Memory Optimization for Adaptive Time-Triggered Systems

Omar Hekal, Daniel Onwuchekwa, Roman Obermaisser
University of Siegen, Chair for Embedded Systems
Siegen, Germany
{omar.hekal,daniel.onwuchekwa,roman.obermaisser}@uni-siegen.de

***Abstract*—Adaptation is crucial in time-triggered systems for maintaining system performance and reliability under varying conditions, such as dynamic workloads or resource failures. Time-triggered systems rely on metascheduling techniques for adaptation; however, utilizing existing metascheduling schemes for time-triggered systems faces storage challenges for adaptation using the resulting schedules. This work presents a Genetic Algorithm (GA)-based metascheduler to tackle the state explosion problem in time-triggered systems. Our proposed method is designed to optimize meeting the application deadline and memory utilization by employing a Multi-Objective Genetic Algorithm (MOGA). Meeting the deadlines is crucial for safety-critical applications. Additionally, minimizing the schedule changes after a context event reduces the memory overhead. We leverage the similarity between successive schedules to store schedules incrementally, retaining only the differences between parent and child schedules. This approach significantly reduces redundant data storage and helps mitigate the state explosion problem. Comparative experiments demonstrate that our MOGA-based metascheduler achieves up to 90% memory savings, outperforming a makespan-optimized metascheduler and a metascheduler optimizing the lateness in a scenario with twenty-two context events. These results highlight the potential of our approach in enhancing the scalability and efficiency of metascheduling techniques in memory-coznstrained environments.**

***Keywords*—metascheduling, adaptation, genetic algorithm, time-triggered systems.**

I. Introduction

Safety-critical systems are frameworks designed to operate in environments where their failure can lead to catastrophic outcomes, such as loss of life, extensive property damage, or significant environmental harm [1]. They are characterized by having hard deadline requirements for which timing guarantees must be provided [2]. Safety-critical systems are found in the aviation, healthcare, nuclear power, and transportation industries. These systems demand the highest levels of reliability and robustness. For instance, in aviation, safety-critical systems are integral to flight control systems, which ensure the safe operation of an aircraft by continuously monitoring and adjusting various parameters such as altitude, speed, and navigation [3].

Reliability in safety-critical systems is motivated by the need to adapt to dynamic and unpredictable environments while ensuring consistent performance and safety. They must effectively respond to various scenarios, including unexpected faults or external disturbances. The system's reliability requirements can be met by incorporating fault tolerance, which can involve adding some form of redundancy. The redundancy can be in hardware, software, information, or a combination of these elements, mitigating faulty system operations [4]. Triple Modular Redundancy (TMR) [5] is a well-known method for addressing single permanent or transient faults. In this configuration, a single module is supplemented with two identical replicas and a majority voter, allowing the correct operation of two redundant modules to outvote a single fault. This strategy remains used in many fields, including aeronautics and space. However, TMR can incur power and area overheads of up to 400% due to the additional voter and spacing requirements, making it inefficient.

Time-triggered systems are pivotal for safety-critical systems to ensure deterministic behaviour and precise control over safety operations, fulfilling the safety-critical system requirements of stable service delivery under different load and fault assumptions [6]. Time-triggered systems offer temporal predictability, implicit synchronization, and avoidance of resource contention [7]. They use time-triggered schedules, computed at design time, to determine task allocations and communication routes while avoiding time conflicts and ensuring all tasks meet their deadlines.

Metascheduling techniques are employed in time-triggered systems to achieve adaptation while preserving temporal predictability [8]. The metascheduler computes a schedule for each context event resulting in a Multi-Schedule Graph (MSG). Upon the occurrence of a relevant event, the current schedule is left, and the system traverses to another schedule of the MSG [6]. This dynamic switching process ensures optimal performance and adherence to timing constraints, reflecting the system's agility and responsiveness.

State explosion in the MSG is a significant challenge for metascheduling techniques as memory resources are limited in many real-time multi-core systems [9]. The MSG's size depends on the size of the application, platform, and context models. The MSG's size increases exponentially with the number of tasks and messages within the application model as the schedule must provide the temporal and spatial allocations for the application model elements [6].

This paper presents a Genetic Algorithm (GA)-based metascheduler to tackle the state explosion problem and minimize memory consumption. Our proposed method is a trade-off between having optimum makespan values and memory utilization. Our proposed metascheduler is based on the idea

of a Multi-Objective Genetic Algorithm (MOGA), where the first objective is to meet the application deadline, and the second objective is to minimize the changes to the current schedule following the occurrence of a context event. We leverage the similarity between the parent and child schedules to store schedules incrementally, retaining only the new information and omitting redundant ones. We compared our MOGA metascheduler to two other metaschedulers, one optimizing the makespan and the other optimizing the lateness with respect to memory consumption. The results showed that our proposed algorithm saves up to 90% of memory space used for schedule storage in a scenario of twenty-two context events compared with the metascheduler optimizing the makespan only.

The rest of the paper is organized as follows. Section II discusses the related work, and section III explains the system model. In section IV, we introduce the proposed method, section V for experiments and results, and finally, section VI concludes the paper.

II. Related Work

Metascheduling is critical for time-triggered systems to enhance their ability to manage complex and dynamic workloads. These systems execute tasks based on precise, predefined schedules, which must ensure predictability and reliability, particularly in safety-critical applications. Metascheduling is the method that allows for the adaptation of schedules to changing system conditions or requirements. Adaptation is essential for accommodating events such as slack or failures while maintaining strict timing constraints. By leveraging metascheduling, time-triggered systems can achieve greater flexibility and resilience, optimizing resource utilization and, most importantly, maintaining system stability.

Sorkhpour et al. [8] introduced a metascheduling technique to utilize energy consumption for time-triggered systems. Their proposed algorithm supports mapping the scheduling of the jobs to network-on-chip architectures to minimize energy consumption while meeting the timing constraints. The scheduler aimed to minimize energy consumption by reducing the core's frequency.

Muoka et al. [7] presented a metascheduler with sample points for energy saving in time-triggered systems. The idea is to minimize the communication overhead by minimizing the adaptation frequency. The authors introduced an offline metascheduler that optimizes static schedules by applying slack events to save energy at global periodic times in the schedule. The adaptation points are mapped to the run time sampling period of adaptation. Slack events are reported synchronously by adaptation units at run time, and adaptation is achieved through the aligned switching of component schedules facilitated by a Fault-Tolerant Agreement Protocol (FTAP). The metascheduler computes a multi-schedule that holds the adapted schedules and describes the run time switching of schedules based on the reported slack events. Results showed the effectiveness of their proposed method in minimizing the communication overhead and saving energy.

However, the authors in [7] and [8] did not address the state space explosion problem, which is a challenge for metascheduling techniques. Besides, their experiments did not explore a wide range of application model sizes.

Machine learning techniques are emerging as a powerful solution for optimizing schedules in complex systems. They can improve the system performance by predicting the optimal scheduling decisions. In [10], the authors introduced an architecture that uses machine learning to infer new schedules at run time, thus eliminating the need to store the schedule sets generated by a metascheduler. They investigated the performance of three machine learning models: Random Forest, Artificial Neural Networks and an Encoder-Decoder model. The results showed the superiority of the Encoder-Decoder model for different job sizes regarding the correct task allocation. Furthermore, in [11], Alshaer et al. proposed a machine learning-based metascheduler based on Graph Neural Networks (GNN). The idea is to use a GNN model to learn the scheduling mechanism of a GA-based metascheduler so that adaptation can occur through the GNN model at run time. Their results showed potential for using machine learning-based metaschedulers for adaptation and minimizing the storage requirements.

However, neither work can guarantee the applicability of machine learning scheduling techniques in safety-critical environments due to the inconsistent model performance. Besides, they did not factor in the deadline, an important parameter when addressing the scheduling problem in such environments.

In [12], the authors presented a solution to tackle the state-space explosion problem in MSGs by introducing a reconvergence of paths algorithm. The reconvergence algorithm reduces the number of schedules resulting from the metascheduler by eliminating redundant schedules that appear on different tree paths by merging the transitions. However, if the new schedule differs partially from the previous schedule, The whole new schedule information will need to be stored.

Our proposed method efficiently tackles the state-space explosion problem. By introducing a more efficient memory-saving approach, we store only the differences between the new and its parent schedules. This technique, called delta encoding [13], minimizes memory usage by capturing incremental changes rather than duplicating redundant data. Storing the changes between successive schedules reduces the number of unique schedules that must be stored, thereby shrinking the state space. Moreover, the proposed method can enhance the scalability of the metascheduler, allowing it to handle immense and more complex system configurations without overwhelming storage resources.

III. System model

In this section, we describe the scheduling problem of the metascheduler. We define three entities: the application model, the platform model and the context model, which together constitute the metascheduler's input. The metascheduler generates a schedule for each context event. These schedules build the

Multi-Schedule Graph (MSG) as shown in Figure 1. In the following lines, we explain each of these entities.

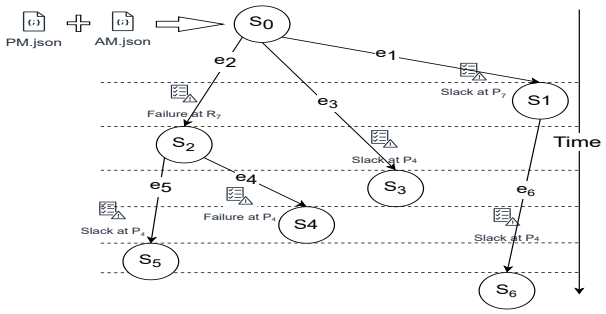
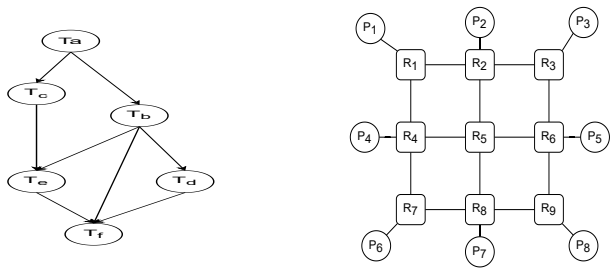


Figure 1. An MSG showing alternative schedules (different systems states) based on context events like failures and slack availability.

A. Input Models

We split the input model into three models, which serve as input for the metascheduler.

1) Application Model (AM): The application model refers to the computational tasks and their specific attributes, such as processing times and resource requirements. Furthermore, the application model details each message, specifying the sender and receiver tasks, therefore highlighting the dependency constraints between the tasks. For instance, if a task (T_a) produces data which serves as input for another task (T_b), the execution of the task (T_b) cannot take place except after the completion of (T_a).



(a) Application Model (b) Platform Model

Figure 2. Comparison of the Application and Platform Models

As can be seen in Figure 2a, the application model can be represented as a Directed Acyclic Graph (DAG), denoted as $\mathcal{G}(\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ represents the graph nodes corresponding to the tasks, and $\mathcal{E}$ denotes the connections between these tasks. A node is a collection of instructions known as a task, and an edge originates from a parent node and terminates at a child node. The graph's edges represent the messages that need to be exchanged between tasks; in other words, they model the communication required for the tasks to interact and coordinate. These edges define the dependencies and data flow between tasks, ensuring the tasks are executed in a specific sequence based on the application's requirements [14] [15].

2) Platform Model (PM): The platform model informs about the computational resources needed to execute the application model. The platform model as in Figure 2b is an undirected graph $\mathcal{P}(\mathcal{N}, \mathcal{L})$, where $\mathcal{N}$ is the set of routers and cores, and $\mathcal{L}$ is a set of bi-directional links connecting them. A path through $\mathcal{P}$ from a core P_i to $P_k \in \mathcal{N}$, is a sequence $\langle R_i, ..., R_k, P_k \rangle$ of vertices connected by links $l_{ik} \in \mathcal{L}$ for $i = 1, 2, ..., k$. In the Figure below, the cores $P_i \in \mathcal{N}$ are indicated with the symbol P and the routers with R.

3) Context model: The context model includes events relevant to the adaptation, including faults, dynamic slack, resource alerts, and environmental changes. Examples of faults are permanent failures of cores and routers, which require recovery action, such as reallocating tasks and messages. Dynamic slack is another example of a context event, where a task finishes before its worst-case execution time (WCET), leading to sending a message before its anticipated transmission time in the time-triggered schedule, which motivates applying energy-saving techniques (e.g., Dynamic Voltage and Frequency Scaling (DVFS)) without causing implications on the rest of the system.

B. Output Model

1) Metascheduler: A metascheduler is an offline scheduling framework for computing time-triggered schedules while considering spatial, temporal, and contextual system dimensions. The metascheduler takes the task application, platform model and context model as inputs for schedule computation. The computed schedules accumulate, forming a Multi-Schedule Graph (MSG).

The MSG is a Direct Acyclic Graph (DAG), where each node in the graph represents a schedule corresponding to a specific combination of context events, and the graph links are the context events. Figure 1 shows an example of an MSG. Each schedule node carries information about the temporal and spatial allocation of the computational and communication resources. The system will be at one of the MSG nodes at any particular point before moving to another node upon the occurrence of a context event. The metascheduler starts in the initial state is S_0 where no context event occurs. Then, it continues to perform time steps till the occurrence of an event.

2) Scheduler: The metascheduler invokes the scheduler repetitively to compute the MSG. The scheduler inputs the application, platform and context models to generate an MSG. The scheduler's output is a time-triggered schedule fulfilling specific pre-defined constraints such as the precedence constraints between tasks, collision avoidance between messages, and the application deadlines. A schedule example can be seen in the Figure 3 below.

The resulting schedule maps the application model to the platform model, showing the start and end times for all the tasks within the application model. The start time of a task

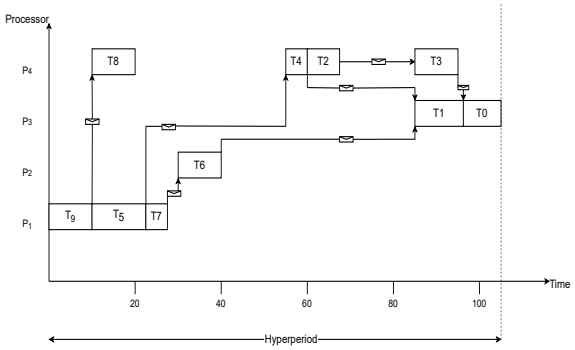


Figure 3. A time-triggered schedule example

T_b with a precedence task T_a can be calculated based on the equation below:

$$st_{T_b}(P_i) \geqslant t_f(m_{ab}, T_a \rightarrow P_i, P_k). \quad (1)$$

where $st_{T_b}(P_i)$ is the start time of task T_b on processor P_i,and $t_f(m_{ab}, T_a \rightarrow P_i, P_k)$ refers to the finish time of a message transfer form P_k to P_i following the completion of T_a.

For every message $m_{ab} \in \mathcal{E}$, where T_a is precedent to T_b a message path through the platform model $\mathcal{P}$ is computed. The shortest message path $\langle R_i, ..., R_k, P_k \rangle$ that includes $T_a \rightarrow P_k$ and $T_b \rightarrow P_i$ is considered when scheduling.

IV. Proposed method

This section presents our novel method: a GA-based metascheduler to tackle the state explosion problem. The proposed method aims to balance the optimization of makespan and memory utilization by using a Multi-Objective Genetic Algorithm (MOGA)-based metascheduler. The algorithm aims to minimize deviations from the current schedule when a context event occurs while ensuring the application meets its specified deadlines. By maximizing the structural similarity between parent and child schedules in the Multi-Schedule Graph (MSG), the method allows for an efficient incremental storage mechanism. In this approach, only the newly introduced information from the updated (child) schedule is stored, while unchanged portions from the previous (parent) schedule are reused, reducing redundant data storage.

A. Genetic Algorithm

1) ***GA overview****:* Genetic Algorithm (GA) is a metaheuristic optimization tool that follows the evolution paradigm. The algorithm utilizes genetic operators such as selection, mutation, and crossover to generate offspring that are fitter than their ancestors. Each generation yields a new individual corresponding to a schedule [15] [16]. Algorithm 1 provides a detailed overview of the genetic algorithm's working principle implemented in this study.

2) ***Genomes description****:* We construct schedules from the genomes in the population. A genome represents a solution to the scheduling problem and comprises four chromosomes.

Algorithm 1 GA for Task Scheduling Optimization

Input: Application model & Platform model.
Output: Optimum genomes

1: Create initial population
Genome→[$Task\_order$,$Processor\_Allocation$, $Path\_index$,$Message\_Oder$]
2: **Function** Evaluate pop():
3: S_n = Reconstruct schedule using Alg.2
4: Compute S_n makespan
5: Similarity score = Compare S_n with $S_{predecessor}$
6: lateness = max (0 , (makespan - deadline))
7: **return** lateness, Similarity score
8: **Set:** generation counter ($g = 0$)
9: **while** termination criteria not met **do**
10: **Selection:**$parents$ fitness = Evalute pop()
11: **Crossover:** Apply to $parents$ to produce $offspring$
12: **Mutation:** Mutate $offspring$
13: **Evaluate:** fitness of $offspring$ = Evalute pop()
14: **Combine:** Combine Pop_{g-1} and $offspring$
15: **Non-Dominated Sorting:** Rank individuals based on Pareto dominance
16: **Survivor Selection:** Select the best individuals to form Pop_g based on Pareto rank (prioritize similarity score)
17: **end while**
18: Construct the optimal schedule using the final individual's genetic representation // Alg.(2)
19: **return** Schedule (S_n)

These chromosomes are responsible for task ordering, processor allocation, message path index, and priority ordering of messages. Figure 4 shows the representation of a genome.

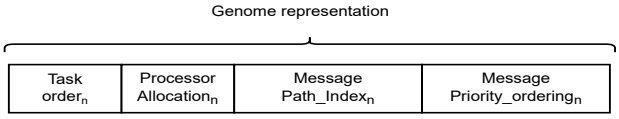


Figure 4. Genome representation

The initial task order chromosome is generated using a random permutation of task identifiers (task_id) within the range zero up to the maximum task ID minus one. The initial chromosome for processor allocation is generated by randomly assigning task allocations to processing element identifiers, with the constraint that only processing elements, excluding routers, are selected. This ensures that tasks are mapped exclusively to valid processing units in the system. The message path index denotes the multi-cast communication route from the source node (processor) to one or more destination processors. This index represents the sequence of communication links used to transmit messages across the network. The path indices are computed in a prior stage using the k-shortest path method [17]. Each path has a cost computed based on the number of hops between the two processors joined via this path. All the possible paths with their associated

costs are stored in a Python dictionary. The GA is designed to prioritize the selection of communication paths that minimize overall communication costs. This optimization process seeks to reduce the latency and resource overhead associated with message transmission across the network, thereby improving system efficiency. The message priority ordering chromosome organizes the sequence of messages to be exchanged between processing elements to optimize task scheduling. This ordering aims to minimize the overall makespan by prioritizing message transmissions to reduce communication delays and enable efficient task execution across the system. The message priority ordering mechanism reduces resource contention by managing messages on intersecting paths based on their assigned priorities. This prioritization scheme ensures orderly handling of message transmissions, minimizes conflicts, and optimizes communication flow within the system.

*3) **Genomes selection**:* In the proposed MOGA-based metascheduler, the selection operator is critical for balancing competing objectives, ensuring that application deadlines are met while minimizing schedule modifications in response to context changes. The selection process prioritizes schedules that optimally satisfy both objectives, assigning higher fitness to those that maintain deadlines and exhibit minimal changes.

Several Multi-Objective Evolutionary Algorithms (MOEA) have been introduced in previous works [18] [19], aimed at addressing similar multi-objective optimization problems. MOEAs can identify multiple Pareto-optimal solutions within a single simulation run by focusing on converging toward the Pareto-optimal front. This capability enables the exploration of trade-offs between competing objectives and the generation of a diverse set of optimal solutions [19].

In our MOGA, Pareto-based selection techniques, such as non-dominated sorting, are utilized to maintain diversity in the solution population. These methods ensure a balanced exploration of the solution space, preserving a variety of trade-offs between makespan optimization and memory utilization. This approach enhances the system's adaptability and efficiency by allowing the algorithm to explore and converge toward optimal solutions across both objectives.

Because of the conflicting nature the two objectives might have, we use the lexicographic ordering optimization method, where the problem's objectives are ordered based on importance from the most important objective to the least one [20]. In our case, we prioritize the schedules' similarity objective.

We used the Jaccard similarity metric to assess the similarity between a successor schedule and its preceding schedule after a specific context event. The general formula for calculating the Jaccard index between two sets is shown in equation 2.

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{2}$$

B. Schedule reconstruction logic

The reconstruction logic outlined in this paper is focused on optimizing task scheduling within a parallel computing environment. The genetic algorithm generates final genomes, which are then used as input parameters for the reconstruction function. The *processing_times* attribute contains the time required to complete each task within the application, and the *message_list* stores details of the messages exchanged between tasks. Both sets of information are derived from the application model. Additionally, the *Path_index_with_cost* provides the routes between processors via routers and the associated communication costs from the platform model. Using these inputs, the reconstruction function maps tasks to processors, scheduling them to execute as soon as their dependencies are resolved, including receiving required messages from other tasks. The reconstruction function also considers communication delays by integrating the costs of transferring messages between processors, ensuring efficient use of time and resources. The algorithm dynamically adjusts the execution start times based on processor availability and the completion of preceding tasks. The pseudo-algorithm for the reconstruction logic is stated in Algorithm (2).

Algorithm 2 Reconstruction Logic

Input: $GA\_genomes$, $Message\_order$, $processing\_times$, $message\_list$, $Path\_index\_with\_cost$
Output: Schedule (S_n)

```
Map each task to the corresponding processor
Arrange the message_list based on Message_order
For each message in Message_list
    Pick a path from Path_Index list
if two messages use the same path then
    The higher priority message has the advantage
end if
Initialize completed_tasks as an empty set
Initialize ready_tasks with all tasks that have complete precedence constraints
while ready_tasks is not empty do
    Select and remove a task from ready_tasks
    Determine processor and predecessors for the task
    if all predecessors are completed then
        Calculate start_time based on predecessors' completion times
        Schedule the task; update task_completion_times and current_time_per_processor
        Add the task to completed_tasks
    else
        Add the task back to ready_tasks
    end if
end while
return Schedule (S_n)
```

C. Metascheduler for memory optimization

The metascheduler is a tool for computing time-triggered schedules to adapt to different context events. It creates schedules for all potential sequences of context events within a hyper-period during the design time. These schedules will be deployed during runtime at the beginning of each hyperperiod. The successive schedule computation following a subsequent

context event leads to the generation of an MSG. As the number of context events increases, the number of generated schedules grows exponentially. This leads to a massive set of schedules that needs to be stored for adaptation during runtime, which is impractical. Our proposed metascheduler addresses this issue in two ways: first, by minimizing schedule changes when a context event occurs (as explained earlier in IV-C)), and second, by storing only the differences between two subsequent schedules.

Algorithm 3 MOGA-based Metascheduler

Input: Application Model (AM) & Platform Model (PM) & Context Model (CM)

Output: Multi-Schedule Graph (MSG)

Function Compute MSG (AM,PM,CM):
  S_o = Invoke GA for computing base schedule // *Alg.(1)*
  If (context_event = none):
    $S_{deployed}$ = S_o
  ElseIf (context_event = True):
    $AM_{updated}$, $PM_{updated}$ = *Apply_Context_event*
    S_{new} = Invoke GA // *Alg.(1)*
    Compute $\Delta_{schedules}$ = S_{new} - ($S_{new} \cap S_{predecessor}$)
    Add the node S_{new} to MSG // Δ information only
return MSG

V. Experiments and Results

This section discusses the experimental procedures and their corresponding results. We used the Stanford Network Analysis Platform (SNAP) to create example graphs for both the application and platform models [21]. SNAP applies network theory, also known as graph theory, and was used in this work to generate various scenarios. It allows for the development of functions to analyze and manipulate large-scale networks. The library processes the original model input and defines parameters such as the number of tasks and participants in the platform model (e.g., processors and routers). We used SNAP functions to produce multiple application and platform model scenarios. The code responsible for generating different application model scenarios was executed using the OMNI cluster at the University of Siegen [22]. Additionally, we implemented the genetic algorithm using the Distributed Evolutionary Algorithm in Python (DEAP) library [23].

We created application models of various sizes using a random forest-fire model. Each model specified the number of nodes, edges, in-degrees, and out-degrees. Our experiments involved application models ranging from twenty to a hundred tasks. The platform model we used had eight homogeneous processing elements connected through a 3x3 mesh network of routers, as illustrated in Figure 2b. Twenty-two context events were used to generate MSGs for each application model size.

We compare our proposed MOGA-based metascheduler to two other GA-based metaschedulers with different objective functions. The objective function of the first is to minimize the schedule's makespan, while the objective function of the other one is to minimize lateness with respect to the application deadline. We assume different application deadline values, where the values were 5% or 10% or 20% or 30% or 40% of the base schedule makespan for each application model.

The objective of the comparison is to evaluate the impact of the proposed method with respect to the memory size required to store the MSG. As explained in IV-C, we store only the changed information about the schedule compared to its previous state in the MSG. Additionally, we compare the average makespan values over the whole MSG for all three metaschedulers.

GA parameters were constant for the varying AM sizes for all three metaschedulers. The parameter settings can be shown in Table I.

TABLE I: GA Parameters

Population Size	Crossover Probability	Mutation Probability	Number of Generations
100	0.4	0.6	400

The results are summarized below in Figures 5 and 6. Figure 5 shows the average makespan values across all the schedules within an MSG for different application model sizes. The makespan is the total time required to complete all the application tasks. Figure 6 shows the total number of bits changed throughout the MSG for different application model sizes. The number of changed bits metric refers to the amount of new information needed to be stored following computing an MSG. The different curves in both figures represent the different settings for the metascheduler objective functions throughout the experiments.

TABLE II: Number of Changed Bits Throughout the Whole MSG

App Model Size / Objective Function	20T	30T	40T	50T	60T	70T	80T	90T	100T
Makespan	4200	2960	7730	11080	9865	11965	15140	16365	14730
MOGA, 5% BsMS	100	360	2435	3610	3585	4980	7530	7770	7980
MOGA, 10 %BsMS	105	395	2750	3580	3485	4920	7455	7720	7760
MOGA, 20 %BsMS	90	320	2565	3585	3415	4905	7305	7925	7830
MOGA, 30%BsMS	100	265	2390	3430	3455	4710	7195	7575	7450
MOGA, 40%BsMS	55	350	2500	3320	3175	4545	7060	7465	7680
Lateness, 5%BsMS	4975	3250	8475	10605	10215	11335	16325	17680	16570
Lateness, 10%BsMS	5220	3560	8710	10520	10225	11320	16555	17445	16460
Lateness, 20%BsMS	5280	3580	8765	10635	10250	11345	16490	17805	16575
Lateness, 30%BsMS	5360	3695	8955	10675	10425	11520	16525	17800	16650
Lateness, 40%BsMS	5445	3685	8820	10580	10260	11630	16565	17905	16530

The results in Table II, and Figure 6 show that our proposed metascheduler (MOGA) remarkably minimizes schedule changes compared to the other metaschedulers, optimizing the makespan and the lateness. This substantial reduction in schedule modifications directly leads to a marked decrease in the memory space required for storing the MSG, thereby improving overall system resource efficiency. Across all application model sizes, MOGA consistently achieves significantly fewer changed bits. In other words, less information is required to be stored following the computation of a new schedule to the respective context event. For example, in the application model of twenty tasks, MOGA with a deadline of 1.05 times the

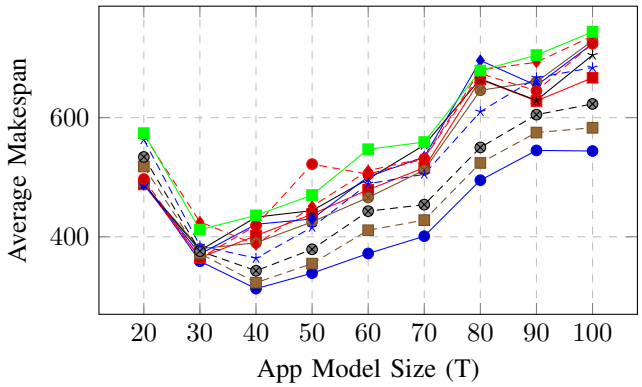


Figure 5. Average Makespan Values

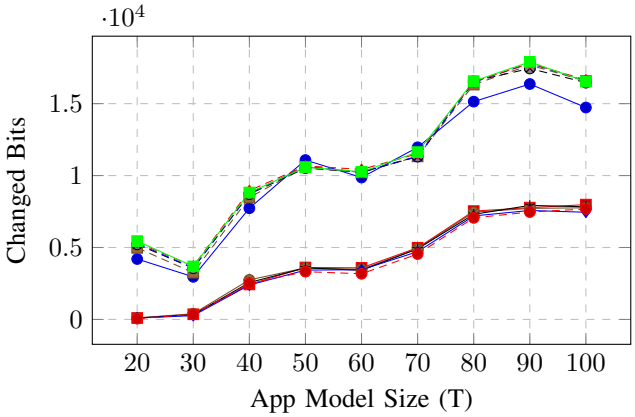
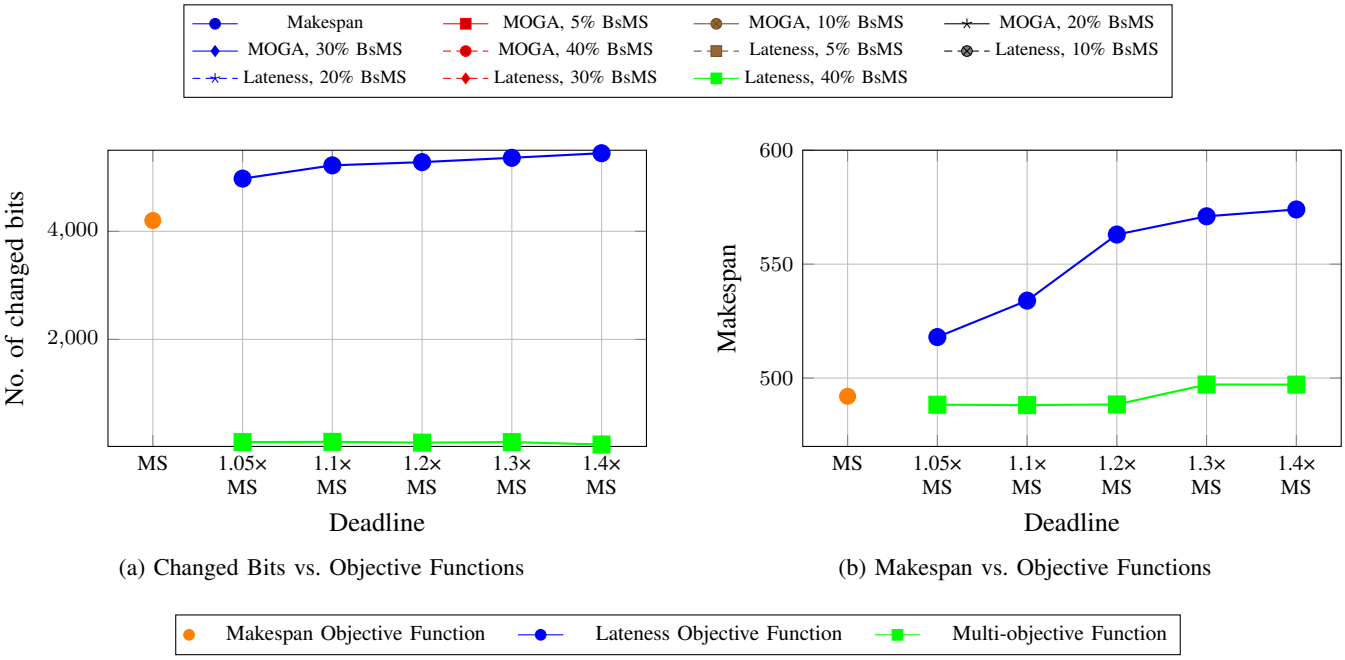


Figure 6. Number of Changed Bits



(a) Changed Bits vs. Objective Functions

(b) Makespan vs. Objective Functions

Figure 7. MOGA achieves over 90% fewer changed bits with under 5% makespan overhead for the 20-task application model.

base schedule makespan (BsMS) results in only 100 changes compared to 4,200 for the metascheduler with makespan minimization objective, a remarkable 97.6% reduction. Similarly, for the application model of a hundred tasks, MOGA, with a deadline of 1.05 times the base schedule makespan (BsMS), achieves 7,980 changes compared to 14,730 for makespan, representing a 45.8% improvement. Moreover, MOGA significantly outperforms the metascheduler, optimizing the lateness objective. For instance, in the case of the application model of twenty tasks, the metascheduler optimizing the lateness with a deadline of 1.05 times the base schedule makespan (BsMS) produces 4,975 changes, 97.9% higher than MOGA's result. The hundred-task application model produces 16,570 changes, more than double that of MOGA.

The results indicate that the metascheduler optimizing the schedule's makespan achieves the lowest average makespan values across all application model sizes, with minimal growth of approximately 10% as the model size increases from twenty to a hundred tasks. MOGA metascheduler performs competitively at smaller model sizes but exhibits a higher makespan overall. The Lateness objective generally results in the highest makespan values, particularly at smaller sizes. However, its growth rate when varying the application model sizes from twenty to hundred tasks is closer to that of Makespan than MOGA.

Figure 7 provides an in-depth comparison of the performance of our proposed MOGA method against the makespan and lateness objectives for the twenty-task application model. Figure 7a shows that MOGA significantly reduces the number of changed bits across all deadlines, achieving far fewer modifications compared to the other methods. Figure 7b illustrates that while the makespan objective achieves the lowest

makespan, MOGA maintains competitive performance with minimal increases, effectively balancing makespan optimization and schedule stability.

These results suggest that while the makespan metascheduler prioritizes minimizing makespan values, our proposed MOGA metascheduler offers a balanced trade-off by achieving competitive makespan values while potentially reducing the memory space required for storing the MSGs.

VI. Conclusion

This paper presents a Multi-Objective Genetic Algorithm (MOGA)-based metascheduler that optimizes memory usage by minimizing schedule disruptions during context events and storing changes incrementally. The algorithm efficiently balances memory utilization and deadline adherence.

Experimental results show that the MOGA-based metascheduler significantly reduces schedule changes compared to makespan and lateness-based metaschedulers. For example, in the twenty-task application model, MOGA achieves a 97.6% and 97.9% reduction in changed bits compared to makespan and lateness-based approaches, respectively. In larger models, such as the hundred-task application, MOGA reduces schedule changes by over 45%, translating to substantial memory savings.

While the makespan-based metascheduler minimizes makespan values, it does so with significantly higher schedule disruptions. Conversely, the lateness-based metascheduler yields higher makespan values and more significant disruptions. Our proposed MOGA-based metascheduler offers a trade-off between maintaining competitive makespan values and minimizing schedule changes, offering a scalable and efficient solution for dynamic scheduling in resource-constrained environments.

Acknowledgment

This work has been supported by the research project EcoMobility in part by the EC under grant number 101112306 and the BMBF under grant number 16MEE0316.

SPONSORED BY THE

References

[1] J. C. Knight, "Safety critical systems: challenges and directions," Proceedings of the 24th International Conference on Software Engineering. ICSE 2002, Orlando, FL, USA, 2002, pp. 547-550.

[2] S. Skalistis and A. Kritikakou, "Timely Fine-Grained Interference-Sensitive Run-Time Adaptation of Time-Triggered Schedules," 2019 IEEE Real-Time Systems Symposium (RTSS), Hong Kong, China, 2019, pp. 233-245, doi: 10.1109/RTSS46320.2019.00030.

[3] Clinton V. Oster, John S. Strong, C. Kurt Zorn, Analyzing aviation safety: Problems, challenges, opportunities, Research in Transportation Economics, Volume 43, Issue 1, 2013, Pages 148-164, ISSN 0739-8859, https://doi.org/10.1016/j.retrec.2012.12.001.

[4] M. Krstic, A. Simevski, M. Ulbricht and S. Weidling, "Power/Area-Optimized Fault Tolerance for Safety Critical Applications," 2018 IEEE 24th International Symposium on On-Line Testing And Robust System Design (IOLTS), Platja d'Aro, Spain, 2018, pp. 123-126, doi: 10.1109/IOLTS.2018.8474178.

[5] Petrovic, V., Krstic, M.D. (2015). Design Flow for Radhard TMR Flip-Flops. 2015 IEEE 18th International Symposium on Design and Diagnostics of Electronic Circuits & Systems, 203-208.

[6] Obermaisser R, Ahmadian H, Maleki A, Bebawy Y, Lenz A, Sorkhpour B. Adaptive Time-Triggered Multi-Core Architecture. Designs. 2019; 3(1):7. https://doi.org/10.3390/designs3010007

[7] Muoka, Pascal & Umuomo, Oghenemaro & Onwuchekwa, Daniel & Obermaisser, Roman. (2023). Adaptation for Energy Saving in Time-Triggered Systems Using Meta-scheduling with Sample Points. 10.1007/978-3-031-34214-1_3.

[8] Sorkhpour, Babak & Obermaisser, Roman & Murshed, Ayman. (2017). Meta-Scheduling Techniques for Energy-Efficient, Robust and Adaptive Time-Triggered Systems. 10.1109/KBEI.2017.8324961.

[9] Yao, G., Pellizzoni, R., Bak, S. et al. Memory-centric scheduling for multicore hard real-time systems. Real-Time Syst 48, 681–715 (2012). https://doi.org/10.1007/s11241-012-9158-9

[10] D. Onwuchekwa, M. Dasandhi, S. Alshaer and R. Obermaisser, "Evaluation of AI-based Meta-scheduling Approaches for Adaptive Time-triggered System," 2023 International Conference on Smart Computing and Application (ICSCA), Hail, Saudi Arabia, 2023, pp. 1-8, doi: 10.1109/ICSCA57840.2023.10087446.

[11] S. Alshaer, C. Lua, P. Muoka, D. Onwuchekwa and R. Obermaisser, "Graph Neural Networks Based Meta-scheduling in Adaptive Time-Triggered Systems," 2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA), Stuttgart, Germany, 2022, pp. 1-6, doi: 10.1109/ETFA52439.2022.9921580.

[12] Muoka, P.; Onwuchekwa, D.; Obermaisser, R. Adaptive Scheduling for Time-Triggered Network-on-Chip-Based Multi-Core Architecture Using Genetic Algorithm. Electronics 2022, 11, 49. https://doi.org/10.3390/electronics11010049

[13] Steven W. Smith. 1997. The scientist and engineer's guide to digital signal processing. California Technical Publishing, USA.

[14] Pranab K. Muhuri, Amit Rauniyar, Rahul Nath, On arrival scheduling of real-time precedence constrained tasks on multi-processor systems using genetic algorithm,Future Generation Computer Systems, https://doi.org/10.1016/j.future.2018.10.013.

[15] O. Hekal, D. Onwuchekwa and R. Obermaisser, "Incremental Scheduling Using Genetic Algorithm," 2024 International Symposium ELMAR, Zadar, Croatia, 2024, pp. 269-275, doi: 10.1109/ELMAR62909.2024.10694420.

[16] F. Pezzella, G. Morganti, G. Ciaschetti, A genetic algorithm for the Flexible Job-shop Scheduling Problem, Computers & Operations Research, Volume 35, Issue 10, 2008, Pages 3202-3212, ISSN 0305-0548, https://doi.org/10.1016/j.cor.2007.02.014.

[17] H. Liu, C. Jin, B. Yang and A. Zhou, "Finding Top-k Shortest Paths with Diversity," in IEEE Transactions on Knowledge and Data Engineering, vol. 30, no. 3, pp. 488-502, 1 March 2018, doi: 10.1109/TKDE.2017.2773492.

[18] Deb, Kalyan. (2001). Multiobjective Optimization Using Evolutionary Algorithms. Wiley, New York.

[19] K. Deb, A. Pratap, S. Agarwal and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," in IEEE Transactions on Evolutionary Computation, vol. 6, no. 2, pp. 182-197, April 2002, doi: 10.1109/4235.996017.

[20] Talebian, Seyed & Abdul Kareem, Sameem. (2010). A Lexicographic Ordering Genetic Algorithm for Solving Multi-objective View Selection Problem. Computer Research and Development, International Conference on. 110-115. 10.1109/ICCRD.2010.81.

[21] Leskovec J, Sosič R. SNAP: A General Purpose Network Analysis and Graph Mining Library. ACM Trans Intell Syst Technol. 2016 Oct;8(1):1. doi: 10.1145/2898361. Epub 2016 Oct 3. PMID: 28344853; PMCID: PMC5361061.

[22] Universitat Siegen, https://cluster.uni-siegen.de.

[23] Kim, J., Yoo, S. Software review: DEAP (Distributed Evolutionary Algorithm in Python) library. Genet Program Evolvable Mach 20, 139–142 (2019). https://doi.org/10.1007/s10710-018-9341-4