \times \mu$ 
35: until  $\Psi_c \leq \Psi_f$ 
36: return the  $SR_b$ 
```

τ_1	τ_2	τ_3		τ_M
1	1	2	...	1

Figure 8: Representation of the assignment of tasks to cores.

- *Problem space:* The set of all possible allocations for a given set of tasks and processing cores is called the problem space.
- *Solution representation:* Each point in the problem space is corresponding to an assignment of tasks to the cores that potentially could be a solution for the problem. The solution representation strongly affects the algorithm performance. We represent each allocation solution with a vector of N_{task} elements, and each element is an integer value between one and N . Since the number of tasks in the initial task set is equal to the number of transactions, then $N_{task} = M$. The vector is called Allocation Representation (AR). Figure 8 shows an illustrative example of an allocation solution. The third element of this example is two, which means that the third task (corresponding to the third transaction) is assigned to the second core. Furthermore, this representation causes satisfaction of the *no redundancy constraint*, meaning that each task should be assigned to no more than one core.
- *Initial solution in SMSA:* The initial solution is generated randomly.
- *Neighborhood structure used by the SMSA function:* SMSA constitutes a subset of the problem space that is reachable by moving any single task to any other processing core as the neighbors of the current solution. Therefore, each solution has $M(N - 1)$ different neighbors, because each task can run on one of the other $N - 1$ cores.
- *Selecting neighbor in SMSA:* SMSA in each step, instead of looking at all neighbors (i.e., $M(N - 1)$ neighbors), selects one task randomly and then it examines all neighbors of the current solution in which the selected task is assigned to another core. Hence, it visits $N - 1$ neighbors, and then the best solution of this subset is designated regardless of whether it is better than the current solution or not. We call this process *stochastic-systematic selection*, because we use a

595

combination of systematic and stochastic process to select the neighbor.

- *Initial temperature of SMSA*: The starting temperature, denoted by ψ_s should not be too high to perform a random search for a period of time, but sufficiently high to be able to explore a broad range of neighborhood. The initial and final temperature can be determined according to the probability of acceptance of a negative neighbor at the beginning and end of the algorithm respectively. It is recommended to set the starting temperature such that at the beginning of the algorithm the probability of acceptance of a worst neighbor is equal to 30% - 40% [51].
- *Cooling schedule in SMSA*: There are two common types of cooling schedules, namely, monotonic and non-monotonic. The cooling schedule of the SMSA in this paper is assumed monotonic in the sense that the temperature of the current iteration is equal to $\mu \times$ the temperature in the previous iteration, where μ is a real value between zero and one.
- *Stopping condition of SMSA*: The algorithm terminates when the current temperature ψ_c becomes less than the final temperature ψ_f .

610

After finishing the allocation phase, a REfinement Function (REF) is applied, trying to merge multiple tasks into merged tasks on each core, according to a predefined metric. Similar to the mechanism applied for initialization of tasks, the refinement function never split up a transaction, avoiding more complexity and overhead in the system. Concisely, REF merges all the mergeable tasks described as follows:

615

- Two tasks are *mergeable* if they are located on the same core, they have the same period, and they communicate with each other or they are dependent.
- Two tasks communicate with each other if and only if at least one of the runnables of the first task communicates with one (or more) of the runnables of the second task.
- Two tasks are dependent if and only if there is at least one shared runnable between the two tasks.

620

The basic notion of REF is that when we merge communicating tasks having equal period, the overall communication time can be decreased, since the communication
625 between these tasks is performed at a lower latency α rather than β . Additionally, merging multiple dependent tasks having an equal period on the same task, the dependent transactions are serialized. The serialization of dependent tasks leads to an elimination of the local blocking time, resulting in the following advantages:

1. Reducing the total blocking time, thus the schedulability of the core hosting
630 the merged tasks goes higher, thereby potentially allowing for more tasks to be allocated to the same core.
2. Decreasing the *response jitter* to complete the tasks; the response jitter is defined as the difference between the worst-case response time of a task and the best-case response time of a task, which is desirable to be minimized to improve the
635 stability of system in several automotive applications [52, 53].

When multiple tasks have different periods, to ensure the fulfillment of timing constraints and also to preserve the periodic nature of the tasks, the period of the new task merging those tasks should be set to the Greatest Common Divisor (GCD) of the periods of those tasks [12]. As a result of this period reduction, the utilization of the new
640 task might be significantly higher than the sum of the utilization of the original tasks. Accordingly, in order to avoid the increase of CPU utilization, REF does not merge the tasks with different periods. The pseudo code of REF is presented in Algorithm 2.

Algorithm 2 REF

```

1: Inputs: a given task set  $\tau$  and a vector indicating allocation of the tasks to cores
2: for each task  $i$  do
3:   for each task  $j$  do
4:     if  $\tau_i$  and  $\tau_j$  are mergeable then
5:       Merge them into the task  $\tau_i$ 
6:       Update tasks' indices
7:       Decrement the number of tasks
8:     end if
9:   end for
10: end for
11: return the updated task set

```

Note that our approach for both the initialization of the task set and the refinement of the task set follows the popular approach in the automotive domain mentioned in Section 2, where a set of runnables with the same period are mapped to the same task. Nevertheless, at the initialization step we do not merge multiple transactions with the same period, giving the allocation algorithm more flexibility to play with the location of tasks to achieve a better allocation solution. In fact, we postpone the investigation of the possibility of merging different transactions with the same period until the last step.

In the first solution, the allocation and the mapping of runnables to tasks are accomplished separately in a subsequent manner. This approach is called non-feedback-based. As an alternative, the mapping and allocation phases can be interleaved, called feedback-based approach. Opposite of the first method, the two next methods are designed according to the feedback-based approach.

4.2. Method 2: The Feedback-based Refinement Approach (SMSAFR)

The initial task set in the second method is generated just similar to the first solution, meaning that each transaction is mapped to a single task. In the first method, since the allocation phase is not aware of the task refinement procedure (not using feedback of the refinement function), it may select a non-optimal solution. For example, let us suppose that X and Y are two candidate solutions for the allocation problem. Before doing the refinement, X outperforms Y , but after refinement, due to a stronger merging applicable on Y , it surpasses. To manage this issue, a feedback-based approach is taken into account in which REF is frequently invoked from the inside of SMSA to refine the task set before evaluation of each individual (candidate solution). In this way, SMSA reflects the effect of task merging in guiding the search towards an optimal solution. This algorithm is called SMSA with Feedback Refinement (SMSAFR). To implement this method, it is sufficient to invoke REF before invocation of the total cost function to refine the task set, and then the total cost value is computed. Hence the pseudo code of the second method is just similar to Algorithm 1 whereas the REF function is invoked (i) before line 5, (ii) before line 13, and (iii) in line 11 to select a neighbor. Recall that according to stochastic-systematic selection, $(N - 1)$ neighbors should be examined to

select a neighbor, i.e., the total cost function should be calculated for $(N - 1)$ neighbors, and for each of these neighbors, the REF function should be invoked first, then
675 the total cost function can be calculated.

Although, SMSAFR is more efficient in comparison to the simple mapping-based approach in terms of both the overall communication time reduction and task synchronization cost reduction, it takes a longer execution time to search the problem space. The longer search time is inherent in the frequency of the REF invocation. In other
680 words, REF is more frequently invoked by SMSAFR compared to the first method invoking only one instance of REF after finishing the allocation phase. The third method attempts to enhance the quality of the achieved solutions even further, and at the same time accelerate the execution time of the search process in comparison to the second method.

685 4.3. Method 3: The Utilization-based Refinement Approach (PUBRF)

The third method called Parallel version of SMSA with Utilization-Based ReFine-ment (PUBRF) is similar to SMSAFR with two principal differences. The first difference is that SMSAFR uses REF as the refinement function which only merges multiple communicating (and dependent) tasks located on the same core with the same period
690 into one task, whereas the third method uses an extended version of the refinement function, called Utilization-Based Refinement (UBR). UBR merges multiple tasks on the same core iff merging them into one task results in the CPU utilization reduction on that core. The basic idea of UBR comes from this fact that multiple tasks with different periods may have a lot of communication to each other, as far as the period
695 reduction can be compensated by the communication cost reduction. In other words, UBR considers a trade-off between the amount of the reduction of execution time of task due to a lower communication time on one side, and the increase of load inherent in the merging of tasks with different periods on the other side. In practice, when the runnable mapping is done manually, this situation happens quite often, particularly
700 when runnables interact heavily with each other [54]. Therefore, not only UBR merges the communicating and dependent tasks with the same period similar to REF, but it also allows to merge tasks with different periods on the same core whenever they have

a lot of communication together.

To form this trade-off, the CPU utilization test is applied, meaning that the algorithm merges multiple tasks into one task whenever the CPU utilization after merging gets lower.² The utilization of the task containing τ_i and τ_j , denoted by $u_{\tau_{ij}}$ is computed by Eq. 15.

$$u_{\tau_{ij}}(SR_z) = \frac{E_i(SR_z) + E_j(SR_z) - (\beta - \alpha) \frac{T_{ij}}{H} \sum_{\forall R_k \in \tau_i} \sum_{\forall R_l \in \tau_j} C_{R_{kl}}(SR_z)}{T_{ij}} \quad (15)$$

$$T_{ij} = GCD(T_i, T_j)$$

where GCD indicates the greatest common divisor. Indeed, the equation expresses that merging communicating tasks leads to diminish the communication time between them since it is performed with a lower latency α instead of β , meanwhile a period reduction may occur. This formulation can be extended to merge more than two tasks iteratively. In Algorithm 3 the pseudo-code of the new refinement function is provided.

The algorithm implies that for a given allocation of tasks to cores, UBR starts from an initial task set for each core, where each task contains one transaction, and gradually trying to group more and more tasks together to minimize the cardinality of the task set. UBR acts in a greedy manner, first selecting a pair of tasks with the maximum gain in terms of CPU utilization reduction. After merging the designated pair of tasks, UBR recomputes the gains of merging the new task with all other tasks. Then it decides which pair of tasks should be merged in the next step. It will continue until there exist no pair of tasks where merging of them would result in an utilization reduction.

The second difference is that PUBRF utilizes a more effective evolutionary algorithm which is able to find high quality solutions in a shorter execution time. The idea is that we can run multiple copies of the SMSA algorithm at the same time on different cores of a multi-core processor. In other words, we exploit the potential of multi-core processors to accomplish a highly efficient search similar to the approach adopted by the automotive industry.

²Multiple dependent tasks will be merged only when they have the same period.

Algorithm 3 UBR

```
1: Inputs: a given task set  $\tau$  and a vector indicating allocation of the tasks to cores
2: Create a strictly upper triangular matrix with the size of  $M$ , named profitMatrix
3: for each task  $i$  do
4:   for each task  $j > i$  do
5:     if  $\tau_i$  and  $\tau_j$  are located on the same core then
6:       profitMatrix[ $i, j$ ] =  $\min\{0, (u_{\tau_i} + u_{\tau_j}) - u_{\tau_{ij}}\}$ 
7:     else
8:       profitMatrix[ $i, j$ ] = 0
9:     end if
10:   end for
11: end for
12: repeat
13:   Find the pair of tasks with the maximum profit value (let us suppose they are  $k$  and  $l$  while  $k < l$ )
14:   Add all the runnables of the  $l$ th task to the  $k$ th task
15:   Set the deleted flag of the  $l$ th task (logically remove)
16:    $T_k = \min\{T_l, T_k\}$ 
17:   Recompute the  $U_{\tau_k}$ 
18:   Set all elements of the  $l$ th row and the  $l$ th column of the profitMatrix equal to zero
19:   Recompute all entries of the  $k$ th row and the  $k$ th column
20: until (The maximum element of the profitMatrix becomes zero)
21: Delete the tasks which have been logically removed
22: return the updated task set
```

The point is that when we run multiple copies of SMSA, we expect a lower number of iterations to find the optimal solution in comparison to when we only use one instance of SMSA, in that sense we gain a shorter execution time. However, experiments have been conducted demonstrating that running multiple copies of SMSA is not as efficient as running one instance with a larger number of iterations. Accordingly, an additional part should be interleaved to the search algorithm which collects the results of all copies of SMSA, then analyzes them and guides the search towards the regions of the search space with a higher probability to find the global optimum. The additional part can be fulfilled by using the idea introduced by the Max-Min Ant algorithm [55]. Therefore, we introduce a max-min ant system being leveraged by an asynchronous parallel version of the SMSA algorithm to effectively search our problem space. A flowchart scheme of this method is given by Figure 9.

As is seen in the flowchart, most parts of this algorithm are developed in a multi-threading manner, where the threads can be executed concurrently on different cores.

Figure 9: Flowchart representation for the PUBRF algorithm.

In fact, only a few relatively lightweight instructions are located in the non-parallel part, such as updating the pheromone trails and selecting the next generation of ants, while the CPU-intensive functions like the total cost and the UBR are handled in the parallel part. Further details regarding the configuration of PUBRF are described into two parts; SMSA settings and configuration of the max-min ant algorithm.

SMSA settings. To enable SMSA to play the rule of an ant in a max-min ant algorithm the following settings are applied.

1. *Initial solution:* The initial solution is turned to an input parameter where at

745 the first iteration of the main algorithm it is generated randomly and passed to
SMSA, while from the second iteration of the main algorithm and later on, the
initial solution of SMSA is generated by the max min ant algorithm.

2. *Initial temperature of SMSA*: Only the first instances of SMSA invoked at the first
iteration of the main algorithm start with the initial temperature Ψ_s , however, in
750 order to make the main algorithm to converge, the initial temperature of SMSA
instances invoked in the next iterations of the main algorithm is set to a lower
value which is equal to half of the initial temperature in the previous iteration.
In other words, when SMSA starts its execution in the first iteration of the main
algorithm with the initial temperature Ψ_s , then SMSA invoked in the second
755 iteration of the main algorithm will start with the initial temperature $\frac{\Psi_s}{2}$.

3. *SMSA+UBR*: In the forth level of the flowchart, SMSA+UBR implies the SMSA
function in which before computing the total cost value, the UBR function is
invoked to create a new task set and then the total cost value is calculated for the
new task set. Hence, in the pseudo core of SMSA (Algorithm 1) before line 5, in
760 line 11, and before line 13, the UBR function should be invoked.

Configuration of the max-min ant algorithm. Now we discuss the elements of
the max-min ant algorithm.

- *Updating the pheromone trails*: Real ants use trails of a chemical substance to
communicate with other ants to inform them about the directions in which food
can be found. Actually, the pheromone trails are a kind of distributed numeric
information which is modified by the ants to reflect their experience achieved
during solving a particular problem. In order to apply the ant system to task
allocation problems, a pheromone matrix Ph with the size of $M \times N$ is required
where the element Ph_{ij} is corresponding to the assignment of the i th task to the
 j th core. Updating the pheromone trails is done first by lowering the pheromone
trails by a constant factor (called evaporation) and then by allowing the best ant
to deposit pheromone on the direction that it has visited (called reinforcement).

In particular, the update can be performed by

$$Ph_{ij}^{l+1} = \varpi \times Ph_{ij}^l + \frac{x_{ij}^{best}}{TC(AR^{best})} \quad (16)$$

where ϖ denotes the evaporation factor, Ph_{ij}^{l+1} indicates the pheromone value for the next iteration, x_{ij}^{best} is a binary variable which is equal to one if in the best solution the i th task is assigned to the j th core, otherwise it is set to zero, and $TC(AR^{best})$ denotes the total cost value for the best solution.

- *Generation of ants:* The ants are created based on a probabilistic decision relevant to the pheromone values. In other words, if the pheromone value for the element Ph_{ij} is a large value then the i th task will probably be assigned to the j th core in the next ants. This concept is reflected by the following formula

$$Prob^k(x_{ij}) = \frac{Ph_{ij}}{\sum_{l=1}^{N_A} Ph_{il}} \quad (17)$$

where $Prob^k(x_{ij})$ denotes the probability of assigning the i th task to the j th core in the k th ant.

- *Stopping condition of the main algorithm:* The algorithm terminates after a specific number of iterations, denoted by υ .

4.4. Illustrative Examples

This subsection provides two examples where the first example illustrates the behavior of the proposed methods, while the second example describes the reason to make a copy from the shared runnables to provide distinct transactions.

4.4.1. Example 1

The first example (Figure 10) includes 12 runnables and 3 transactions. R_2 is shared between Γ_1 and Γ_2 . A copy of R_2 , called R_2' , is made. The initial task set is generated according to "one transaction (either actual or dummy) to a single task", hence we have 6 tasks. τ_1 and τ_2 are dependent, in the sense that they share a critical section that needs to be mutually exclusive, while other tasks are independent.

785 Let us start with the investigation of the behavior of Method 1 (i.e., SMSA). After creating an initial task set, SMSA allocates the given set of tasks among core 0 and core 1. Since it is a small example, finding an optimal allocation of tasks to cores is straight forward. Suppose that $\tau_1, \tau_2, \tau_3, \tau_4, \tau_5$ do not fit into a single core, meaning that they need to be executed on more than one core. Since τ_1 and τ_2 are dependent, in
790 order to avoid the remote blocking time, they are allocated to the same core. As cr_{24} is the highest communication rate, τ_3 is also assigned to core 0. Since cr_{57} is the second highest communication rate, τ_4 and τ_5 are preferably assigned to the same core, thus, they are allocated to core 1. Finally, τ_6 that communicates with τ_5 is assigned to core 1. After completing the allocation phase, the refinement function is invoked. At core
795 0, only τ_1 and τ_2 are mergeable, thus they are merged into a single task, and there is no mergeable task at core 1. According to our experiments, SMSA achieves the same result.

Now we examine the behavior of Method 2 (i.e., SMSAFR). SMSA, in the process of allocation of tasks to cores, neglects the possibility of merging tasks. Therefore,
800 due to the heavy data exchange between τ_1 and τ_3 , τ_3 is assigned to core 0. However, if τ_3 is allocated to core 1, then τ_3 and τ_4 can be merged into a single task. Let us suppose that merging τ_3 and τ_4 not only compensates the inter-core communication cost for exchanging $cr_{24} - cr_{45}$ byte data, but it also significantly reduces the total CPU load. Since SMSAFR considers this trade-off during the allocation phase, in contrast
805 to SMSA, τ_3 is assigned to core 1, then τ_3 and τ_4 , as they share the same period, are merged into a single task. Here again, τ_1 and τ_2 are also merged into a single task. Although τ_4 and τ_5 have a heavy data exchange, they are not merged because of having different periods.

Method 3 (i.e., PUBRF) does not only, similar to SMSAFR, assign τ_3 to core 1, but
810 it also merges τ_3, τ_4 and τ_5 into a single task. Indeed, it considers a trade-off between (i) the execution of τ_5 more frequently than what is needed (let assume $P_2 < P_3$), and (ii) the reduction of communication cost as the result of placing τ_4 and τ_5 onto the same task. In this example, the latter factor dominates the former, and that is the reason behind the decision of PUBRF for merging τ_3, τ_4 and τ_5 . Note that since τ_6 has
815 a small data exchange rate to τ_5 and they have different periods, they are not merged

together. Another interesting point regarding PUBRF is that if merging τ_3 with $\tau_{1,2}$ would dominate merging τ_3 with $\tau_{4,5}$ in terms of the total CPU utilization, PUBRF would have assigned τ_3 to core 0. It demonstrates the noticeable strength of PUBRF for combining mapping of runnables to tasks and allocation of tasks to cores.

820 4.4.2. Example 2

Figure 11 depicts the second example where the transactions have different periods, which in this case we must execute the runnable with the shortest period (i.e., $P = 2$). If we do not make a copy from the shared runnable, we would face a challenge to assign those transactions to tasks. In this case either

- 825 • R2 should be assigned to a separate task for which, in this case, transaction 1 is split to 3 different tasks (task 1 containing R1, task 2 containing R2 and task 3 containing R3). There is a precedence constraint among these three tasks in terms of the execution order. In other words, we need to assure that the scheduler executes task 1 before task 2 and task 2 before task 3; or
- 830 • All the runnables of transaction 1 should also be executed with period = 2, i.e., two times more frequent than what is really needed.

In the first case, a higher complexity for allocation of tasks to cores comes to play, whilst in the second case, a considerable overhead in terms of the CPU utilization is imposed. Because of escaping the overhead and the complexity, we suggest to make a
835 copy from the shared runnables.

5. Performance Evaluation

In this section, we first describe three alternative frameworks to deal with the problem; one being the framework proposed in this paper and the other two are representing relevant alternative solutions to the problem. Then the benchmark application specifications and target hardware architecture are presented. Finally the comparison results
840 are discussed.

Figure 10: An illustrative example to explain the proposed methods.

Figure 11: An illustrative example for dependent transactions with a different period.

5.1. Alternative Frameworks

To demonstrate the quality of our solution framework, given best practices, we develop three alternative frameworks containing solutions to both mapping of runnables to tasks as well as task allocation.

The first alternative framework is designed according to the traditional well known mechanisms for both mapping of runnables to tasks and allocation of tasks to cores. It is called Common Framework (CF). CF maps runnables to tasks according to the common mapping approach (i.e., all runnables with the same period are mapped to a single task) as the initialization phase. Then it utilizes one of the bin packing algorithms - Rate Monotonic Best Fit (RMBF) [56] - as the allocation phase. Among the bin packing algorithms we choose RMBF since it attempts to consolidate the tasks onto a minimum number of cores, potentially decreasing the cost of both inter-core communication and global task synchronization. Moreover, there is no refinement step in CF to shrink the size of task set.

RMBF starts with sorting tasks according to non-decreasing periods, and the tasks will be scheduled in that order. Note that as we merge transactions with the same period into a single task, each task has a unique period. Then RMBF finds the processor p_j with the maximum utilization, such that the task τ_i , together with all the tasks that have been assigned to the processor p_j can be feasibly scheduled according to the extended

rate monotonic schedulability test.

$$\sum_{l=1}^i \frac{E_l^w}{T_l} + \frac{B_i^w}{T_i} \leq i(2^{1/i} - 1) \quad (18)$$

Note that in our problem, before the completion of assigning tasks to cores, we know neither the execution time of tasks nor the blocking times, since they pertain to the allocation of other communicating and dependent tasks. To address this issue, we define E_i^w and B_i^w which are used when the assignment of tasks to cores is not completed yet. Indeed, when a task is supposed to be scheduled, we assume that all unassigned communicating and dependent tasks are assigned on other cores (worst case). In other words, all data communications between the unassigned tasks and this task are performed with the maximum latency γ and all the runnables shared between unassigned tasks and this task suffer a remote blocking. It is obvious that the actual blocking times and communication times between the current task and the assigned tasks are known and we do not need a conservative estimation for them. Accordingly, as far as RMBF progresses the pessimism decreases and actual communication and blocking times are used.

The second alternative framework developed to compare with the proposed framework is called Exhaustive Framework (EF). As its name indicates, this framework applies an exhaustive search in the problem space to find the exact solution. At the beginning, a task set is generated according to the common mapping approach, similar to CF presented above. Afterwards, an exhaustive search, based on Back-Tracking (BT) search, is invoked. Similar to CF, there is no refinement step here.

BT traverses a search tree where leaves correspond to potential solutions to the task assignment problem. It also leverages a fast bounding method that prunes unpromising branches that cannot lead to an optimal solution. To find out whether a vertex is promising or not, the response time of all assigned tasks should be computed, and if at least one of them is greater than the task deadline, then the solution is unpromising, and otherwise it is a promising solution. To compute the response time of the assigned tasks we need to calculate the tasks' execution times for each vertex of the search tree. This simple method not only consumes a long time to compute tasks' execution times for

each vertex, but it also requires an estimation for communication times and blocking
890 times of the unassigned tasks. A faster and smarter way could be as follows:

1. The EF framework assigns all runnables with the same period to one task, then
the number of tasks is equal to the number of different periods in the system,
and will not change during the execution of the search algorithm. Accordingly,
the problem space can be extremely diminished by considering a task interac-
895 tion graph instead of the runnable interaction graph. In the following we show
that in the benchmark with 10 different periods, a system with more than 1000
runnables can be reduced to only 10 tasks before starting the exhaustive search
algorithm. In this case the number of combinations would be N^{10} rather than
 N^{1000} (N is the number of cores).
- 900 2. Before starting the search algorithm, we compute a minimum execution time of
each task irrespective of the partitioning of tasks. To do so, we assume that firstly
all inter-task communications will be carried out with the cost β (communicating
tasks are assigned to the same core), secondly there is no blocking time. It leads
to a minimum utilization for each vertex and if the minimum utilization of a
905 core is greater than one, then the actual utilization is definitely equal or higher,
and thus the vertex is unpromising. It should be noted that only for the leaves,
the total cost function (Eq. 9) is invoked which of course works with the actual
utilization.

The third alternative solution framework works just like the third method of the
910 proposed solution framework with the difference that we replace the parallel version of
Max-Min Ant system intensified by SMSA with a genetic algorithm. This framework is
called Genetic Algorithm Framework (GAF), concentrating on the allocation algorithm
to demonstrate the performance of the proposed meta-heuristic algorithm.

5.2. Application and Hardware Specifications

915 To evaluate the performance of the proposed solution framework for automotive
applications, a real-world benchmark is required. To highlight the difference between

the efficiency of various frameworks and methods targeting this problem, an automotive benchmark including a large number of highly-connected runnables is suitable, which is the case for the Engine Management System (EMS) application. When we
920 have an automotive system with rarely inter-runnable communications, even ignoring the communication and synchronization costs may result in a negligible impact on the efficiency of the system. Accordingly, in this work we use one of the EMS benchmarks proposed by Bosch corporate research [26] to conduct the application specifications.

Three types of labels in terms of access type are used in automotive systems; read-
925 only, write-only and read-write labels. To exchange data in between the runnables read-write labels are used, and the interaction between runnables and basic software components can be performed through read-only, write-only and read-write labels. According to the benchmark 50% of the labels are read-only and write-only, while the remaining 50% are read-write. The size of the labels are stated in Table 1.

Table 1: Distribution of label sizes.

# of labels	size
35%	1 byte
49%	2 byte
13%	4 byte
0.8%	5-8 byte
1.3%	9-16 byte
0.5%	17-32 byte
0.2%	33-64 byte
0.2%	>64 byte
Average	2.4 byte

930 Table 2 shows the period and the range of the execution time of runnables along with the data communication specifications in the benchmark. The total number of runnables is 1000 (i.e., $m = 1000$). Note that, in the benchmark, 15% of the tasks are activated by asynchronous events where their inter-arrival times depend both on revolution per minute (rpm) and the number of cylinders of the engine in an engine

management system. In order to integrate them into our periodic task model, the minimum inter-arrival time between two consecutive instances of such tasks are calculated. To do so, we assume that the maximum number of cylinders are eight and the maximum rpm is 10,000 which yields the period equal to 1.5 msec, mentioned in the second row of Table 2. Additionally, the average size of labels (2.4 byte) was used to calculate the average number of used labels which is also equal to the number of sending data. For example, the fifth row of the table implies that the runnables with period = 10ms send data in total in the range of [1401,2850] bytes and the average number of sending data for all runnables with this period is 1180. If we divide it by the number of runnables with this period we can say how many other runnables that each runnable in average send data to.

Table 2: Application parameters and the corresponding value ranges.

Period (msec)	# of runnables	execution time(μ sec)	total size of sent data(byte)	average num- ber of sending data
1	3%	[0.34,30.11]	$40 \leq$	17
1.5	15%	[0.45,88.58]	[812,2160]	618
2	2%	[0.32,10.69]	$30 \leq$	11
5	2%	[0.36,83.38]	[240,320]	100
10	25%	[0.21,309.87]	[1401,2850]	1180
20	25%	[0.25,291.42]	[1360,2640]	833
50	3%	[0.29,92.98]	[91,360]	94
100	20%	[0.21,420.43]	[1792,3200]	1044
200	1%	[0.22,21.95]	[40,50]	21
1000	4%	[0.37,0.46]	[232,770]	208

As the number of shared runnables with internal states (runnables with mutually exclusive requirements) was not mentioned in the benchmark, we assume that the dependency ratio is $\kappa = 10\%$, meaning that in average $\kappa\%$ of runnables are subject to mutually exclusive requirements, i.e., sharing of runnables among multiple transactions.

950 In addition, in the following, playing with the dependency ratio value, we will show
the effect of the dependency ratio on the total utilization of the system. Let us presume
that in average a shared runnable belongs to 3 transactions. There are 60 transactions
in the benchmark where the number of runnables per transaction (excluding dummy
transactions), according to the benchmark, is in the range of [2,10] with an average
955 of 5. Therefore, in average 30% of the runnables belong to at least one transaction
(recall that for each of the remaining 70% of the runnables that are not included in any
transaction, we make a dummy transaction containing only that one runnable).

Let us call the mentioned benchmark as *BenchM* hereon. We also create two vari-
ants of *BenchM*, denoted by *BenchM*⁺ and *BenchM*⁺⁺ respectively. *BenchM*⁺ and
960 *BenchM*⁺⁺ both share the same specifications as *BenchM* with the only difference in
that the total size of communication data in *BenchM*⁺ is half of that in *BenchM*, and
the total size of communication data in *BenchM*⁺⁺ is two times greater than that in
BenchM. In other words, the average total size of communication in *BenchM*⁺⁺ is four
times greater than that in *BenchM*⁺.

965 As the ECU hardware specification we consider a quad-core processor in which the
first level cache is private for each core, the second level cache is shared among each
pair of cores and the third level cache is shared among all cores, similar to Figure 6.
In order to measure the inter-core and intra-core latency values we performed some
empirical experiments on a 32-bit version of the Ubuntu 12.04 LTS operating system
970 (kernel version 3.2.29) patched with the PREEMPT RT patch (version 3.2.29-rt44)
platform to transfer 1kB data in different communication scenarios on a 2GHz quad-
core with a three-level cache (intra-task, intra-core, inter core through L2 and inter-
core). To investigate the first scenario we created two functions in a single Posix thread.
The first function writes 1kB data in a shared array and then the second function is
975 invoked and read the data from the array. Both the writing and reading are done by
the sequential data access pattern. Other scenarios were implemented through a pair
of event triggered tasks in which when the first task write data on the shared memory,
the second task is triggered to read the data. Additionally, to reduce the probability of
unwanted interference we put the tasks in the highest priority level.

980 The algorithm parameters have a substantial impact on the performance of both

SMSA and the Max-Min Ant Optimization algorithm. We strive to check all possible values in a reasonable range for every parameter to select the best value for them. Algorithm parameters are listed in Table 3. The only point related to parameters which may need more clarification is that the cooling factor of SMSA is chosen 0.995 while in the
985 third solution it is set to 0.9 – it is the main reason for the speedup of the third solution in comparison to the first and the second solution. However, the two first solutions are unfortunately not working properly with a cooling factor lower than 0.995.

Table 3: Algorithm parameters and the corresponding value ranges.

Parameters	Description	Value
ψ_s	start temp.	$\sum_{\forall tasks} \frac{E_i^w + B_i^w}{T_i}$
ψ_f	final temp.	$\frac{1}{m} = 0.001$
μ	cooling factor of SMSA	0.995
μ'	cooling factor of PUBRF	0.9
Q	size of queue	m
ϖ	evaporation factor	0.9
N_A	the number of ants	4
ν	the number of iterations	10
σ	penalty coefficient	$10 \times \psi_s$
α	intra-task latency	4.0 η sec per cache line
β	inter-task latency	7.0 η sec per cache line
θ	inter-core latency (through L2)	14.5 η sec per cache line
γ	inter-core latency	16.2 η sec per cache line

The penalty coefficient should be determined such that all the feasible solutions have a lower total cost in comparison to infeasible solutions, meaning that all the fea-
990 sible solutions are always preferred to an infeasible solution. On the other hand, the penalty coefficient should not be extremely large since it hinders the search algorithm to search among infeasible solutions to find a way to reach the global optimum. Figure 12 shows an example of a problem space where the vertical axis shows the total cost value derived by Eq. 9 for three different values of the penalty coefficient $\sigma_1 < \sigma_2 < \sigma_3$, and

the horizontal axis shows multiple points (candidate solutions) in the problem space. In this example, SR_4 , SR_5 , and SR_6 are infeasible solutions for the problem, in the sense that the penalty function of these solutions is greater than zero, while for the other candidate solutions the penalty function is equal to zero. As is seen in the figure, SR_8 is the best feasible solution, however, when the penalty coefficient is set to a small value σ_1 , the search algorithm converges to SR_6 as the best solution, while it is an infeasible solution. When the value of the penalty coefficient is extremely high (i.e., it is set to σ_3), the probability of accepting infeasible solutions (SR_4, SR_5, SR_6) is dramatically reduced, and as long as SR_4 , SR_5 and SR_6 are not visited we could not reach the best feasible solution SR_8 .

Figure 12: An example of a problem space corresponding to three different values of the penalty coefficient.

Our experiments also verify this theoretical discussion. We observed that when $\sigma = 10 \times \psi_s$ we always converge to a feasible solution and when it is set to a lower value, in some experiments, we converge to an infeasible solution. We also observed that when $\sigma = 100 \times \psi_s$ the deviation from the best total cost found in multiple experiments goes up, meaning that the average total cost is increased, thereby, the quality of the solutions is decreased.

5.3. Comparison Results

To conduct the simulation process, all the methods and frameworks are implemented in C++ and executed on a PC with 3.2 GHz six-core Intel Core i7 and 8 GB of RAM memory. For each problem size, all the three proposed methods and the Genetic

1015 Algorithm Frameworks were run 20 times to reach 95% confidence interval, while the
CF and EF have deterministic nature and thus only require one run.

5.3.1. Comparison Between the Three Proposed Methods

As Table 4 shows, the average utilization of the four core processors for all bench-
marks are improved by SMSAFR rather than the simple mapping while PUBRF gen-
1020 erates the best solutions in terms of the processor utilization for all the benchmarks.
It is also observed that the difference between the efficiency of the methods gets more
considerable with increasing the communication size, meaning that in the $BenchM^{++}$
(having the biggest communication size) the difference between average utilization
reaches to 2%. The third method can also achieve a solution using a lower number of
1025 cores, for instance, PUBRF provides a solution for $BenchM^+$ on only one core while
Simple Mapping and SMSAFR need at least two cores to find a feasible solution.

Time complexity analysis. To provide a comparison between the three proposed
methods in terms of processing time, we first present a time complexity analysis to
specify the order of magnitude for the processing time of the proposed methods and
1030 then we report the execution time of the methods derived from the experiments.

In the first method, the number of iterations of SMSA is $\log_{\mu}(\psi_f/\psi_s)$. By replacing
 $\psi_f = 1/m$ and $\psi_s = O(N)^3$ we reach $O(\log_{\mu}(m.N))$ (recall that m , M and N denote
the number of runnables, the number of tasks, and the number of cores respectively).
The dominant factor in each iteration of SMSA is to calculate the total cost function
1035 for $(N - 1)$ neighbors. The time complexity of the total cost function is $O(M.m^2 +$
 $N.M^2)$. Let $M = O(m)$, and we also know that the number of cores (N) is significantly
lower than number of runnables, hence the time complexity of the total cost function
is $O(m^3)$. Accordingly, the time complexity of SMSA is $O(N.m^3.\log_{\mu}(m.N))$. The
number of iterations in the REF function is $M^2 = O(m^2)$, however, since in the first
1040 method the REF function is not invoked within the SMSA function (it is executed after
SMSA), and the time complexity of SMSA dominates that of REF, the time complexity

³ ψ_s is set with regard to the total utilization of the given workload. Since the maximum utilization of a
given workload could not be larger than the number of cores, we consider it as $O(N)$.

Table 4: Experimental results for the engine management benchmarks running on a four-core processor.

	Method1(Simple Mapping)		Method2(SMSAFR)		Method3(PUBRF)	
Bench- mark	Processor utilization	Size of task set	Processor utilization	Size of task set	Processor utilization	Size of task set
M^+	0.1782	14	0.1751	12	0.1642	9
M	0.2854	15	0.2756	13	0.2684	11
M^{++}	0.5454	19	0.5326	17	0.5262	14

of the first method is equal to the time complexity of SMSA.

In the second method, the REF function is invoked in each iteration of SMSA and before the calculation of the total cost value. Nevertheless, since the time complexity of the total cost function dominates that of REF, the time complexity of the second method is equal to the first method, i.e., $O(N.m^3.\log_{\mu}(m.N))$.

In the PUBRF algorithm, the number of iterations of the main algorithm is v . At each iteration of PUBRF, the dominant factor affecting the time complexity is SMSA+UBR (we assume that the number of ants is considerably lower than the number of runnables). The time complexity of the UBR function is $O(m^4)$. Thus, the time complexity of the third method is $O(v.N.m^4.\log_{\mu}(m.N))$.

According to the experiments, for the main benchmark the execution time of the three methods are 30.31 sec, 33.16 sec and 273.35 sec. Note that although the allocation algorithm of the third method works relatively faster than SMSA, the significant overhead imposed by the more complex merging algorithm (UBR) results in a longer execution time in overall.

5.3.2. Comparison Against the Alternative Frameworks

Now let us have a look at the performance of the proposed framework using the third method in comparison to the three alternative frameworks. In Fig. 13 the results generated by the three alternative frameworks along with the proposed framework outputs are listed, where vertical axis indicates the average processor utilization of the four processing cores. As we already anticipated, the proposed framework outperforms all the three alternative frameworks in all experiments. For the main benchmark,

in comparison to the CF (common method applied in practice), the proposed frame-
 1065 work reduces the CPU utilization on each core by 2.8%, meaning that in the four core
 processor, the total processor utilization is reduced by 11.2%. The utilization improve-
 ment becomes even more considerable when enlarging the communication size; for
 the *BenchM⁺⁺* the average CPU utilization reduction reaches to 3.68%, i.e., the total
 CPU utilization reduction reaches to 14.7%. Table 5 also represents the data derived
 1070 from Fig. 13 in terms of the improvement percentage of PUBRF against the alternative
 frameworks to provide a more clear view of performance improvement.

Figure 13: Average utilization results.

The interesting point is that even though the EF framework uses an exhausting
 search algorithm, it is not able to achieve the same or a better solution than the proposed
 framework, substantiating that even with a perfect task allocation algorithm we can not
 1075 achieve a highly efficient solution for the problem as long as mapping and allocation
 are not interleaved.

This chart also manifests that if we replace the genetic algorithm as a well-known

Table 5: The improvement percentage of PUBRF against the alternative frameworks in terms of the average utilization results.

	BenchM+	BenchM	BenchM++
CF	12.24%	9.42%	6.32%
EF	6.65%	5.96%	2.89%
GAF	1.86%	1.90%	2.80%

meta-heuristic algorithm with the version of Max-Min Ant System introduced in this paper, then the quality of solutions decreases. In other words, not only the merging
1080 method and interleaving the mapping and allocation are effective in the performance of the solution framework, but also the quality of the search algorithm has a great impact. There is no need to say that in the proposed framework we are not able to replace our algorithm with an exhaustive search algorithm (like the BT algorithm) since in terms of the time complexity it is impossible to execute an exhaustive search for a real
1085 world benchmark such as the EMS benchmark that we are using here. Regarding the time complexity of the GAF, it takes 1654 sec in average which is severely worse than 273.35 sec that it takes for the proposed framework.

Now let us have a look at the effect of the number of shared runnables on the performance of the proposed framework. In following we show that when the the
1090 dependency ratio (κ) is changed how the processor utilization is affected. To do so, we consider the main benchmark (*BenchM*) and we just play with κ , while we attempt to keep the total communication size constant.

According to Figure 14 the proposed framework surpasses all the alternative frame-
1095 works for all various dependency ratios. As we intuitively expected, since the CF framework does not care about task dependencies to select the nominated core for putting a task on that, when increasing the dependency ratio the CF results are deviating more from the solutions generated by the proposed framework. Accordingly, an automotive system with a larger communication size and a more intense dependency ratio motivates us to apply the proposed framework to enhance resource efficiency of
1100 such systems in terms of processor utilization. In addition, as the proposed framework attempts to reduce the inter-core communication and remote blocking, it indirectly im-

proves memory efficiency as well.

Figure 14: Average utilization results.

6. Conclusion

In this paper we have investigated challenges related to utilization of multi-core platforms in the design of highly efficient and predictable AUTOSAR-based software applications. Specifically, we have looked into the challenges of designing a resource efficient solution in terms of minimizing both the overall communication time inherent in communication among AUTOSAR runnables executing on a multi-core processor, and the synchronization cost imposed by dependent transactions. An abstract communication time analysis was introduced which is able to cover most of the common multi-core architectures. A solution framework to address the problem was presented where both mapping of AUTOSAR runnables to tasks and task partitioning are interleaved to find a highly efficient solution. Although the framework concentrates on minimization of the total load of the processing cores, it also reduces the number of required cores, diminishes the inter-core communications and improves the blocking times.

The framework contains three methods, each of which can by itself be a complete solution for the problem, while the third method is an extended version of the two

former methods. The mapping of runnables to tasks is performed in two steps where the
1120 first step is to create an initial task set which maps transactions to tasks, and the second
step is to update the initial task set by merging tasks with respect to the allocation of
tasks to cores. Furthermore, the allocation of tasks to cores is performed by a meta-
heuristic algorithm which exploits a feedback on the mapping of runnables to tasks.
The reason to include the first two methods in this paper (not only the third one) is
1125 that the first method conforms with the common best practice approach in the literature
where the mapping and allocation are not interleaved, and the second method implies
when we neglect the effect of communication time on CPU utilization.

In the future, we plan to investigate a method which is able to split a transaction
into more than one task. It not only enables us to handle very large transactions, but
1130 it can eliminate the inefficiency whenever multiple runnables in a transaction interact
with various BSW components located on different cores. For example, few of the
runnables of a transaction interact with the BSW located on the core i while the other
runnables of the transaction interact with the BSW located on the core j . This limita-
tion can be relaxed and a trade-off can be considered between the increase of processor
1135 utilization due to precedence requirements between the tasks, on one side, and the com-
munication time reduction due to allocation of different runnables of a transaction to
multiple cores whenever they require services from different cores. It is also desirable
to investigate automotive systems where multiple runnables in a transaction can have
different periods corresponding to their functionality in the system.

1140 **Acknowledgment**

The work presented in this paper has been supported by Mälardalen University and
Vinnova via the FFI initiative "AUTOSAR for Multicore in Automotive and Automa-
tion Industries".

Appendix A. Abbreviations

1145 The abbreviations used in this paper are listed in Table A.6.

References

- [1] AUTOSAR methodology, AUTOSAR std.
URL <http://www.autosar.org>
- [2] K. Klobedanz, C. Kuznik, A. Thuy, W. Mueller, Timing modeling and analysis
1150 for autosar-based software development: a case study, in: Proceedings of the
Conference on Design, Automation and Test in Europe, European Design and
Automation Association, 2010, pp. 642–645.
- [3] K. S. Lakshmanan, G. Bhatia, R. Rajkumar, Autosar extensions for predictable
task synchronization in multi-core ecus, Tech. rep., SAE Technical Paper (2011).
- [4] M. Paolieri, E. Quiñones, F. J. Cazorla, G. Bernat, M. Valero, Hardware sup-
1155 port for wcet analysis of hard real-time multicore systems, in: ACM SIGARCH
Computer Architecture News, Vol. 37, ACM, 2009, pp. 57–68.
- [5] A. Ferrari, M. Di Natale, G. Gentile, G. Reggiani, P. Gai, Time and memory
tradeoffs in the implementation of autosar components, in: Proceedings of the
1160 Conference on Design, Automation and Test in Europe, European Design and
Automation Association, 2009, pp. 864–869.
- [6] J. Y.-T. Leung, J. Whitehead, On the complexity of fixed-priority scheduling of
periodic, real-time tasks, *Performance evaluation* 2 (4) (1982) 237–250.
- [7] T. Langer, L. Osinski, J. Mottok, A survey of parallel hard-real time schedul-
1165 ing on task models and scheduling approaches, in: International Conference on
Architecture of Computing Systems (ARCS), VDE, 2017, pp. 1–8.
- [8] C. Guo, X. Hua, H. Wu, D. Lautner, S. Ren, Best-harmonically-fit periodic task
assignment algorithm on multiple periodic resources, *IEEE Transactions on Par-
allel and Distributed Systems* 27 (5) (2016) 1303–1315.
- [9] M. Bambagini, J. Lelli, G. Buttazzo, G. Lipari, On the energy-aware partition-
1170 ing of real-time tasks on homogeneous multi-processor systems, in: International

Conference on Energy Aware Computing Systems and Applications (ICEAC),
IEEE, 2013, pp. 69–74.

- 1175 [10] S. K. Dhall, C. Liu, On a real-time scheduling problem, *Operations research*
26 (1) (1978) 127–140.
- [11] M. Panić, S. Kehr, E. Quiñones, B. Boddecker, J. Abella, F. J. Cazorla, Runpar:
An allocation algorithm for automotive applications exploiting runnable paral-
lelism in multicores, in: *International Conference on Hardware/Software Code-
sign and System Synthesis*, ACM, 2014, pp. 29–37.
- 1180 [12] H. R. Faragardi, B. Lisper, T. Nolte, Towards a communication-efficient mapping
of AUTOSAR runnables on multi-cores, in: *Emerging Technologies and Factory
Automation (ETFA)*, IEEE, 2013, pp. 1–5.
- [13] H. R. Faragardi, B. Lisper, K. Sandstrom, T. Nolte, A communication-aware solu-
tion framework for mapping autosar runnables on multi-core systems, in: *Emerg-
ing Technology and Factory Automation (ETFA)*, IEEE, 2014, pp. 1–9.
- 1185 [14] A. Colin, A. Kandhalu, R. Rajkumar, Energy-efficient allocation of real-time ap-
plications onto heterogeneous processors, in: *International Conference on Em-
bedded and Real-Time Computing Systems and Applications (RTCSA)*, IEEE,
2014, pp. 1–10.
- 1190 [15] P. M. Yomsi, Y. Sorel, Schedulability analysis for non necessarily harmonic real-
time systems with precedence and strict periodicity constraints using the exact
number of preemptions and no idle time, Ph.D. thesis, INRIA (2008).
- [16] D.-T. Peng, K. G. Shin, Static allocation of periodic tasks with precedence con-
straints in distributed real-time systems, *IEEE transaction on software engineer-
ing* 23 (2) (1997) 745–758.
- 1195 [17] J. M. Rivas, J. J. Gutiérrez, J. C. Palencia, M. González Harbour, Schedulability
analysis and optimization of heterogeneous edf and fp distributed real-time sys-
tems, in: *Euromicro Conference on Real-Time Systems (ECRTS)*, IEEE, 2011,
pp. 195–204.

- 1200 [18] Q. Zhu, H. Zeng, W. Zheng, M. Di Natale, A. Sangiovanni-Vincentelli, Optimization of task allocation and priority assignment in hard real-time distributed systems, *ACM Transactions on Embedded Computing Systems (TECS)* 11 (2) (2012) 85–97.
- [19] K. Tindell, J. Clark, Holistic schedulability analysis for distributed hard real-time
1205 systems, *Microprocessing and microprogramming* 40 (2) (1994) 117–134.
- [20] K. W. Tindell, A. Burns, A. J. Wellings, Allocating hard real-time tasks: an np-hard problem made easy, *Real-Time Systems* 4 (2) (1992) 145–165.
- [21] J.-M. Chang, M. Pedram, Codex-dp: co-design of communicating systems using dynamic programming, *Computer-Aided Design of Integrated Circuits and
1210 Systems*, *IEEE Transactions on* 19 (7) (2000) 732–744.
- [22] Y. Yang, Software synthesis for distributed embedded systems, Ph.D. thesis, Ph.D. thesis/Yang Yang (2012).
- [23] L. D. Briceño, J. Smith, H. J. Siegel, A. A. Maciejewski, P. Maxwell, R. Wakefield, A. Al-Qawasmeh, R. C. Chiang, J. Li, Robust static resource allocation
1215 of dags in a heterogeneous multicore system, *Journal of Parallel and Distributed Computing* 73 (12) (2013) 1705–1717.
- [24] A. Saifullah, J. Li, K. Agrawal, C. Lu, C. Gill, Multi-core real-time scheduling for generalized parallel task models, *Real-Time Systems* 49 (4) (2013) 404–435.
- [25] T. C. Xu, P. Liljeberg, J. Plosila, H. Tenhunen, Exploration of heuristic scheduling
1220 algorithms for 3d multicore processors, in: *International Workshop on Software and Compilers for Embedded Systems*, ACM, 2012, pp. 22–31.
- [26] S. Kramer, D. Ziegenbein, A. Hamann, Real world automotive benchmarks for free, in: *International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, IEEE, 2015, pp. 16–22.
- 1225 [27] Z. Ran, H. Yan, H. Zhang, Y. Li, Approximate optimal autosar software components deploying approach for automotive e/e system, *International Journal of Automotive Technology* 18 (6) (2017) 1109–1119.

- [28] L. Ran, W. Junfeng, W. Haiying, L. Gechen, Design method of can bus network communication structure for electric vehicle, in: International Forum on Strategic Technology (IFOST), IEEE, 2010, pp. 326–329.
- [29] J. Han, J. Pei, M. Kamber, Data mining: concepts and techniques, Elsevier, 2011.
- [30] A. Monot, N. Navet, B. Bavoux, F. Simonot-Lion, Multisource software on multi-core automotive ecus combining runnable sequencing with task scheduling, IEEE Transactions on Industrial Electronics 59 (10) (2012) 3934–3942.
- [31] A. Bertout, J. Forget, R. Olejnik, Minimizing a real-time task set through task clustering, in: International Conference on Real-Time Networks and Systems, ACM, 2014, pp. 23–38.
- [32] A. Bertout, J. Forget, R. Olejnik, A heuristic to minimize the cardinality of a real-time task set by automated task clustering, in: Annual ACM Symposium on Applied Computing, ACM, 2014, pp. 1431–1436.
- [33] E. Wozniak, A. Mehiaoui, C. Mraidha, S. Tucci-Piergiovanni, S. Gerard, An optimization approach for the synthesis of autosar architectures, in: Emerging Technologies & Factory Automation (ETFA), 2013 IEEE 18th Conference on, IEEE, 2013, pp. 1–10.
- [34] S. Kehr, M. Panić, E. Quiñones, B. Böddeker, J. B. Sandoval, J. Abella, F. J. Cazorla, G. Schäfer, Supertask: Maximizing runnable-level parallelism in autosar applications, in: Design, Automation & Test in Europe Conference & Exhibition (DATE), IEEE, 2016, pp. 25–30.
- [35] S. Kehr, E. Quiñones, B. Böddeker, G. Schäfer, Parallel execution of autosar legacy applications on multicore ecus with timed implicit communication, in: Annual Design Automation Conference, ACM, 2015, pp. 42–50.
- [36] N. Böhm, D. Lohmann, W. Schröder-Preikschat, Multi-core processors in the automotive domain: An autosar case study, Proceedings Work-in-Progress Session (2010) 1–6.

- 1255 [37] A. Davare, Q. Zhu, M. Di Natale, C. Pinello, S. Kanajan, A. Sangiovanni-Vincentelli, Period optimization for hard real-time distributed automotive systems, in: Annual Design Automation Conference, ACM, 2007, pp. 278–283.
- [38] N. Feiertag, K. Richter, J. Nordlander, J. Jonsson, A compositional framework for end-to-end path delay calculation of automotive systems under different path semantics, in: Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS), Diva, 2008, pp. 1–8.
- 1260 [39] N. Serreli, G. Lipari, E. Bini, Deadline assignment for component-based analysis of real-time transactions, in: Workshop on Compositional Real-Time Systems, Washington, DC, USA, Citeseer, 2009.
- 1265 [40] P. Dziurzynski, A. K. Singh, L. S. Indrusiak, B. Saballus, Benchmarking, system design and case-studies for multi-core based embedded automotive systems, arXiv preprint arXiv:1601.03708.
- [41] S. Saha, L. Das, Blocking time analysis of multiprocessor priority ceiling protocol in real-time multiprocessor system, in: International Conference on Advanced Communication Control and Computing Technologies (ICACCCT), IEEE, 2014, pp. 984–988.
- 1270 [42] P. Gai, G. Lipari, M. D. Natale, Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip, in: Real-Time Systems Symposium (RTSS), IEEE, 2001, pp. 73–83.
- 1275 [43] F. Nemati, M. Behnam, T. Nolte, Independently-developed real-time systems on multi-cores with shared resources, in: Euromicro Conference on Real-Time Systems (ECRTS), IEEE, 2011, pp. 251–261.
- [44] Software techniques for shared-cache multi-core systems.
URL [https://software.intel.com/en-us/articles/](https://software.intel.com/en-us/articles/software-techniques-for-shared-cache-multi-core-systems)
1280 [software-techniques-for-shared-cache-multi-core-systems](https://software.intel.com/en-us/articles/software-techniques-for-shared-cache-multi-core-systems)
- [45] Performance analysis guide for intel core i7 processor and intel xeon 5500 processors.

URL <http://software.intel.com/sites/products/collateral/hpc/vtune/performanceanalysisguide.pdf>

- 1285 [46] J. Feljan, J. Carlson, The impact of intra-core and inter-core task communication on architectural analysis of multicore embedded systems, in: International Conference on Software Engineering Advances (ICSEA), IEEE, 2013, pp. 402–407.
- [47] C. L. Liu, J. W. Layland, Scheduling algorithms for multiprogramming in a hard-real-time environment, *Journal of the ACM (JACM)* 20 (2) (1973) 46–61.
- 1290 [48] AUTOSAR consortium, AUTOSAR release 4.0, specification on multi-core os architecture v1.0, 2009.
- [49] H. R. Faragardi, R. Shojaei, M. A. Keshtkar, H. Tabani, Optimal task allocation for maximizing reliability in distributed real-time systems, in: International Conference on Computer and Information Science, IEEE, 2013, pp. 513–519.
- 1295 [50] H. R. Faragardi, K. Sandstrom, B. Lisper, T. Nolte, Communication-aware scheduling of autosar runnables on multi-core systems, in: International Workshop on Design Space Exploration of Cyber-physical, Springer, 2014, pp. 11–16.
- [51] E.-G. Talbi, *Metaheuristics: from design to implementation*, Vol. 74, John Wiley, 2009.
- 1300 [52] R. J. Bril, E. F. Steffens, W. F. Verhaegh, Best-case response times and jitter analysis of real-time tasks, *Journal of Scheduling* 7 (2) (2004) 133–147.
- [53] O. Redell, M. Sanfridson, Exact best-case response time analysis of fixed priority scheduled tasks, in: Euromicro Conference on Real-Time Systems, IEEE, 2002, pp. 165–172.
- 1305 [54] S. Schliecker, J. Rox, M. Negrean, K. Richter, M. Jersak, R. Ernst, System level performance analysis for real-time automotive multicore and network architectures, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28 (7) (2009) 979–992.

- 1310
- [55] T. Stützle, H. H. Hoos, Max–min ant system, *Future generation computer systems* 16 (8) (2000) 889–914.
- [56] Y. Oh, S. H. Son, Allocating fixed-priority periodic tasks on multiprocessor systems, *Real-Time Systems* 9 (3) (1995) 207–239.

Table A.6: Abbreviations used in the paper

Acronym	Description
AR	Allocation Representation
AUTOSAR	Automotive Open System Architecture
BB	Branch and Bound
BSW	Basic Software Component
BT	Back-Tracking
CF	Common Framework
ECU	Embedded Control Unit
EDF	Earliest Deadline First
EF	Exhaustive Framework
GA	Genetic Algorithm
GAF	Genetic Algorithm Framework
GCD	Greatest Common Divisor
LCM	Least Common Multiple
OS	Operating System
PCP	Priority Ceiling Protocol
PUBFR	Parallel Version of SMSA with Utilization-based Refinement
REF	Refinement Function
RIG	Runnable Interaction Graph
RMBF	Rate Monotonic Best Fit
RTE	AUTOSAR Run-time Environment
SA	Simulated Annealing
SMSA	Systematic Memory-based Simulated Annealing
SMSAFR	SMSA with Feedback Refinement
SWC	Software Component
UBR	Utilization-based Refinement
VFB	Virtual Function Bus
WCET	Worst-Case Execution Time