Contention-Free Execution of Automotive Applications on a Clustered Many-Core Platform

Matthias Becker*, Dakshina Dasari†, Borislav Nikolić‡, Benny Akesson‡, Vincent Nélis‡, Thomas Nolte*
*MRTC / Mälardalen University, Sweden {matthias.becker, thomas.nolte}@mdh.se
† Research and Technology Centre, Robert Bosch, India dakshina.dasari@in.bosch.com
‡CISTER/INESC-TEC, ISEP, Portugal {borni, kbake, nelis}@isep.ipp.pt

***Abstract*—Next generations of compute-intensive real-time applications in automotive systems will require more powerful computing platforms. One promising power-efficient solution for such applications is to use clustered many-core architectures. However, ensuring that real-time requirements are satisfied in the presence of contention in shared resources, such as memories, remains an open issue.**

This work presents a novel contention-free execution framework to execute automotive applications on such platforms. Privatization of memory banks together with defined access phases to shared memory resources is the backbone of the framework. An Integer Linear Programming (ILP) formulation is presented to find the optimal time-triggered schedule for the on-core execution as well as for the access to shared memory. Additionally a heuristic solution is presented that generates the schedule in a fraction of the time required by the ILP. Extensive evaluations show that the proposed heuristic performs only 0.5% away from the optimal solution while it outperforms a baseline heuristic by 67%. The applicability of the approach to industrially sized problems is demonstrated in a case study of a software for Engine Management Systems.

I. Introduction

The automotive domain is witnessing a surge of innovation as new advanced driver assistance systems and autonomous vehicles shape the demand for more functionality. This demand has already resulted in modern cars with 80-100 Electronic Control Units (ECU) [1], [2] that manage different subsystems, such as the power-train, the chassis, active safety, driver assistance and infotainment. In tandem with this development, automotive architectures are also witnessing a major paradigm shift from multiple scattered single-core ECU's (connected over multiple busses) to hierarchical multi-core Domain Controllers (DC) with the intent of ECU consolidation. It is envisioned that future cars will consist of multiple DCs interconnected over a deterministic Ethernet backbone [1], [3], [4], where each of these domain controllers will be a many-core platform that caters to the needs of a specific subsystem. The architectural needs for such a domain controller can be closely mapped to newer clustered many-core architectures, such as the MPPA-256 from Kalray [5], Intel's SCC [6] and Tilera Tile64 [7]. These platforms provide clusters of cores, each capable of hosting a different domain application. Additionally, the multiplicity of cores provides the required computing capabilities within the desired power envelope of embedded automotive applications.

However, the transition to multi/many-core platforms is not straightforward, since resources (e.g. memory subsystems, the interconnect medium and the I/O subsystem) are shared among applications. Although resource sharing provides benefits in terms of cost and energy savings, it may cause complex interference scenarios between sharing applications, which may lead to missed deadlines and/or yield incorrect outputs in a multi-core setting [8]. This is a serious problem in some areas, including automotive systems, where applications are often *safety-critical* and have *strict timing requirements*.

Deriving tight bounds for an unconstrained execution on such a platform is often difficult, if not impossible. This paper addresses these problems by proposing a *contention-free execution framework for automotive applications on a clustered many-core architecture*. The five main contributions of this work are:

1) A cluster organization based on two pillar concepts: (i) *privatization* of memory banks to ensure interference-free execution, and (ii) *sharing* of other memory banks to support communication between software components. These concepts are suitable for automotive applications and constitute the basis of the framework.
2) Assuming this organization, we propose an Integer Linear Programming (ILP) formulation that *optimally* maps AUTOSAR runnables to cores of a cluster and provides a contention-free time-triggered schedule that distinguishes and overlaps memory access phases and execution phases.
3) A memory-centric heuristic that finds a runnable to core mapping and a contention-free time-triggered schedule in a fraction of the time required by the ILP method, and scales up to handle industrial use-cases in the automotive domain.
4) We experimentally show that the proposed memory-centric heuristic significantly outperforms a baseline core-centric heuristic, while only sacrificing 0.5% in average schedulable utilization compared to the ILP.
5) Finally, we demonstrate the applicability of our approach through a case study of a realistic engine control application deployed on a clustered many-core architecture, resembling a Kalray MPPA-256 [5].

The rest of this paper is organized as follows. Section II presents related work, followed by a description of our system model in Section III. The proposed contention-free execution framework is introduced in Section IV, after which the ILP- and the heuristic-based mapping and scheduling approaches are presented in Section V. We experimentally evaluate our approach in Section VI and demonstrate its applicability via a

case study. Lastly, conclusions are presented in Section VII.

II. Related Work

Contention analysis of shared resources in multi-/many-core Commercial Of-The-Shelf (COTS) platforms has received significant attention in recent years. Most analyses consider multi-core systems with a simple bus providing access to a single shared memory [9], [10], [11], [12]. However, contention analysis of clustered many-core platforms has also been explored, as in [13], [14], [15], [16], where the former two are the most relevant for this work, as they focus on Kalray MPPA-256, which is the platform considered in this paper.

A response time analysis for different resource access models is presented in [17] and it is shown that applications following a read-execute-write semantic perform best. The PRedictable Execution Model (PREM) [18] builds on this result by proposing to divide applications into dedicated non-preemptive memory and execution phases, where all cache-lines required for non-blocking execution are fetched during the memory phase. Although the original work only considers fixed-priority scheduling on a single-core system, the concept has been extended to multi-core systems in [19], [20], [21] and applied to a heterogeneous many-core system in [22]. Our work is related to the PREM effort in the sense that we consider AUTOSAR applications, where the execution of runnables is divided into memory phases and execution phases. However, the state-of-the-art works considering PREM are currently limited to independent periodic/sporadic tasks, while our work relaxes that assumption and considers runnables that share data. This addition is highly relevant in the context of automotive applications, where a single application may require thousands of shared variables.

There are several existing approaches to mapping applications on multi-core platforms, some heuristic and others exact. We will discuss these two categories of solutions in turn, starting with the former. Faragardi et al. [23] heuristically map an AUTOSAR application in which event-chains/transactions are clearly specified with associated deadlines and periods. Similarly to Monot et al. [24], who use bin-packing to map applications, they assume that transactions are independent and therefore group all runnables within a transaction to a given task. However, the assumption of independent transactions does not hold for complex applications in the automotive context due to a high degree of coupling among runnables. This issue is addressed in [25], which also maps based on bin-packing, although while respecting precedence constraints between tasks. A drawback of this approach is that it does not efficiently deal with communication delays, since they inflate the Worst-Case Execution Time (WCET) of the runnable by assuming maximum interference for every access. In contrast, our contention-free approach separates execution and communication into distinct phases where cores are only blocked when a runnable is in the execution phase. This enables a more efficient use of resources. A communication-aware mapping of dependent tasks to a Kalray-like platform is presented in [13]. This work is related to ours as it models the MPPA-256 and its associated resources, although the approach is contention-aware as opposed to contention-free. An algorithm based on general simulated annealing is proposed for mapping tasks to cores and data to memory banks. Similarly, Dziurzanski et al. [26] derive mappings by using a heuristic based on genetic algorithms. However, in contrast to our work, all the aforementioned heuristic approaches are not compared to an optimal formulation, and the quality of the proposed heuristics is hence not quantified.

While the quality of a heuristic approach is difficult to establish, exact approaches to complex problems typically suffer from scalability issues and cannot provide optimal solutions for large problem instances in reasonable time. This is apparent in [27], [28], where ILP formulations are proposed for mapping and scheduling runnables on multi-core and distributed automotive architectures, respectively. The approaches are demonstrated for a small automotive applications, although results clearly show that a basic ILP formulation does not scale to complex applications with hundreds or thousands of runnables that are individually mapped and scheduled. Existing work address this scalability issue by optimizing their formulations (ILP or otherwise) to refine constraints and remove symmetry [29], decomposing the problem into smaller, possibly communicating, sub-problems [30], [31], or by finding and addressing the minimum reason for constraint violations [32], [30] and exclude it in future searches. These types of optimizations help improve the scalability of exact approaches, but have not been shown to scale to large industry-sized applications.

Our method is different from existing work in the sense that it proposes a contention-free execution environment for a clustered many-core architecture based on a combination of bank privatization and time-triggered scheduling. Unlike most previous approaches, our work deals with code and communication-data placement, data dependencies between runnables, and it includes costs for fetching data/code from off-chip memory. Additionally, we do not place any restriction on the execution of runnables on a particular core, but instead we consider the compute cluster as a pool of available resources (similar to global scheduling). A fast heuristic algorithm is proposed to address the mapping and scheduling problem of individual runnables of complex applications, and its quality is compared to optimal solutions for smaller use-cases (due to the scalability issue of optimal techniques).

III. System Model

This section presents the system model used in this paper. First, the platform model is introduced in Section III-A, followed by the software model in Section III-B.

A. Platform Model

We consider a domain controller model very similar to Kalray MPPA-256 Bostan [5], [33], which is a clustered many-core platform organized as illustrated in Fig. 1. Cores are grouped in clusters connected by a Round-Robin (RR) arbitrated Network-on-Chip (NoC) in a 2D-torus topology [14].

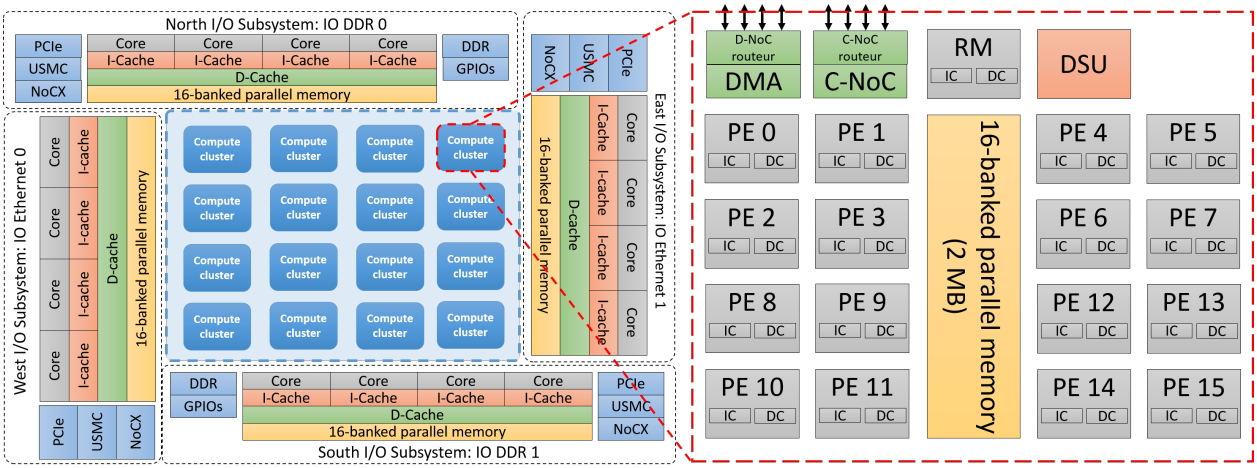

Fig. 1: Outline of the architecture of the Kalray MPPA-256.

Each cluster contains $(n + 1)$ identical processing elements ($n = 16$ in the MPPA-256), of which n are *compute cores*, dedicated to general-purpose computations. Every compute core has a private instruction and data cache. The *Resource Manager* (RM) core is identical to the compute cores, but has a different purpose – it manages processor resources on behalf of the entire cluster (maps and schedules runnables on compute cores), and also organizes the communication between its hosting cluster and other clusters on the chip, as well as with the main off-chip memory. All cores are fully timing compositional [34] in the sense that they do not exhibit timing anomalies. Additionally each cluster also contains a Debug Support Unit (DSU), a network interface for receiving data requests from the Data-NoC (D-NoC) and a DMA engine used for data transmission over the D-NoC.

Regarding the organization of the memory subsystem, each cluster has a local shared memory comprising n banks, each with a capacity of S_{bank}, for a total memory capacity of $n \times S_{\text{bank}}$ in each cluster. In the MPPA-256, $S_{\text{bank}} = 128$ KB for a total memory capacity of 2 MB per cluster. Although the cluster address space can be divided among banks in an interleaved fashion (useful for high-performance and parallel applications), this work uses the blocked memory mode where the address space is divided in a sequential manner. This results in more predictable system behavior, which is a desired characteristic in the safety-critical domain. The mapping of data and code to memory banks can then be done by the use of linker scripts.

The arbitration of memory requests to a cluster's memory bank is performed in four levels (stages), as depicted in Fig. 2. The first three levels use the RR arbitration scheme. The first level arbitrates between memory requests from the data cache and instruction cache of each compute core. At the second level, the requests issued from each compute core compete against requests from other compute cores. At the third level, requests from all compute cores compete against requests from the RM, the DSU, and the DMA. Finally, at the fourth and last level, the scheduled requests compete with those coming from the D-NoC (Rx) under static-priority arbitration, where requests from the NoC always have higher priority. Note that in order to minimize contention, all arbitration levels are replicated for each memory bank.

The large number of memory clients sharing each memory bank give rise to a large amount of possible contention for each memory access. In the worst case, all memory clients try to access the same memory bank at the same time. In this case, a cache of one of the cores has to compete with 31 other caches in the first two levels of RR arbitration, each of which could be preceded by an access by the RM, DSU, or DMA in the third arbiter. Even if the interference from the NoC receiver in the static-priority arbiter is ignored, this results in a worst-case latency of $16 \cdot 2 \cdot 2 - 1 = 63$ memory accesses. Assuming accesses of a single word, which takes 10 cycles to serve (9 cycles latency with 8 bytes fetched on each consecutive cycle, as reported in [35]), this corresponds to 630 clock cycles, or 1575 ns at 400 MHz. In contrast, an access without contention would finish in just 10 cycles, or 25 ns. Based on this interference analysis, we infer that the penalty for a cache miss may be very high unless the worst-case situation is somehow prevented. This work proposes to address this problem by creating a contention-free execution environment based on a combination of bank privatization and time-triggered scheduling, as further detailed in Section IV.

B. Software Model

We consider an AUTOSAR application, which at the lowest level consists of a number of *runnables*. A runnable is a schedulable entity that can be thought of as a C-function. We characterize a runnable r_i by the tuple $\{T_i, C_i, S_i, \mathcal{R}_i, \mathcal{W}_i\}$, where T_i is the period of the runnable, C_i is its worst-case execution time, and S_i is its memory footprint. Each runnable is implicitly assigned a deadline equal to its period.

We adopt the read-execute-write semantics of execution, which is a standard execution model for AUTOSAR applications [36], [37]. Under these semantics, the execution of a runnable is logically divided into three distinct and consecutive

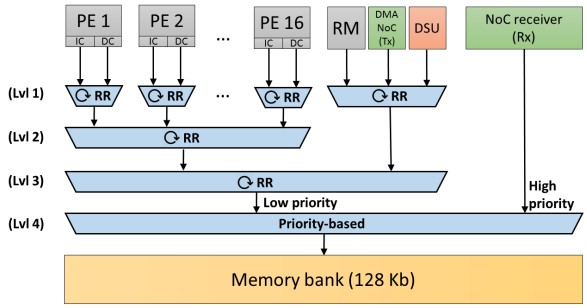

Fig. 2: Memory request arbitration for the memory banks on a cluster of the MPPA-256.

phases, namely the *read*, *execute*, and *write* phases. During the *read* phase of a runnable, the code and all data needed for the execution of the runnable are copied into the memory space allocated to it. In the *execution* phase, the runnable is executed in a non-blocking manner. Finally, in the *write phase*, all output variables are written. We denote by C_i^{read}, C_i^{exec}, and C_i^{write} the WCET of each phase of runnable r_i, with $C_i = C_i^{\text{read}} + C_i^{\text{exec}} + C_i^{\text{write}}$.

Runnables communicate through variables (called *labels* in the automotive domain) stored in a shared memory. Each runnable description therefore includes a set of input labels $\mathcal{R}_i$ and a set of output labels $\mathcal{W}_i$. The access type of a runnable to a label can be read, write, or both, and a label can be accessed from any number of runnables, without precedence constraints. A label itself is denoted as $L_k = \{M_k\}$, where M_k is its size in bytes.

IV. Contention-Free Execution Framework

The previous section showed that the maximum contention for shared memory banks is prohibitively large unless accesses to the memory banks are constrained. This section explains how we address this problem by proposing a contention-free execution framework for automotive applications in clustered many-core systems. The two key ingredients to achieve this are: 1) eliminate interference from other cores using *memory bank privatization*, while enabling communication between cores using a *shared communication bank*, 2) *read-execute-write semantics* for runnables to get coarse-grained memory phases that can be scheduled on the label bank and off-chip memory in a mutually exclusive manner by a *time-triggered scheduler*. These mechanisms are further detailed below.

A. Memory Bank Privatization

The idea behind *bank privatization* is to avoid interference between runnables executing on different cores by preventing them from accessing the same memory bank at the same time. Privatization is achieved by statically assigning each core to one bank with exclusive access, i.e. the bank assigned to each core cannot be accessed by any other core. This is a viable approach since the considered hardware platform provides as many memory banks as compute cores. Such a bank privatization technique totally eliminates interference from other cores and interfaces at the first, second, and third levels of arbitration, as those arbiters are replicated for each individual bank (see Fig. 2).

To enable communication between cores, a memory bank among the set of available banks is arbitrarily selected and dedicated to the storage of all the labels (hereafter this bank is casually referred to as the *label bank*). Unlike the private banks, all cores can access the label bank and use it as shared memory for communication. The labels are statically loaded into the label bank when the system boots and are never evicted at run-time. To increase predictability, we sacrifice performance and do not perform any computation on the compute core associated with this label bank, thus this core remains unused[1].

Communication in our framework follows the implicit communication model of AUTOSAR, where runnables always operate on local copies of the labels [37], [36]. This means that if a runnable r_i has read access to a specific label L_k (i.e. $L_k \in \mathcal{R}_i$) the label is copied from the label bank into the private bank of its assigned core. During execution, solely the local copy is accessed by the runnable. Similarly, if a runnable r_i has write access to a label L_k (i.e. $L_k \in \mathcal{W}_i$) the runnable writes to a variable located in the local memory bank. Once the execution of a job ends, these values are committed to the label bank. Note that, as per this implicit communication model of AUTOSAR, if a runnable reads and writes the same variable, two local copies are used to store the input and output value respectively. This is done such that the runnable always operates on the same input value during the execution of one job.

To avoid interference on the label bank, it must be guaranteed that there cannot be two runnables that retrieve/commit a value simultaneously from/to the label-bank. This is ensured in our model by explicitly defining different runnable phases and by scheduling these phases suitably, as explained in the following sections.

B. Read-Execute-Write Semantic

As previously mentioned, our approach uses read-execute-write semantics of execution [36], [37] to benefit from coarse-grained memory phases that can be orchestrated in software. During the *read* phase of a runnable, all data needed for its execution is written into the private memory bank of the core to which the runnable has been assigned. This includes the code and input data of the runnable (that may have to be fetched from the off-chip memory), but also the labels used for the communication with other runnables. Input labels needed by the runnable during its execution are copied from the common label bank to the private bank of the core. Similarly, buffers for output labels are configured with their respective initialization values. In the *write* phase, the modified non-shared data is written back to the off-chip memory and the

[1]Note that if the labels do not fit within one bank, more banks could be selected and dedicated to the storage of the labels, which also means that more compute cores would be unused.

output labels in the label bank are updated with their new values from the private bank.

The division into these three execution phases has three main advantages:

1) Since no information (code and data) of a runnable is kept in the cores' private banks after its execution, the framework yields independence between the runnables and the cores. Runnables can therefore be assigned to any core and subsequent executions of a same runnable can be assigned to different cores. This results in extra freedom during scheduling, thereby increasing the chances of satisfying the timing requirements of a given set of runnables.
2) By generating a time-triggered schedule for the three phases of every runnable without overlap (as discussed in the next section), it is possible to provide exclusive access to off-chip memories and the label bank, thereby *eliminating all memory contention*.
3) It ensures that the required data is always available to the runnable before it starts executing and it will not stall once it has started. This increases the predictability of the system by reducing uncertainties and thus pessimism in the analysis, as well as unpredictable overheads at runtime.

C. Time-Triggered Scheduler

Our next directive is to compute a schedule of the runnables on the cores and enforce that schedule at run-time. Here, "schedule" refers to the set of time-instants at which each runnable job starts to execute. A time-triggered scheduler enforces predictable system behavior and does not require complex scheduling decisions at run-time. It also eliminates the need for synchronization constructs, such as mutexes and spin-locks. The time-triggered schedule proposed here is defined with the objective of avoiding resource access conflicts (and hence interference), both on the cores and the memory.

Each of the three phases of the runnables (read, execute, and write) is a schedulable entity that starts at a specifically computed time-instant and executes non-preemptively. The schedule is constructed according to the following 3 rules: (1) the required data and code sections of every runnable are loaded into the private memory bank of its assigned core before it starts its execution in order to enable non-blocking execution, (2) read and write phases of any two runnables do not overlap in time to avoid interference in label banks and off-chip memory, and (3) the schedule is guaranteed to preserve the timeliness of the execution. The actual algorithms for deriving such a schedule are described in the next section.

V. Generation of the Time-Triggered Schedule

Generating a time-triggered schedule is an NP-hard problem, as it involves mapping jobs to cores and deciding the execution order, while at the same time satisfying dependencies. This section proposes two different approaches towards solving this problem. The first approach adopts Integer Linear Programming (ILP) while the second approach proposes a memory-centric scheduling heuristic.

A. The ILP Approach: Finding an Optimal Solution

In this approach, we construct a time-triggered schedule by formulating the problem as an objective function, subject to linear constraints. Then, we feed that formulation into an ILP solver. This approach has the benefit of finding the optimal solution, i.e. a schedule is found, as long as a solution exists, but it suffers from the well-known limitation of having its computation time growing drastically with the problem size under consideration.

The objective function: Since our goal is to find *any* schedule that meets all timing requirements (i.e. fulfill all the constraints), we do not need to optimize any criteria and we use a constant objective function, e.g. "**maximize** 1".

The variables: The objects to be scheduled are the (potentially many) executions of every runnable within a given time window of length H. Given that runnables execute periodically in our model, we define H as the hyper-period of all runnables, i.e. $H \stackrel{\text{def}}{=} LCM(T_i)$, $\forall i$. We denote by $\mathcal{J}$ the set of all runnable executions in H and by $r_{i,j} \in \mathcal{J}$ the j^{th} execution of runnable r_i in the time interval H (also called its j^{th} *job*). For each job $r_{i,j}$, we define two constants $\mathrm{rel}_{i,j}$ and $\mathrm{dead}_{i,j}$ which denote the time-instants of its release and deadline, respectively. The variables $\mathrm{start}_{i,j}$ and $\mathrm{end}_{i,j}$ are introduced to denote the time when $r_{i,j}$ starts and ends its execution (including the read and write phase). Further, we introduce the decision variables $\mathrm{mapped}_{i,j,k}$ to indicate whether or not the job $r_{i,j}$ is assigned to core k, i.e.

$$\mathrm{mapped}_{i,j,k} = \begin{cases} 1, & \text{if } r_{i,j} \text{ is mapped to core } k \\ 0, & \text{otherwise} \end{cases}$$

The constraints: Some constraints that we use in our formulation require two logical operators, logical *and* – ($\wedge$), and logical *or* – ($\vee$). These operators are not directly applicable to ILPs, and therefore should be linearized beforehand (Equations (1)−(2) demonstrate how that can be done for two binary variables a and b).

$$a \wedge b = 1 \quad \Rightarrow \quad a + b \geq 2 \tag{1}$$

$$a \vee b = 1 \quad \Rightarrow \quad a + b \geq 1 \tag{2}$$

In the remainder of this work, for clarity purposes, we use logical operators to express constraints. Note that some programs for implementing and solving ILPs (e.g. [38]) support logical operators and automatically perform linearizations.

In our implementation, there are two types of constraints.

1) *Job-to-core assignment constraints*: At run-time, every job of a runnable must be executed on exactly one core, but the execution model does not require all the jobs of the same runnable to be assigned to the same core, i.e. runnable migration is allowed while job-level migration is forbidden. This is enforced by adding the following constraint to the model:

$$\forall r_{i,j} \in \mathcal{J}: \qquad \sum_{k=1}^{n} \mathrm{mapped}_{i,j,k} = 1$$

Every job must execute entirely within its execution window delimited by the time-instants of its release and deadline:

$$\forall r_{i,j} \in \mathcal{J}: \quad \mathrm{start}_{i,j} \geq \mathrm{rel}_{i,j}$$
$$\forall r_{i,j} \in \mathcal{J}: \quad \mathrm{end}_{i,j} \leq \mathrm{dead}_{i,j}$$

Additionally, every job must occupy the core for at least its minimum execution time. There might be benefits by not minimizing $\mathrm{end}_{i,j}$. Delaying $\mathrm{end}_{i,j}$ will implicitly postpone the write phase of the job, which can improve schedulability (e.g. in case another job $r_{k,l}$ executing on an other core needs to urgently schedule its write phase in order to reach its deadline. Delaying the write phase of job $r_{i,j}$ after the write phase of $r_{i,k}$ will thus allow for $r_{k,l}$ to finish in time.).

$$\forall r_{i,j} \in \mathcal{J}: \quad \mathrm{end}_{i,j} - \mathrm{start}_{i,j} \geq C_i$$

Moreover, the execution of any two jobs assigned to the same core cannot overlap: $\forall r_{i,j}, r_{x,y} \in \mathcal{J}$ with $r_{i,j} \neq r_{x,y}$, $\forall k \in [1,n]$:

$$\mathrm{mapped}_{i,j,k} + \mathrm{mapped}_{x,y,k} \leq 1 \quad \vee \quad \mathrm{overlap}_{i,j}^{x,y} = 0$$

with

$$\begin{aligned} \mathrm{overlap}_{i,j}^{x,y} = \quad & \mathrm{start}_{x,y} \leq \mathrm{start}_{i,j} < \mathrm{end}_{x,y} \\ \vee \quad & \mathrm{start}_{x,y} < \mathrm{end}_{i,j} \leq \mathrm{end}_{x,y} \\ \vee \quad & (\mathrm{start}_{i,j} < \mathrm{start}_{x,y} \wedge \mathrm{end}_{i,j} > \mathrm{end}_{x,y}) \end{aligned}$$

2) *Memory constraints*: The following constraints relate to the memory and data dependencies. First, we introduce a constraint to prevent any overlap between memory read and write phases, ensuring that the shared memory banks or off-chip memory are never accessed simultaneously: $\forall r_{i,j}, r_{x,y} \in \mathcal{J}$ with $r_{i,j} \neq r_{x,y}$, $\forall k \in [1,n]$: $\text{R-overlap}_{i,j}^{x,y} = 0 \wedge \text{W-overlap}_{i,j}^{x,y} = 0$, with $\text{R-overlap}_{i,j}^{x,y} \stackrel{\text{def}}{=}$

$$\begin{aligned} & \mathrm{start}_{x,y} \leq \mathrm{start}_{i,j} < \mathrm{start}_{x,y} + C_x^{\mathrm{read}} \\ \vee \quad & \mathrm{start}_{x,y} < \mathrm{start}_{i,j} + C_i^{\mathrm{read}} \leq \mathrm{start}_{x,y} + C_x^{\mathrm{read}} \\ \vee \quad & (\mathrm{start}_{i,j} < \mathrm{start}_{x,y} \wedge \mathrm{start}_{i,j} + C_i^{\mathrm{read}} > \mathrm{start}_{x,y} + C_x^{\mathrm{read}}) \end{aligned}$$

and $\text{W-overlap}_{i,j}^{x,y} \stackrel{\text{def}}{=}$

$$\begin{aligned} & \mathrm{end}_{x,y} - C_x^{\mathrm{write}} \leq \mathrm{end}_{i,j} - C_i^{\mathrm{write}} < \mathrm{end}_{x,y} \\ \vee \quad & \mathrm{end}_{x,y} - C_x^{\mathrm{write}} < \mathrm{end}_{i,j} \leq \mathrm{end}_{x,y} \\ \vee \quad & (\mathrm{end}_{i,j} - C_i^{\mathrm{write}} < \mathrm{end}_{x,y} - C_x^{\mathrm{write}} \wedge \mathrm{end}_{i,j} > \mathrm{end}_{x,y}) \end{aligned}$$

B. Memory-Centric Scheduling Heuristic (MCH)

In this approach, we construct an offline time-triggered schedule using a memory-centric scheduling heuristic. The central idea behind this approach is that the crucial resources to be scheduled within the cluster are not the cores (which are plentifully available) but the single NoC channel which transfers data between the cluster and the external off-chip memory and the common label banks on the cluster which must be accessed exclusively to avoid contention. MCH therefore aims to create a time triggered schedule of (exclusive contention free) accesses to the NoC and the label banks.

1) Algorithm Overview: The input to Algorithm 1 is a set of runnable instances (jobs) that must be scheduled on the compute cluster during one hyperperiod. Each job is considered to be a scheduling entity and the algorithm aims at assigning jobs to the available cores, while scheduling their memory access phases on the off-chip memory via the NoC, and the local memory banks. The algorithm follows a global scheduling approach, where different jobs of a given runnable may be assigned to *different* cores. In order to decouple the computation from memory accesses, the algorithm splits each job into three *logical* (sub-) jobs, a read job, an execute job and a write job. Note that all these sub-jobs are *always* assigned to the same core. The assignment of the three phases of the job follows the read-execute-write semantic. A specific core is selected from which the read job (code and data fetch) is initiated, which implicitly decides where the corresponding execute and write jobs are carried out. As a result of the bank privatization described earlier, it follows that the code/label section of the job is allocated to the private bank of that core. In principle, there is no explicit assignment of the execute and write portions of the job. As we describe shortly, each of these sub-jobs are also assigned a deadline, release-time and execution time, making it a well-defined schedulable entity.

2) Algorithm Setup: A current time variable "ctime", initialized to zero at the outset, is used to track the progress of the algorithm from time zero to the computed hyperperiod. Additionally, two queues are used to manage the unscheduled jobs. The queue Q_{job} contains all jobs that are not yet ready, meaning that their release time is after (greater than) the time indicated by ctime. A second queue, Q_{ready} contains all jobs that are ready to be scheduled at the current algorithm time. With the progress of time, jobs subsequently move from Q_{job} to Q_{ready} and the algorithm assigns jobs from Q_{ready} to the cores. A set S that is initially empty is used to hold the final *memory access schedule*. Hence, S holds the start times of all *read* and *write* phases. This implicitly defines the start times of the execution phases or a window in which it can be started unless they are tightly packed, since no other job can utilize the core while the associated private memory bank is accessed by either read or write phase. Thus S does not include the start time of the execution phase of a job.

3) Algorithm Description: After initialization, the algorithm progresses by populating the queue Q_{job} of jobs to be scheduled within the hyperperiod (see line 5). The next step is to create and define read sub-jobs, which is realized by the function `generateReadJobs`($\mathcal{J}$). This function generates a job $j_{i,k}^{\mathrm{read}}$ for every job $r_{i,k}$ in $\mathcal{J}$, corresponding to the read phase of $r_{i,k}$. As illustrated in Fig. 3, the release time $\mathrm{rel}_{i,k}$ of the read job $j_{i,k}^{\mathrm{read}}$ is set to the release time of its associated job $r_{i,k}$ and its deadline $\mathrm{dead}_{i,k}$ is set to the release time of the next job of the same runnable, after deducting the execution time of its execute and write phases, i.e. $\mathrm{dead}_{i,k} = \mathrm{rel}_{i,k+1} - (C_i^{\mathrm{exec}} + C_i^{\mathrm{write}})$. The WCET of $j_{i,k}^{\mathrm{read}}$ is simply set to the WCET C_i^{read} of the read job of $r_{i,k}$. Every such read job $j_{i,k}^{\mathrm{read}}$ is enqueued in Q_{job}. At this point, the queue Q_{ready} is initially empty since there are no ready jobs. The following

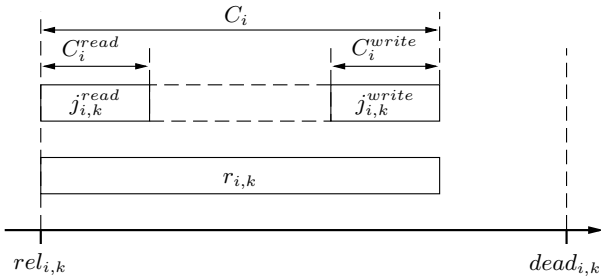

Fig. 3: Relation between the runnable jobs and the generated jobs.

Algorithm 1: MCH-GenerateMemSchedule($\mathcal{J}$)

```
   input : J, the set of jobs
1  begin
2      S ← ∅; // empty schedule
3      ctime ← 0; // current time
4      HP ← getHyperperiod(J);
5      Q_job ← generateReadJobs(J);
6      Q_ready ← ∅; // empty queue of ready jobs
7      while (ctime ≤ HP) do
8          if (no read/write job is running at time ctime) then
9              Q_ready.add({j_{i,k} ∈ Q_job | rel_{i,k} = ctime}) ;
10             j_curr ← getNextJob(Q_ready);
11             if (j_curr is a read job) then
12                 if (there is a core available) then
13                     assign j_curr to an available core;
14                     S.add(j_curr);
15                     ctime ← ctime + C_curr;
16                 else ctime ← ctime + 1;
17             else if (j_curr is a write job) then
18                 S.add(j_curr);
19                 ctime ← ctime + C_curr;
20             else ctime ← ctime + 1;
21         if (a read or write job is running and
           is finishing at time ctime) then
22             j_run ← the running job;
23             if (j_run is a read job) then
24                 j_write ← generateWriteJob(j_run);
25                 Q_job.add(j_write);
26             if (j_run is a write job) then
27                 if (j_run finished after its deadline) then
28                     return UNSCHEDULABLE;
29                 mark its assigned core as free;
30     if (Q_ready ≠ ∅ ∨ Q_job ≠ ∅ ∨ Active job ≠ ∅) then
31         return UNSCHEDULABLE;
32     return S;
```

explains the algorithm in five steps (a-e).

a) Empty ready queue: At line 8, when there are no jobs running (it is for instance the case at the first iteration in the while loop), the algorithm advances the current time ctime, up to the point when one of the jobs in Q_{job} can be moved to Q_{ready}, which happens when ctime is equal to the release time of that job (line 9).

b) Single read job in the ready queue: If the job in the ready queue is a read-job (line 11), the algorithm assigns it to an available core (if any), implying that the execute phase of that job will run on that core. Additionally, the memory channel is assigned to complete the read job (line 14), which is thus added to S. The algorithm time ctime is advanced to account for the completion of the read job.

c) Multiple jobs in the ready queue: If there are multiple jobs in the ready queue, the selection of the next job to be executed (see line 10) is carried out by the rules outlined in Algorithm 2 and explained below. The ready queue is sorted in a non-decreasing order based on the deadline of the jobs. The algorithm then selects the first read job and the first write job in the sorted queue. A selection between the candidate read- and write-jobs is carried out by applying the following three rules:

1) If all the cores are used (line 4-5), schedule any write job (if any).
2) If there is an available core and a read and a write job have the same deadline, prioritize the write job (line 8).
3) If there is an available core, prioritize the job with the smaller deadline (line 10-13).

The main intuition behind prioritizing a write job over a read job is that once completed, a write job releases the occupied core, whereas executing an additional read job locks up another core. Not prioritizing a write-job may lead to indefinite blocking in the case where all cores are currently in job's execution phase or before a job's write phase (hence busy or waiting for the write job to be scheduled) and a read job has the smallest deadline in Q_{ready}. If write jobs are not prioritized, the system blocks indefinitely waiting for a core to become free.

d) Job completion: As the current time ctime advances, the algorithm checks if it aligns with the completion time of any read or write job that was scheduled to run (line 21). If the completing job is a read job (line 23), the algorithm creates a corresponding write job j^{write} (line 24). The release time of that write job is set to the finishing time of the read job, plus the time for the corresponding execution phase. The deadline of j^{write} is set to the deadline of the parent runnable job. Job j^{write} is then enqueued in Q_{job} (line 25). There is no explicit mapping step needed for a write job, since a core is already assigned to its (parent) job in an earlier phase. Likewise, the start time of the execution phase is implicitly set to the finishing time of its corresponding read job.

If the completed job is a write job (line 26), its assigned core needs to be freed (line 29), because this phase marks the end of the execution of the entire job. At this point, the heuristic also checks if the deadline of the write job was missed, in which case the algorithm terminates, returning the status "unschedulable" (line 28).

e) End of hyperperiod: Once ctime exceeds a hyperperiod, the algorithm checks if there is any job in Q_{ready} or Q_{job} as well as if there are currently executing jobs on one of the cores or on the memory (line 30). The presence of any such job implies that the algorithm was unsuccessful in scheduling it within the hyperperiod and it terminates with the result "unschedulable". In the case of successfully scheduling

Algorithm 2: MCH-getNextJob(Q_{ready})

```
   input : Q_ready, queue with ready jobs
1  begin
       // get index of first read and write job
          without dependencies to active jobs
2      i_write ← getJobIndex(Q_ready, "write");
3      i_read ← getJobIndex(Q_ready, "read");
4      if (there is no core available) then
5          return remove(Q_ready, i_write)
       // deadline of selected read & write job
6      D_read ← getDeadline(Q_ready, i_read);
7      D_write ← getDeadline(Q_ready, i_write);
       // getDeadline() returns −1 if no job found
8      if (D_write = D_read ∧ D_write ≠ −1) then
9          return remove(Q_ready, i_write)
10     if (D_read < D_write ∧ D_read ≠ −1) then
11         return remove(Q_ready, i_read);
12     if (D_read > D_write ∧ D_write ≠ −1) then
13         return remove(Q_ready, i_write);
14     return ∅
```

all jobs, the algorithm returns the generated schedule S.

VI. Experiments

In this section we evaluate the proposed execution framework as well as the heuristic to generate the time-triggered schedule. We describe the experimental setup and compare our proposed memory centric heuristic (MCH) against a core-centric heuristic (CCH) (described later in this section) and the ILP formulation. Additionally, the applicability to industrially sized applications is compared for the two heuristics and the ILP formulation using an automotive case study.

A. Experimental Setup

1) Instance Generation: In the first set of experiments, we generate synthetic data sets with parameters that conform to automotive applications [37]. The runnable utilizations were generated by UUniFast [39] and periods were selected such that the complete runnable set comprises 1 x 100 ms (1 runnable of period 100 ms), 5 x 1000 ms, 1 x 50 ms, 3 x 200 ms, and 1 x 20 ms. Thus, each runnable set comprises 100 jobs over the entire hyperperiod of 1000 ms.

2) Baseline Core-Centric Heuristic (CCH): Methods described in related work either assume different platforms or application models, which renders them unsuitable for any meaningful comparison. Hence, we propose a baseline heuristic for comparison. The underlying premise of this baseline core-centric heuristic (CCH) is that the main resources to be scheduled are the available *cores* in the compute cluster; therefore this heuristic is built around conventional scheduling parameters i.e., deadlines of jobs. A global runnable job queue is maintained by the heuristic and jobs are sorted in a non-decreasing order by their relative deadlines. In alignment with our model, CCH also conforms to the read-execute-write semantics. It schedules the memory access by maintaining a list of currently available free memory access intervals.

The aim of this heuristic is to derive an offline schedule table that spans the Hyper-Period (HP) of all the jobs. To do so, a "current time" ($ctime_i$) variable traces the run of the algorithm from 0 up to HP on each available core i. The heuristic proceeds by assigning the job at the head of the job queue to a core when the job is ready to execute. The core is selected based on the $ctime_i$ variable, where it is assigned to the core with the smallest $ctime_i$ within the cluster, thus following a first-fit approach. After the core is assigned to a job, in order to prefetch the required data into the local memory banks (read phase), the heuristic finds the earliest free memory interval (beginning at time, say t_1, where t_1 is larger than or equal than $ctime_i$ and the release time of the job), which is large enough to complete the read job. After the read phase is completed, the execute phase proceeds and accordingly, the variable $ctime_i$ is set to $ctime_i = t_1 + C^{read} + C^{exec}$. The heuristic then finds the first free memory interval, say t_2, after the updated $ctime_i$ of the core, which is large enough to complete the write phase and $ctime_i$ is then advanced to account for the completion of the write job ($ctime_i = t_2 + C^{write}$).

If the time to complete the write part of the job exceeds its deadline, the heuristic flags a failure implying that the job-set is "unschedulable" and terminates. If all the jobs in the job-queue have been assigned to a core when (or before) all $ctime_i$ are equal to the hyper-period, the heuristic has successfully found a schedule for each of the jobs. However, if there are unscheduled jobs in the job-queue after the HP is exceeded, it flags a failure ("unschedulable") and eventually terminates.

During the evaluations several sorting policies were examined (e.g. sorting of the job queue by release time or slack time). The deadline-based heuristic presented here outperformed all other tested candidate policies, hence we only use this policy for the further comparisons.

B. Synthetic Experiments

In order to compare the proposed heuristics, we performed a sensitivity analysis by varying (i) the cluster size and (ii) application characteristics.

1) Varying the Cluster Size: The objective of this experiment is to understand the maximum utilization achievable by the proposed heuristic on a given cluster size. In this experiment, we examined runnable sets with a (5%:90%:5%) read-execute-write ratio. For a given cluster size, the simulations were initiated by considering a runnable set with low utilization. The execution times of the jobs were then incrementally inflated until the last schedulable instance was found. This utilization was recorded. The simulations were repeated by increasing the number of cores available on the cluster in order to show the implications of different cluster sizes. The cluster size was varied in the range of 1 to 14 cores. For *each data point* in the graph, 500 random runnable sets were evaluated.

The results are depicted in Fig. 4. It can be seen that the Core-Centric Heuristic (*CCH*) has difficulties utilizing the hardware platform, even if a large number of cores are available. This is an outcome of the core-centric scheduling.

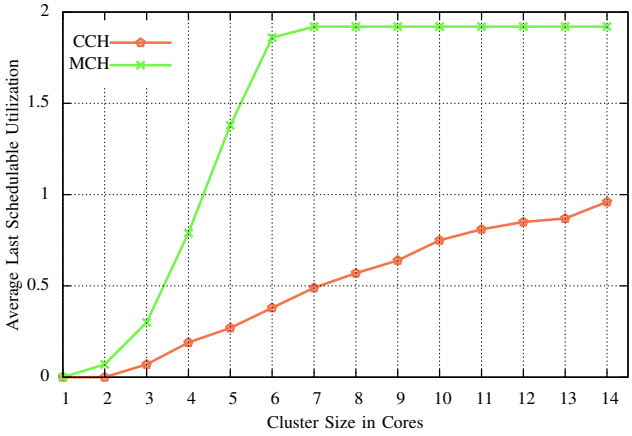

Fig. 4: Average schedulability of the Memory-Centric Heuristic (MCH) and the Core-Centric Heuristic (CCH) for task sets with with a (5%:90%:5%) read-execute-write ratio of each task, when the cluster size is changed.

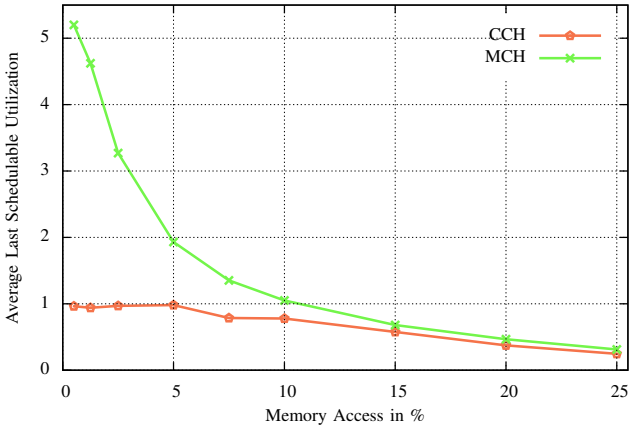

Fig. 5: Average last schedulable utilization of the Memory-Centric Heuristic (MCH) and the Core-Centric Heuristic (CCH) with different percentages of memory access relative to the execution time.

Jobs are assigned to cores and they access the next free memory slot for their read and write access, making the memory access uncoordinated between the different cores, resulting in large stall times. The Memory-Centric Heuristic (*MCH*) outperforms CCH for all cluster sizes. The benefits of scheduling the scarce resource, i.e. the memory, become clearly visible with this approach. It can also be seen that the proposed heuristic reaches saturation once the cluster size reaches 7 cores. At this point the memory schedule is filled densely, which hinders the inflation of the memory phases further. Consequently, inflating the task set further leads to an unschedulable systems.

2) Changing the Job Characteristics: Given the read-execute-write semantics of the underlying application, the objective of the experiment was to observe how the heuristic utilizes the available platform when the execution-to-memory access ratios are varied in the runnable set. For these experiments, we generate random runnable sets as described earlier. The cluster size is fixed to 14 cores and 2 label banks. The ratio of the memory access phase in comparison to the jobs execution phase is varied. Note that a memory access ratio of 5% means that the read and write phase *each* comprise 5% of the generated execution time of UUniFast for the respective runnable.

To observe the impact, the memory access ratio is varied between 0.5% and 25%. The results are presented in Fig. 5. It can be seen that the maximum average schedulable utilization drops for *both* heuristics as the read and write phases of the jobs increase. The main reason behind this drop in utilization is the growing granularity of memory accesses that need to be scheduled, i.e. the access to memory itself grows and the memory fragmentation increases, which results in inefficient use of a scarce shared resource.

A value of 25% hence means that half the execution time of the job is spent on accessing the shared memory, which is a single resource shared among all cores. This means, even an optimal algorithm that utilizes the memory 100% yields only a core utilization of 200%. This is because no other runnable can utilize the core during the memory access of the read and write portion. Further, the results show clear benefits of MCH compared to CCH when the memory access phase becomes small. The way the memory accesses are scheduled for CCH leads to fragmentation, which in turn renders the runnable sets unschedulable, since the heuristic cannot find a continuous free slot of the memory access size in the memory schedule. MCH, on the other hand, primarily schedule the memory, which creates an organized access to memory and in turn allows for higher utilization of the system.

3) Quality of the Heuristic Solution: The memory-centric algorithm presented in Section V is a heuristic and therefore may not yield optimal solutions. In order to compare the quality of this heuristic against the optimal case, we generate small runnable sets of the size solvable by the ILP formulation in reasonable time.

We use the same method as described in Section VI-A1 to generate our sample runnable sets. The runnable sets included two runnables of period 100 ms, three runnables of period 20 ms, three runnables of period 10 ms, and one runnable of period 50 ms. This distribution of periods is representative for the automotive domain and in line with [37]. The memory access times are again distributed between read, execute, and write phase in the ratio 5%:90%:5%. The CPLEX optimizer [38] version 12.6.2 was used to solve the ILP formulations, and all experiments were performed on a system containing an Intel i5 CPU (2 cores at 2.7 GHz), and 8GB of RAM. The results present the average values out of 100 randomly generated runnable sets. The utilization of a generated runnable set is then increased and we record the last schedulable utilization value for each algorithm.

The results are presented in Table 1. The first column presents our proposed memory-centric heuristic, the baseline heuristic, and the ILP solution. Column 2 presents the average time for each solution to find a schedule for a schedulable instance. If no solution was found the time was excluded from the average solving time. The third column presents the average Last Schedulable Utilization (LSU) of

TABLE I: Comparison of solving time, last schedulable utilization, and improvement of MCH for the different methods when 14 cores are available.

Algorithm	Average Solving time	LSU	MCH vs. X
CCH	35 ms	276%	67%
MCH	130 ms	463%	0%
ILP	21087ms	486%	-0.5%

the two heuristics and the ILP solution. Finally, the last column presents the improvements of solutions compared to our proposed heuristic. The improvement is calculated as $(MCH\ vs.\ X) \stackrel{def}{=} \frac{LSU_{MCH} - LSU_X}{LSU_X}$.

For 4 out of the 100 runnable sets the memory-centric heuristic finds a larger LSU than the ILP. This is the case because the ILP was aborted if no solution was found within 2 hours. For 24 out of the 100 runnable sets both memory-centric heuristic and ILP, return with the same LSU.

From the data, we can see that there is a tradeoff between the solution efficiency and the computation time of the algorithms. The increased time to arrive at a solution for MCH compared to CCH can be explained by the additional need to manage the different queues used in the heuristic. Considering the improvements of MCH over the baseline heuristic, it is seen that MCH clearly outperforms the baseline heuristic. However, the heuristic performs marginally worse (0.5%) than the optimal solution found by the ILP. This is a reasonable cost for scaling to industrially-sized applications. We will demonstrate this capability in a case study in the following section.

C. Case Study

The application in this case study is a software for Engine Management Systems (EMS), which is one of the most complex ECU's in a car. A modern EMS involves 40-50 sensors and multiple actuators [40]. The associated software spans around 2000 modules (atomic SW components), more than 5000 source files and half a million lines of C-code. Depending on different variants, nearly 2000-4000 runnables communicate over more than 20000-60000 data labels. The memory footprint is approximately 1.5 MB to 2.5 MB of program flash, and 750 KB to 1.5 MB of RAM, together with around 250 KB of calibration data.

The parameters of the hardware platform are chosen based on the Kalray MPPA-256 Bostan as presented in [35], [33]. Each local memory bank has a capacity of 128 KB. Without contention, access to local memory banks has a 9 cycle latency with 8 bytes fetched on each consecutive cycle. Access to off-chip memory is more expensive having a 55 cycle latency after which 4 bytes of data are fetched on each consecutive cycle. The memory and compute cores are clocked at 400 MHz. All numbers are in line with measurements performed on the Kalray hardware platform.

The EMS application used in this case study contains 2000 runnables and 50000 shared labels. The access to labels is divided into *read*, *write*, and *read-write* access with a partitioning of 40%, 10%, and 50% respectively [37].

The proposed memory-centric heuristic (MCH) successfully finds a time-triggered schedule for this application in 52 minutes. The generated schedule utilizes all 14 cores with a per core utilization between 23.3% and 26.1%, while the memory has a utilization of 26.4%.

The core-centric heuristic, on the other hand, fails to find a solution. By changing the system frequency to 3200 MHz, this heuristic manages to find a schedulable solution. Similarly, a schedulable solution is found if the number of cores available for execution within the cluster is increased to 93. While it is possible to change the platform parameters such that the core-centric heuristic finds a solution, these changes are severe (scaling the system frequency by a factor of 8 or the number of cores by a factor of 6.64), which in turn leads to a heavily under utilized system.

The ILP formulation does not terminate in reasonable time. Even for a reduced job set of 1000 jobs (out of the 171631 jobs which need to be scheduled within one hyperperiod) no solution is found within the first 64 hours.

These observations also bring forth the strengths of the proposed memory-centric heuristic and demonstrate its applicability to real-world problem scenarios.

VII. Conclusions

This paper presents a contention-free execution framework on a clustered many-core platform, tailored for automotive applications. One of the main issues for real-time applications on such platforms is the large number of possible sources for interference on the path to shared memory. The execution framework presented in this paper privatizes memory resources to allow contention-free access during the execution of runnables. Access to shared memory resources is done at the beginning and end of the job. Orchestrating these memory accesses hence becomes the main challenge, considering the large number of shared labels commonly found in automotive applications. We present an ILP formulation to generate a time-triggered schedule, taking the runnable-to-core mapping as well as the access to shared memory into account. A heuristic solution, where the time-triggered schedule is constructed based on the memory accesses, is furthermore presented. We experimentally evaluate the execution framework as well as the proposed heuristic. The results show that the heuristic provides near-optimal solutions (within 0.5%), while requiring only a fraction of the time required by the ILP. Further experiments confirm that the bottleneck in such architectures is the memory and not the compute resources. Finally, we demonstrate that the approach is applicable to industrial-sized problems – and computes a solution within acceptable time for an Engine Management System with 2000 runnables and 50000 labels on the cluster.

Acknowledgment

The work presented in this paper was partially supported by the Swedish Knowledge Foundation via the research project PREMISE and by National Funds through FCT/MEC (Portuguese Foundation for Science and Technology) and co-financed by ERDF (European Regional Development Fund) under the PT2020 Partnership, within the CISTER Research Unit (CEC/04234), and also by FCT/MEC and the EU ARTEMIS JU within project ARTEMIS/0001/2013 - JU grant nr. 621429 (EMC2).

REFERENCES

[1] Freescale, "Future advances in body electronics," 2013. [Online]. Available: http://cache.freescale.com/files/automotive/doc/white_paper/BODYDELECTRWP.pdf

[2] Roland Berger Strategy Consultants, "Need for consolidation in vehicle electronics," 2015. [Online]. Available: http://www.greencarcongress.com/2015/07/20150729-berger.html

[3] R. Grave, "Software integration challenge multi-core experience from real world projects," in *Embedded Multi-core Conference,Munich Germany*, 2015.

[4] D. Reinhardt and M. Kucera, "Domain controlled architecture-a new approach for large scale software integrated automotive systems." in *Proceedings of the 3rd Conference on Pervasive and Embedded Computing and Communication Systems (PECCS)*, 2013, pp. 221–226.

[5] B. D. de Dinechin, D. van Amstel, M. Poulhiès, and G. Lager, "Time-critical computing on a single-chip massively parallel processor," in *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*, 2014, pp. 97:1–97:6.

[6] *Intel Single Chip Cloud Computer*, last access October 2015, available at www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/intel-labs-single-chip-cloud-article.pdf.

[7] *Tile Processor Architecture Overview for the TILEPro Series*, last access October 2015, available at http://www.tilera.com/scm/docs/UG120-Architecture-Overview-TILEPro.pdf.

[8] D. Dasari, B. Akesson, V. Nelis, M. Awan, and S. Petters, "Identifying the sources of unpredictability in COTS-based multicore systems," in *Proceedings of the 8th IEEE International Symposium on Industrial Embedded Systems (SIES)*, 2013, pp. 39–48.

[9] S. Schliecker and R. Ernst, "Real-time performance analysis of multiprocessor systems with shared memory," *ACM Transactions in Embedded Computing Systems*, vol. 10, pp. 22:1–22:27, 2011.

[10] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele, "Worst case delay analysis for memory interference in multicore systems," in *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*, 2010, pp. 741–746.

[11] D. Dasari and V. Nelis, "An analysis of the impact of bus contention on the WCET in multicores," in *Proceedings of the 9th IEEE International Conference on Embedded Software and Systems (HPCC-ICESS)*, 2012, pp. 1450 –1457.

[12] S. Chattopadhyay, L. K. Chong, A. Roychoudhury, T. Kelter, P. Marwedel, and H. Falk, "A unified WCET analysis framework for multicore platforms," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 4s, pp. 124:1–124:29, Apr. 2014.

[13] G. Giannopoulou, N. Stoimenov, P. Huang, L. Thiele, and B. D. de Dinechin, "Mixed-criticality scheduling on cluster-based manycores with shared communication and storage resources," *Real-Time Systems*, 2015.

[14] B. D. de Dinechin, Y. Durand, D. van Amstel, and A. Ghiti, "Guaranteed services of the NoC of a manycore processor," in *Proceedings of the International Workshop on Network on Chip Architectures (NoCArc)*, 2014, pp. 11–16.

[15] B. Nikolić, P. M. Yomsi, and S. M. Petters, "Worst-case memory traffic analysis for many-cores using a limited migrative model," in *Proceedings of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2013, pp. 42–51.

[16] B. Nikolić and S. M. Petters, "Real-time application mapping for many-cores using a limited migrative model," *Real-Time Syst.*, vol. 51, no. 3, pp. 314–357, 2015.

[17] A. Schranzhofer, J.-J. Chen, and L. Thiele, "Timing analysis for TDMA arbitration in resource sharing systems," in *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2010, pp. 215–224.

[18] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley, "A predictable execution model for COTS-based embedded systems," in *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2011, pp. 269–279.

[19] G. Yao, R. Pellizzoni, S. Bak, E. Betti, and M. Caccamo, "Memory-centric scheduling for multicore hard real-time systems," *Real-Time Systems*, vol. 48, no. 6, pp. 681–715, 2012.

[20] A. Alhammad and R. Pellizzoni, "Schedulability analysis of global memory-predictable scheduling," in *Proceedings of the 14th International Conference on Embedded Software (EMSOFT)*, 2014, pp. 20:1–20:10.

[21] A. Alhammad, S. Wasly, and R. Pellizzoni, "Memory efficient global scheduling of real-time tasks," in *Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2015, pp. 285–296.

[22] P. Burgio, A. Marongiu, P. Valente, and M. Bertogna, "A memory-centric approach to enable timing-predictability within embedded many-core accelerators," in *Proceedings of the CSI Symposium on Real-Time and Embedded Systems and Technologies (RTEST)*, 2015, pp. 1–8.

[23] H. R. Faragardi, B. Lisper, K. Sandstrom, and T. Nolte, "A communication-aware solution framework for mapping autosar runnables on multi-core systems," in *Proceedings of the IEEE International Conference on Emerging Technology and Factory Automation (ETFA)*, 2014, pp. 1–9.

[24] A. Monot, N. Navet, B. Bavoux, and F. Simonot-Lion, "Multisource software on multicore automotive ECUs - combining runnable sequencing with task scheduling," *IEEE Transactions on Industrial Electronics*, vol. 59, no. 10, pp. 3934–3942, Oct 2012.

[25] M. Panić, S. Kehr, E. Quiñones, B. Boddecker, J. Abella, and F. J. Cazorla, "Runpar: An allocation algorithm for automotive applications exploiting runnable parallelism in multicores," in *Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis (CODES)*, 2014, pp. 29:1–29:10.

[26] P. Dziurzanski, A. K. Singh, L. S. Indrusiak, and B. Saballus, "Benchmarking, system design and case-studies for multi-core based embedded automotive systems," in *2nd International Workshop on Dynamic Resource Allocation and Management in Embedded, High Performance and Cloud Computing (DREAMCloud)*, 2016.

[27] S. E. Saidi, S. Cotard, K. Chaaban, and K. Marteil, "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*, 2015, p. 6.

[28] M. Lukasiewycz, R. Schneider, D. Goswami, and S. Chakraborty, "Modular scheduling of distributed heterogeneous time-triggered automotive systems," in *Proceedings of the 17th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2012, pp. 665–670.

[29] W. Puffitsch, E. Noulard, and C. Pagetti, "Off-line mapping of multi-rate dependent task sets to many-core platforms," *Real-Time Systems*, vol. 51, no. 5, pp. 526–565, 2015.

[30] M. Lukasiewycz and S. Chakraborty, "Concurrent architecture and schedule optimization of time-triggered automotive systems," in *Proceedings of the 8th International Conference on Hardware/software codesign and system synthesis (CODES)*, 2012, pp. 383–392.

[31] S. Wildermann, M. Glaß, and J. Teich, "Multi-objective distributed run-time resource management for many-cores," in *Proceedings of the conference on Design, Automation & Test in Europe (DATE)*, 2014, p. 221.

[32] F. Reimann, M. Lukasiewycz, M. Glass, C. Haubelt, and J. Teich, "Symbolic system synthesis in the presence of stringent real-time constraints," in *Proceedings of the 48th Design Automation Conference (DAC)*, 2011, pp. 393–398.

[33] S. Saidi, R. Ernst, S. Uhrig, H. Theiling, and B. D. de Dinechin, "The shift to multicores in real-time and safety-critical systems," in *Proceedings of the 10th International Conference on Hardware/Software Codesign and System Synthesis (CODES)*, 2015, pp. 220–229.

[34] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand, "Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 7, pp. 966–978, 2009.

[35] D. Kanter and L. Gwennap, "Kalray clusters calculate quickly," *Microprocessor Report*, 2015.

[36] *AUTOSAR - Specification of RTE*, AUTOSAR Std. 4.2.2, 2014.

[37] S. Kramer, D. Ziegenbein, and A. Hamann, "Real world automotive benchmarks for free," *6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2015.

[38] IBM, "Ilog cplex optimizer," 2016. [Online]. Available: http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/

[39] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems Journal*, vol. 30, no. 1-2, pp. 129–154, 2005.

[40] D. Claraz, S. Kuntz, U. Margull, M. Niemetz, and G. Wirrer, "Deterministic execution sequence in component based multi-contributor powertrain control systems," in *Proceedings of the Embedded Real Time Software and Systems Conference (ERTS)*, 2012, pp. 1–7.