# Task and Memory Mapping of Large Size Embedded Applications over NUMA architecture*

Alessandro Druetto[1], Enrico Bini[1], Andrea Grosso[1]
Stefano Puri[2], Silvio Bacci[2], Marco Di Natale[2,3], Francesco Paladino[3]
[1]University of Turin [2]Huawei Research Center, Pisa [3]Scuola Superiore Sant'Anna, Pisa
Italy

## ABSTRACT

Multicore architectures provide the increased performance required by modern embedded real-time systems. Most platforms exhibit a non-uniform memory access (NUMA). In NUMA, memory banks with different access time can be explicitly addressed. Such an architecture, however, is challenging predictability given the significant impact of the allocation of variables on the execution times.

At software level, real-world embedded applications (e.g. automotive) are composed by thousands of functions often communicating through shared variables stored in memory, with a variable access time because of NUMA.

This paper addresses the mapping of complex embedded applications onto NUMA multicore architectures. The developed problem formulation offers a solution to the following problems: (i) allocating variables (called *labels* in the automotive context) over memories of different characteristics, (ii) mapping functionalities (called *runnables*) onto CPUs, (iii) creating OS tasks from runnables, and (iv) assigning priorities to tasks. Our developed implementation is capable to handle an application composed by 1K+ runnables, all sharing 10K+ labels and finds a solution in at most 3 minutes on a standard laptop, enabling interactive design space exploration.

**ACM Reference Format:**
Alessandro Druetto[1], Enrico Bini[1], Andrea Grosso[1] and Stefano Puri[2], Silvio Bacci[2], Marco Di Natale[2,3], Francesco Paladino[3]. 2023. Task and Memory Mapping of Large Size Embedded Applications over NUMA architecture. In *The 31st International Conference on Real-Time Networks and Systems (RTNS 2023), June 07–08, 2023, Dortmund, Germany.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3575757.3593650

## 1 INTRODUCTION

In many embedded systems, including most of automotive real-time controls, the fundamental problems for designers are (i) the definition of the task model from the set of functions that need to be executed, (ii) the allocation of those tasks to the available CPUs, and (iii) the mapping of data onto memory, including all variables shared among functions.

In automotive systems, the application definition is in most cases formalized by the AUTOSAR standard, in which a system is defined as a collection of components. The internal behavior of each component consists of a set of runnables (functions) activated in response to events. In this context, the problem input consists of a set of runnables to be executed according to some specified event (e.g. periodic, or upon the completion of another one, or upon receiving some input, or call request). A set of tasks needs to be defined to execute these runnables. These tasks need to be allocated to CPUs, and the data (including the program and communication variables) need to be allocated in memory.

Solving this problem is not easy and is key to achieve good performance of the application. It is the concern of most designers how NUMA architectures can result in a large variation in the execution times if the memory allocation is not carefully managed.

The problem has been investigated along several lines by the research community. However, the delivered performance and insights of current methodologies accounting for the memory allocation are not fully satisfactory, as described next.

### 1.1 Related works

A large number of previous works makes use of stochastic optimization techniques. McLean et al. [29] use a Simulated Annealing algorithm to assign tasks to CPUs and to generate a static schedule of tasks per CPU that is compatible with the communication pattern. An algorithm based on Simulated Annealing was proposed in [14] to partition non-independent tasks over a multicore platform and assign them a priority value. Similar to our case, the objective of the mapping is to maximize the robustness of the application against execution overruns, computed as the time units the execution time of each task can be inflated while still retaining the schedulability of the whole taskset. Genetic algorithms (GAs) were also used to map tasks over heterogeneous processors [3], though ignoring the placement of shared labels. Among the meta-heuristics, Systematic Memory-based Simulated Annealing was also used to partition AUTOSAR applications [13]. In the context of the AMALTHEA Projects [40], GAs were also proposed to solve the mapping problem [10]. Finally, Bouaziz et al. [7] proposed a Multi-Objective Evolutionary Algorithm to find the Pareto front of the runnables to tasks mapping, whereas Ferrandi et al. proposed an ant-colony approach to the problem of mapping both tasks and communication messages over an heterogeneous architecture [16]. In all of these papers, not only the optimization engine is different. Also, the impact of label allocation on the execution time of runnables is either modeled as a generic communication cost to be minimized (with an extremely simplified representation of the

*Patent pending with World Intellectual Property Organization (WIPO).


*RTNS 2023, June 07–08, 2023, Dortmund, Germany*

ACM ISBN 978-1-4503-9983-8/23/06…$15.00
https://doi.org/10.1145/3575757.3593650

memory accesses overheads and their dependency on the memory structure), or ignored altogether.

Based on past experience and our own, we claim that *stochastic optimization methods are in general unfit for the whole mapping problem*, since it is very difficult to select the right set of transitions (or mutation/crossover) operators to escape from local optima. Consequently, the quality of the found solution is extremely difficult to evaluate. In fact, our Simulated Annealing (SA) alternate solver remained stuck at a local minimum despite executing for more than 20 hours (more details are reported in Section 8).

Other methods presented in the past operate in the context of specific architectures or programming paradigms. Kobayashi et al. [22] propose the Model-Based Parallelizer (MBP), which maps C applications generated by Simulink's Embedded Coder onto Kalray's MPPA2-256, which is composed by 16 clusters of 16 cores. Becker et al. [4] propose to partition memories into banks and to schedule accesses to the same bank at different instants. Pazzaglia et al. [34] implements a partitioning scheme that includes memory allocation using the Logical Execution Time (LET) paradigm.

Methods that use constraint programming have been proposed, without exploring the complex dimension of the mapping of the variables to memory. Perret et al. [35] proposed a constraint programming approach to map a large application over a massively parallel architecture. An ILP formulation of the runnable-to-task-to-core mapping was proposed by [36]. However, the addressed use case was two orders of magnitude smaller than our target automotive application. Optimal partitioning and priority assignment using mathematical optimization is also proposed by Zhao and Zeng [42] and by Casini et al. [8].

Methods to adapt the application partitioning to dynamic workload do exist [33]. Fernandez et al. [15] identified the "cyclic dependency" between the execution cycles of tasks and their partitioning (later represented in Figure 3) as one of the main challenges. They proposed to break this dependency by establishing ties between tasks of varying strength. However, none of these works explored the mapping of labels with awareness of NUMA.

The closest contribution to ours is probably [20], in which the partitioning of automotive applications is addressed by (i) allocating runnables into tasks based on their activation pattern, (ii) creating DAGs based on the shared labels, and (iii) finally mapping DAGs onto the CPUs. Performing such a grouping of runnables based on the activation pattern, however, does not necessarily lead to a lower resource utilization. Also, mapping DAGs injects an unnecessary level of complexity which prevents the applicability to large applications.

Commercially, there are patents that cover some of the steps that are needed for optimization, but in very general terms and not with a realistic modeling of the memory costs. The patents apply to a specific method for allocating tasks to cores, which does not control over the use of the memory resources (and therefore the actual runnable execution time e.g. [32] and [18]). Other methods consider the optimization of the memory allocation of data and code (e.g. [28] and [25]) by assuming a given assignment of runnables to tasks and tasks to CPUs. Instead, the originality of our contribution is in the capacity to address all dimensions of the mapping (both runnables and labels) and to achieve superior results with a run-time orders of magnitude smaller than existing meta-heuristic-based approaches.

*Contribution of the paper.* Summarizing, the large majority of related works have ignored the impact of the placement of shared variables over the NUMA architectures. The few works which have considered this additional dimension, have proposed a GA formulation. GAs, however, ignore the peculiarities of the mapping of embedded applications and then hinder a full understanding of the role played by the many available tuning parameters of the model. Hence, to best of our knowledge, our ILP approach is the first one proposing a tractable problem formulation that wholly addresses the mapping of the runnables over the available cores, the mapping of the labels over the NUMA architecture, and the aggregation of runnables into tasks. Compared to GA formulations, our approach has the advantage of being computationally more efficient and more transparent to the designer, and it enables interactive design space exploration through the linear constraints and cost which can be tuned at design time.

## 2 SYSTEM MODEL

AUTOSAR (AUTomotive Open System ARchitecture)[1] is a standardized software architecture for the definition of automotive components and for providing the foundation platform for their execution [17]. The essential element of AUTOSAR that is relevant for our problem is the concept of *runnables*, that are functions to be executed in response to events. For the sake of our work, we are interested in capturing the nature of these activation events, assumed as periodic or sporadic. AUTOSAR runnables communicate by means of data ports or by client-server interactions. In both cases, memory locations shall be identified (in a stage called RTE, or Run-Time Environment generation) to store the values communicated over the ports or the arguments of the call.

In the AMALTHEA Projects (Model Based Open Source Development Environment for Automotive Multi Core Systems) and its follow-up APP4MC [1], the problem is abstracted by analyzing the results of the RTE generation stage directly. The model contains the elements that provide the minimal level of details necessary to setup a partitioning problem [40]. The App4MC platform was used to introduce the model of the 2017 WATERS challenge [23], which is used as our reference use case.

### 2.1 Hardware model

The AMALTHEA/App4MC hardware model is depicted in Figure 1. It provides the key features with an impact on timing analysis, without delving into too fine-grained details, which have little impact on the mapping problem. According to this view:

- A set of the identical CPUs, denoted here by $\mathcal{M}$, is available for processing instructions;
- Each CPU $k \in \mathcal{M}$ has a directly connected *Local RAM (LRAM)* of size $S_k^{\text{cpu}}$, which can be accessed at higher speed;
- A *Generic RAM (GRAM)* is available. We assume its size is large enough to accommodate data as needed;
- A crossbar switch enables all CPUs to access both the GRAM and the LRAM of the others with dedicated virtual channels;

[1] https://www.autosar.org/

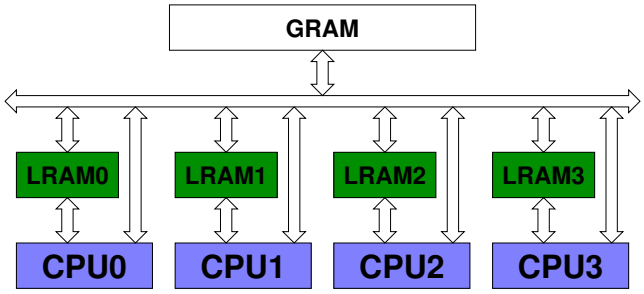


**Figure 1: Abstract hardware model.**

| | |
|---|---|
| Number of labels in $\mathcal{L}$ | 10000 |
| Total memory of labels [bytes] | 27363 |
| Number of runnables in $\mathcal{N}$ | 1250 |
| Number of tasks in $\mathcal{T}$ | 21 |
| Number of different release patterns | 19 |
| Number of accesses by runnables to any label | 15255 |

**Table 1: Key data of an automotive reference application [23].**

- All LRAMs and the GRAM are mapped to a unique address space, making them accessible from any CPU.

## 2.2 Software model

The software model used in AMALTHEA, which is also relevant to our purposes, includes the following terms:

- A *label* is a data element used by the application code; it has a type, which determines its memory size. The set of labels is denoted by $\mathcal{L}$ and $s_\ell$ is the size of label $\ell \in \mathcal{L}$.
- A *runnable* is a function implemented by sequential code. The set of runnables is denoted by $\mathcal{N}$. A runnable may read or write labels with a given frequency. Communication between runnables is implemented by writing/reading shared labels. As illustrated later in Section 4, a subset of labels $\mathcal{L}_i \subseteq \mathcal{L}$ is attached to the runnable $i \in \mathcal{N}$.
- $T_i$ denotes the *period* or minimum interarrival time of runnable $i$. Later, we may use the notation $f_i = 1/T_i$ to denote the maximum *frequency* of activation.
- A *task* corresponds to the operating system notion of thread. Its code corresponds to a sequence of runnables invoked sequentially. The set of tasks is denoted by $\mathcal{T}$.
- Other notions will be used in the paper such as the *gain* $g_{i,\ell}$ for runnable $i$ to have fast access to label $\ell$ or the *execution cycles* $C_i^0$ of runnable $i$. However, we postpone their precise definitions to the context when they will be needed (Sections 4 and 5, respectively)

Table 1 reports the size of the 2017 WATERS challenge [23] embedded application, which we extensively use in this paper.

Finally, tasks execute on statically assigned CPU (partitioned scheduling) and are scheduled by Fixed Priority.

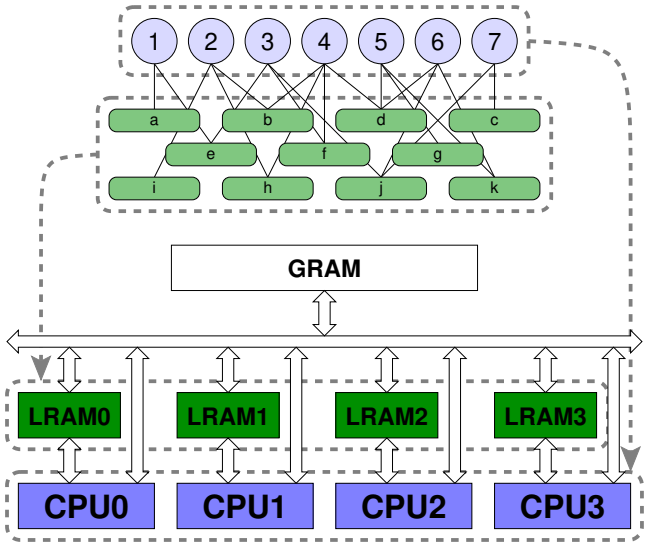


**Figure 2: The mapping problem. Runnables are azure circles ("1", "2", ...), labels are mint green rounded boxes ("a", "b", ...). The mapping (of runnables to CPUs and labels to LRAMs) is represented by thick dashed gray arrows.**

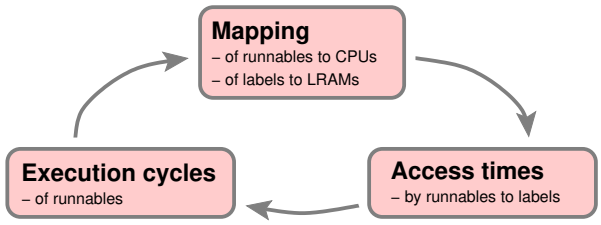


**Figure 3: The cyclic dependency of the mapping problem.**

## 3 THE PROBLEM

The problem addressed in this paper is the mapping of an application modeled by runnables $\mathcal{N}$ and labels $\mathcal{L}$ over the CPUs $\mathcal{M}$ and their associated LRAMs, respectively (as depicted in Figure 2.)

The main difficulty of the problem is due to the accesses that runnables make to labels. Depending on the mapping, these accesses may happen either locally (from a CPU to its directly connected LRAM, such as LRAM2 from CPU2 in Figure 1) or remotely. Since access times vary by one order of magnitude (please refer to Table 2), they do affect the overall execution cycles of runnables. In turn, the execution cycles of runnables do affect the mapping as any knapsack problem is affected by the size of the items to be packed. Such a cyclic dependency is represented in Figure 3. Finally, the size of real-world problem (please refer to Table 1) is about two orders of magnitude above the size of tractable problems of this kind.

Hence, we decompose the mapping in the following stages.

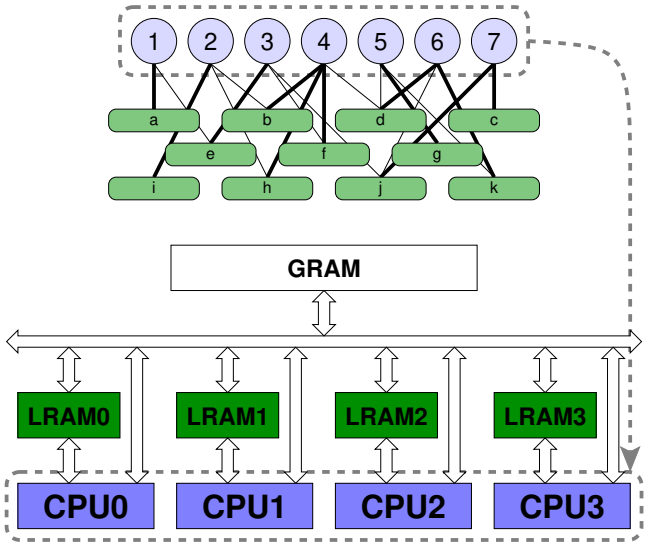


**Figure 4: In our methodology: (i) labels are first bound to the runnable which benefits the most (represented by a thick link), then (ii) runnables only are mapped to CPU, the bound labels will follow to the linked LRAM.**

(1) First, we address the problem of binding labels to runnables (in Section 4). This is the key enabler of the significant reduction in complexity. In Figure 4, labels bound to a runnable are represented by a thick link.
(2) Then runnables are mapped to CPUs (Section 5). As illustrated in Figure 4, every runnable carries the bound labels which are then implicitly mapped to the corresponding LRAM.
(3) Finally, runnables are assigned to tasks (Section 6) and, then tasks are assigned a priority (Section 7).

As it will be illustrated in greater details in the next section, the guiding principles that drive all optimization stages are

- the **minimization of the resource utilization**, and
- the **maximization of the slack** so that further upgrades or extensions may be accommodated more easily.

Without loss of generality, we identify the elements in any set $\mathcal{N}$ by the integers $1, \ldots, |\mathcal{N}|$. Also, to lighten the presentation, we use the same notation of any set $\mathcal{N}$ to denote the number of elements as well. In short, we consider correct to write $\mathcal{N} = \{1, 2, \ldots, \mathcal{N}\}$.

## 4 BINDING LABELS TO RUNNABLES

The key phase that makes the overall methodology feasible for the large scale automotive use case, is the binding of labels to runnables. In fact, given the size of realistic applications (of 1K+ runnables and 10K+ labels, please refer to Table 1 for details), a unique ILP formulation for the joint mapping of the labels and runnables is not tractable with the computing capacity available at time of writing. Hence, we bind labels to runnables (as represented by thick black links between runnables and labels in Figure 4). When a runnable $i$ is mapped to CPU $k$, then all the labels $\mathcal{L}_i$ bound to it are mapped to the LRAM directly linked to CPU $k$.

We formulate the binding problem as follows. For each pair $(i, \ell) \in \mathcal{N} \times \mathcal{L}$,

- if the label $\ell$ is used by the runnable $i$, we define the *gain* $g_{i,\ell}$ as the saved execution cycles by one invocation of the runnable $i$ when the label $\ell$ is allocated to the LRAM linked to the CPU where the runnable $i$ is mapped,
- we set $g_{i,\ell} = 0$ if the label $\ell$ is not used by runnable $i$.

The gain $g_{i,\ell}$ is expressed in clock cycles. Its calculation depends on many factors: type of access, size $s_\ell$ of $\ell$, frequency of access, etc. Later, in Section 8.1, we illustrate the gain models used in the experiments. We remark that our proposed methodology is independent of such a choice.

*Variables.* For the purpose of partitioning labels among runnables, we introduce the following variables

$$i \in \mathcal{N}, \ell \in \mathcal{L}, \quad x_{i,\ell} = \begin{cases} 1 & \text{label } \ell \text{ bound to runnable } i \\ 0 & \text{otherwise,} \end{cases} \tag{1}$$

and we define the partition of labels by

$$i \in \mathcal{N}, \qquad \mathcal{L}_i = \{\ell \in \mathcal{L} : x_{i,\ell} = 1\}.$$

*Constraints.* If we denote by $S_i$ the amount (unknown) of LRAM assigned to labels in $\mathcal{L}_i$, then the following constraint

$$i \in \mathcal{N}, \quad \sum_{\ell \in \mathcal{L}} s_\ell x_{i,\ell} \leq S_i \tag{2}$$

ensures that the total memory local to the runnable $i$ is not exceeded, while the next one

$$\sum_{i \in \mathcal{N}} S_i \leq \sum_{k \in \mathcal{M}} S_k^{\text{cpu}} \tag{3}$$

is needed not to exceed the total LRAM. If needed, our formulation can also include a constraint on the maximum amount of memory $S_i$ needed by runnable $i$.

Finally, it is certainly needed to assign a label to at most one runnable, that is

$$\ell \in \mathcal{L}, \quad \sum_{i \in \mathcal{N}} x_{i,\ell} \leq 1. \tag{4}$$

Notice that the subsets $\mathcal{L}_i$ are not a partition (that is Equation (4) is not written with the "=" sign) as there may be some labels that are not bound to any runnable.

*Goal function.* The natural aim of the binding is to minimize the resource usage by runnables. In fact, wherever every runnable $i$ is mapped, we are certain that **accesses to labels in $\mathcal{L}_i$ are through local links**. Since each runnable executes with frequency $f_i$, the metric to be maximized is

$$\sum_{i \in \mathcal{N}} f_i \sum_{\ell \in \mathcal{L}} g_{i,\ell}\, x_{i,\ell}. \tag{5}$$

The rationale of the cost of Eq. (5) is to bind label $\ell$ to the runnable $i$ that can benefit the most in terms of saving CPU utilization.

*Size of the problem.* For this problem, the number of variables $x_{i,\ell}$ (1) is $\mathcal{N} \times \mathcal{L}$, while the number of constraints is $\mathcal{N} + \mathcal{L} + 1$ following from (2)–(4).

### 4.1 Polynomial-time algorithms

If the overall amount of LRAM is sufficient to store all labels, then the constraint of Eq. (3) is never active. This means that the only active constraint remains (4) and that the optimal solution is:

$$\ell \in \mathcal{L}, \quad x_{i,\ell} = 1 \;\Leftrightarrow\; i = \operatorname*{argmax}_{j \in \mathcal{N}} \{f_j\, g_{j,\ell}\} \tag{6}$$

which is found in $O(\mathcal{N} \times \mathcal{L})$ time. For such a solution, every label $\ell$ brings the maximum saving of resource usage, which is $G_\ell = \max_j \{f_j\, g_{j,\ell}\}$.

If instead the constraint of (3) is active, then the problem becomes the knapsack problem in which "*a person is planning a hike and has decided not to carry more than 70 lb of different items, such as bed roll, Geiger counters (these days), cans of food, etc.*" [11]. The continuous relaxation of this problem is solved exactly in $O(\mathcal{L}\ \log(\mathcal{L}))$ time as suggested by Dantzig [11] and described below:

(1) Labels are sorted by decreasing *gain density* as follows

$$\ell, \ell' \in \mathcal{L}, \ell < \ell', \quad {}^{G_\ell}/_{s_\ell} \geq {}^{G_{\ell'}}/_{s_{\ell'}}. \tag{7}$$

(2) Labels are selected, and assigned to the runnable of (6), following the ordering of (7) until the memory capacity constraint (3) is not violated.
(3) Let $z \in \mathcal{L}$ be the *critical item*, which is the first label that, according to the ordering of (7), violates the capacity constraint of Eq. (3).

If we set

$$S^{\text{slack}} = \sum_{k \in \mathcal{M}} S_k^{\text{cpu}} - \sum_{\ell=1}^{z-1} s_\ell,$$

which is the remaining memory capacity after allocating the first $z-1$ labels, then the optimum is found by taking a fraction ${}^{S^{\text{slack}}}/_{s_z}$ of the label $z$. The value of the maximized metric of Eq. (5) for such an optimal solution is

$$\sum_{\ell=1}^{z-1} G_\ell + \frac{S^{\text{slack}}}{s_z} G_z.$$

In our polynomial time greedy algorithm:

(1) we drop the critical item "label $z$" and bind labels up to the one in position $z-1$ in the ordering of (7), and
(2) we fill up the remaining capacity $S^{\text{slack}}$ with any label that fits.

*Distance to optimality.* The maximum penalty of this solution is

$$\frac{S^{\text{slack}}}{s_z} G_z \leq G_z \leq \max_{\ell \in \mathcal{L}} G_\ell = \max_{\ell \in \mathcal{L}, i \in \mathcal{N}} \{f_i\, g_{i,\ell}\}.$$

If the granularity $f_i g_{i,\ell} \ll 1$, as in our use case [23], the penalty is negligible. Our experiments confirm the quality of the polynomial greedy algorithm, since its solution and the optimal one are nearly indistinguishable.

## 5 MAPPING RUNNABLES TO CPUS

The mapping of the runnables in $\mathcal{N}$ over the available CPUs is formalized as a Binary ILP (BILP) problem.

*Variables.* We model the mapping over the CPUs by $\mathcal{N} \times \mathcal{M}$ variables $y_{i,k}$ with the following interpretation

$$i \in \mathcal{N}, k \in \mathcal{M}, \quad y_{i,k} = \begin{cases} 1 & \text{runnable } i \text{ mapped to CPU } k \\ 0 & \text{otherwise.} \end{cases} \tag{8}$$

We remind that if $y_{i,k} = 1$, then the bound labels in $\mathcal{L}_i$ will be mapped to the LRAM associated to CPU $k$.

The distinguishing feature that needs to be captured in the problem of mapping runnables to CPUs is whether or not any pair of runnables is mapped onto the same core. In fact, if runnable $i$ is on the same CPU as runnable $j$, it may save processing time if it uses labels in $\mathcal{L}_j$. Motivated by this observation, we add the following variables to the problem formulation

$$\begin{matrix} i, j \in \mathcal{N}, \\ i < j, \end{matrix} \quad x_{i,j}^{\text{same}} = \begin{cases} 1 & \text{runnables } i \text{ and } j \text{ on same CPU} \\ 0 & \text{otherwise.} \end{cases} \tag{9}$$

Finally, an additional continuous *slack* variable $z$ is added. Such a variable represents the "extensibility" of software for future updates and it is going to be maximized. A negative value of $z$ indicates an infeasible design.

*Constraints.* If runnables $i$ and $j$ are bound to the same CPU $k$, then the corresponding variable $x_{i,j}^{\text{same}}$ must be equal to one:

$$i, j \in \mathcal{N}, i < j, \quad x_{i,j}^{\text{same}} = \max_{k \in \mathcal{M}} (y_{i,k} + y_{j,k}) - 1, \tag{10}$$

which can also be written as linear constraint, by adding extra variables for each $k \in \mathcal{M}$.

Each runnable is mapped over one CPU only, that is

$$i \in \mathcal{N}, \qquad \sum_{k \in \mathcal{M}} y_{i,k} = 1. \tag{11}$$

Notice that if a runnable must be necessarily mapped to some specific CPU (e.g. some of the available CPUs offer some HW features or accelerator which are necessary to the runnable), it is possible to encode such a constraint by setting $y_{i,k} = 1$.

The variables $x_{i,j}^{\text{same}}$ as defined by (9) clearly imply an equivalence relation. Hence, we enforce the following properties

- **reflexivity** is enforced implicitly by omitting the variables $x_{i,i}^{\text{same}}$, as it would always be $x_{i,i}^{\text{same}} = 1$
- **symmetry** is enforced implicitly by having only one variable $x_{i,j}^{\text{same}}$ for both ordered pairs $(i, j)$ and $(j, i)$. To have a more convenient notation, we may be using $x_{i,j}^{\text{same}}$ with $i > j$. When this happens, we mean $x_{j,i}^{\text{same}}$.
- **transitivity** that is

$$(x_{i,j}^{\text{same}} = 1) \wedge (x_{j,\ell}^{\text{same}} = 1) \Rightarrow (x_{i,\ell}^{\text{same}} = 1).$$

Transitivity is not explicitly enforced because implied by (10) and (11). In fact, if $x_{i,j}^{\text{same}} = 1$, from (10) it must exist $k_1 \in \mathcal{M}$ such that $y_{i,k_1} = y_{j,k_1} = 1$. For the same reason, $x_{j,\ell}^{\text{same}} = 1$ implies that it must exist $k_2 \in \mathcal{M}$ such that $y_{j,k_2} = y_{\ell,k_2} = 1$. Constraint (11) implies that $k_1 = k_2$ and then from $y_{i,k_1} = y_{\ell,k_1} = 1$ we have $x_{i,\ell}^{\text{same}} = 1$.

Furthermore, applications may require two or more runnables to be scheduled together over the same CPU. For example, in the automotive AUTOSAR standard, runnables may belong to Software Components (SWCs), which need to be mapped to the same

core. This can be easily encoded constraining $x^{\text{same}}_{i,j} = 1$ for all the runnables $i$ and $j$ belonging to the same SWC.

Before formulating the constraints on the CPU capacity, let us introduce the following notation.

- $C^0_i$ denotes the execution cycles of runnable $i$, assuming that:
  - all labels in $\mathcal{L}_i$, bound to runnable $i$ as described in Section 4, are stored in LRAM and then enjoy a faster access
  - all other labels are stored in GRAM.
- $\Delta C_{i,j}$ denotes the execution cycles saved by one invocation of runnable $i$ if runnable $j$ executes over the same CPU. $\Delta C_{i,j}$ is non-zero, if runnable $i$ happens to use any label in $\mathcal{L}_j$. $\Delta C_{i,j}$ is written as function of the gains $g_{i,\ell}$ introduced in Section 4, as follows

$$i \in \mathcal{N}, \qquad \Delta C_{i,j} = \sum_{\ell \in \mathcal{L}_j} g_{i,\ell}. \tag{12}$$

- the variable $C_{i,k} \geq 0$ represents the number of execution cycles required by runnable $i$ over CPU $k$, that is

$$k \in \mathcal{M}, \qquad C_{i,k} \geq C^0_i \, y_{i,k} - \sum_{j \in \mathcal{N}, j \neq i} \Delta C_{i,j} \, x^{\text{same}}_{i,j}. \tag{13}$$

  It is worth noting that, if the solver does not map runnable $i$ to CPU $k$, the value of $C_{i,k}$ is set to zero.

As stated in Section 4, when runnable $i$ is partitioned it also carries an amount of needed local memory $S_i$. The constraint of limited size of the local memory is formulated as

$$k \in \mathcal{M}, \qquad \frac{1}{S^{\text{cpu}}_k} \sum_{i \in \mathcal{N}} y_{i,k} \, S_i \leq 1 - \alpha^{\text{mem}} z, \tag{14}$$

which requires to account for some slack in the memory constraint on CPU $k$. Analogously, the constraint on the CPU capacity is

$$k \in \mathcal{M}, \qquad \sum_{i \in \mathcal{N}} f_i \, C_{i,k} \leq 1 - \alpha^{\text{cpu}} z. \tag{15}$$

The weights $\alpha^{\text{mem}}$ and $\alpha^{\text{cpu}}$ represent the relevance of the slack in each constraint and can be freely chosen by the designer. A large value of $\alpha^{\text{mem}}$ or $\alpha^{\text{cpu}}$ encodes the goal of having much slack in the constraint, while a value of zero informs the solver that the constraint can also hold tightly.

*Goal function.* The goal of the design is to maximize the "extensibility" of software for future updates, that is

$$\text{maximize } z. \tag{16}$$

If the optimal $z^*$ found is negative then the problem is not feasible. If $z^* \geq (1-U_{\text{LL}})/\alpha_{\text{cpu}}$, with $U_{\text{LL}}$ equal to the Liu and Layland [26] utilization upper bound $\log 2 \approx 0.693$ the problem is feasible. Otherwise, schedulability is ensured by the next step of the design (the assignment of priority described in Section 7). Notice that the proposed design goal generalizes the typical optimization goal borrowed from the literature [34].

*Size of the problem.* Summarizing, the total number of variables is $2(\mathcal{N}\times\mathcal{M})$, counting $y_{i,k}$ of Eq. (8) and $C_{i,k}$ of Eq. (13). The number of constraints is $O(\mathcal{N}^2)$, dominated by Eq. (10). In real-world scenarios, $\mathcal{N} \approx 1000$ as shown in Table 1, making the number of constraints in the order of millions. When such a problem becomes intractable, a different approach, illustrated next, is required.

## 5.1 Hierarchical Clustering

To mitigate the issues due to the size of the problem, we borrow the *hierarchical clustering* from the literature [31]. In hierarchical clustering, runnables are aggregated into clusters of tunable size. Then the mapping described earlier in Section 5 is applied to the fewer clusters, rather than to all runnables. An advantage of hierarchical clustering is that the size of clusters can be set by the designer to trade accuracy vs. tractability. Also, hierarchical clustering is particularly well suited for partitioning very many "small" items, as in our use case.

**Algorithm 1** Hierarchical Clustering

```
1:  function HC(N, U)                          ▷ runnables, their utilizations
2:      clusters ← {{i} : i ∈ N}                          ▷ initialize clusters
3:      utils ← U                              ▷ utilization of singleton clusters
4:      mergeTree ← [ ]                          ▷ tracking of cluster merges
5:      for n from |N| − 1 to 1 do                ▷ need |N| − 1 merges
6:          c_min ← argmin(utils)                ▷ min cluster utilization
7:          U_avg ← (sum(utils) − utils[c_min])/n
8:          maxGain ← −1
9:          for c in clusters \ {c_min} do
10:             if utils[c] > U_avg then continue
11:             end if
12:             g ← utilGain(c_min, c)
13:             if g > maxGain then
14:                 maxGain ← g
15:                 c_best ← c
16:             end if
17:         end for
18:         clusters ← clusters \ {c_min} \ {c_best}
19:         c_new ← c_min ∪ c_best                    ▷ merge clusters
20:         clusters ← clusters ∪ {c_new}             ▷ add new cluster
21:         utils[c_new] ← utils[c_min] + utils[c_best] − maxGain   ▷
    update utilization of new cluster considering gain.
22:         mergeTree[n] ← clusters              ▷ save found i clusters
23:     end for
24:     return mergeTree
25: end function
```

The full procedure of hierarchical clustering is outlined in Algorithm 1. The initial clustering set clusters is initialized (line 2) by considering all runnables as singleton clusters. The array utils contains the CPU utilization of all clusters (line 3) and for any cluster $c$ we denote its utilization of utils$[c]$. The array of clusters mergeTree (initialized at line 4) is meant to contain the set of all found clusters. More specifically, mergeTree$[n]$ reports the solution of how all runnables in $\mathcal{N}$ are partitioned in $n$ clusters.

The loop from line 5 to line 23 picks a pair of clusters and merges them until a unique cluster with all runnables is created. At every iteration, our algorithm first picks the cluster $c_{\text{min}}$ with the lowest utilization (line 6). The second cluster $c_{\text{best}}$ to be merged with $c_{\text{min}}$ satisfies two properties:

(1) it has utilization no greater than the average utilization of clusters (enforced by the condition at line 10), and
(2) it has the highest gain maxGain if paired with $c_{\text{min}}$ (enforced by the condition at line 13).

The choice of this merging rule was proven to keep a very balanced utilization of the clusters, while leading towards the best possible decrease in utilization for all merge operations.

Once the joining pair of clusters is chosen, they are removed from the set (line 18), all runnables that they contain are merged in a new cluster $c_{\text{new}}$ (at line 19), which is then added to the set (line 20). At line 21, the utilization of the new cluster $c_{\text{new}}$ is computed accounting for the utilization of each of the merged clusters and the gain maxGain they have from being together.

An immediate advantage of hierarchical clustering is the availability of the entire hierarchical tree recording the history of the merges. Such a tree enables the designer to choose the desired level of granularity. Once a level of granularity has been chosen, the newly formed clusters are considered as "runnables" for the model described in Section 5 and the problem of mapping runnables to CPUs is solved with a significantly smaller set $\mathcal{N}$.

## 6 AGGREGATION OF RUNNABLES INTO TASKS

Runnables, we remind, are equivalent to functions to be properly invoked. The division of software in runnables responds to specifications and principles at application design level. At the lower OS level, instead, it may be infeasible (due to the potentially large number of runnables) and it is certainly inefficient (due to context switches) to dedicate an OS task to each runnable. For example, in the 2017 WATERS Challenge [23], there are 1250 runnables, but 21 tasks only executing them, as reported in Table 1. It is then necessary to establish criteria to aggregate runnables.

Our approach to aggregate runnables in tasks is applied after the optimal mapping of runnables (or clusters of runnables) to CPUs is performed as described in Section 5. We assume then to have a solution to the mapping represented by the variables $y_{i,k}$ and $x^{\text{same}}_{i,j}$ defined in (8) and (9), respectively. Let us now formalize the aggregation of runnables.

- The set of tasks is denoted by $\mathcal{T}$.
- The set of runnables in $\mathcal{N}$ to form task $t$ is denoted by $\mathcal{N}_t \subseteq \mathcal{N}$. The subsets in $\{\mathcal{N}_t\}_{t\in\mathcal{T}}$ form a partition of $\mathcal{N}$ that is every runnable $i \in \mathcal{N}$ belongs to one and only one subset $\mathcal{N}_t$.
- The *equivalence relation* $\sim$ over the pairs of runnables $\mathcal{N}\times\mathcal{N}$ encodes the aggregation of runnables. In our case, $i \sim j$ if "the two runnables $i$ and $j$ have the same release period and none of them self-suspends". We remark that our methodology works for any other choice $\sim$ providing the properties of equivalence relations.
- The runnables in $\mathcal{N}_t$ belonging to the same task $t$ are defined as follows

$$i, j \in \mathcal{N}_t \quad \Leftrightarrow \quad (i \sim j) \ \wedge \ x^{\text{same}}_{i,j} = 1 \tag{17}$$

  with $x^{\text{same}}_{i,j}$ being the variables representing the optimal mapping found in Section 5.

From our definition of $\sim$ above, it follows that

$$i \sim j \ \Rightarrow T_i = T_j, \tag{18}$$

meaning that two runnables with different period, cannot be aggregated together in the same task. Such assumption holds for the 2017 WATERS Challenge [23] and is recommended in software design. Still, having runnables with different period in the same task is possible. In such a case, the task implementation simulates the different periods by counting the invocations of each runnable and the execution pattern of the task becomes analogous to the multi-frame task model [30]. The analysis of this case, however, is left to future investigations due to the lack of space.

The definition of Equation (17) partitions runnables to tasks. We can now define the parameters of tasks, starting from the parameters of the runnables

- From (13), the execution cycles of task $t \in \mathcal{T}$ are

$$C_t = \sum_{i\in\mathcal{N}_t} \left( C^0_i - \sum_{j\in\mathcal{N}, x^{\text{same}}_{i,j}=1} \Delta C_{i,j} \right) \tag{19}$$

- Because of (18), all runnables of a task have the same period (or minimum interarrival time). Hence, $\forall t \in \mathcal{T}$, we set $T_t = T_i$, picking any $i \in \mathcal{N}_t$
- The deadline of task $t \in \mathcal{T}$ is

$$D_t = \min_{i\in\mathcal{N}_t} D_i.$$

- Finally, it is useful to introduce the partition of tasks over the CPUs in $\mathcal{M}$. We define

$$\mathcal{T}_k = \{t \in \mathcal{T} : i \in \mathcal{N}_t, y_{i,k} = 1\}.$$

  Notice that this is a good definition because if $y_{i,k} = 1$ for some runnable $i \in \mathcal{N}_t$, then $y_{j,k} = 1$ for all $j \in \mathcal{N}_t$.

**Algorithm 2** Robust Priority Assignment [12]

```
1:  function RPA(T_k)                          ▷ tasks mapped to CPU k
2:      pri ← lowest priority available
3:      T_unassigned ← T_k
4:      while T_unassigned not empty do
5:          z_best ← −∞
6:          for i in T_unassigned do
7:              T_hp ← T_unassigned \ {i}
8:              z_i ← slack of task i              ▷ RHS of Eq. (21)
9:              if z_i > z_best then               ▷ found a better task
10:                 z_best ← z_i
11:                 i_best ← i
12:             end if
13:         end for
14:         priority[i_best] ← pri
15:         pri ← next priority higher than pri
16:         T_unassigned ← T_unassigned \ {i_best}
17:     end while
18:     return priority
19: end function
```

## 7 ASSIGNING PRIORITIES TO TASKS

For each CPU $k$, the priorities of the tasks in $\mathcal{T}_k$ are assigned based on Robust Priority Assignment (RPA) [12], recalled in Algorithm 2. The only adaptation w.r.t. the original RPA is the computation of the maximum slack $z_i$ at line 8. Rather than using binary search as originally proposed [12], we borrow the *sensitivity analysis* [6] to find the exact expression of the per-task slack $z_i$. More, precisely,

from the exact schedulability condition [24] properly modified to account for the per-task weighted slack $z_i$

$$\exists t \in \mathcal{P}_i, \quad \frac{1}{t}\left(C_i + B_i + \sum_{j \in \mathcal{T}_{\text{hp}}} \left\lceil \frac{t}{T_j} \right\rceil C_j\right) \leq 1 - \alpha_i^{\text{sched}} z_i$$

with

$$\mathcal{P}_i = \{D_i\} \cup \{kT_j : j \in \mathcal{T}_{\text{hp}},\ 0 < kT_j < D_i,\ k \in \mathbb{N}\}, \tag{20}$$

we find

$$z_i \leq \max_{t \in \mathcal{P}_i} \frac{t - \left(C_i + B_i + \sum_{j \in \mathcal{T}_{\text{hp}}} \left\lceil \frac{t}{T_j} \right\rceil C_j\right)}{\alpha_i^{\text{sched}} t}, \tag{21}$$

which is the expression used at line 8 of Alg. 2 for computing $z_i$.

The blocking time $B_i$ in Eq. (21), is the time spent by task $i$ *waiting* for the execution of lower priority tasks on the same CPU or any task on other CPUs. This may happen because of:

- An attempt to access to a shared resource locked by:
  - a lower priority task within the same CPU, or
  - a task executing on a different CPU;
- The invocation of a blocking system call such as a remote procedure call (allowed by the AUTOSAR standard, for example).

Several protocols can be used to protect resources shared globally. In the case of the AUTOSAR standard, a discussion on their applicability to AUTOSAR and blocking times can be found in [39] and [41], where linear formulations for lock-based protocols that admit a bounded blocking time are presented. AUTOSAR requires that a task can be terminated at any time, even when waiting (in a spin lock) for a global resource. This can be in principle solved by using the protocols in [9] and [37]. We remark that, shared resource protocols for multiprocessor systems are outside the scope of this paper and that our methodology can integrate any protocol above.

*Complexity.* Algorithm 2 is pseudo-polynomial as the cardinality of $\mathcal{P}_i$ of Eq. (20). In our experiments, Algorithm 2 completed in a matter of tenths of a second.

# 8 EXPERIMENTS

## 8.1 The use case

Our experiments are based on the use case of an automotive application provided by the WATERS 2017 challenge [23], whose size was reported earlier in Table 1. Our methodology is relevant for those applications which make an intensive use of labels in memory. Applications with CPU-bound work and few memory operations do not particularly benefit by our approach since the impact of the mapping of labels is negligible.

In our experiments, we borrowed the values of the memory access times from the datasheet of TC39xx architectures. Table 2 reports the stall cycles to access a 32-bit word. In the table

- The cycles for read and write accesses are reported;
- Since write operations are buffered, stall cycles "5,3" mean:
  - 5 cycles for the first write of a 32-bit word,
  - 3 cycles for the next consecutive writes;
- The column "local CPU" reports the access time from a CPU to the directly connected to LRAM (direct accesses to GRAM are not possible as shown in the architecture of Figure 1);

| | local CPU | | remote CPU | |
|---|---|---|---|---|
| Memory type | read | write | read | write |
| LRAM | 0 | 0 | 7 | 5, 3 |
| GRAM | n.a. | n.a. | 7 | 5, 3 |

**Table 2: Stall cycles for memory accesses in TC39x. LRAM denotes: Data Scratchpad RAM (DSPR) and distributed Local Memory Units (dLMUs). GRAM denotes: Local Memory Units (LMUs)) and Default Application Memory (DAM). (source: Table 72 of User Manual [21])**

- The column "remote CPU" reports the time for accesses from a CPU to LRAM or GRAM which need to traverse the crossbar (shown as a horizontal arrow in Figure 1).

Given these characteristics, we model the gain $g_{i,\ell}$ of the cycles saved by one invocation of runnable $i$ when accessing the label $\ell$ in the directly connected LRAM, by

$$g_{i,\ell} = \begin{cases} 7 \times \mathsf{num}_i \times \lceil s_\ell/4 \rceil & \text{read accesses} \\ 5 + 3 \times (\lceil s_\ell/4 \rceil - 1) & \text{write accesses} \end{cases}$$

with

- $s_\ell$ denoting the size of the label $\ell$ in bytes
- $\lceil s_\ell/4 \rceil$ accounting for accesses made by 4-bytes words
- $\mathsf{num}_i$ denoting the number of reads by the one invocation of runnable $i$
- and the expression of the write cycles accounting for the different number of write cycles to the first word and the following ones in presence of write buffers.

The model of the gain $g_{i,\ell}$ assumes that a runnable writes to a label only once per invocation. This assumption originates from an inspection of the WATERS 2017 use case, which does not contain any information about the access statistics for the labels to be written. Clearly, our approach can account for labels to be written more than once whenever this information is available. Moreover, we underline that our approach is compatible with any model of the memory access times such as the one proposed in the WATERS 2017 challenge (which is 1 cycle to access to the directly linked LRAM, 9 cycles for other accesses) or others. In fact, the gains $g_{i,\ell}$ are fed as input to our problem.

To avoid trivial solutions (in which, for example, all runnables fit onto the same core or the mapping is infeasible), we scaled the execution cycles of all runnables such that:

- if no label is in LRAM, then the total utilization $\sum_{i \in \mathcal{N}} f_i C_i$ is equal to 2.110 (still fitting on 4 CPUs)
- if all runnables have their used labels in the directly linked LRAM, then the total utilization is equal to 1.479 (not fitting a single CPU)

These values allow exploring rich scenarios for the mapping over 4 identical CPUs, and set upper and lower bounds to the total utilization of any solution.

## 8.2 A Simulated Annealing approach

To evaluate the quality of the ILP solution, we implemented a simulated annealing (SA) [2] mapping optimizer as baseline. The choice upon a SA algorithm in spite of other meta-heuristics or genetic

algorithms is twofold: SA is a generally applicable and easy-to-implement stochastic approximation approach, and it is able to produce good solutions for an optimization problem even if the underlying structure of the problem is not obvious nor easily understandable. Moreover, SA algorithms in the past history [5] have outperformed the best known heuristics for several problems, while for other problems their performance was comparable to specialized heuristics finely-crafted to solve exactly those specific problems.

SA belongs to the category of *randomized optimization* techniques and aims at optimizing a given objective function by performing a sequence of random changes to the system configuration, generating at every iteration a new mapping solution. At each step, the new configuration is evaluated and retained if its performance is better than the previous solution. If the new solution is worse, it can still be conditionally accepted with a probability $P$ computed by

$$P = e^{\frac{-\Delta V}{T}} \tag{22}$$

which exponentially decreases with "temperature" $T$. This prevents being stuck in a local optimum. $\Delta V$ is the difference between the current and the new performance values.

The performance metric is the same as the ILP (maximization of the slack as in Eq. (16)). The constraints are not explicit, to let the algorithm explore unfeasible regions of the solution space and guarantee the reachability of any possible configuration. However, unfeasible configurations are penalized in the objective function, hence the SA moves towards feasible solutions in the end.

The random modifications to the system configuration are realized by 4 *transition operators*.

(1) *Task priority*: this operator picks a task of the system and sets its priority to a different value.
(2) *Runnable-to-task mapping*: this operator chooses a runnable of the system and moves it to another task. The runnable can be moved to an existing task, or a newly created one. If the task where the runnable was originally allocated contains no runnables after the transition, it is removed from the system.
(3) *Task-to-CPU mapping*: this operator picks a task of the system and moves it to a different CPU.
(4) *Label-to-memory mapping*: this operator chooses a label and places it to a different memory where it can fit according to the currently available space.

At every step, the algorithm randomly chooses the number of consecutive transition operators to apply to the current solution (between 1 and 3) and then randomly picks the operators. Applying more than one operator at a time helps the SA escape local optima during the search.

The algorithm was programmed to run with an initial temperature of 20000, a final temperature of 0.0001, a cooling rate of 0.94, a maximum number of temperature values (MAXNUMCHAINS) of 4000000, a maximum number of iterations for each temperature value (MAXTRY) of 400000, a maximum number of acceptable configurations for each temperature value (MAXCHANGE) of 20000 and a penalty multiplier 10 (in cost) for unfeasible configurations.

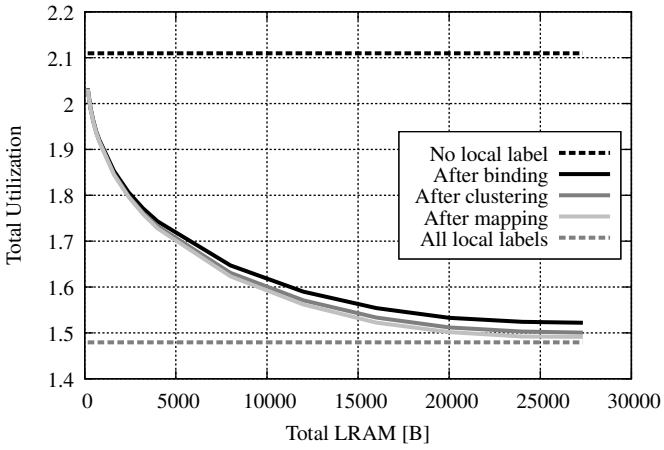


**Figure 5: Utilization as function of the allocated memory, with 40 clusters.**

## 8.3 Setup

The executions of the solver are performed over one of the cores available in the 11th Generation Intel Core i7-11700, 2.5 GHz desktop, with Linux kernel 5.15.0.

To ease the process of data extraction and analysis of the 2017 WATERS challenge, described in XML, we implemented the parsing process, the two polynomial algorithms to bind labels to runnables (Section 4.1) and to aggregate runnables in clusters (Section 5.1), and the pseudo-polynomial algorithm to assign priorities (described in Section 7) in `Python 3` [38]. The solver of the ILP problem to map clusters to CPUs of Section 5 is made with `COIN-OR Cbc` [27]. `Cbc` is one of the best open-source integer optimization solvers, and it is developed and maintained explicitly for research by the non-profit COIN-OR Foundation. The communication between the main Phyton code and solver is made by Python's library `Pyomo` [19]. The solver was invoked as a single thread.

We underline that we purposely targeted a not-so-performing implementation (e.g. Phyton, single-thread solver) to focus exclusively on the optimization problems and leave room to further performance improvements by those willing to make an industrial product out of our research prototype.

The weights to the slack $\alpha^{\text{mem}}$, $\alpha^{\text{cpu}}$, and $\alpha_i^{\text{sched}}$ of Equations (14), (15), and (21) are set as follows:

- $\alpha^{\text{mem}} = 0$ and $\alpha^{\text{cpu}} = 1$ meaning that we aim at balancing the total utilization among CPUs and
- $\alpha_i^{\text{sched}} = 1$ meaning that we equally weight all tasks.

We tried different weights, which did not highlight any different behavior.

## 8.4 Results

The experiment of Figure 5, made with 40 clusters, shows the impact of the amount of allocated LRAM onto the total utilization of the application. For reference, we also plot the upper and lower bounds to the utilization found as described in Section 8.1. The "After binding" plot corresponds to the total utilization of the whole application after labels are bound to runnables (as described in Section 4). Then,

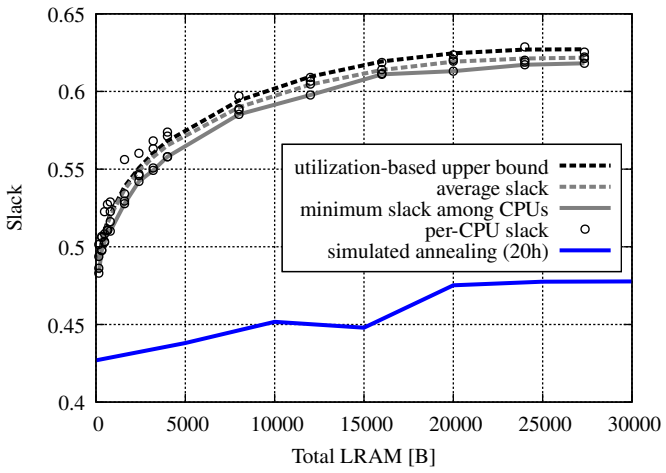


**Figure 6: Slack of the mapping, with 40 clusters.**

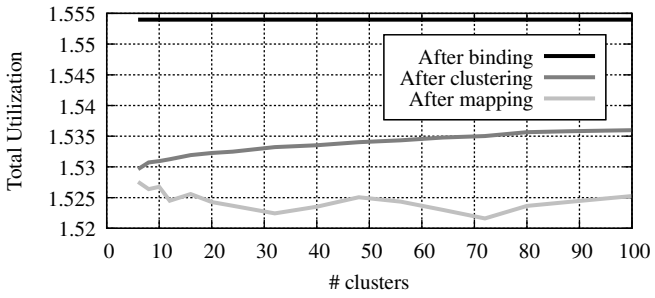


**Figure 7: Utilization as function of the number of clusters.**

"After clustering" accounts for the extra utilization savings achieved by hierarchical clustering (Section 5.1). And finally, "After mapping" is the utilization after the clusters are mapped to CPUs. As expected, the more LRAM is available for storing labels, the lower the final utilization it is. We also observe that a much steeper descent is achieved for the low values of LRAM. Our explanation is that storing in LRAM a very frequently used label has a greater impact on utilization than storing another one with the same size, but used less frequently.

Figure 6 shows the achieved slack. As the ILP optimzation targets the maximization of the minimum slack among CPUs, we observe that the per-CPU slacks are quite balanced. It is very striking to observe that SA, despite running about 1500 times longer than the proposed ILP approach, is always achieving significantly inferior results.

The impact of the number of clusters on the utilization (Figure 7) is as expected. As clusters get merged, their total utilization decreases, because the merged pair takes advantage of the commonly used labels. On the other hand, if clusters are too few, then the mapping has really little maneuvering margin to allocate clusters which are then very coarse grained. To our experience, a number between 20 and 40 demonstrated to be a good compromise when mapping over 4 CPUs.

| Phase | average [msec] | std deviation [msec] |
|---|---|---|
| Binding labels | 10.7 | 8.1 |
| Clustering runnables | 338.6 | 40.0 |
| Priority assignment | 17.4 | 12.1 |

**Table 3: Run-time of binding, clustering, and priority assignment.**

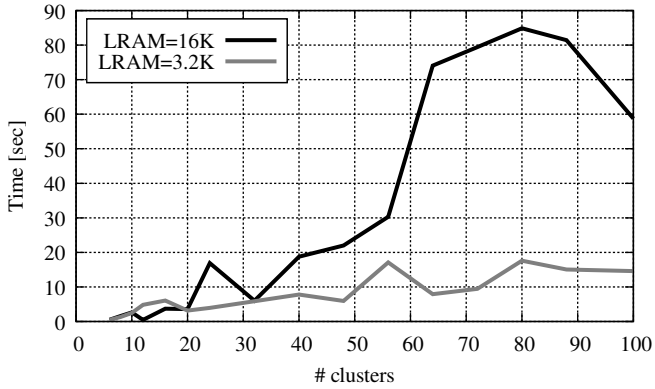


**Figure 8: Run-time of mapping of clusters.**

Let us now examine the run-time of the whole method. The run time taken by the binding (Section 4.1), the clustering (Section 5.1), and the priority assignment (Section 7) is negligible compared to the time taken to map clusters over CPUs. Table 3 reports the average and standard deviation of their run-times. Figure 8 shows the run-time of the mapping of clusters (of runnables). Not surprisingly, it grows rapidly with the number of clusters. We also observe that if the amount of allocated LRAM is smaller (3.2K instead of 16K) then the run-time is also smaller. This happens because if the LRAM is large, then there are more pairs of clusters with potential utilization savings by staying together. Our explanation for the observed decrease of the run time when the number of clusters approaches 100 is that for such a value, the clusters gets smaller and smaller. Hence, the pruning rules of the solver operate very effectively.

## 9 CONCLUSIONS

In this paper we have illustrated a whole methodology for mapping complex embedded software over NUMA architectures, exploiting the features of the different memory areas. The key innovation of our approach resides in the "binding" phase that reduces the complexity of the problem by orders of magnitude.

In the future, we may exploit the efficiency of our method by integrating it with other tools including measurement-based timing analysis tools. Also, we may be investigating the adaptation of the mapping in response to variations in the application features or in the processing capacity. Finally, a valuable direction of further investigation is the possibility to add end-to-end constraints, which are very typical of automotive applications.

## ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers of previous versions of the paper for suggesting the usage of RPA [12] and for fixing a few technical details.

## REFERENCES

[1] 2017. Eclipse APP4MC. Eclipse APP4MC Website, https://www.eclipse.org/app4mc/.

[2] Emile Aarts and Jan Korst. 1989. *Simulated Annealing and Boltzmann Machines*. Wiley & Sons.

[3] Adrian Alexandrescu, Ioan Agavriloaei, and Mitică Craus. 2011. A genetic algorithm for mapping tasks in heterogeneous computing systems. In *15th International Conference on System Theory, Control and Computing*. IEEE, 1–6.

[4] Matthias Becker, Dakshina Dasari, Borislav Nicolic, Benny Akesson, Vincent Nélis, and Thomas Nolte. 2016. Contention-free execution of automotive applications on a clustered many-core platform. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 14–24.

[5] Dimitris Bertsimas and John Tsitsiklis. 1993. Simulated Annealing. *Statist. Sci.* 8, 1 (1993), 10–15. https://doi.org/10.1214/ss/1177011077

[6] Enrico Bini, Marco Di Natale, and Giorgio Buttazzo. 2008. Sensitivity analysis for fixed-priority real-time systems. *Real-Time Systems* 39, 1–3 (2008), 5–30. https://doi.org/10.1007/s11241-006-9010-1

[7] Rahma Bouaziz, Laurent Lemarchand, Frank Singhoff, Bechir Zalila, and Mohamed Jmaiel. 2018. Multi-objective design exploration approach for ravenscar real-time systems. *Real-Time Systems* 54, 2 (2018), 424–483.

[8] Daniel Casini, Paolo Pazzaglia, Alessandro Biondi, and Marco Di Natale. 2022. Optimized partitioning and priority assignment of real-time applications on heterogeneous platforms with hardware acceleration. *Journal of Systems Architecture* (2022), 102416.

[9] Travis S. Craig. 1993. Queuing spin lock algorithms to support timing predictability. *1993 Proceedings Real-Time Systems Symposium* (1993), 148–157.

[10] Pedro Cuadra, Lukas Krawczyk, Robert Höttger, Philipp Heisig, and Carsten Wolff. 2017. Automated scheduling for tightly-coupled embedded multi-core systems using hybrid genetic algorithms. In *International Conference on Information and Software Technologies*. Springer, 362–373.

[11] George B. Dantzig. 1957. Discrete-variable extremum problems. *Operations research* 5, 2 (1957), 266–288.

[12] Robert I Davis and Alan Burns. 2007. Robust priority assignment for fixed priority real-time systems. In *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*. IEEE, 3–14.

[13] Hamid Reza Faragardi, Björn Lisper, Kristian Sandström, and Thomas Nolte. 2014. An efficient scheduling of AUTOSAR runnables to minimize communication cost in multi-core systems. In *7'th International Symposium on Telecommunications (IST'2014)*. IEEE, 41–48.

[14] Frédéric Fauberteau and Serge Midonnet. 2010. Robust Partitioned Scheduling for Static-Priority Real-Time Multiprocessor Systems with Shared Resources. In *18th International Conference on Real-Time and Network Systems*. 217–225.

[15] Gabriel Fernandez, Jaume Abella, Eduardo Quinones, Luca Fossati, Marco Zulianello, Tullio Vardanega, and Francisco J Cazorla. 2015. Seeking time-composable partitions of tasks for cots multicore processors. In *2015 IEEE 18th International Symposium on Real-Time Distributed Computing*. IEEE, 208–217.

[16] Fabrizio Ferrandi, Pier Luca Lanzi, Christian Pilato, Donatella Sciuto, and Antonino Tumeo. 2010. Ant colony heuristic for mapping and scheduling tasks and communications on heterogeneous embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 29, 6 (2010), 911–924.

[17] Simon Fürst, Jürgen Mössinger, Stefan Bunzel, Thomas Weber, Frank Kirschke-Biller, Peter Heitkämper, Gerulf Kinkelin, Kenji Nishikawa, and Klaus Lange. 2009. AUTOSAR — A Worldwide Standard is on the Road. In *14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden*, Vol. 62. 5.

[18] Mo Guan and Tong Tong. 2016. Ant colony algorithm based optimization method for real-time task scheduling of multi-core system.

[19] William E Hart, Jean-Paul Watson, and David L Woodruff. 2011. Pyomo: modeling and solving mathematical programs in Python. *Mathematical Programming Computation* 3, 3 (2011), 219–260.

[20] Robert Höttger, Lukas Krawczyk, and Burkhard Igel. 2015. Model-based automotive partitioning and mapping for embedded multicore systems. In *International Conference on Parallel, Distributed Systems and Software Engineering*, Vol. 2. 888.

[21] Infineon [n. d.]. *AURIX™TC3xx User's Manual*. Infineon. Available at https://www.infineon.com/cms/en/product/microcontroller/32-bit-tricore-microcontroller/32-bit-tricore-aurix-tc3xx/aurix-family-tc39xxx/.

[22] Yutaro Kobayashi, Kentaro Honda, Sasuga Kojima, Hiroshi Fujimoto, Masato Edahiro, and Takuya Azumi. 2022. Mapping Method Usable with Clustered Many-core Platforms for Simulink Model. *Journal of Information Processing* 30 (2022), 141–150.

[23] Simon Kramer, Dirk Ziegenbein, and Arne Hamann. 2017. Automotive application model based on APP4MC (WATER17). available at https://www.ecrts.org/forum/viewtopic.php?f=31&t=108&sid=9e9dc98cfb2dac9e2606ef421789ceeb.

[24] John P. Lehoczky, Lui Sha, and Ye Ding. 1989. The Rate-Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. In *Proceedings of the 10$^{th}$ IEEE Real-Time Systems Symposium*. Santa Monica (CA), U.S.A., 166–171.

[25] Liu Liping. 2017. CPU (Central Processing Unit) performance optimization method and device based on NUMA (Non-uniform Memory Access) architecture.

[26] Chung Laung Liu and James W. Layland. 1973. Scheduling Algorithms for Multiprogramming in a Hard real-Time Environment. *Journal of the Association for Computing Machinery* 20, 1 (Jan. 1973), 46–61.

[27] Robin Lougee. 2003. The Common Optimization INterface for Operations Research: Promoting open-source software in the operations research community. *IBM Journal of Research and Development* 47 (02 2003), 57 – 66. https://doi.org/10.1147/rd.471.0057

[28] Matias Maspoli, Matthias Knauss, and Marcin Nowacki. 2017. Method and device for operating a many-core system.

[29] Shane D. McLean, Silviu S. Craciunas, Emil Alexander Juul Hansen, and Paul Pop. 2020. Mapping and Scheduling Automotive Applications on ADAS Platforms using Metaheuristics. In *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, Vol. 1. IEEE, 329–336.

[30] Aloysius K. Mok and Deji Chen. 1997. A multiframe model for real-time tasks. *IEEE Transactions on Software Engineering* 23, 10 (Oct. 1997), 635–645.

[31] Fionn Murtagh and Pedro Contreras. 2011. Methods of Hierarchical Clustering. *Computing Research Repository - CORR* (04 2011). https://doi.org/10.1007/978-3-642-04898-2_288

[32] Suzuki Noriaki, Edahiro Masato, and Sakai Junji. 2012. Real time system task configuration optimization system for multi-core processors, and method and program.

[33] Suraj Paul, Navonil Chatterjee, Prasun Ghosal, and Jean-Philippe Diguet. 2020. Adaptive Task Allocation and Scheduling on NoC-based Multicore Platforms with Multitasking Processors. *ACM Transactions on Embedded Computing Systems (TECS)* 20, 1 (2020), 1–26.

[34] Paolo Pazzaglia, Alessandro Biondi, and Marco Di Natale. 2019. Optimizing the functional deployment on multicore platforms with logical execution time. In *2019 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 207–219.

[35] Quentin Perret, Pascal Maurère, Éric Noulard, Claire Pagetti, Pascal Sainrat, and Benoît Triquet. 2016. Mapping hard real-time applications on many-core processors. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems*. 235–244.

[36] Salah Eddine Saidi, Sylvain Cotard, Khaled Chaaban, and Kevin Marteil. 2015. An ILP approach for mapping autosar runnables on multi-core architectures. In *Proceedings of the 2015 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*. 1–8.

[37] H. Takada and K. Sakamura. 1994. Predictable spin lock algorithms with preemption. In *Proceedings of 11th IEEE Workshop on Real-Time Operating Systems and Software*. 2–6. https://doi.org/10.1109/RTOSS.1994.292571

[38] Guido Van Rossum and Fred L. Drake. 2009. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA.

[39] A. Wieder and B. Brandenburg. 2013. On spin locks in AUTOSAR: Blocking analysis of FIFO, unordered, and priority-ordered spin locks. In *Proceedings of the IEEE 34th Real-Time Systems Symposium*. 45–56.

[40] Carsten Wolff, Lukas Krawczyk, Robert Höttger, Christopher Brink, Uwe Lauschner, Daniel Fruhner, Erik Kamsties, and Burkhard Igel. 2015. AMALTHEA — Tailoring tools to projects in automotive software development. In *2015 IEEE 8th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, Vol. 2. IEEE, 515–520.

[41] M. Yang, A. Wieder, and B. Brandenburg. 2015. Global real-time semaphore protocols: A survey, unified analysis, and comparison. In *Proceedings of the IEEE 36th Real-Time Systems Symposium*. 1–12.

[42] Yecheng Zhao and Haibo Zeng. 2018. The concept of unschedulability core for optimizing real-time systems with fixed-priority scheduling. *IEEE Trans. Comput.* 68, 6 (2018), 926–938.