Mapping and Integration of Event- and Time-triggered Real-time Tasks on Partitioned Multi-core Systems

Carlo Meroni
Technical University of Denmark
Kongens Lyngby, Denmark
meronicarlo98@gmail.com

Silviu S. Craciunas
TTTech Computertechnik AG
Vienna, Austria
silviu.craciunas@tttech.com

Anaïs Finzi
TTTech Computertechnik AG
Vienna, Austria
anais.finzi@tttech.com

Paul Pop
Technical University of Denmark
Kongens Lyngby, Denmark
paupo@dtu.dk

***Abstract*—In order to meet the requirements of critical applications, modern multi-core multi-SoC real-time systems must handle both periodic and sporadic events within specified deadlines. A two-level scheduling hierarchy that combines time-triggered and fixed-priority scheduling is effective for managing periodic time-triggered (TT) and sporadic event-triggered (ET) tasks, respectively. We introduce two polling-based approaches, called simple and advanced polling, for guaranteeing both TT and ET task deadlines within single core systems. We then propose an optimization heuristic for the task-to-core allocation problem based on genetic algorithms for fully partitioned multi-core systems that can be applied to both polling methods. We evaluate the schedulability and runtime performance of the polling approaches and investigate the effectiveness of the allocation heuristic using synthetic test cases based on real-world application characteristics. The results show that both polling approaches achieve high schedulability with low runtime, and the allocation heuristic can generate good solutions for fully partitioned systems. Furthermore, we show that the server design problem of the advanced polling approach can be integrated into the allocation heuristic to achieve better solutions in terms of schedulability and that our choice of the genetic algorithm has a good speedup when being parallelized.**

***Index Terms*—real-time, time-triggered, event-triggered**

I. Introduction

Modern multi-core multi-SoC real-time systems, such as those found in automotive applications [1], must respond to a mixed set of periodic and sporadic events to fulfill the requirements of critical applications. Usually, a mix of time-triggered scheduling (TT) and event-triggered (ET) tasks are used to process these periodic and sporadic events. There are many trade-offs between the time- and event-triggered approaches. Time-triggered systems tend to be more predictable, more stable, support a compositional approach, and have low jitter and high determinism [2], [3], [4], [5], [6]. Time-triggered (TT) systems are commonly used in the safety-critical aerospace domain due to stringent certification requirements [7], [8]. On the other hand, event-triggered systems allow more flexibility to respond quickly to sporadic events and result in lower average response times for tasks. More recently, many automotive applications have moved to integrate a mix of time-triggered and event-triggered scheduling [9], [1] to achieve a more deterministic temporal behavior while not sacrificing flexiblity [5], [4]. Moreover, modern real-time systems are also distributed and multi-core, especially in the automotive domain, and feature a distributed multi-SoC multi-core platform (c.f. [9], [1]) running a mixed time-and event-triggered task set with complex dependencies (e.g., cause-effect chains [10]). Hence, the multi-core task allocation and scheduling problem, which is NP-complete, is especially important in emerging applications like Advanced Driver-Assistance Systems (ADAS) [9], [1]. Modern safety-critical real-time systems will therefore require mixing the time-triggered and event-triggered approaches in a multi-core platform, merging the strict determinism of time-triggered dispatching with the flexibility of event-based handling. In such mixed systems, periodic time-triggered (TT) tasks are handled within predefined TT slots that are part of the static schedule table, and sporadic event-triggered (ET) tasks are executed within dedicated polling slots within the static schedule table. Moreover, ET tasks can also reuse slack generated within TT slots when periodic tasks finish their execution early. Traditionally, sporadic ET tasks are integrated into time-triggered systems via a feedback loop integrated into the TT schedule generation [11], [12].

In this paper, we investigate multi-core mixed event-and time-triggered systems where sporadic arbitrary-deadline and periodic constrained-deadline tasks must meet their predefined deadlines. We first introduce two polling-based approaches, called simple polling (SPoll) and advanced polling (AdvPoll), which guarantee the schedulability of the mixed sporadic and periodic task set in single core systems. Next, we introduce a genetic task-to-core allocation heuristic for the partitioned multi-core scheduling approach that can be applied to both polling methods. We note that TT tasks are always partitioned or semi-partitioned since the execution is based on statically-defined schedules, and there cannot be any runtime migration based on dynamic decisions like in global scheduling. For ET tasks, we also assume a fully-partitioned approach and leave a global scheduling solution for future investigation, noting that a (semi-)partitioned approach may be enough for most systems [13]. We investigate the trade-offs in terms of schedulability and runtime of both polling-based approaches and demonstrate in a series of experiments that we can efficiently solve the allocation problem for partitioned multi-core systems using our genetic algorithm. Finally, we also investigate the usefulness of integrating the server-design problem for AdvPoll into the genetic algorithm search.

We describe the system model in Sec. II and related work in Sec. III. We present the polling-based approaches in Sec. IV and detail our allocation heuristic in Sec. V. We finally evaluate our solution in Sec. VI and conclude in Sec. VII.

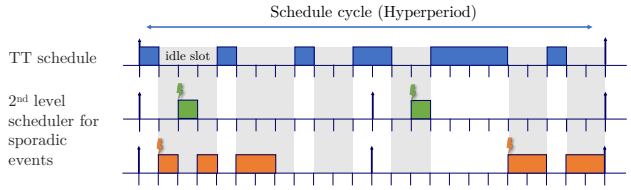

Fig. 1: TT schedule with 2^{nd}-level fixed-priority scheduling.

II. System model

We assume a (distributed) multi-core system in which, on each core of each node, there is a table-driven dispatcher (cyclic executive) which executes periodic TT tasks according to a statically computed and configured time-triggered schedule table. In the idle slots that are left within the TT table, the dispatcher implements a 2^{nd}-level preemptive fixed-priority scheduler (c.f. Fig. 1) running sporadic ET tasks. The TT scheduler always takes priority over the 2^{nd}-level fixed-priority scheduler, meaning that any ET task will be preempted whenever a TT task needs to be executed. We denote with $\mathcal{T}^{TT}$ and $\mathcal{T}^{ET}$ the sets of TT and ET tasks, respectively. A TT or ET task τ_i is modeled using the tuple $(C_i,\ T_i,\ D_i)$, where C_i represents the worst-case execution time and D_i is the relative deadline. For TT tasks, T_i denotes the period of the task, while for the sporadic ET tasks, T_i describes the minimal inter-arrival time (MIT), which represents the minimum time at which an event of the type can arrive after the previous arrival. TT tasks have a constrained deadline (i.e., $D_i \leq T_i$), and for ET tasks, the deadline can be larger than the inter-arrival time (arbitrary-deadline model). An ET task τ_i also has a priority value denoted with $p(i)$. ET tasks are sorted and indexed in the order of their priority, i.e., if τ_i has a higher priority than τ_j ($p(i) > p(j)$), then $i > j$ (FIFO order for equal priorities).

The granularity of the timeline for scheduling is based on microticks where a microtick mt (slot length) is the smallest scheduling granularity for tasks [12], and we assume that all task parameters are already scaled to be a multiple of the microtick mt. Each schedule repeats after a certain time period called the schedule cycle (or hyperperiod). As stated before, we assume a fully partitioned approach, meaning that both TT and ET tasks need to be assigned to the cores and nodes at design time, and there is no migration at runtime.

III. Related work

Integrating sporadic event-based tasks in systems with a time-triggered scheduler is more challenging than in purely fixed- or dynamic-priority systems. The mixed time- and event-triggered application model in [11] is similar to ours. However, the task model in [11] assumes a very restrictive task model in which ET tasks need to be periodic, whereas we work with a more generic and realistic sporadic task model. Using this periodic ET arrival model, the authors present an allocation and scheduling heuristic in which each candidate solution for the time-triggered schedule is investigated using a schedulability test for event-triggered tasks. The static table is regenerated if the ET tasks are not schedulable with the current candidate TT schedule solution. The approach in [11] attempts to generate correct TT schedules for the periodic tasks while trying to maximize the schedulability of ET tasks. However, they do not guarantee to find feasible static schedule tables that also fully fulfill the ET task deadlines. In contrast, our polling-based approaches only accept solutions where both TT and ET tasks fulfill their respective deadlines.

Our advanced polling approach can be viewed as a simplified version of hierarchical scheduling approaches such as [14], [15], [16] (c.f. Sec. IV-B) with 2 levels, a time-triggered scheduler on the bottom level, and a fixed-priority scheduler for ET tasks on the top level. For advanced polling, we also use the periodic resource abstraction (or periodic server) to decouple TT schedule generation from ET task schedulability analysis as in [14], [15], [16]. Using the worst-case service pattern for the periodic resource abstraction, as in [14], has the downside of the abstraction overhead (c.f. [16]) and, additionally, the server design problem [17] makes the problem difficult to solve, even for bandwidth-optimal approaches such as [18]. Moreover, in hierarchical approaches, there is the additional complexity of the allocation problem, which is not usually addressed. We base our AdvPoll method on [14] and use their schedulability test for fixed-priority systems within the periodic resource abstraction.

Slot-shifting [19], [12] introduces more flexibility in the time-triggered approach by allowing TT tasks to execute outside their assigned TT slots. While this improves the flexibility of TT schedules and ET task response times, the added dynamic runtime behavior may not be allowed for high-criticality functions that require safety assurance. Nevertheless, the work in [19], [12] is related to ours in that they introduce an efficient static schedulability test for ET tasks under the assumption of a given TT schedule. The test only needs to check so-called critical slots instead of all possible arrival times for ET tasks, making it more efficient in terms of runtime. Another more flexible approach for TT systems has been introduced in [20] where the schedule is safely adapted at runtime to allow for improved Quality-of-Service or the execution best-effort tasks.

IV. Polling-based integration of TT and ET tasks

We introduce two polling-based approaches based on prior work for single-core systems that are orthogonal to the allocation problem for multi-core systems presented in Sec. V.

A. Simple polling

An obvious and computationally "cheap" method, which we call *simple polling* **(SPoll)**, is to let each ET task $\tau_i \in \mathcal{T}^{ET}$ be handled by its own polling TT task τ_i^p. The over-sampling period T^p is easily derived for, e.g., out of the availability function in [14] as $T_i^p = \lfloor \frac{D_i + C_i}{2} \rfloor$. For sporadic ET tasks with constrained deadlines, the polling task has a computation time $C_i^p = C_i$. For sporadic ET tasks with arbitrary deadlines, we need to consider how many previous job releases there can be within any polling period. Hence we have $C_i^p = \lceil \frac{T_i^p}{T_i} \rceil \cdot C_i$. This approach can be very pessimistic for tasks with a

short deadline and long MIT/period or a long deadline and short period/MIT, leading to reduced schedulability. Hence, this approach results in a large over-utilization and will most likely not result in any feasible TT schedule creation.

B. Advanced polling

A more precise approach, which we call *advanced polling* **(AdvPoll)**, is a simplified version of the hierarchical scheduling paradigm [15], [16], [17] with 2 levels, a 2^{nd}-level fixed-priority (FP) dispatcher for ET tasks, and a time-triggered dispatcher on the lowest level, similar to [14]. Our reference method for the advanced polling is [14], where the schedulability of a set of constrained deadline sporadic tasks is verified under a server with a given capacity and period. Similar to [14], we define a periodic resource abstraction (basically a budget and period) for the polling task such that the sporadic ET tasks are still schedulable if the polling task gets the desired budget in the given period. The offline schedule synthesis step for TT tasks can then readily include the polling task when generating the schedule table as another periodic (set of) TT task(s), e.g., using exact methods or heuristics (c.f. [21]). Naturally, there can be more than one polling task, each of them handling a disjoint subset of the ET tasks. Using multiple polling tasks and deciding the mapping of ET tasks to them has an equivalent effect to re-assigning tasks' priorities since low-priority ET tasks can execute before higher-priority ET tasks. However, we have seen in our experiments that if priorities are assigned in a deadline-monotonic fashion, using multiple polling tasks yields worse or equal (in the best-case) results compared to using a single polling task.

We assume that there is only one polling task τ_p handling the entire set $\mathcal{T}^{ET}$ of ET tasks for which C^p and T^p have to be determined. While in [14] the polling task (periodic resource) is defined by a budget C^p and a period T^p, a more general model called Explicit Deadline Periodic (EDP) [18], [22], [17] can be used in which the server also has a deadline $D^p \leqslant T^p$. While this extension may increase the search space for possible TT schedules (and, therefore, schedulability), it will also result in a more complex server design problem (see below). The lower supply bound function $slbf(t)$ from [14] of a polling task τ^p in any time window of length $t \geqslant 0$. The exact expression of $slbf(t)$ can be found in [14], based on the characteristic function from [17]. To reduce complexity, the $slbf(t)$ is usually bound linearly from below by the so-called linear supply lower bound function $lslbf(t)$ (c.f. [17]) defined in [14] using $\alpha = \frac{C^p}{T^p}$ and $\Delta = 2 \cdot (T^p - C^p)$, as

$$lslbf(t) = max\{0, (t - \Delta) \cdot \alpha\}. \tag{1}$$

Following the method in [14], we compute for each ET task $\tau_i \in \mathcal{T}^{ET}$ and for each instant t the maximum load of task τ_i and all higher and equal priority tasks (maximum load of level-i) $H_i(t)$. We can use the classical definition of the maximum load of level-i from [23] for constrained deadline tasks, namely

$$H_i(t) = \sum_{j=1}^{i} \left\lceil \frac{t}{T_j} \right\rceil \cdot C_j. \tag{2}$$

The schedulability condition for an ET task $\tau_i \in \mathcal{T}^{ET}$ is defined in Lemma 1 of [14]. The worst-case response time R_i for task $\tau_i \in \mathcal{T}^{ET}$ can be calculated by determining the earliest time instant in which the maximum load of level-i $H_i(t)$ intersects the linear supply bound function $lslbf(t)$ of the polling task from Eq.(1), as follows [14]:

$$R_i = \text{earliest } t : t = \Delta + H_i(t)/\alpha. \tag{3}$$

While this method has the advantage of decoupling the schedulability of ET tasks from the generation of the TT schedule, the parameters of the polling task need to be found (which is, in essence, the non-trivial server design problem [15], [14]). In classical hierarchical scheduling, the aim is to find the resource abstraction with the least impact on other components, i.e., the best (C^p, T^p) where the utilization is just large enough to respect the ET deadlines. The search for the best (C^p, T^p) may be complex since we have to iterate not only through T^p, but for every T^p, we need to find C^p (and potentially D^p). If TT tasks have implicit deadlines, we can use a simplification here to eliminate the binary search for C^p for every T^p since we can use the maximum C^p that does not lead to overutilization, i.e., $C^p = \lfloor (1 - U^{TT}) \cdot T^p \rfloor$.

Once the polling task has been found, the schedule generation step can be done by including the polling server as another TT task along the existing TT tasks and simulating EDF/LLF scheduling until the hyperperiod of the resulting task set (e.g. [24], [25]). The schedule generation is relatively efficient, especially for harmonic TT task periods where the hyperperiods are (relatively) small [26] or when period re-dimensioning is possible to reduce the hyperperiod [27]. However, the main drawback of this approach is that we have to find the server parameters of the polling task. While there are specific optimizations that can be employed (e.g., using external points [14] or the methodology from [17]), the approach can be computationally intensive for large systems or when considering the task-to-core allocation problem as an additional dimension (c.f. Sec. V).

V. Genetic algorithm for the task-to-core allocation problem

So far, we have focused on the single-core case. However, modern real-time systems feature distributed multi-SoC multi-core platforms (c.f. [9], [1]) running mixed sets of tasks with complex dependencies (e.g., multi-rate cause-effect chains [10]). In such systems, the time-triggered paradigm implies solving the NP-complete allocation problem in partitioned or semi-partitioned solutions and then generating a correct schedule table for each core. Hence, the multi-core task allocation/scheduling and inter-dependence problems are, in essence, orthogonal to the presented polling approaches.

We propose a genetic algorithm with k-tournament selection [28] to optimize the task-to-core and scheduling problems for partitioned multi-core systems that integrate TT and ET tasks. The algorithm begins with an initial population (initial task-to-core mapping) that can be created using 4 different methods (c.f. Sec. V-D). A "chromosome" is an unique list

of values that describes a single solution, i.e., the target core ID for each TT and ET task, and, for the optimization of the Polling Servers parameters, the budget, period and relative deadline of each server. Each solution is then evaluated based on a fitness function (described below). The fittest chromosomes (candidate solutions) are then selected for reproduction through a process called crossover. During crossover, pairs of chromosomes exchange pieces of their genetic material to create new offspring chromosomes. Additionally, genetic algorithms also include a mutation step, which introduces random changes in the genetic material of the offspring chromosomes. Through this randomization, local optima are avoided and we encourage search space exploration. The selection, crossover, and mutation sequence is repeated for multiple generations, with the hope that the population will evolve toward a (near-)optimal solution. The algorithm terminates either when a solution meets a specified fitness threshold or when a predefined number of iteration have elapsed. We use a genetic algorithm with k-tournament selection because it provides a good mix between exploration and exploitation, good scalability, and a suitable degree of parallelization [28]. The pseudocode in Alg. 1 describes our multi-threaded genetic algorithm with k-tournament selection.

Algorithm 1 Genetic algorithm with k-tournament selection

Input: maxIterations, population

1: $i \leftarrow 0$
2: $father \leftarrow$ **copy**($population$[**getGoodChromosomeID**()])
3: $mother \leftarrow$ **copy**($population$[**getGoodChromosomeID**()])
4: **while** i $<$ maxIterations **do**
5: $mutationRate \leftarrow$ **mutationSchedule(**$\frac{i}{maxIteration}$**)**
6: $roundCount \leftarrow$ **roundSchedule(**$\frac{i}{maxIterations}$**)**
7: $son \leftarrow$ **crossingOver(**$mother, father$**)**
8: $son \leftarrow$ **mutate(**$son, mutationRate$**)**
9: $sonFitness \leftarrow$ **computeFitness(**son**)**
10: mutex.lock()
11: $victimId \leftarrow$ **getBadChromosomeID()**
12: $population[victimId] \leftarrow son$
13: $populationFitness[victimId] \leftarrow sonFitness$
14: $father \leftarrow$ **copy**($population$[**getGoodChromosomeID**()])
15: $mother \leftarrow$ **copy**($population$[**getGoodChromosomeID**()])
16: mutex.unlock();
17: $i \leftarrow i + 1$;
18: **end while**

The algorithm begins by initializing a counter of the current iteration (i) to 0. It then selects two good-performing chromosome from the current population using the *getGoodChromosomeID* function (c.f. Sec. V-A), which returns the chromosome with the highest fitness value among a random subset of size k of the population (where k is the round count). The algorithm then enters the main loop, which runs until the predefined number of loops specified by *maxIterations* has elapsed. Within each iteration, the algorithm determines the mutation rate and the number of rounds (tournaments) using the *mutationSchedule* and *roundSchedule* functions. The crossover is then performed on the parent chromosomes to create a new offspring chromosome. The mutation operation is then applied to the resulting offspring chromosome with the mutation rate computed in the previous step.

The fitness of the offspring chromosome is evaluated (and stored) using the *computeFitness* function (c.f. Sec. V-B), which returns a fitness value based on how well the chromosome solves the problem (see below). The algorithm then selects a victim chromosome from the population to be replaced by the offspring chromosome using the *getBadChromosomeID* function, which returns the ID of the chromosome with the lowest fitness value among a random subset of the population.

When running the algorithm on multiple cores, we avoid race conditions by locking the population data structure before replacing the victim chromosome with the offspring chromosome and updating its fitness value. The algorithm then selects new parent chromosomes from the population using the *getGoodChromosomeID* function. This process repeats until the maximum number of iterations is reached. Note that many possible stop criteria can be used, e.g., time or reaching a certain solution quality. The genetic algorithm uses a combination of selection, crossover, and mutation to evolve a population of potential solutions to a given problem. The fittest chromosomes are selected for reproduction, which increases the likelihood of producing even fitter offspring in the next generation. The algorithm continues to evolve the population for a specified number of iterations, after which the best-performing chromosome is returned as the solution.

A. Finding good and bad chromosomes

A chromosome is considered "good" when it has a high fitness value compared to a randomly selected subset of chromosomes, and is therefore used as a parent to create a new chromosome. To find a good chromosome, an individual is chosen randomly from the population and designated as the current best candidate. This candidate is then compared to another randomly chosen individual, and the solution with higher fitness becomes the current best candidate with a selection probability known as "*selectionProbability*". This process is repeated for a specified number of rounds, with the best candidate from all comparisons being deemed the "good" chromosome. Two parameters control the selective pressure during this process:

- *roundsCount*: controls the number of rounds (or tournaments) to be held in the k-tournament selection process. Increasing the number of rounds will increase the number of tournaments, creates a stronger selection pressure and may lead to faster convergence, but it may also increase the risk of premature convergence and the loss of genetic diversity.
- *selectionProbability*: controls the probability that a good chromosome (i.e., a winner of a tournament) is selected as a parent for the next generation. Changing this parameter results in a similar trade-off as the roundsCount parameter.

These parameters can be adjusted to balance the trade-off between convergence speed and genetic diversity. A "bad" chromosome, on the other hand, has relatively low fitness compared to a randomly selected subset of size k in the population. It can be found using the same method as selecting a good chromosome, but instead of maximizing fitness, the goal is to minimize it.

B. Fitness function

The fitness value indicates how well the individual solves the problem at hand and is usually problem-specific. Here, the optimization goal is to find a solution where the task set is schedulable and the response time of all tasks is minimized. In particular, the objective is to minimize the average worst-case response time of both Time-triggered and Event-triggered tasks. The fitness function proposed is comprised of two parts: $f_1(x)$ evaluates the worst-case response time of all tasks, while $f_2(x)$ evaluates the schedulability of TT and ET tasks on each core. Given the set of tasks $\mathcal{T}^{TT}$ and $\mathcal{T}^{ET}$, $R_i^{TT}(x)$ and $R_i^{ET}(x)$ the worst case response time of the i-th task given the solution x, D_i^{TT} and D_i^{ET} the deadline of the i-th task, ω^{TT} and ω^{ET} the weight factor for TT tasks and ET tasks, then $f_1(x)$ is defined as follow:

$$f_1(x) = \frac{\omega^{TT}}{|\mathcal{T}^{TT}|} \times \sum_{i=1}^{|\mathcal{T}^{TT}|} \frac{R_i^{TT}(x)}{D_i^{TT}(x)} + \frac{\omega^{ET}}{|\mathcal{T}^{ET}|} \times \sum_{i=1}^{|\mathcal{T}^{ET}|} \frac{R_i^{ET}(x)}{D_i^{ET}(x)}$$

The goal of $f_1(x)$ is to let the meta-heuristic algorithm minimize the worst-case response time of all tasks while also shifting the priority of responsiveness more towards either TT or ET tasks. For e.g., when $\omega^{TT} = \omega^{ET} = 1$ both TT and ET tasks are optimized to have a smaller response time, while when $\omega^{TT} = 1$ and $\omega^{ET} = 0$ only the TT tasks are optimized.

The function $f_2(x)$ is used to penalize solutions that are not schedulable. The schedulability of time-triggered (TT) tasks and event-triggered (ET) tasks mapped on each core and polling server, respectively, are represented by $u_i(x)$ and $v_i(x)$ for a given solution x. The penalty factors for unschedulable TT and ET tasks are denoted by λ^{TT} and λ^{ET}, respectively. Specifically, a penalty of λ^{TT} is added for each core where one or more TT tasks are not schedulable, and a penalty of λ^{ET} is added for each core where one or more ET tasks are not schedulable. This penalty scheme incentivizes the optimization process to converge quickly toward schedulable solutions. In this context and given K the number of cores in the target system, the expression for $f_2(x)$ is given by:

$$f_2(x) = \lambda^{TT} \times \sum_{j=1}^{K} u_j(x) + \lambda^{ET} \times \sum_{k=1}^{K} v_k(x)$$

Genetic algorithms are designed to maximize the fitness of chromosomes. In the case of minimizing both $f_1(x)$ and $f_2(x)$, it becomes necessary to convert the problem from a minimization problem to a maximization one. This can be achieved by negating both expressions. The final fitness function with both terms combined is the following:

$$f(x) = -\left[f_1(x) + f_2(x)\right] \tag{4}$$

C. Mutation and crossover functions

A mutation function is a genetic operator that introduces random changes to an individual's genetic material (i.e., the chromosomes) in a population. The mutation function works by randomly selecting one or more parameters within an individual's chromosome and changing their value in some way. In the proposed algorithm, mutations are applied to different task-to-core mappings but can also be applied to, e.g., different server parameters for the AdvPolling method. Optionally, ET task priorities can be mutated. The tasks mapping to cores are mutated with a probability of $\frac{1}{|\tau^{TT}|}$ for TT tasks and $\frac{1}{|\tau^{ET}|}$ for ET tasks, so that, on average, the mapping of one TT task and one ET tasks are mutated for each iteration. To mutate the mapping of a single task, a different core is randomly picked amongst the cores the fulfill the mapping constraints. For the Advanced Polling solution, polling servers are mutated with a probability of $\frac{1}{K}$ where K is the number of cores in the system. The mutation involves either adding a random number picked from a uniform distribution in the range of $(-1-9\mu, 1+9\mu)$ or multiplying for a random number in the range of $(0.5, 1.5)$, either the budget or the period.

Crossover is a fundamental genetic operator that combines genetic information from two parent solutions to create a new offspring solution. The choice of crossover function can significantly impact the convergence speed, diversity of the population, and ultimately the quality of the solutions obtained. The proposed crossover function combines different task-to-core mappings by selecting the core ID of each task from either one of the parent solutions, with equal probability. Additionally, for the Advanced Polling approach, each Polling Server parameter of the offspring solution is picked from one of the parent solutions, with equal probability.

D. Generating initial mappings

Four different methods of task-to-core mapping initialization are considered to generate initial task-to-core mappings for both TT and ET tasks. These methods are as follows:

1) Random mapping: Task-to-core mappings are generated randomly without regard to the current workload on each core. This method is not recommended as it may result in imbalanced workloads on the cores and produce many solutions that are not schedulable. On the other hand, it creates an initial population that has high diversity, which might help the optimization algorithm to find better solutions.

2) Laxity round-robin mapping: In this method, task-to-core mappings are generated using a round-robin approach where tasks are assigned to cores in a cyclic manner. The laxity (or slack time) of a task is defined as the difference between its deadline and its cost (or duration): $L_i = D_i - C_i$. The task with the highest laxity is selected from the set of tasks that can be executed on the core. This method is simple to implement and can result in a good initial workload balance. However, it does not take into account the utilization of the tasks, which may result in sub-optimal performance.

3) Load balancing mapping: In this method, task-to-core mappings are generated to balance the core utilization. This method can result in a good workload balance between the system cores, but it may not take into account the deadlines of the tasks, which could result in missed deadlines.

4) Delay minimizing mapping: The delay-minimizing task-to-core mapping initialization algorithm sorts all tasks in decreasing order of duration and then assigns each task to the

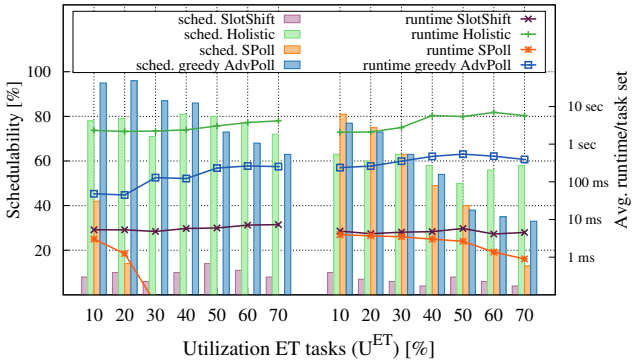

Fig. 2: Single-core schedulability and avg. runtime with $250\mu s$ microtick, ET & TT task periods $\{5, 10, 20, 40, 80\}ms$.

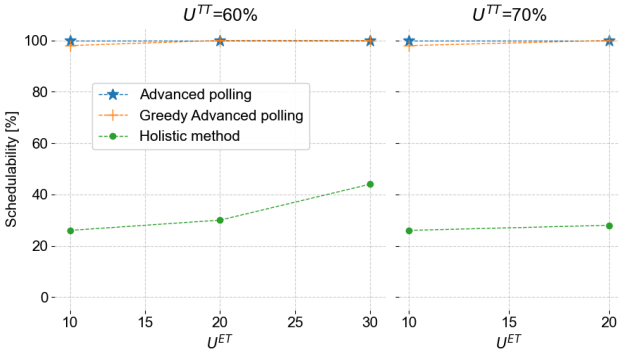

Fig. 3: Comparison of greedy AdvPoll vs. optimized AdvPoll vs. Holistic scheduling.

core that would minimize the total delay of execution caused by existing tasks on that core. For each task, the algorithm calculates the delay that would be caused by mapping the task on each core by summing the duration of all tasks on the core that would be handled first by EDF. The algorithm then assigns the task to the core where the delay of execution is minimum. This ensures that the tasks are assigned to cores in a way that minimizes the total delay caused by already existing tasks, which can be beneficial for meeting task deadlines.

VI. Experiments

A. Single-core comparison of polling approaches

For the single-core experiments (Fig. 2), we compare **SPoll** and **AdvPoll** against **Slot-Shifting** [29], and **Holistic scheduling** [11] in terms of schedulability (left y-axis) and runtime (right logarithmic y-axis) for constrained-deadline (left sub-plot) and arbitrary-deadline (right sub-plot) ET tasks. We implemented SPoll such that the polling period of each ET task does not lead to an explosion in the hyperperiod HP, selecting the largest value lower than the ideal polling period for which the new hyperperiod is smaller than $4HP$. We implemented AdvPoll as a greedy search by iterating through 200 possible equidistant polling task periods T^p up to the hyperperiod of TT tasks. After finding the polling task(s), we use a simple LLF simulation until the hyperperiod for both SPoll and AdvPoll to generate the static TT schedule. We also implemented the offline schedulability test for ET tasks of Slot-Shifting [29] and used an LLF simulation to generate the initial TT schedule with TT tasks as input. We note that we cannot directly compare to holistic scheduling [11] since their schedulability test does not work for our more generic sporadic ET task model. Nevertheless, we implemented the ALAP/ASAP-based scheduling heuristic from [11], replacing the original schedulability test for periodic tasks with a simulation-based test for sporadic tasks. For arbitrary deadlines, we use $min(D_i, T_i)$ for AdvPoll, SlotShifting, and the schedulability test of holistic scheduling, since they do not support arbitrary deadlines. We use 30 TT and 20 ET tasks per task set with periods selected from the set $\{5, 10, 20, 40, 80\}ms$ which correspond to periods found in a real-world automotive use-case. We generated 100 task sets per test case and use a microtick of $250\mu s$. For the constrained ET deadline test cases, D_i is uniformly selected in the upper half of the interval $[C_i, T_i]$, and for arbitrary ET deadlines, we use $D_i \in [C_i, 5 \cdot T_i]$. We keep the TT task utilization at 20% and increase the utilization of ET tasks as seen on the x-axis of Fig. 2. These experiments were run on an Apple MacBook M1 Pro 10-core (3.12GHz) / 16GB RAM.

We see in Fig. 2 that SPoll is very fast but only has acceptable schedulability (left y-axis) for low system utilization or when considering arbitrary deadlines. For constrained deadline ET tasks, SPoll does not perform very well in terms of schedulability, while AdvPoll can schedule even systems with high overall utilization. For arbitrary deadlines, SPoll performs better than AdvPoll and even Holistic scheduling for up to 50% system utilization while being significantly faster (right logarithmic y-axis). For highly utilized systems, the holistic method is better than the greedy search of AdvPoll but at the cost of a very significant runtime increase (avg 200 ms for AdvPoll vs. avg 3.5 sec for Holistic Scheduling). We note that holistic scheduling cannot be directly applied to sporadic task sets, and hence we have to use a very exact but very computationally expensive schedulability test which results in slightly better schedulability for highly-utilized systems at the expense of a significantly increased runtime (by a factor of $100 - 1000$). This increased runtime may especially hinder the solution quality when adding the task-to-core allocation dimension to the scheduling problem. We show that we can improve the schedulability of AdvPolling by including the server design problem in the genetic algorithm (c.f. Sec. VI-B).

B. Integration of allocation and AdvPoll server design

We compare the schedulability of three methods when applied to a multi-core setting. The task-to-core mapping is optimized using our genetic algorithm, while the integration of TT and ET tasks is managed using (a) AdvPoll with server parameters optimized by a greedy search (c.f. Sec VI-A), (b) AdvPoll with server parameters optimized using a genetic algorithm, and (c)the holistic method [11] with the simulation-based schedulability test (c.f. Sec VI-A). The task sets used in this experiment comprise 30 TT tasks and 30 ET tasks, targeting a two-core system with $U^{TT} = 60\%$ and $U^{TT} = 70\%$

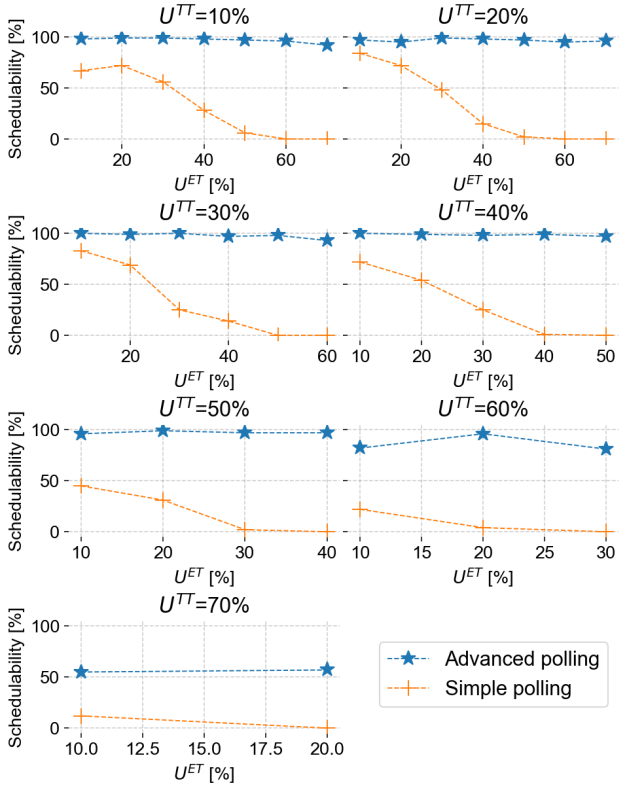

Fig. 4: Schedulability after 2500 iterations of optimization on different task sets with 8 cores and varying U^{TT} and U^{ET}

and varying degrees of U^{ET} utilization. For each task set, the optimization process runs for a maximum time interval of one second, after which it is halted.

Results shown in Fig. 3 indicate that the AdvPolling methods exhibit similarly high schedulability rates, even though the greedy search for server parameters is significantly slower. This is because, at each optimization iteration, optimal parameters must be re-evaluated. In this experimental setup, the holistic method demonstrates a lower schedulability rate compared to the AdvPolling approaches, particularly when dealing with high TT utilizations as tested in this experiment. To achieve similar results to the other methods, the holistic method might require more optimization iterations or an extended optimization time.

C. Schedulability performance

Before, we evaluated the schedulability of the proposed polling methods for single-core systems. We now assess the schedulability of the proposed genetic approach for multi-core systems when TT and ET tasks are integrated using the proposed polling mechanisms. In order to examine this, the schedulability percentage is calculated for 3400 task sets with varying TT and ET task set utilization, denoted as U^{TT} and U^{ET}, respectively. Each task set consists of 120 constrained-deadline TT tasks and 120 arbitrary-deadline ET tasks running on an 8 core platform. The genetic algorithm

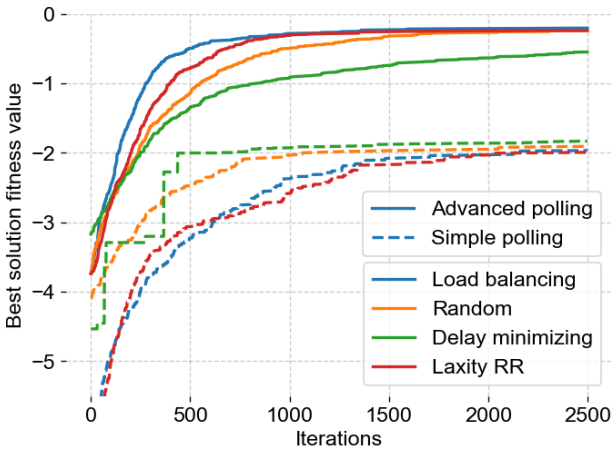

Fig. 5: Fitness value while optimizing solutions using different task-to-cores mapping initializations (higher is better).

is set up with a population size of $N = 50$, a selection probability of 1.0, a linearly increasing round count schedule of $k(t_0) = 1 + (0.05 * N - 1) * t_0$ and a linearly decreasing mutation schedule of $\mu(t_0) = 1 - t_0$ (where t_0 is the work done defined as $t_0 = \frac{i}{maxIterations}$). The optimization process is executed for a total of 2500 iterations on each task set. A solution is deemed valid if the algorithm can find a valid schedule for all of the 8 cores (i.e., both TT and ET tasks meet their deadlines on all cores).

Fig. 4 presents the experimental results in seven different subplots, each corresponding to a group of task sets with distinct TT task utilization U^{TT}. The optimized AdvPoll solution exhibits a high schedulability percentage, averaging close to 100% schedulability rate on these challenging task sets. In cases where the task set comprises TT tasks that consume a significant portion of the available resources (e.g., when $U^{TT} = 70\%$), the schedulability rate can drop to an average of 55% given the specified number of iterations. We see that the proposed optimization technique can achieve a good schedulability rate on a multi-core system, even when employing a simple integration method for TT and ET tasks like the SPoll approach. The optimized AdvPoll solution performs particularly well, even for highly utilized systems.

D. Impact of mapping initializations on the optimization

We evaluate the impact of different task-to-core mapping initializations on the convergence of the optimization process using SPoll (c.f Sec. IV-A) and AdvPoll (c.f Sec. IV-B). We use our genetic algorithm described in Sec. V to optimize the task-to-core mapping. To improve schedulability, we also optimize, as part of the genetic algorithm, the polling server parameters of AdvPoll instead of using the greedy search described in Sec. VI-A. Please note that we search only for the server period T^p since the deadline of the polling server is assumed to be equal to the period and the server budget C^p is set to $C^p = \lfloor (1 - U^{TT}) \cdot T^p \rfloor$. As before, we set up the genetic algorithm with a population size of $N = 50$, a

Cores	1	2	3	4	5	6	7	8
Speedup	1	1.93	2.73	3.74	4.12	4.52	5.03	5.33

TABLE I: Speedup with increasing number of cores.

selection probability of 1.0, a linearly increasing round count schedule of $k(t_0) = 1 + (0.05 * N - 1) * t_0$ and a linearly decreasing mutation schedule of $\mu(t_0) = 1 - t_0$ (where t_0 is the work done defined as $t_0 = \frac{i}{maxIterations}$).

The four mapping initializations discussed in Sec. V-D are compared for both approaches. The task set used for the experiment consists of 120 constrained-deadline TT tasks and 120 arbitrary-deadline ET tasks running on a 8 core system. The optimization process is executed for a total of 2500 iterations. Fig. 5 displays the best solution fitness curve for each experimental setup, averaged over 50 optimization runs.

As can be seen, employing various mapping initialization heuristics leads to different convergence rates during the optimization process. When optimizing the multi-core schedule using AdvPoll, the Load Balancing mapping initialization outperforms the others by discovering superior solutions with fewer iterations. Conversely, when utilizing the SPoll approach, the Delay Minimizing mapping heuristic yields the best results. These findings highlight the significance of selecting the task-to-core mapping initialization heuristic in the optimization process depending on the underlying polling method used. The choice of heuristic plays a crucial role in the convergence rate and the quality of the resulting solution.

E. Scalability

We have selected a genetic approach with k-tournament selection because it is highly parallelizable. Through parallelization, our method can leverage the computational power of multiple cores, leading to better solutions and improved scalability, especially for larger and more complex target systems. We have investigated the degree of speedup achieved by distributing the computation over multiple cores. We used a task set consisting of 240 TT tasks and 160 ET tasks running on an 8 core system. The optimization was done over 10000 iterations. The experiment was conducted on a Mac Mini with an M2 CPU and 16GB of RAM. Results show that we can achieve a speedup of up to $5.3x$ when increasing the number of cores to 8 (c.f. Table I). However, the speedup begins to taper off as more cores are used, which can be attributed to the overhead caused by thread synchronization. For more complex problems, the parallel efficiency improves, as the time required to evaluate, combine, and mutate chromosomes outweighs the delay caused by thread synchronization.

VII. Conclusion

We have addressed the problem of integrating event- and time-triggered tasks in partitioned multi-core real-time systems. We have introduced two polling-based approaches, called simple and advanced polling, that guarantee both TT and ET task deadlines within single core systems. Furthermore, we have proposed a genetic algorithm for the task-to-core allocation problem that can be applied to both polling methods (as well as to other well-known single-core approaches). We have shown that the polling-based approaches scale well in terms of schedulability and runtime compared to existing methods. We have also demonstrated that the allocation heuristic can achieve good solutions for fully partitioned systems and that the server design problem of the advanced polling approach can be integrated into the allocation heuristic to achieve better solutions in terms of schedulability.

References

[1] S. D. McLean *et al.*, "Configuring ADAS platforms for automotive applications using metaheuristics," *Front. Robot. AI*, vol. 8, 2022.

[2] J. Xu and D. L. Parnas, "Priority scheduling versus pre-run-time scheduling," *Real-Time Syst.*, vol. 18, no. 1, p. 7–23, 2000.

[3] J. Xu, "Satisfying complex timing constraints with pre-run-time scheduling," *IFAC Proceedings Volumes*, vol. 35, no. 1, pp. 273–278, 2002.

[4] F. Sagstetter *et al.*, "Schedule integration framework for time-triggered automotive architectures," in *Proc. DAC*, 2014.

[5] M. Lukasiewycz *et al.*, "Modular scheduling of distributed heterogeneous time-triggered automotive systems," in *Proc. ASP-DAC*, 2012.

[6] A. Mehmed *et al.*, "A time-triggered middleware for safety-critical automotive applications," in *Proc. AEiC*, 2017.

[7] T. Fleming *et al.*, "Improving the schedulability of mixed criticality cyclic executives via limited task splitting," in *Proc. RTNS*, 2016.

[8] C. Deutschbein *et al.*, "Multi-core cyclic executives for safety-critical systems," *Science of Computer Programming*, vol. 172, 2019.

[9] T. Fleming and A. Burns, "Investigating mixed criticality cyclic executive schedule generation," in *Proc. WMC*, 2015.

[10] M. Becker *et al.*, "End-to-end timing analysis of cause-effect chains in automotive embedded systems," *J. Syst. Archit.*, vol. 80, no. C, 2017.

[11] T. Pop *et al.*, "Schedulability analysis for distributed heterogeneous time/event triggered real-time systems," in *Proc. ECRTS*, 2003.

[12] D. Isović and G. Fohler, "Handling mixed sets of tasks in combined offline and online scheduled real-time systems," *Real-Time Syst.*, vol. 43, no. 3, 2009.

[13] B. B. Brandenburg and M. Gül, "Global scheduling not required: Simple, near-optimal multiprocessor real-time scheduling with semi-partitioned reservations," in *Proc. RTSS*, 2016, pp. 99–110.

[14] L. Almeida and P. Pedreiras, "Scheduling within temporal partitions: Response-time analysis and server design," in *Proc. EMSOFT*, 2004.

[15] I. Shin and I. Lee, "Periodic resource model for compositional real-time guarantees," in *Proc. RTSS*. IEEE, 2003.

[16] ——, "Compositional real-time scheduling framework with periodic model," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, 2008.

[17] G. Lipari and E. Bini, "Resource partitioning among real-time applications," in *Proc. ECRTS*, 2003.

[18] A. Easwaran *et al.*, "Compositional analysis framework using edp resource models," in *Proc. RTSS*, 2007.

[19] D. Isović and G. Fohler, "Efficient scheduling of sporadic, aperiodic, and periodic tasks with complex constraints," in *Proc. RTSS*, 2000.

[20] S. Skalistis and A. Kritikakou, "Dynamic interference-sensitive run-time adaptation of time-triggered schedules," in *Proc. ECRTS*, 2020.

[21] K. Schild and J. Würtz, "Scheduling of time-triggered real-time systems," *Constraints*, vol. 5, no. 4, p. 335–357, 2000.

[22] A. Aminifar *et al.*, "Designing bandwidth-efficient stabilizing control servers," in *Proc. RTSS*. IEEE, 2013.

[23] J. Lehoczky *et al.*, "The rate monotonic scheduling algorithm: exact characterization and average case behavior," in *Proc. RTSS*, 1989.

[24] S. D. McLean *et al.*, "Mapping and scheduling automotive applications on ADAS platforms using metaheuristics," in *Proc. ETFA*. IEEE, 2020.

[25] S. S. Craciunas *et al.*, "Optimal static scheduling of real-time tasks on distributed time-triggered networked systems," in *Proc. ETFA*, 2014.

[26] M. Nasri *et al.*, "A framework to construct customized harmonic periods for real-time systems," in *Proc. ECRTS*, 2014.

[27] M. Nasri and G. Fohler, "An efficient method for assigning harmonic periods to hard real-time tasks with period ranges," in *ECRTS*, 2015.

[28] D. E. Goldberg and K. Deb, "A comparative analysis of selection schemes used in genetic algorithms," ser. Foundations of Genetic Algorithms, G. J. Rawlins, Ed. Elsevier, 1991, vol. 1, pp. 69–93.

[29] D. Isović and G. Fohler, "Handling sporadic tasks in off-line scheduled distributed real-time systems," in *Proc. ECRTS*, 1999, pp. 60–67.