Scheduling Multi-Rate Real-Time Applications on Clustered Many-Core Architectures with Memory Constraints

Matthias Becker[1], Saad Mubeen[1], Dakshina Dasari[2], Moris Behnam[1], Thomas Nolte[1]
[1]MRTC/Mälardalen University, Västerås, Sweden
{matthias.becker, saad.mubeen, moris.behnam, thomas.nolte}@mdh.se
[2]Corporate Research, Robert Bosch GmbH, Renningen, Germany
dakshina.dasari@de.bosch.com

Abstract— Access to shared memory is one of the main challenges for many-core processors. One group of scheduling strategies for such platforms focuses on the division of tasks' access to shared memory and code execution. This allows to orchestrate the access to shared local and off-chip memory in a way such that access contention between different compute cores is avoided by design. In this work, an execution framework is introduced that leverages local memory by statically allocating a subset of tasks to cores. This reduces the access times to shared memory, as off-chip memory access is avoided, and in turn improves the schedulability of such systems. A Constraint Programming (CP) formulation is presented to select the statically allocated tasks and to generate the complete system schedule. Evaluations show that the proposed approach yields an up to 19% higher schedulability ratio than related work, and a case study demonstrates its applicability to industrial problems.

I. Introduction

Many-core processors are becoming increasingly relevant for industrial domains, such as the automotive, or avionics domain [1, 2]. The vast availability of massive computational power on many-core platforms is welcomed by industry, as industrial systems move from distributed to integrated architectures, where multiple applications are consolidated on the same compute platform (for example Domain Controller in the automotive industry [3] or Integrated Modular Avionics (IMA) for the avionics industry [4]). Clustered many-core platforms [5], where a number of compute cores build a subsystem with local memory, are candidates to consolidate applications as they provide isolated islands of computation [3, 6].

The transition to integrated architectures is driven by the increasing number, and complexity of todays applications which are often subject to stringent timing requirements. Such timing requirements manifest themselves not only through the well-known constraints on tasks' response times, but also on timing constraints on the data propagation delay through chains of tasks [7, 8, 9, 10].

Executing these applications on a many-core platform is not trivial, as contention on shared resources can influence the Worst-Case Execution Time (WCET) of tasks that are executing in parallel on different cores. Accounting for the worst-case access time to shared resources is generally too pessimistic (overestimated) due to the large number of contenders on these platforms [11, 6, 12]. Several strategies exist to minimize or remove the contention on shared resources. The focus of this work is on the execution frameworks that schedule tasks in a way such that contention on shared resources is avoided by design.

Typically, local memory is a scarce resource on many-core processors, and embedded systems in general. For example, one cluster of the Kalray MPPA® many-core processor hosts 2 MB of local memory, while, the software of an automotive Engine Management System (EMS) has a total memory footprint of up to 4 MB [6]. This highlights the need to access off-chip memory during the execution. The Contention-Free Execution Framework (CEF) [6] is based on time-triggered non-preemptive scheduling. Each core is assigned a private memory bank, and access to local and off-chip memory is exclusive between all cores of a cluster. In the CEF, for each execution, the code of a task is loaded into the core's private memory bank, i.e. only one tasks' footprint is present in a core's memory bank at a time. Scheduling all memory accesses is one of the main challenges and was identified as the bottleneck in [6].

This work builds on the main principles of the CEF. It is observed that, by *statically allocating selected tasks* footprint to the local memory banks, the core's local memory banks are utilized more efficiently and the WCET of exclusive memory accesses that need to be scheduled is reduced, as the code of statically allocated tasks does not need to be preloaded from off-chip memory.

The main contributions of this work are:

- An execution model that statically allocates a subset of tasks to a cores' local memory, which increases the usage of local memory while, at the same time, reduces the expensive access to off-chip memory.
- We ensure that data propagation delay constraints are met, as the approach considers specified job-level dependencies that impose a partial ordering on the tasks jobs.
- A Constraint Programming (CP) formulation for the task mapping together with the generation of the time-triggered schedule is presented.
- Evaluations demonstrate an improvement in schedulability ratio of 19% compared to the CEF. A case study further demonstrates the applicability to an automotive benchmark application with data propagation delay constraints.

The rest of the paper is organized as follows. Related work is discussed in Section II. In Section III the rules of the CEF are introduced. Section IV presents the system model. The memory aware execution framework is presented in Section V, followed by Section VI which discusses the system and schedule generation. Evaluations are presented in Section VII, and conclusions are drawn in Section VIII.

II. Related Work

The study of execution strategies for real-time workload on a many-core processor is receiving growing attention due to their possible benefits for industry [3, 13, 14, 15]. One of the main challenges faced on many-core platforms is the access to shared resources, such as memory [11, 16, 6].

Schranzhofer et al. examine different execution models and show that best results are obtained when access to shared memory is confined to the beginning and end of a jobs execution [16]. Such an execution model is for example used in AUTOSAR [8], in the PRedictable Execution Model (PREM) [17], or in the CEF [6].

Several works focus on multi- and many-core execution frameworks that are based on time-triggered execution [18, 1, 2, 6, 12]. Integer Linear Programming (ILP) formulations are used in [19, 6]. Due to scalability issues an alternative heuristic solution is presented in [6]. Constraint Programming frameworks scale better than ILP frameworks, and results are reported that scale to industrial sized applications [18, 2]. Becker et al. [6] consider code pre-fetching from off-chip memory, while an application is mapped to one compute cluster only. Puffitsch et al. [18], and Perret et al. [1, 2] consider mapping of applications onto several tiles of a many-core processor where communication is handled over the Network-on-Chip (NoC).

Precedence constraints are considered in [18, 1, 2]. Such constraints are necessary to meet timing constraints on the data propagation through a chain of independently triggered tasks that can execute at different periods [8, 7, 9, 10]. In [9, 10], it is shown how to translate timing constraints on the data propagation delay to a set of precedence constraints between tasks that potentially execute at different periods.

The method presented in this paper is different from the above in the sense that the task code can be mapped to local or off-chip memory, depending on the execution of tasks jobs. The access time to shared memory then depends on this memory locality.

III. Recapitulation of the Contention-Free Execution Framework

This section recapitulates the Contention-Free Execution Framework (CEF) that is presented in [6]. The framework is build on three main concepts that address the execution of independent real-time tasks on clustered platforms under the presence of shared resources. In the following all three principles are discussed in detail.

Memory Bank Privatization:
With memory bank privatization, each compute core is statically assigned one of the clusters local memory banks. A compute core can only access this private memory bank, thus there is no interference in the access to memory on this bank. The global copies of all communication variables are located on a distinct memory bank within the cluster, the *global bank*. This memory bank can be accessed by all compute cores.

Read-Execute-Write Semantic:
An application's memory requirement can exceed the available local memory. To tackle this shortcoming, the application's code is located outside the compute cluster in (remote) global memory and dynamically loaded to the compute core's private memory bank before the execution of the task's job. This reduces the memory requirements for the compute cluster and also allows to schedule consecutive jobs of the same task on different compute cores.

The read-execute-write semantic is used and extended to prefetch the task's code in addition to creating copies of all input variables that are then stored in the core's private memory bank. During the prefetching step, the task's code is loaded from the remote memory into the private memory bank of the core. Fig. 1 visualizes the execution of one job in the framework, where the different memory access phases are shown and mapped to the read, execute, and write phase.

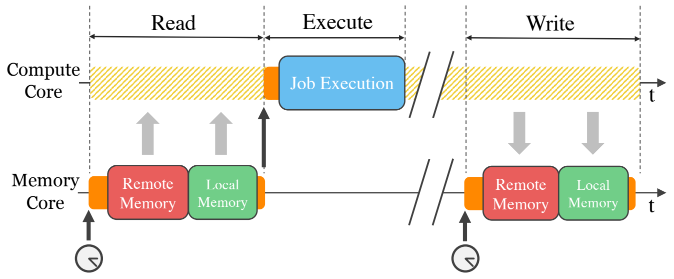

Fig. 1.: Execution of one job divided into its memory and execution phases.

Time-Triggered Schedule:
The previously described concepts lead to a contention-free execution during the execute phase of each task. Contention is still possible when global memory is accessed (during prefetching of code and reading/writing to shared variables). This contention is resolved by a time-triggered schedule that orchestrates the access to shared resources in a way such that no read- or write-phase of any two tasks executions overlaps with each other (see Fig. 2). This schedule is executed by a core on the platform that is solely dedicated for managing the access to the shared memory resources.

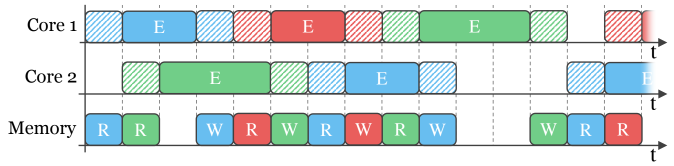

Fig. 2.: Example of three tasks that are executed on two cores. Read, execute, and write phases of the different jobs are marked by R, E, and W.

IV. System Model

A. Application Model

An application is defined as the set of non-preemptive periodic dependent tasks Γ, together with the set of job-level dependencies Θ.

A task τ_i is represented by the tuple $\{C_{i,E}, \overrightarrow{C_{i,R}}, \overrightarrow{C_{i,W}}, T_i, S_i\}$. In this model, the tasks execution is divided into a read, execute, and write phase. The vectors $\overrightarrow{C_{i,R}}$ and $\overrightarrow{C_{i,W}}$ represent the time required to access shared memory before and after the tasks execution and depend on the tasks type (a detailed explanation is provided in Section V). The execution time of the task on a compute core is independent of the task's type and stays constant with $C_{i,E}$ time units. The function $\min(\overrightarrow{C_i})$ returns the minimum total execution time of the task (i.e., the sum of $C_{i,R}+C_{i,E}+C_{i,W}$) depending on the state. And $\max(\overrightarrow{C_i})$ returns the maximum total execution time respectively.

Each task is periodically invoked with a period of T_i time units and the deadline of each task is assumed to be equal to its period, i.e., $D_i = T_i$. The tasks do not have offset relations. The footprint of a task, denoted by S_i, represents the number of bytes that are required to store the task's code as well as private and communication variables.

The function $\mathrm{LCM}(\tau_1, \tau_2, ...\tau_i)$ returns the least common multiple of the tasks' periods. Hence, the hyperperiod of the taskset Γ is obtained by $\mathrm{LCM}(\Gamma)$.

Communication between tasks is realized by *register communication*. A register is a global variable. A sending task writes an updated value to the variable, and a receiving task reads the current value from the variable. I.e. there is no signaling between communicating tasks and the last-is-best semantic applies. To increase predictability on the communication delays, tasks communicate via the *read-execute-write* model. All communication variables with read access are copied into local variables during the read phase. The task uses the local variables during the execution phase. Finally, in the write phase, the values of the local variables are copied to the global variables with write access.

We denote the j^{th} instance of a task τ_i by job $\tau_{i,j}$. To ease the presentation we define $rel_{i,j}$ and $d_{i,j}$ as the absolute release and deadline of the job $\tau_{i,j}$. Tasks may have precedence constraints that are specified by job-level dependencies. A job-level dependency $\tau_A \xrightarrow{i,j} \tau_B \in \Theta$ specifies a partial ordering of τ_A's and τ_B's jobs. Such a dependency specifies that the job $\tau_{A,i}$ must be executed before the job $\tau_{B,j}$, where i and j refer to the jobs within $\mathrm{HP}(\tau_A, \tau_B)$ and repeat for each consecutive hyperperiod of the two tasks.

B. Platform Model

The application executes on a compute cluster that contains $m+1$ identical cores. Each core has a private memory bank that has a capacity of b bytes that is used to store task code. All cores operate on synchronous clocks. One core, designated as the *management core* in the cluster, is responsible for managing the access to shared resources, for e.g., memory that is shared among the compute cores or the memory that is external to the cluster. The management core can access external memory to copy data into the local memory banks. The remaining m cores are designated *compute cores*. They are dedicated to execute the application tasks.

Such a platform represents, for example, one compute cluster of the Kalray MPPA® many-core platform [5, 6].

V. Memory Aware Contention-Free Execution Framework

In the CEF, each job can be executed on any of the compute cores. During the read-phase, the complete code of the task is prefetched from external memory, in addition of the copies that are created for communication variables. In [6] it is shown that the size of read and write phases has a large influence on the system schedulability, as memory is an exclusive resource. Hence, if the memory cannot accommodate the required read and write phases in a timely manner, the system is deemed to be unschedulable, even if enough computational power is available, in terms of compute cores (since the cores stall waiting for data).

Although the above method works reasonably it should be noted that the cores' local memory banks effectively host only the memory that is required by one task at any given time. Typically the required memory is small in comparison to the memory bank size. As an example, the automotive application presented in a case study by Dziurzanski et al. [15] reports individual footprints in the range of 7 KB to 17 KB, while one memory bank on a compute cluster of the Kalray MPPA® has a capacity of 128 KB. Thus, less than 15% of each of the core local memory banks are utilized in CEF.

In the following, the rules of the Memory Aware Contention-Free Execution Framework (MCEF) are discussed, that are additionally specified to the mechanisms described for the CEF in Section III. The MCEF distinguishes tasks as *static* and *dynamic* depending on their memory usage. The following provides a definition of the two task types:

Definition 1 *A task τ_i is said to be* ***static****, if all jobs $\tau_{i,j}$ within the hyperperiod $\mathrm{LCM}(\Gamma)$ are executed on the same compute core and the required memory of the task S_i is statically allocated to the core's private memory bank.*

Definition 2 *A task τ_i is said to be* ***dynamic****, if either the jobs $\tau_{i,j}$ within the hyperperiod $\mathrm{LCM}(\Gamma)$ are executed on more than one core, or the required memory is not statically assigned to the core's private memory bank.*

Depending on the task type, the read and write phases have different sizes, which are captured by the different parameters of the task model $C_{i,R}^{static}$ and $C_{i,R}^{dynamic}$ (where $C_{i,R}^{static} \leq C_{i,R}^{dynamic}$) as well as by the parameters for the write access $C_{i,W}^{static}$ and $C_{i,W}^{dynamic}$ (where $C_{i,W}^{static} \leq C_{i,W}^{dynamic}$). The different access times result from the fact that, if a task is static, task private data and code do not need to be prefetched from off-chip memory, as it is already available on the core's memory bank. Hence, the expensive access to external memory is not required.

In contrast to the CEF which only schedules independent periodic tasks, the MCEF considers specified job-level dependencies during the generation of the schedule. This is an important improvement, as applications in many areas are subject to data propagation delay constraints in addition to the deadline constraints on the response time of each individual task [7, 9, 10].

In the next section, we propose a method using constraint programming which computes a time-triggered schedule that orchestrates read, execute, and write phases of each job of the applications tasks, such that individual deadlines and specified job-level dependencies are met. The approach further allocates the application's tasks statically or dynamically in the cluster.

VI. Schedule Generation

This section describes the CP formulation that is used to assign an execution type (i.e., static or dynamic) to each task, and generate the time-triggered schedule for the MCEF. We use CP with *conditional time-intervals* to create the time-triggered schedule of the execution framework. These variable types are designed for constraint based scheduling problems [20, 21], where they represent jobs that may or may not be executed by the final schedule. The state-of-the-practice CP solver IBM ILOG CP Optimizer directly supports this variable type which allows to efficiently solve scheduling problems in contrast to conventional CP formulations [22, 2].

A. Decision Variables

In the given scheduling problem three types of decision variables need to be solved by the CP solver.

A.1 Task Jobs as Conditional Time-Intervals

A conditional time-interval variable $\underline{a}$ defines an activity that should be scheduled within a predefined interval. Let $\mathrm{s}(\underline{a})$ be the start of the interval and $\mathrm{e}(\underline{a})$ its end. $\mathrm{l}(\underline{a})$ returns the duration of the interval. The function $\mathrm{x}(\underline{a})$ describes if the interval is present in the solution or not (i.e. it is 1 or 0) [20].

A task's job $\tau_{i,k}$ can be represented by the tuple $\{s, f, l\}$, where the start of the jobs' execution is defined by $s \in [rel_{i,k}, d_{i,k} - \min(\overrightarrow{C_i})]$, the finishing time of the jobs' execution is defined by $f \in [rel_{i,k} + \min(\overrightarrow{C_i}), d_{i,k}]$, and the length of the execution is defined by $l \in [\min(\overrightarrow{C_i}), T_i]$. Hence, the job is not allowed to start before its release time $rel_{i,k}$, and it must end latest at its deadline $d_{i,k}$. In any schedule, the duration of the job must be at least the length of the minimum execution time.

Lets define J_{τ_i} as the set of jobs containing all jobs for the task τ_i during the hyperperiod $\mathrm{LCM}(\Gamma)$ of the taskset. The jobs to be scheduled on one core can be defined as $J_\Gamma = \{J_{\tau_i} | \tau_i \in \Gamma\}$. As a job can be scheduled on any of the m cores, the set J_Γ is defined for each core, which we denote as J_Γ^c, for the jobs that execute on core $c \in [1, m]$. To ease the presentation we denote the job $\tau_{i,j}$ on core c as $\tau_{i,j}^c$.

A.2 Static Tasks

The decision variable $\delta_i \in [0, 1]$ models the type of a task $\tau_i \in \Gamma$. If this variable has the value 0 the task is dynamic, and if the variable has the value 1 the task is static.

A.3 Memory Access

Depending on the type δ_i of a task τ_i, the memory access at the beginning and at the end of one job varies. In order to represent the exact length of this memory access, depending on the tasks type, two variables are introduced for each task.

The length of the task's read phase is represented by $\phi_i \in [C_{i,R}^{static}, C_{i,R}^{dynamic}]$ and the write phase by $\psi_i \in [C_{i,W}^{static}, C_{i,W}^{dynamic}]$.

B. Constraint Formulation

Four types of constraints are specified. The basic scheduling constraints, together with the global memory access constraints model basic parts of the framework. The local memory bank constraints model the dynamic and static tasks, including their memory allocation, and finally dependencies between tasks are modeled by the job-level dependency constraints.

B.1 Basic Scheduling Constraints

The multi-core scheduling problem is modeled by one time-interval variable for each job on each core. Exactly one of these possible jobs must be present in the final schedule, as each job can only be executed once. The operator $\mathrm{alt}(\underline{a}, \{\underline{a}_1, ..., \underline{a}_n\})$ provides a constraint that states, if interval variable $\underline{a}$ is present in the solution, exactly one of the possible interval variables $\{\underline{a}_1, ..., \underline{a}_n\}$ is present [20]. a then inherits the start and end times from this interval variable. The first constraint specifies that for each job exactly one of the possible jobs is present in the final solution. This selected job is represented by $\tau_{i,j}$:
$\forall \tau_i \in \Gamma \;\; \forall \tau_{i,j} \in J_{\tau_i}$

$$\mathrm{alt}(\tau_{i,j}, \{\tau_{i,j}^c | c \in [1, m]\})$$

The operator $\mathrm{noOverlap}(\underline{a}_1, ..., \underline{a}_n)$ provides a constraint that states that the execution of the interval variables $\{\underline{a}_1, ..., \underline{a}_n\}$ do not overlap [21]. As each job is executing non-preemptively, the following constraint specifies that jobs on the same core are not allowed to overlap in their execution:
$\forall c \in [1, m]$

$$\mathrm{noOverlap}(\tau_{i,j} \in J_\Gamma^c)$$

B.2 Global Memory Access Constraints

Access to the shared memory is exclusive between all tasks. The timed interval variables represent the complete interval the core is busy in executing all phases of one job (i.e. read, execute, and write). With these constraints, it is guaranteed that only one task accesses the shared memory at a time. Note that the length of the read and write phase is itself a decision variable that depends on the type of the task a job belongs to.

Several operators are available to specify constraints that impact the temporal ordering of time intervals [20]. The $\mathrm{startBeforeStart}(\underline{a}, \underline{b}, t)$ constraint specifies that $\underline{a}$ must start at least t time units before $\underline{b}$ starts: $\mathrm{s}(\underline{b}) - \mathrm{s}(\underline{a}) \geq t$. The $\mathrm{startBeforeEnd}(\underline{a}, \underline{b}, t)$ constraint specifies that $\underline{a}$ must start at least t time units before $\underline{b}$ ends: $\mathrm{e}(\underline{b}) - \mathrm{s}(\underline{a}) \geq t$. The $\mathrm{endBeforeEnd}(\underline{a}, \underline{b}, t)$ constraint specifies that $\underline{a}$ must end at least t time units before $\underline{b}$ ends: $\mathrm{e}(\underline{b}) - \mathrm{e}(\underline{a}) \geq t$. The $\mathrm{endBeforeStart}(\underline{a}, \underline{b}, t)$ constraint specifies that $\underline{a}$ must end at least t time units before $\underline{b}$ starts: $\mathrm{s}(\underline{b}) - \mathrm{e}(\underline{a}) \geq t$.

To achieve exclusive access to the shared memory, constraints are specified between all jobs that are scheduled within the hyperperiod. It can be observed that memory access intervals of two jobs can only have a potential conflict if the job's execution intervals (i.e. the time between the job's release and its deadline) overlap. Two jobs $\tau_{i,j}$, and $\tau_{o,p}$ do not overlap if: $d_{o,p} \leq rel_{i,j} \vee d_{i,j} \leq rel_{o,p}$. Thus, the negation of this statement must be true in order to specify the constraints between these two tasks.

This constraint models the maximum distance between the start of the two jobs' read phases such that they are at least ϕ_i or respectively ϕ_o time units apart:
$\forall \tau_{i,j}, \tau_{o,p} \in J_\Gamma,\ \tau_{i,j} \neq \tau_{o,p},\ \neg(d_{o,p} \leq rel_{i,j} \vee d_{i,j} \leq rel_{o,p})$

$$\begin{aligned} & \text{startBeforeStart}(\tau_{i,j}, \tau_{o,p}, \phi_i) \\ \vee\ & \text{startBeforeStart}(\tau_{o,p}, \tau_{i,j}, \phi_o) \end{aligned}$$

The same reasoning applies for the jobs write phases that must be ψ_i or respectively ψ_o time units apart:
$\forall \tau_{i,j}, \tau_{o,p} \in J_\Gamma,\ \tau_{i,j} \neq \tau_{o,p},\ \neg(d_{o,p} \leq rel_{i,j} \vee d_{i,j} \leq rel_{o,p})$

$$\begin{aligned} & \text{endBeforeEnd}(\tau_{i,j}, \tau_{o,p}, \psi_o) \\ \vee\ & \text{endBeforeEnd}(\tau_{o,p}, \tau_{i,j}, \psi_i) \end{aligned}$$

Read and write intervals are also not allowed to overlap. Thus, this constraint models that the read interval of $\tau_{i,j}$ does not overlap with the write interval of $\tau_{o,p}$. As the constraint is specified between each task, all respective combinations are covered:
$\forall \tau_{i,j}, \tau_{o,p} \in J_\Gamma,\ \tau_{i,j} \neq \tau_{o,p},\ \neg(d_{o,p} \leq rel_{i,j} \vee d_{i,j} \leq rel_{o,p})$

$$\begin{aligned} & \text{startBeforeEnd}(\tau_{i,k}, \tau_{o,p}, \phi_i + \psi_o) \\ \vee\ & \quad \text{endBeforeStart}(\tau_{o,p}, \tau_{i,j}, 0) \end{aligned}$$

B.3 Local Memory Bank Constraints

To model the memory bank constraints for static tasks we define the following constraints.

$\text{taskOnCore}(\tau_i, c)$ evaluates to $true$[1] if all jobs of τ_i are present, i.e., executing on core c:

$$\text{taskOnCore}(\tau_i, c) \models \bigwedge_{\forall \tau^c_{i,j} \in J^c_{\tau_i}} \text{x}(\tau^c_{i,j})$$

$\text{taskMapped}(\tau_i)$ evaluates to $true$, if the task τ_i is statically mapped to any of the m compute cores:

$$\text{taskMapped}(\tau_i) \models \bigvee_{\forall c \in [1,m]} \text{taskOnCore}(\tau_i, c)$$

A task can only be of type static if all its jobs execute on the same core. Note that this is only a necessary condition as a task can be of type dynamic in the case that the required memory to accommodate the task's footprint is not available on that core. The following constraint can be introduced:
$\forall \tau_i \in \Gamma$

$$\delta_i \leq \text{taskMapped}(\tau_i)$$

On each core's memory bank only b bytes can be allocated statically. The sum of the footprint of all tasks that are executing on the core and that are of type static must be less or equal to the memory capacity:
$\forall c \in [1, m]$

$$b \geq \sum_{\forall \tau_i \in \Gamma} \text{taskOnCore}(\tau_i, c) \cdot \delta_i \cdot S_i$$

The type of a task (static or dynamic) affects its total execution length. The minimum length of a timed interval $\tau_{i,j}$'s execution is $\min(\overrightarrow{C_i})$. This results if the task is static and uses the static version of the read and write phase. A dynamic task has a larger execution time $\max(\overrightarrow{C_i})$ as the dynamic read and data intervals are used. A constraint is specified that restricts the minimum length of jobs that are dynamic:
$\forall \tau_{i,j} \in J_\Gamma$

$$\delta_i = 0 \Rightarrow l(\tau_{i,j}) \geq \max(\overrightarrow{C_i})$$

B.4 Job-Level Dependency Constraints

Dependencies on the execution order of tasks jobs that are specified by job-level dependencies can be represented by the $\text{endBeforeStart}(\tau_A, \tau_B, z)$ operator without offset z. Such constraints are added for each job-level dependency and are then instantiated for the length of the task-sets hyperperiod. The number of required repetitions can be computed by $HP(\Gamma)/LCM(\tau_A, \tau_B)$.
$\forall \tau_A \xrightarrow{i,j} \tau_B \in \Theta \quad \forall k \in \left[0, \frac{LCM(\Gamma)}{LCM(\tau_A, \tau_B)} - 1\right]$

$$\text{endBeforeStart}(\tau_{A,\omega_A}, \tau_{B,\omega_B}, 0)$$

The indexes of the constraint jobs are represented by ω_A and ω_B respectively. Their calculation are shown below:

$$\omega_A = k \cdot \frac{LCM(\tau_A, \tau_B)}{T_A} + i \qquad \omega_B = k \cdot \frac{LCM(\tau_A, \tau_B)}{T_B} + j$$

C. Objective Function

The main objective of the schedule generation is to find *any* valid schedule. Thus, "**minimize** 1" is used as optimization function.

Alternatively the objective function "**maximize** $\sum_{i \in [1,n]} \delta_i$" can be used to allocate as many tasks statically as possible. This then reduces the load on the external resources (e.g. interconnect, memory controller, etc.) that may be shared between several compute clusters.

VII. Evaluation

Evaluations focus on the improvement of the MCEF compared to the CEF. We first present synthetic experiments that focus on the schedulability ratio, as well as on the solving time. Additionally, the application presented in the case study is analyzed using the presented framework.

A. Synthetic Experiments

In these experiments, the MCEF is compared against the CEF presented in [6]. In order to compare the solving time, the time-triggered schedule for both frameworks is generated by CP.

[1]Binary values are defined as $true \hat{=} 1$ and $false \hat{=} 0$, and binary variables can be used in arithmetic constraints, as possible in the IBM ILOG CP solver.

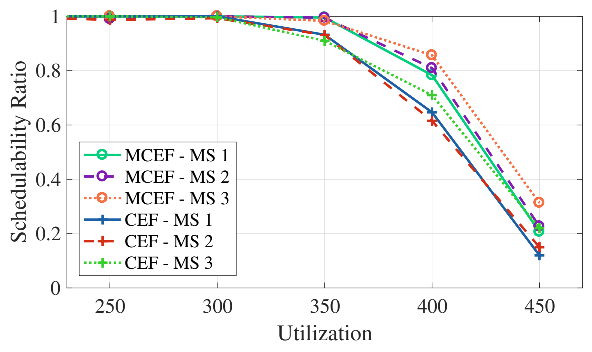

Fig. 3.: Schedulability ratio of MCEF and CEF for the 3 memory scenarios at varying utilization.

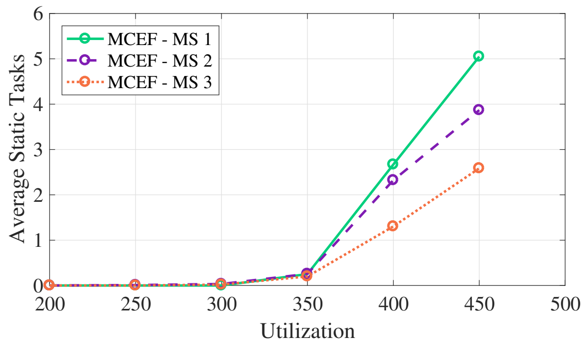

Fig. 4.: Avrg. number of statically allocated tasks in the MCEF approach at varying utilization.

The platform contains 5 compute cores and operates at a clock frequency of 400 MHz. Access to local memory takes 1 cycle to fetch 8 bytes, access to off chip memory is three times slower[2]. Each compute core's memory bank has a size of 64 KB. It is assumed that all systems can fit into the memory outside the compute cluster.

Task sets are generated randomly, where each set contains 10 tasks. Task periods are taken from the set $[1, 2, 5, 10, 20, 50, 100, 200]$ ms, this represents a subset of common periods found in the automotive industry [8]. The task utilizations are generated by UUniFast [23]. The length of the memory access regions is varied in the experiments, where 3 groups are analyzed. The footprint of a task in memory scenario 1 (MS 1) is in the range $[6, 30]$ KB, and the local data in the range of $[64, 512]$ bytes. Memory scenario 2 (MS 2) has a footprint in the range of $[6, 60]$ KB and between $[64, 1024]$ bytes of local data. Memory scenario 3 (MS 3) has a footprint in the range of $[6, 90]$ KB, and between $[64, 2048]$ bytes of local data. All values are chosen based on a uniform distribution.

For each data point 300 systems are generated. The IBM ILOG CP solver is used with a limited solving time of 10 minutes. The experiments are performed using an Intel i7 CPU (4 cores at 2.8 GHz), and 16 GB of RAM.

Fig. 3 compares the schedulability ratio (the ratio between schedulable and unschedulable systems) that is achieved by MCEF and CEF for the three described memory scenarios. For low utilizations, both approaches achieve high schedulability. Deviations can be observed for high system utilizations, where

[2]Note that this is a simplification of the memory access latencies. As the evaluations focus on the qualitative comparison of the two approaches detailed explanations of memory access latencies are omitted.

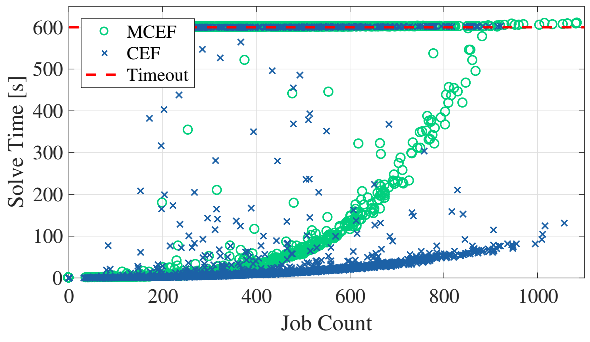

Fig. 5.: Comparison of the computation times in relation to the number of job in the schedule.

the MCEF outperforms the CEF by up to 19% for the system with memory scenario 2. Fig. 4 shows the average number of static tasks of the schedulable systems. It can be seen that, with increasing utilization, a larger number of tasks need to be allocated statically in order to find a schedulable solution. This demonstrated that, by taking advantage of local memory, the schedulability can be improved.

Fig. 5 depicts the execution times recorded during all experiments. The dashed line marks the timeout which was set for the experiments. A relation of the solving time to the number of jobs that need to be scheduled within the hyperperiod of the taskset can be seen. Solving times for the MCEF increase faster than the CEF, as the additional rules need to be taken into account during the solving process. This shows that there is a trade-off between the achieved improvements in the schedulability ratio compared to the required computation time. As a system is generated only once, this trade-off is negligible compared to the improvements that are achieved.

B. Case Study

The case study is based on an automotive engine control application that is presented in [15]. The application consists of 18 real-time tasks that communicate over 62 shared variables (793 bytes in total). The total memory required to store the task's code is 178 KB, and the task-set hyperperiod is 100 ms, in which 155 jobs need to be scheduled. In addition, three data propagation delay constraints are specified, one constrains the maximum reaction time, and two constrain the maximum data age [7] (see Fig. 7).

Platform parameters are chosen in line with the Kalray MPPA® many-core processor [5, 14]. The compute cluster has 15 compute cores, where each core has a private memory bank of 128 KB of which 64 KB can be used to store the code of static tasks. Access to off-chip is possible over a Network-on-Chip that connects to DDR3-1600 SDRAM. The system frequency is set to 400 MHz. The access time to read and write data to remote memory is computed for a NoC that is following the partitioning described in [24]. This allows to compute tighter bounds on the memory access times compared to the general NoC. To further minimize the access times, a maximum request size of 1KB is used. Requests of larger size are sent in sequence until the complete data is read or written.

The 62 data labels are allocated to the external memory or to the dedicated memory bank within the cluster. A label is

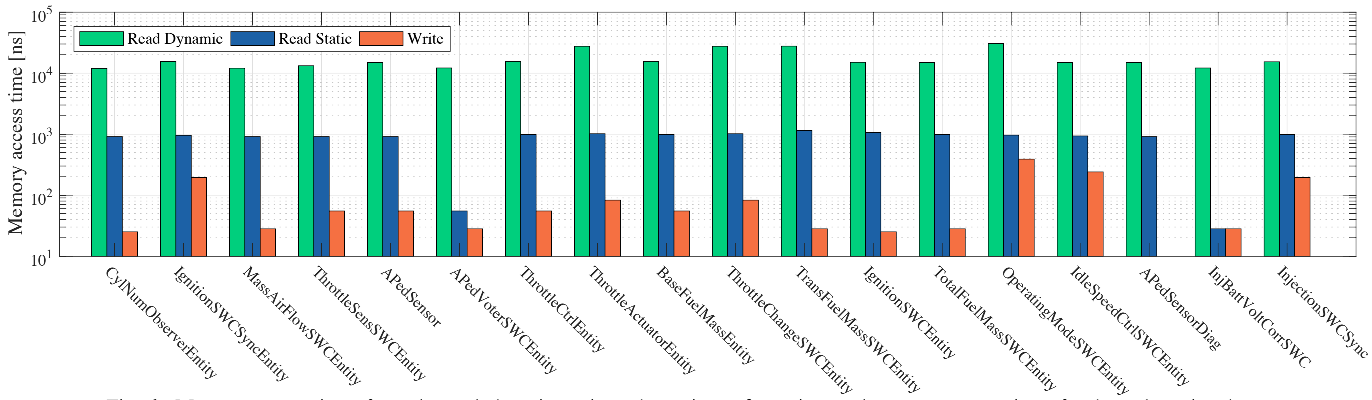

Fig. 6.: Memory access times for tasks read phase in static or dynamic configuration, and memory access times for the tasks write phase.

allocated to the external memory if it is only accessed by either write or read operations. In these cases the label is used as *input* or *output* of the application, therefore it needs to be accessed from outside the cluster. All other labels are used for internal communication by the application, hence they are mapped to the global memory bank within the cluster. In the case study, 34 labels are mapped to external memory and 28 labels are mapped to the cluster's local memory bank. A detailed description of the task's access to labels and the size of labels can be found in [15]. The resulting task parameters are presented in Table I. Note that tasks that are not periodically activated in [15] are assigned a period of 100 ms in this case study. Fig. 6 further visualizes the difference in memory access times, it can be seen that tasks face significantly longer read times if they are dynamic in contrast to their static versions due to the expensive access to external memory.

Fig. 7 visualizes the task chains that are subject to data propagation delay constraints. In order to meet all specified data propagation delay constraints, 7 job-level dependencies are generated using the methods presented in [9, 10]. These job-level dependencies are highlighted in the respective communication links between the tasks in Fig. 7.

On the described platform, the execution phases lead to a utilization of $U_{Execute} = 301,95\%$. The memory access has a utilization of $U_{Mem}^{Dynamic} = 2,67\%$ and $U_{Mem}^{Static} = 0,15\%$.

A time-triggered schedule can be generated for both settings, MCEF and CEF. The number of active cores is gradually reduced and both solutions can generate the system schedule up to a system size of 4 compute cores in $2,1$ and $2,8$ seconds respectively (average out of 100 samples). However, the data propagation delay constraints cannot be met for the CEF, where all three constraints are violated. The MCEF on the other hand takes the specified job-level dependencies into account during the schedule generation and hence satisfies all specified data propagation delay constraints. Satisfying these constraints is crucial for the industrial applicability of the approach as these constraints are an essential part to guarantee correct functionality [7, 8, 9, 10].

TABLE I
: Engine Management System – Case Study.

	C_E [μs]	$C_R^{stat.}$ [ns]	$C_R^{dyn.}$ [ns]	C_W [ns]	T [ms]	S [B]
CylNumObserverEntity	573	11970	908	25	100	6950
IgnitionSWCSyncEntity	2461	15543	958	195	100	9064
MassAirFlowSWCEntity	86	12070	908	28	5	7076
ThrottleSensSWCEntity	169	13188	908	55	5	7352
APedSensor	482	14858	908	55	5	8286
APedVoterSWCEntity	144	12140	55	28	10	7104
ThrottleCtrlEntity	2892	15413	990	55	10	8868
ThrottleActuatorEntity	2957	27593	1013	83	10	16058
BaseFuelMassEntity	2892	15413	990	55	10	8868
ThrottleChangeSWCEnt.	2957	27593	1013	83	10	16058
TransFuelMassSWCEnt.	3188	27730	1148	28	10	16058
IgnitionSCWEntity	2269	15058	1060	25	10	8348
TotalFuelMassSWCEnt.	677	14948	988	28	10	8304
OperatingModeSWCEnt.	19641	30470	965	390	20	17424
IdleSpeedCtrlSWCEntity	843	14945	933	240	20	8372
APedSensorDiag	118	14858	908	0	100	8286
InjBattVoltCorrSWC	274	12130	28	28	100	7116
InjectionSWCSync	1651	15300	985	195	100	8728

VIII. Conclusions and Future Work

This work presents a memory aware execution framework for clustered many-core platforms. As the contention on access to shared memory is one of the main challenges for such platforms a contention-free execution paradigm is applied, dividing the tasks execution into shared memory access, and execution. A non-preemptive time-triggered schedule is utilized to orchestrate the access to the shared memory. As the local memory is generally not large enough to host the complete applications, access to off-chip memory is needed. Such access impacts on the schedulability of the system due to the increased

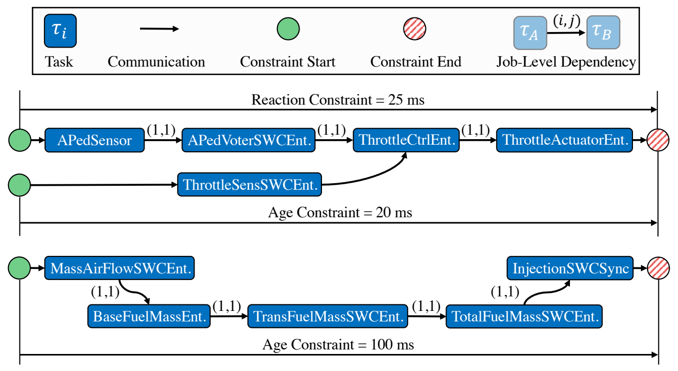

Fig. 7.: View of the data propagation constraints and the involved tasks. Additionally the generated job-level dependencies are shown.

size of the memory access phases. We tackle this challenge by observing that local memory banks can be used to statically allocate a subset of all tasks, which then removes the need for expensive off-chip memory access. The framework further takes job-level dependencies into account that are used to guarantee that timing constraints in the data propagation through chains of tasks are met. A constraint programming formulation of the problem is provided that generates the time-triggered schedule. Evaluations show that the improved memory aware framework outperforms related work.

Future work will investigate heuristic solutions to allow the approach to scale to industrial sized applications. Additionally, we will address the problem of task grouping, that can minimize the communication data and also reduce the number of jobs that need to be scheduled during one hyperperiod which in turn improves scalability.

Acknowledgement

The work presented in this paper is supported by the Swedish Knowledge Foundation (KKS) through the projects PREMISE, PreView, and DPAC. The work in this paper is also partially supported by Wenner-Gren Stiftelserna, grant number RSh2017-0031.

References

[1] Q. Perret, P. Maurere, E. Noulard, C. Pagetti, P. Sainrat, and B. Triquet, "Temporal isolation of hard real-time applications on many-core processors," in *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016, pp. 1–11.

[2] Q. Perret, P. Maurère, É. Noulard, C. Pagetti, P. Sainrat, and B. Triquet, "Mapping Hard Real-Time Applications on Many-Core Processors," in *Proceedings of the 24th International Conference on Real-Time Networks ans Systems (RTNS)*, 2016, pp. 235–244.

[3] P. Gai and M. Violante, "Automotive embedded software architecture in the multi-core age," in *Proceedings of the 21th IEEE European Test Symposium (ETS)*, 2016, pp. 1–8.

[4] C. B. Watkins and R. Walter, "Transitioning from federated avionics architectures to integrated modular avionics," in *Proceedings of the 26th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, 2007, pp. 2.A.1–1–2.A.1–10.

[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] M. Becker, D. Dasari, B. Nicolic, B. Åkesson, V. Nélis, and T. Nolte, "Contention-free execution of automotive applications on a clustered many-core platform," in *Proceedings of the 28th Euromicro Conference on Real-Time Systems (ECRTS)*, 2016, pp. 14–24.

[7] N. Feiertag, K. Richter, J. Norlander, and J. Jonsson, "A compositional framework for end-to-end path delay calculation of automotive systems under different path semantics," in *Proceedings of the 1st International Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS)*, 2008.

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

[9] M. Becker, D. Dasari, S. Mubeen, M. Behnam, and T. Nolte, "Synthesizing job-level dependencies for automotive multi-rate effect chains," in *Proceedings of the 22nd IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2016, pp. 159–169.

[10] M. Becker, S. Mubeen, D. Dasari, M. Behnam, and T. Nolte, "A generic framework facilitating early analysis of data propagation delays in multi-rate systems (invited paper)," in *Proceedings of the 23th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2017, pp. 1–11.

[11] R. Wilhelm and J. Reineke, "Embedded systems: Many cores – many problems," in *Proceedings of the 7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*, 2012, pp. 176–180.

[12] V. A. Nguyen, D. Hardy, and I. Puaut, "Cache-Conscious Offline Real-Time Task Scheduling for Multi-Core Processors," in *Proceedings of the 29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), M. Bertogna, Ed., vol. 76. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, pp. 14:1–14:22.

[13] C. Pagetti, D. Saussi, R. Gratia, E. Noulard, and P. Siron, "The rosace case study: From simulink specification to multi/many-core execution," in *Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2014, pp. 309–318.

[14] 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*, vol. 52, no. 4, pp. 399–449, 2016.

[15] 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 *Proceedings of the 2nd International Workshop on Dynamic Resource Allocation and Management in Embedded, High Performance and Cloud Computing (DREAMCloud)*, 2016.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] P. Laborie and J. Rogerie, "Reasoning with conditional time-intervals." in *Proceedings of the 21st Florida Artificial Intelligence Research Society Conference (FLAIRS)*, 2008, pp. 555–560.

[21] P. Laborie, J. Rogerie, P. Shaw, and P. Vilím, "Reasoning with conditional time-intervals. part ii: An algebraical model for resources." in *Proceedings of the 22nd Florida Artificial Intelligence Research Society Conference (FLAIRS)*, 2009, pp. 201–206.

[22] P. Laborie, "Ibm ilog cp optimizer for detailed scheduling illustrated on three problems," in *Proceedings of the 6th International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems (CPAIOR)*. Springer, 2009, pp. 148–162.

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

[24] M. Becker, B. Nikolić, D. Dasari, B. Åkesson, V. Nélis, M. Behnam, and T. Nolte, "Partitioning and analysis of the network-on-chip on a cots many-core platform," in *Proceedings of the 23rd IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2017, pp. 101–112.