Optimizing Application Distribution on Multi-Core Systems within AUTOSAR

Wenhao Wang*, Sylvain Cotard*
VALEO Group Electronics Expertise and Development Services
Créteil, France
wenhao.wang@ensea.fr

Fabrice Gravez*, Yael Chambrin*
VALEO Engine and Electrical Systems
Cergy, France
firstname.name@valeo.com*

Benoît Miramond
LEAT Lab, CNRS UMR 7248
University of Nice Sophia antipolis
Nice, France
bmiramond@unice.fr

***Abstract*—Multi-core platforms have gained in popularity in nowadays automotive domain. But, even if multi-core architectures are now supported by the AUTOSAR framework, this migration remains a great challenge. First of all, software designers need new methods to fill the gap between application description and tasks deployment. The use of multiple cores has also to remain compatible with real-time and safety design constraints. Finally, developers need tools to assist them in the new steps of the design process. We propose in this paper a partitioning method integrated in the AUTOSAR design flow acting as a decision guide for the distribution of complex and real world control applications onto automotive multi-core systems.**

***Keywords*—Multi-core, Partitioning, AUTOSAR, Metaheuristics.**

I. Introduction

Nowadays, the multiplication of electronic features in smart engine control implies the execution in real-time of complex computational models. To face this evolution, cars embed ever more ECUs (Electronic Control Units), increasing again the part of embedded software development in the design costs of new generations of vehicles. In the same time, a trend in automotive industries is the adoption of multi-core architecture in critical embedded systems. Now is the time to put it all together by proposing novel design methods facing the scale up of applications, adapting the design process to face the distribution and prediction issues coming from the multi-core advent, while still ensuring the functional safety standards (ISO26262) of the automotive domain.

AUTomotive Open System ARchitecture (AUTOSAR) [1] contributes to meet the increasing complexity in nowadays' automotive electrical and electronic systems. To achieve the technical goals of modularity, scalability, transferability, and function reusability, AUTOSAR standardizes the software development by separating the application and infrastructure. This allows applications to exist and communicate independently of a particular infrastructure. Since its revision 4.0 AUTOSAR has been introducing a new design dimension by supporting multi-core architectures.

On the one hand, regarding the scheduling policies in multi-core systems, AUTOSAR still adopts the static allocation and static priority for the tasks in the system. Static scheduling has been widely studied in the literature, and it remains an efficient way to address the difficult issues of prediction and validation of complex interactions between tasks and shared resources.

On the other hand, multi-core introduces additional challenges that are still difficult to deal with in real world industrial domains where applications exhibit high complexity and special cases features that do not fit with theoretical models. Thus, the shift towards multi-core systems in the automotive industry has revived the challenge of application partitioning to enhance productivity, reusability and predictibility.

This paper proposes a method for the distribution of command and control applications into multi-core architectures, in the purpose of partitioning the computations on the different cores in a near optimal way. We model the problem considering the AUTOSAR specificities and apply metaheuristic algorithms to solve it. This paper presents the first results of such optimization methods on industrial applications of engine control.

The rest of the paper is organized as follows. Section II presents the automotive context and the industrial design flow in which our contribution takes place. We also describe in this section the state of the art on distribution automation in automotive multi-core systems and we explain why current methods are not applicable for concrete automotive projects. Section III presents the formal modeling of the multi-core problem as a Combinatorial Optimization problem. We present our developed tools for partitioning into multi-core platform in section IV. In Section V, we present the automotive case studies considered to evaluate our approach and the quality of the design solutions explored by our tool. Finally, conclusions and future works are discussed in section VI.

II. Automotive Context & Problem Description

A. Automotive application design using AUTOSAR

AUTOSAR mitigates the problems existing in the automotive systems design process by its standardized three-layer software architecture, i.e., the Application layer, the Basic Software layer (BSW) and the RunTime Environment layer (RTE). In the application layer, the applications encapsulate functionalities within a collection of software components. AUTOSAR follows a software component approach as in several description languages. The software components in AUTOSAR (SWC) can interact independently of a particular infrastructure through an abstract environment called Virtual Functional Bus (VFB). Each SWC contains one or more runnables. These runnables are composed of the pieces of codes that can be executed and scheduled independently. Figure 1 depicts such software architecture. All the runnables

are triggered by one or several events, such as timing event for periodic runnables [2], data received event for data reading notification and operation invoked event for server (function) calls by clients. The communication between runnables is done by writing and reading the variables. For intra-component communications, these variables are labeled as InterRunnable-Variables (IRV) that can only be shared by the runnables in the same software component. Inter-component communications are realized through Ports and Interfaces.

B. Configuration of the embedded software

The implementation of the VFB is realized by the generation of the Run-Time Environment (RTE). RTE is mainly responsible for linking the application to the BSW including Operating System (OS). It also involves the realization of communication between components and the generation of all the RTE events that activate the behavior of runnables.

The configuration of BSW for a specific hardware platform consists in the configuration of the OS and other BSW stacks (communication stacks, memory stacks and I/O stacks). The OS is responsible for the execution of real-time tasks containing executable entities. Each task defines an execution sequence of the runnables mapped to it. The introduction of multi-core in AUTOSAR leads to additional works in the configuration: (1) Software allocation to cores, (2) Task set definition configuration and mapping the runnables to tasks, (3) Variables distribution to memories in the case of hardware architecture with memories hierarchy, (4) Synchronization of the execution flow in multi-core systems.

We consider step (1) and (3) in this paper, as shown in Figure 2. To achieve this goal, real-time scheduling techniques need to be considered in order to adapt the application to the targeted multi-core platform. The considered multi-core platform is composed of a set of 32 bits superscalar cores (from 3 to 8 cores) and a set of associated closely coupled memory local memories. As depicted in Figure 2, data exchanged for inter-core communications are stored at the second level into a shared memory. A typical target is the Aurix architecture from Infineon[1].

In the automotive domain, only static scheduling policies are supported by AUTOSAR. That is, all the runnables are statically attached to cores, and the runnables in each core are scheduled by a local scheduler. One of the core often executes in lockstep the critical parts of the application, which then needs redundancy. Reliability is not considered in the current version of the tool, but will be considered in the future works. In the design flow presented in section III, all these decisions have to be explored by the proposed automation method.

[1]http://www.infineon.com

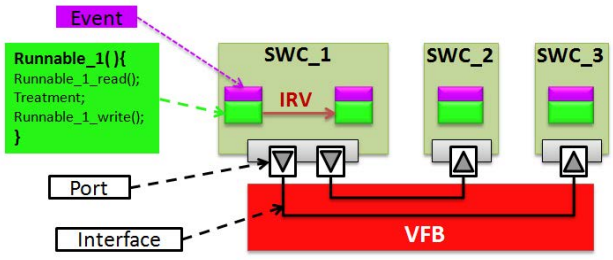

Figure 1. AUTOSAR software architecture

C. Application partitioning

In this work, we focus on partitioning applications driven by control and data flow (e.g. engine control, brake control etc.). For that type of command and control applications the order in which the individual statements are executed is very important and the proportion of parallel code is often hard to identify. In consequence, the partitioning of automotive applications into multiple cores requires a fine analysis of the dependencies between runnables and tasks. The paper studies in what extent a design automation method can be employed for that purpose.

D. Related works

The theoretical formulation of application partitioning has been widely studied in the past either in the domain of multiprocessor computing [3] or in hardware/software co-design [4]. But the proposed partitioning methods rapidly faced a major limitation considering the lack of real use cases integrated in a full industrial working process. The explored solutions at high-level were too abstract to be really considered. Moreover, when considered alone, the formal optimization clears out the designer from the problem and neglects that not all the design considerations can be theoretically formulated.

In recent years, the adoption of multicore architectures in critical embedded systems has revived the need of design flows fully integrating the exploration phase. So, several works have dealt with the partitioning problem of AUTOSAR applications onto multi-core systems. So, in [5] authors developed heuristic algorithms for mapping runnables into different cores. In this paper, runnables are grouped into clusters before being distributed across cores by optimizing a specific objective function. The works of Faragardi et. al [6] and Saidi et. al [7] proposed a heuristic algorithm to create a task set according to the mapping of runnables on the cores. With the goal

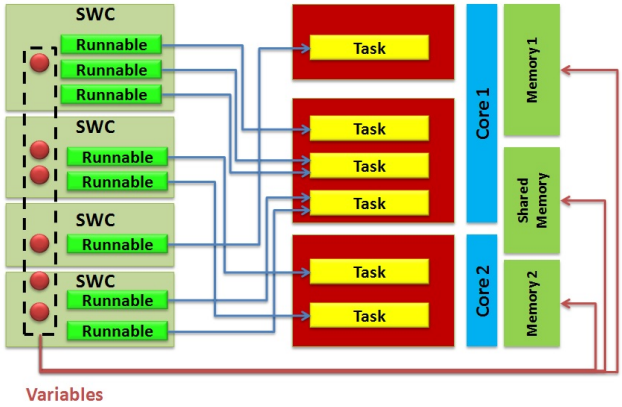

Figure 2. Application partitioning on the targetted multi-core platform

of minimizing the communications between runnables, the problem is classically formulated as an Integer Linear Programming (ILP). Therefore, conventional ILP solvers can be easily applied to derive a solution. In [8], Genetic Algorithms (GA) are applied to partition the application in an optimal way. The results of task allocation are evaluated by their simulation tool TA-Toolsuit. A demonstration version is available in [9] but only simplistic applications are provided. The limitations of the demonstration version avoid any comparison on real applications.
However, all the partitioning methods proposed in the literature only consider the optimization formulation without considering the full design flow. Compared to the existing research work, the proposed method is fully thought into an industrial V-cycle development process. Our contributions are then the following:

- a full working process composed of 5 main phases (see figure 5): application description, dependency analysis, design space exploration, configuration of the executive layer, validation onto the target device;
- back-annotation from the validation phase, enabling optimisation of the cost function from real and credible measurements;
- proposition of a cost function mixing functional and non-functional criteria;
- validation of the solutions explored at high-level thanks to a fully automated refinement process; The detailed description of our working process in section IV will explain how to achieve this goal.

We summarize in table I the properties of the partitioning methods existing in the literature in order to point out our contributions.

Reference	Cost function f	Optimization method	Target architecture	Associated design flow	Validation
[3]	Intercore Communication Overhead (ICO)	ILP	Heterogeneous multicore	No	No
[4]	Total execution time	*SA	Heterogeneous Hw/Sw	No	No
[5]	ICO	heuristic	Automotive multicore	No	No
[9]	Response time, ICO	GA	Automotive multicore	partial	high-level simulation
Ours	Load balancing, ICO, real-time constraints, response time	SA, GA, *TS	Automotive multicore	full	cycle-accurate

Table I. COMPARISONS OF SOFTWARE PARTITIONING METHODS IN THE STATE OF THE ART. *SA IS SIMULATED ANNEALING ALGORITHM, TS IS TABU SEARCH ALGORITHM

III. INTEGRATION OF A SW DISTRIBUTION METHOD IN THE AUTOSAR DESIGN FLOW

Our proposed automation of the partitioning first asks to formalize the design constraints as a combinatorial optimization problem which mainly relies on the definition of the objective function.

A. Combinatorial Optimization theories

Minimizing the objective function involves researching an optimal combination of runnables to cores as well as variables to memories. This problem is assimilated to a Combinatorial Optimization (CO) problem, where solutions are encoded with discrete variables. A model $\mathcal{P} = (S, \Omega, f)$ of a CO problem consists in:

- S: a search space where a finite set of discrete variables are defined;
- Ω: a feasible domain defined by a set of constraints;
- f: an objective function to be minimized.

As the CO problems are NP-hard [10], the complete methods that search for every instance to find optimal solution might need exponential computation time in the worst case. For practical purpose, we often prefer to get a good solution (not the optimal solution) in a significantly reduced amount of time, even though finding optimal solution is not guaranteed. Metaheuristic is this kind of approximate algorithm that aims at exploring the search space efficiently and effectively. This class of algorithms includes - but not restricted to Simulated Annealing (SA), Tabu Search (TS) and Genetic Algorithm (GA).

Simulated Annealing is inspired by the physical annealing process of solids. It accepts solutions according to an acceptance probability computed following the Boltzmann distribution $e^{-\frac{f(s')-f(s)}{T}}$, where s' is a neighbor solution of the current solution s, and T represents the temperature.

Tabu search maintains a tabu list and allows adopting the best solution in the neighborhood in condition that it does not exist in the tabu list. This solution is then added into the tabu list after this iteration.

Unlike SA and TS that deal with one single solution at each iteration, Genetic Algorithm treats a population of potential solutions at each iteration. GA uses ideas from biological evolution that includes three main steps: selection, reproduction and replacement. More details on these classical methods can be found in [11], [12], [13].

De-facto, this optimization problem has been modeled in industrial contexts. Reference [14] applies GA to solve the optimization issue of the SWC-to-ECU mapping, and reference [8] applies GA to optimize the task allocation for multi-core processors.

B. Application and architecture modeling

The software architecture is modeled using a directed graph $G(V, E)$, such that V is a set of nodes (set of runnables here for AUTOSAR application) and E is a set of edges, also called transitions (links between runnables). A node is modeled as an execution time, a trig mode, a period. A transition has a weight that depends on the size of data transmitted, the period of the producer, etc. The graph size is optimized by the creation of buses between nodes.

We assume that each node V is associated with a period T_i. For the runnables activated by a periodic event, T_i is the period

of the activating event. Similarly, for the runnables activated in response to another runnable's result or request, T_i denotes the period of the runnable invoking it or, if it is still not a period event, our partitioning tool identify the one invoking it and so on iteratively.

Each runnable is also associated with execution information that contains two parts: variable accessing time T_a and execution time T_e.

The accessing time T_a mentions the time for a runnable to read or write its related variables located in the memories. In our multi-core architecture, each core is associated with a local distributed memory. Runnables can also access data in shared memories. It is worth to mention that all the memories can be accessed by all the runnables distributed to all the cores, which implies that the accessing time for a runnable to write or read a variable varies with the location of the runnable as well as the location of its variable. In Figure 2, the right part represents a simple model of our architecture with 2 cores: each core has a local memory and there is one shared memory in the system. The accessing time for runnable ρ to access variable θ depends on the location of ρ and θ. All the potential cases are shown in Table II, where $T_{a_\theta}(i,j)$ means the accessing time for ρ located in ith core to access θ located on jth memory. It is obvious that T_a is much shorter if we locate θ into the local memory of the core where ρ is located. Accessing a variable in the local memory of another core is much slower; and accessing to shared memory is dedicated to data exchanged between cores.

The execution time T_e represents the time for a runnable to execute some instructions. T_e is influenced by two factors. One is the performance of the core on which the runnable is located in. The higher computing power, the faster the runnable will finish its corresponding treatment. In a real-life automotive system, the real-time constraints also depend on the execution modes, such as the engine speed or driving modes. E.g. the amount of executed codes depend on the vehicle speed. In the following we denote these contexts cases, and it is the second factor that influences T_e. A weight w is associated to each case to model its importance in the system (high value of w means high importance). So for a given runnable ρ, $T_{e_\rho}(i,n)$ varies with its location (ith core) and the nth case, an example with 3 cases is shown in Table III.

The communications between nodes are presented as transitions E. Each transition contains two nodes ρ_i and ρ_j, $(\rho_j, \rho_j \in V)$, model $\rho_i \mapsto \rho_j$ present the dependency between ρ_i and ρ_j, where ρ_i is the predecessor of ρ_j and ρ_j is the successor of ρ_i. The predecessor ρ_i sends a set of variables that are received by the sucessors. The sum of the size of these variable is noted as S_s. So the sent data rate for the predecessor ρ_i is

$$s_{\rho_i} = \frac{S_s}{T_i} \tag{1}$$

Similarly, the received a set of variable from predecessors. The sum of the size of these variable is noted as S_r, and received data rate for the the successor ρ_j is

$$r_{\rho_j} = \frac{S_r}{T_j} \tag{2}$$

Table II. ACCESSING TIME FOR RUNNABLE ρ TO VARIABLE θ

Variable θ	Mem_1	Mem_2	Share Mem
Core_1	$T_{a_\theta}(1,1)$	$T_{a_\theta}(1,2)$	$T_{a_\theta}(1,3)$
Core_2	$T_{a_\theta}(2,1)$	$T_{a_\theta}(2,2)$	$T_{a_\theta}(2,3)$

Table III. EXECUTION TIME FOR RUNNABLE ρ

Runnable ρ	Case_1	Case_2	Case_3
Core_1	$T_{e_\rho}(1,1)$	$T_{e_\rho}(1,2)$	$T_{e_\rho}(1,3)$
Core_2	$T_{e_\rho}(2,1)$	$T_{e_\rho}(2,2)$	$T_{e_\rho}(2,3)$

C. Cost function formalization

According to the discussion above, we give the formulation of the problem as follows:

The multi-core architecture is composed of a set of cores $\{\pi_1, ...\pi_I\}$ and a set of memories $\{M_1, ...M_J\}$, with $J > I$ and M_1 to M_I are attached to the local memories of cores π_1 to π_I, while M_{I+1} to M_J represent the shared memories. The partitioning involves the distribution of a set of runnables $\{\rho_1, ...\rho_K\}$ to the cores and also a set of variables $\{\theta_1, ...\theta_L\}$ to the memories. We note $\rho_{k,i}$ when the kth runnable is distributed to ith core and $\theta_{l,j}$ when the lth variable is distributed to jth memory. $T_{a_{\theta_l}}(i,j)$ mentions the accessing time for the runnable located on the ith core to access the variable θ_l located on jth memory. We also define a set of contexts cases $\{K_1, ..., K_N\}$, and w_n is the weight for the nth case. Then, $T_{e_k}(i,n)$ represents the execution time for kth runnable located in the ith core and in the nth case. Thus when we distribute a runnable ρ_k to core π_i, based on its execution time, accessing time and period, this runnable results in a load $u_{\rho_{k,i}}$:

$$u_{\rho_{k,i}} = f\left(T_{a_{\theta_l}}(i,j), T_{e_k}(i,n), T_k\right) \tag{3}$$

The load of core π_i is the sum of the loads caused by the runnables distributed to this core, mentioned as u_{π_i}:

$$u_{\pi_i} = \sum_k u_{\rho_{k,i}} \tag{4}$$

Considering α as the max load ratio of a core, the load of each core must respect

$$\forall i : u_{\pi_i} < \alpha \tag{5}$$

Based on the loads of each runnable in (4) and the weight w_n of each case, we can deduce the load for the entire multi-core system.

The load of the multicore distribution must be well balanced, with a tolerated deviation of 2%. It appears as the main design constraint in the optimisation formulation.

We also define the size of memory M_j as S_j and the size of variable θ_l as S_{θ_l}. The maximum occupation ratio of each memory is noted β. So the occupation ratio of each memory should not exceed it:

$$\forall j : \frac{\sum_l S_{\theta_l}}{S_j} < \beta \tag{6}$$

The intercore communications represent the main challenge to pass from monocore to multicore architectures. They are estimated by summing the number of data access per

millisecond. Minimizing this overhead, under load balancing constraints, corresponds to the objective function that evaluates the performance of our partitioning solutions:

$$\mathcal{F} = g\left(u_{\rho_{k,i}}, w_n\right) \tag{7}$$

Equation (7) shows that the cost value of the objective function is decided by the loads generated by the runnable ($u_{\rho_{k,i}}$) in every execution context (weighted by w_n). The loads consider two elements: the CPU utilization computed as $\frac{T_e}{T_k}$ and the communication overhead that is influenced by accessing time of the variables. It is obvious that different ways of partitioning will change the cost value of objective function.

Figure 3 (a) shows a simple example: the application contains 3 runnables ρ_1, ρ_2 and ρ_3. ρ_1 send variable θ_1 to ρ_2 and θ_2 to ρ_3. The hardware model shown in Figure 3 (b) consists in a 2-core system with a shared memory M_3. Besides, each core is attached to a local memory M_1 and M_2. We assume that the execution time for each runnable at each core is identical. The objective is to distribute the application to this 2-core system. Solution in Figure 3 (c) allocates all the runnables in one core, and distributes the variables in its local memory. This could minimize the accessing time, so the communication overhead is low. But the loads of CPU are not well balanced as the other core is empty. Solution in Figure 3 (d) allocates the runnable ρ_3 to the other core, so when runnable ρ_1 finishes its execution, ρ_2 and ρ_3 can execute parallel. Therefore the loads of CPU are better balanced. However, the communication overhead is increased as the accessing time for the variables allocated at the shared memory is much longer. This compromise is considered in our objective function.

In this work, we aim at developing a practical policy for partitioning software applications, composed of several hundreds of nodes, onto multiple cores that will minimize this objective function, while respecting the dependencies and the constraints in AUTOSAR and also verifying the rules in (5) and (6).

D. Description of the optimum solutions searching method

The partitioning solution is represented as a vector in which each element presents the position for runnables or variables. The vector is an ordered list with the length of $l = L + K$, where the L represents the number of the variables and K is the number of runnables to be distributed. In the position i of the vector, $i \in [0, L)$, a memory is distributed for the corresponding variable and in position j, $j \in [L, l)$, a core is attached to the corresponding runnable. The different combinations of the memories and cores will change the value of objective function. In order to deal with this combinatorial optimization problem, we take the metaheuristic algorithms as a solver. The method to search the optimum solution is described as follows:

- the **initial solution** can be obtained in a random way as well as by heuristic guide. The quality of the initial solution would affect final solution;
- the **neighbourhood structure** of a solution defines its possible move direction for improvement, which involves 2 operators: operator $N1$ changes only the memory attached to one single variable to another memory or operator $N2$ changes only the core attached to one single runnable to another core. The move will choose one operator randomly each time;
- the **constraints** guarantee the viability of solutions on each move proposed by the neighbourhood operator: all the solutions (including the initial solution) shall respect all the defined constraints;
- the **metaheuristic algorithms** provide the searching policies to find the optimum (or good) solutions in an efficient way: starting at the initial solution, the improvement is effectuated by a single move (defined by neighbourhood structure) each iteration.

In this work, we apply three metaheuristic algorithms : SA, GA and TS. All the algorithms share the same framework such as initial solution, neighbourhood structure. Each algorithm effectuates different searching policies to find the final solution. The evolution of solutions iteration by iteration is illustrated in Figure 4, which shows the convergence of optimization process by our objective function with the goals that both benefit the acceleration of performance from multi-core and respect the real-time constraints on the dependant tasks.

The results obtained with this method show the contributions of our work :

- the **quality** of the solutions explored according to the cost function;
- the **diversity** of the solutions around the optimum at the convergence of the method. This diversity will

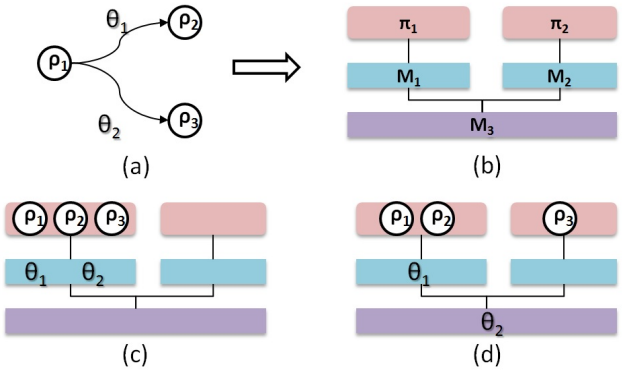

Figure 3. Explanation for objective function (a) Application; (b) Hardware model; (c) and (d) Solutions considering different criteria

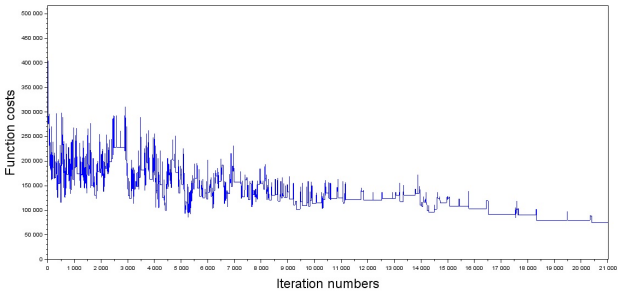

Figure 4. An example of research result by SA

provide the designer the guide needed to take its final decision [15];

- the **scalability** of the method over complex AUTOSAR applications potentially composed of several hundreds of runnables and several thousands of transitions.

IV. Presentation of the Partitioning Tool

Our partitioning tool presented in Figure 5 is designed to analyze the automotive applications in AUTOSAR and distribute them automatically onto cores. The application targeted in these experiments is composed of a set of software components (SWCs) described in the input AUTOSAR XML files (.arxml). The tool is based on eclipse and written in Java. It allows to analyze a software application by parsing the AUTOSAR XML files. The working process of the tool is described as follows.

A. Dependency analysis

As the high sensibility of the execution order and low proportion of parallelism exist in the targeted applications, the partitioning of automotive applications into multiple cores requires a fine analysis of the dependencies between functional elements. For this reason, the tool analyzes the features by the following steps:

- re-works the software architecture: modeling the application as a directed graph presented in the section III-B;
- determines the levels of dependency: building statistics on each transition in the graph;
- analyzes the data information for each transition such as data size, data rate, data unit ;
- identifies the sequences of communications: extraction of data flows.

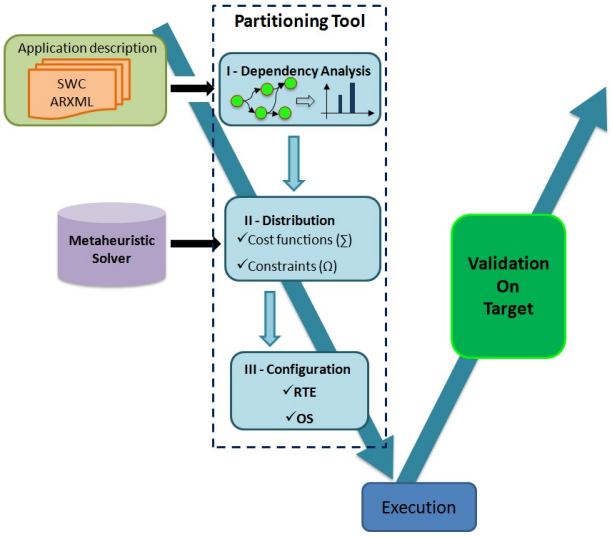

Figure 5. Working process for partitioning automotive application onto multicore architectures

The results of the analysis of dependencies drive the distribution step. More precisely, the level of dependencies and data information are used to evaluate the communication overhead; the sequence of execution would guide the distribution tool to determinate the response time for sequence chains.

B. Software distribution

For the distribution part, the tool performs design space exploration (DSE) of the graph designed in dependency analysis step, to distribute the applications into multi-core systems. As stated in the section III, the problem is formalized as a combinatorial optimization problem, which mainly relies on two essential elements: the definition of objective function and a given set of constraints that each solution shall respects. Therefore, applying the metaheuristic algorithms, the tool researches the solutions by evaluating the defined objective function that was presented in (7). Every research step has to respect the constraints presented in (5) and (6).

As to the granularity of element for the distribution, a preparation step is involved in order to minimize the inter connection between the cores. For doing this, the tool determines the dependencies between runnables based on the results obtained by dependency analysis step such as the communication between runnables or the chains of event, etc. Then the tool groups runnables according to the level of dependency between clusters. AUTOSAR SWC is the atomic element that is not allowed to be divided into multiple partitions, thus, all the runnables in the same SWC shall be mapped into the same partition. Respecting this constraint, the tool then gathers again certain clusters into groups. By doing this, we obtain the atomic elements to distribute into cores. These elements are referred as CpuEntities. Then the tool distributes the runnables, or more precisely the CpuEntities, into cores. It also distributes the variables to the different memories. To do this, the tool applies the selected metaheuristic algorithms to find the optimal combination for runnables and cores.

The output of this tool will provide the designer a set of distribution solutions. Each solution is represented as a vector in which each element presents the position for runnables or variables. The designer can then analyze the subset of near-optimal solutions to finally select the best distribution according to non-formalized criteria (designer experience, reusability, management...). For these reasons, we developed our partitioning tool as a decision guide environment. Thus, the expected behaviour of the underlying optimisation heuristics is not to provide only the optimal solution but also the subset of near-optimal solutions.

C. Configuration of the executive layer

This step contains the configuration of RTE, OS and other BSW stacks. The partition solutions that provide the allocation information on each core update the configuration of RTE and OS. The configuration of RTE consists in the mapping of runnables into tasks. The configuration of OS includes the terms of priority definition for tasks, tasks partition, allocation of resources, communication and synchronization between tasks. After that, embedded source code of the solution is generated, compiled and downloaded on the target architecture for the final validation of both the real-time and the functional exigencies.

V. Experimental Results

We now describe the experiments leaded to determine the optimization method the best adapted to our context and to validate the explored solutions.

A. Results of dependency analysis

The method has been evaluated with three application descriptions. The first one labeled as *App_1* is composed of a small amount of components. This application is built in a random way and the exploration space for this application is exhaustive thanks to its small quantity. Besides, this application contains 3 context cases for the execution time. We have other two applications (labeled as *App_2* and *App_3*) correspond to bigger real industrial use-cases which represent a portion of a full application of engine control. For these two application, we consider only one running execution mode, therefore there is only one context case:

- *App_1* contains 15 SWCs with 32 runnables. After analyzing this application, the tool generates 6 CpuEntities with 7 variables;
- *App_2* contains 25 SWCs and 208 runnables, the tool generates 14 CpuEntities with about 493 variables;
- *App_3* contains 68 SWCs and 562 runnables, the tool generates 21 CpuEntities with about 1358 variables.

The tool also analyzes the transitions information for each application and classifies these transitions according to the different level of dependency. The results for the three tests are shown in Table IV.

B. Results of distribution exploration

The next step consists in distributing the application into a specific multi-core architecture. Our targeted multi-core architecture contains 3 cores, a shared memory and each core is assigned to a local memory. In order to distribute these CpuEntities into 3 cores and the variables into 4 memories, the tool applies the selected metaheuristics: SA, TS and GA. The small application allows us to obtain independently all the possible combinations and to calculate their cost based on (7). Thus we can identify the optimal solution with the smallest cost values among all the potential solutions. The distribution of cost values for all the partitioning solutions of the first application (noted *App_1*) is illustrated in Figure 6. The figure exposes the complexity of the problem even when considering an AUTOSAR application composed of only 32 runnables. The number of feasible solutions exceed several hundreds of thousands solutions (279888 exactly), and so the optimal solution (with value of cost at the left side in Figure 6) only represents $0,0357\%$ of the landscape.

We then apply each algorithm 10 times on application *app_1*. The cost bands of solutions found by each algorithm are compared to the previous distribution of cost values as shown in Figure 6. The more precise results are shown in Table V. GA (in red in Figure 6) always find the optimal solution. SA also find the optimum and other solutions with a cost between $4,02$ and $4,2$. Finally TS never find the optimal solution, but only solutions with costs between $4,1$ and $4,25$. From these results, we can notice that GA can always find the best solution in a longer running time. SA runs faster with a chance less than 50% to find the optimal solution. Considering TS, unfortunately, we never get the optimal solution, but solutions very close to it.

Table IV. Application analysis results

Application	Number of SWC	Number of runnables	Number of transitions	Number of CpuEntities
App_1	15	32	27	6
App_2	25	208	1558	14
App_3	68	562	6826	21

Table V. Optimization results for application APP_1 by GA, SA and TS Metaheuristics. Only Ga works on a population size of 10. SA and TS only explore 1 solution per iteration.

Algorithms	Deviation to best solution	Optimal solution finding times/10	Average Run Time (ms)	Number of explored solutions
SA	0.0	4	243.52	1000000x1
GA	0.0	10	279362.09	100000x10
TS	1.97%	0	7467.08	1000000x1

For the two other applications, we considered real industrial use-cases and focus on quantitative results. We applied only SA and GA, as TS does not show its capability to find the optimum for the small application. We remind that we consider constraints of loads balancing for each solution, data for inter-core communication are allocated in the shared memory, and the cost function minimizes inter-core communication overhead (using IOC). With the growth of the application size, it becomes impossible to obtain all the solutions in the exhaustive way as we did on the small application. So, the optimal solution can not be exactly determined. Thus, we used a different criteria to evaluate the quality criteria of the optimization methods. We focused on the standard deviation between the costs of solutions obtained by each algorithm and the cost of the best solution it ever found. The results for the two applications are shown in Table VI and Table VII. From these results, GA can no longer find better solutions than SA. Besides, the run time of GA is much longer. The average run time for both algorithms increases with the size of application, this is shown in Figure 7.

As previously explained, the goal of our partitioning tool is not to still reach the optimum but rather to prune the design space, and only present to the designer the most promising solutions according to a specific objective function. Only the designer can then identify feasible solutions and take the final decision. Nevertheless, from the optimization point of view,

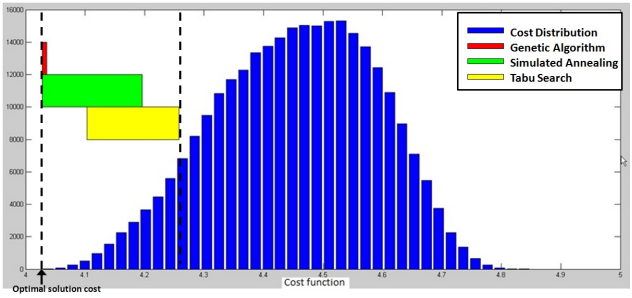

Figure 6. Distribution of the costs of all the partitioning solutions for application *app_1*. The cost band on the left represent the subset of solutions found by the GA, SA and TS methods.

these experiments allowed to identify the algorithm the best adapted to this design problem, even if each of them could be tuned to reach better results. Hence, for this use case, SA shows its ability to provide both the optimal solution and a set of other solutions approaching the optimal one. SA also seems to better scale with the application complexity. The analysis of performances metrics (cores loads, memory occupation, execution time) then allows finer selection.

After the distribution phase, the embedded source code of the solution is generated, compiled and downloaded on the target architecture for the final validation of both the real-time and the functional exigencies.

C. Results of the validation

The target hardware platform is a TC27x tri-core microcontroller. There are two category of memories: the local memories attached to each core and the global memories. There are three cores in this architecture, two identical cores TC1.6P and another core TC1.6E. All these three cores execute the same set of instruction. There are two independent on-chip buses in the tri-core architecture: Shared Resource Interconnect (SRI) and System Peripheral Bus (SPB). The SRI is the crossbar based high speed system bus for TC 1.6.x CPU based devices. The SPB connects the TC1.6 CPUs and the general purpose DMA module to the medium and low bandwidth peripherals. More details can be seen in [16].

We deployed the application App_2 onto this multi-core platform to measure the communication overheads and CPU loads for several distributions. After starting the execution, the trace information were obtained by the vendor tool - Lauterbach Trace32. We present in this section the results obtained for two specific solutions:

- initial solution: it is the first generated solution from which the metaheuristic algorithms search the near-optimal distributions;
- optimised solution: the best solution founded by SA and GA. As shown in the section V-B, the two algorithms could find the same optimised solution for this App_2.

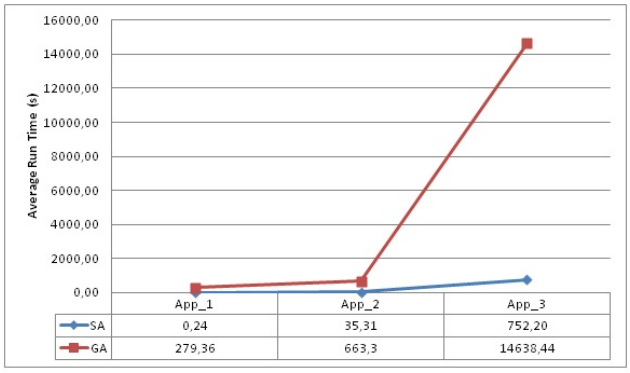

Figure 7. Scalability of the execution time of GA and SA optimization methods. The average run time is plotted according to the application complexity. The figure specifies the average measured values.

Table VI. OPTIMIZATION RESULTS FOR APPLICATION APP_2 BY GA AND SA METAHEURISTICS

Algorithms	Deviation to best found solution	Best solution found	Average Run Time (ms)	Number of explored solutions
SA	0.12%	8	35305	1000000x1
GA	2.83%	7	663305.2	100000x10

Table VII. OPTIMIZATION RESULTS FOR APPLICATION APP_3 BY GA AND SA METAHEURISTICS

Algorithms	Deviation to best found solution	Best solution found	Average Run Time (ms)	Number of explored solutions
SA	21.23%	1	752202.4	1000000x1
GA	10.48%	0	14355693.8	100000x10

The source code of all the solutions found by the exploration tool can be generated and associated to the code of the embedded executive layers. Once compiled, the binary file is downloaded onto the device. We aim at comparing the estimated and real (measured) performances of the explored solutions. The measured communication overhead for the two solutions specifically studied in this paper are given in Table VIII. Estimated values are given by considering the number of data access per millisecond (taking into account the number of fetches required to get data, i.e. the size of data). Measurements are done onto the platform using Trace32 tool and provide the exact amount of time used for intercore communication. It appears in Table VIII as a percentage of the total application execution time. The trace of execution are extracted and analysed in a pseudo-automatic manner. We can for example compute the average load per intercore communication functions (called IOC), and per core by identifying the individual IOC calls, and their execution time, during a period of time.

By comparing real values with estimated values, we can observe that the optimization done by the tool is confirmed by the experiments despite an estimation error. More precisely,

- Table VIII represents the intercore communication cost for each source core (executing the producers of data)
- Table IX shows the associated core loads,

both for the initial and optimised solutions. More precisely, we present in Table VIII the following results of the intercore communications for both solutions:

- the transition counts represent the number of transitions between cores. Each transition is related to 2 IOC functions: send and receive;
- the estimated overhead considers the number of data access per millisecond (taking into account the number of fetches required to get data, i.e. the size of data);
- the measured overhead is the load of IOC functions measured on the target. We can observe in this table that measured overhead is correlated with both transition counts and estimated overhead.

These results show a systematic reduction of the communication and the load metrics, and allow to evaluate the error of estimation.

Table VIII. ESTIMATION AND VALIDATION RESULTS OF THE COMMUNICATION OVERHEAD ON THE AURIX TRICORE TARGET.

	Initial Solution			Optimized Solution		
Cores	transition counts	estimated overhead	measured overhead	transition counts	estimated overhead	measured overhead
Core_0	144	26,25	3,25%	114	26,03	2,0%
Core_1	99	37,20	3,23%	67	22,68	0,94%
Core_2	110	23,50	1,37%	78	15,00	1,2%
Total	353	86,95	7,85%	259	63,71	4,14%
Gain				26,63%	26,73%	47,26%

Firstly, according to the Table VIII, the optimized solutions are better, about 26% more efficient from the partitioning tool point of view, and about 47% in the real platform. It corresponds to about 26% of minimization of the number of intercore transitions. Even if communications are not represented with the same unit in Table VIII we can observe a difference in the global gain. This error of estimation is not very surprising. Performance estimation is currently computed only from the amount of data exchanged between cores. In fact, the count of transitions impacts also the communication overhead. This explains why in Table VIII the decrease of estimated overhead does not necessarily improve the measured overhead while the transition count is increased. Besides, additional features such as the OS services and the memory protection unit (MPU) increase the communication overhead. These overheads should be modeled in the next version of the tool.
Moreover, the on-board profiling showed that, as a system call is done each time the application needs an inter-core communication, it could be more efficient to have 2 data accesses in one communication channel than having 2 communication channels with 1 data access in each. This new optimization will be added as a new type of move (section III-D) during the exploration.

Table IX. ESTIMATION RESULTS OF THE CPU LOADS ON THE AURIX TRICORE TARGET

Cores	Initial Solution (estimated)	Initial Solution (measured)	Optimised Solution (estimated)	Optimised Solution (measured)
Core_0	4.62%	21,8%	5.34%	20,0%
Core_1	6.51%	21.1%	4.66%	13.3%
Core_2	4.66%	14.4%	5.78%	15.6%
Total	15,79%	57,3%	15,78%	48,9%

Secondly, table IX shows the estimated CPU load for initial and optimized solution. The partitioning tool considers the CPU load balancing as one of the design constraints, and ensures a global load balancing between cores (with a 2% tolerated deviation). The results show that this constraint is respected by the partitioning tool, since based on estimations. The load of cores is measured with Trace32 using dedicated scripts whereas we only consider the load generated by applicative runnables in the estimations. The loads of these runnables were previously measured with Trace 32 onto a single-core distribution (without intercore communication) and back annotated into the application description file.

Thus, the other parts of code executed by the application, such as BSW, OS and other stacks are not considered in the estimations computed by the partitioning tool. On the other hand, real CPU loads are obtained on-board by measuring the time spent in the idle task, and by subtracting the load dedicated to the BSW tasks (main functions). If the current measure provides a best precision compared to high-level estimations, it can still be improved since OS features and other modules are counted in the application load. This explains the differences in the results presented in Table IX. Precisely, we can observe a constant global load according to estimations whereas measures point out the consequences of the distribution onto the core load, due to OS and communication overheads. The execution time of the functional code of the runnables only represents 30% of the global load of this automotive system.

We are now working on adding an intermediate fast validation phase between the distribution and the validation phase to improve the quality of our estimations during exploration. We are developing a SystemC transactional simulator of the multicore software distribution. Besides, similarities between the SystemC language and AUTOSAR have already been demonstrated [17]. At this level, the hardware architecture can be essentially abstracted. The concurrency is modeled at the core level, the goal being to reduce the estimation error on communication costs, to explore more accurately the scheduling of tasks, and to identify in the early phase of the design the conflict of resources. This new simulation step will allow short and long validation cycles in the same multicore design flow.

VI. CONCLUSION

We described in this paper the issues in the partitioning of engine control applications in multi-core automotive systems. The proposed partitioning method is the first one fully compatible with the constraints imposed by the AUTOSAR architecture both in terms of software architecture and design process. The corresponding partitioning tool can thus be integrated in a seamless AUTOSAR design flow, from application description to software deployment onto multi-core architectures. Hence, classical optimization methods have been adapted to the automotive context and its specific real-time constraints in an efficient exploration tool. The entire working process has been validated onto real world applications from the AUTOSAR descriptions to the on-board profiling.
The results obtained on complex motor control applications show the benefits of the optimization phase. A 47 % gain has been obtained by minimizing the intercore communication. These first results, obtained on the recent intercore release of AUTOSAR, also point out an increase of the core load when migrating from a monocore to a multicore deployment.

After having proposed a pseudo-automatic top-down refinement process in this paper, we aim at recovering the results obtained by real measurements up to the partioning tool in order to improve the precision of the performance estimations. Moreover, thanks to a multi-criteria formulation of the future version of the cost function, we will be able to take into account several criteria to evaluate multicore distributions such as OS overhead, memory usage, resource conflicts, safety...

REFERENCES

[1] AUTOSAR, http://www.autosar.org, Tech. Rep., last visited: 01/03/2015.

[2] ——, "Specifiaciton of timing extensions, version 2.1.0," www.autosar.org, Tech. Rep., 2014.

[3] Y. Yi, W. Han, X. Zhao, A. T. Erdogan, and T. Arslan, "An ilp formulation for task mapping and scheduling on multi-core architectures," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '09. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2009, pp. 33–38.

[4] B. Miramond and J.-M. Delosme, "Design space exploration for dynamically reconfigurable architectures," pp. 366–371, 2005.

[5] A. Monot, N. Navet, B. Bavoux, and F. Simonot-Lion, "Multi-source software on multicore automotive ecus - combining runnable sequencing with task scheduling," *IEEE Trans. Ind. Electron.*, vol. 59, no. 10, pp. 3934–3942, October 2012.

[6] H. R. Faragardi, B. Lisper, and T. Nolte, "Towards a communication-efficient mapping of autosar runnables on multi-cores," *IEEE 18th Conf. on Emerging Technologies & Factory Automation (ETFA)*, vol. 18, pp. 1–5, September 2013.

[7] S. E. Saidi, S. Cotard, K. Chaaban, and K. Marteil, "An ilp approach for mapping autosar runnables on multi-core architectures," *Proceedings of the 2015 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, 2015.

[8] A. Sailer, S. Schmidhuber, L. Deubzer, M. Alfranseder, M. Mucha, and J. Mottok, "Optimizing the task allocation step for multi-core processors within autosar," *International Conference on Applied Electronics*, 2013.

[9] Amalthea-project, http://amalthea-project.org, Tech. Rep., last visited: 02/02/2015.

[10] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP Completeness*. W. H. Freeman and Company, 1979.

[11] S. Kirkpatrick, C. D. Gelatt Jr, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220 (4598), pp. 533–549, 1983.

[12] F. Glover, "Future paths for integer programming and links to artificial intelligence," *Computers and Operations Research*, vol. 13 (5), 1986.

[13] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Professional, 1989.

[14] M. Zhang and Z. Gu, "Optimization issues in mapping autosar components to distributed multithreaded implementations," *22nd International Symposium on Rapid System Prototyping (RSP)*, pp. 23 – 29, 2011.

[15] B. Miramond and J.-M. Delosme, "Decision guide environment for design space exploration," 2005.

[16] 32-bit TriCore Microcontroller, infineon, http://www.infineon.com/, Tech. Rep.

[17] M. Krause, O. Bringmann, A. Hergenhan, G. Tabanoglu, and W. Rosentiel, "Timing simulation of interconnected autosar software-components," *Design, Automation & Test in Europe, 2007.*, vol. 1, no. 6, pp. 16–20, 2007.