Incremental Multi-Schedule Graph for Memory Optimization in 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*—Safety-critical systems operate in environments where failures can lead to catastrophic consequences, making reliability and efficiency crucial. These systems often rely on time-triggered scheduling to ensure deterministic execution and fault tolerance. Multi-Schedule Graphs (MSG), resulting from deploying metaschedulers, provide adaptive behaviour for time-triggered systems. However, MSGs are prone to state explosion in dynamic environments with increasing tasks and events. This rapid growth in the MSG's complexity increases the memory requirements and makes real-time adaptation challenging to manage efficiently. So far, mitigation strategies for avoiding state explosion have not been introduced for systems with incremental scheduling. This paper introduces an Incremental Multi-Objective Genetic Algorithm (MOGA)-based metascheduler to address the state explosion problem while ensuring timeliness. Our approach provides a trade-off between optimal makespan and memory utilization, leveraging incremental scheduling to retain only new information while minimizing redundant storage. We support scenarios where an application is added incrementally, and its corresponding MSG needs to be computed alongside an existing base application MSG, ensuring seamless integration of new tasks and context events. Experimental results show that our incremental MOGA-based metascheduler reduces MSG size by up to 60% compared to a traditional makespan-based metascheduler, significantly optimizing memory usage while preserving execution efficiency. The proposed approach maintains near-optimal makespan values. The results highlight the effectiveness of incremental metascheduling in balancing scalability, resource efficiency, and execution performance for dynamic safety-critical systems.**

***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 essential for ensuring consistent performance and safety in dynamic environments. To achieve reliability, it is important to incorporate fault-tolerance mechanisms, such as redundancy. This can be implemented through hardware, software, information, or a combination of these methods to mitigate the effects of faulty operations [4]. A well-known approach for addressing both permanent and transient faults is Triple Modular Redundancy (TMR). TMR adds two replicas and a majority voter to a module, ensuring functionality despite a single fault. TMR is widely used in fields such as aeronautics and space. However, this method has significant drawbacks, including power and area overheads that can increase up to 400% due to the additional voter and the necessary spacing requirements, which makes it inefficient [5]. Overhead is particularly undesirable in environments with limited resources; for example, in space, the capacity of solar panels constrains energy resources. Power limitations are even more stringent in mobile applications like Internet of Things (IoT) devices.

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 optimized 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, particularly in dynamic environments characterised by heterogeneous task arrivals requiring schedules to be frequently adapted [9]. In such scenarios, the increasing complexity of tasks, messages, and dependencies makes it even more challenging to manage the MSG effectively, as the size of the MSG depends on the application, platform, and context models. The MSG's size grows expo-

nentially with the number of tasks and messages within the application model, as the schedule must account for both temporal and spatial allocations for the application model elements [6]. This rapid growth and limited memory resources in real-time multi-core systems increase the challenge of achieving efficient and responsive scheduling.

Incremental scheduling is updating an existing schedule to accommodate changes, such as new tasks, revised constraints, or unforeseen disruptions, without fully recomputing it. This approach is vital for maintaining efficiency and avoiding revalidation costs in dynamic environments like manufacturing, transportation, healthcare, and satellite communications, where conditions often shift. It offers high schedule stability, essential for uninterrupted execution, even when faced with unexpected events. For example, managing communication events for NASA's Goddard Space Flight Center (NASA-GSFC) Tracking and Data Relay Satellite System (TDRSS) is challenging due to limited resources such as antennas, ground equipment, and bandwidth, as well as constraints like TDRS visibility and the precise timing of requests. These constraints often interact dynamically, requiring careful placement of new tasks relative to existing ones. Incremental scheduling provides a valid solution by enabling real-time adjustments as changes or new requests arise, minimizing disruptions, optimizing resource usage, and ensuring smooth operations in such complex and dynamic environments [10].

This paper presents an incremental metascheduler to tackle the state explosion problem in dynamic environments. The proposed metascheduler is an Incremental Genetic Algorithm (GA)-based metascheduler that offers a trade-off between having short makespan values and memory utilization. We assume a scenario where an application comes into action, and the multi-schedule graph of the newly added application is calculated on top of that for the existing base schedule. This process ensures that the resulting MSG incorporates scenarios for both applications. By integrating the new application's requirements with the existing schedule, the metascheduler effectively adapts to dynamic changes while maintaining a balance between performance and memory constraints. 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 maximize the stability of the currently running 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 incremental MOGA-based metascheduler to a metascheduler that optimizies the makespan with respect to memory requirements. The results showed that our proposed incremental-MOGA metascheduler consistently outperforms the makespan-based metascheduler regarding memory requirements for different application sizes, indicating an efficient adaptation solution to dynamic environments. Specifically, for the large base and incremented application sizes, Our proposed incremental MOGA-metascheduler reduces the MSG size by up to 60% compared to the makespan-based metascheduler. This reduction highlights the algorithm's ability to maintain schedule stability while incorporating new tasks or context events. Furthermore, it maintains competitive makespan values, with only a marginal increase compared to the makespan-based metascheduler, ensuring near-optimal execution times.

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 plays a vital role in improving time-triggered systems' adaptive capacities. These systems operate based on strictly defined schedules, ensuring predictability and reliability, which are crucial for safety-critical applications. Metascheduling enables schedule switching in response to changing system conditions or requirements, allowing the system to accommodate factors such as slack or failures while adhering to strict timing constraints. By incorporating metascheduling, time-triggered systems gain enhanced flexibility and resilience, leading to optimized resource utilization and, most importantly, 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. Moreover, they did not consider scenarios where a new application is introduced to the system, which is a critical aspect of dynamic systems. The inability to handle the state space explosion problem limits the scalability of their approaches, making them less practical for large and complex workloads. Additionally,

their methods focus primarily on energy efficiency without considering the impact on schedule stability and adaptability when new tasks or applications enter the system.

To address these gaps, we propose an incremental MOGA-based metascheduler that efficiently adapts schedules while mitigating the state space explosion problem. Unlike previous approaches, our method explicitly considers incremental tasks, integrating new applications into the existing schedule with minimal disruption. By constructing a multi-schedule graph (MSG), our approach ensures efficient adaptation without requiring complete rescheduling besides maintaining timing guarantees. This strategy enhances system scalability and adaptability, making it well-suited for dynamic, time-triggered environments.

While machine learning-based scheduling approaches have shown promise in optimizing schedules and reducing storage requirements, their effectiveness in safety-critical environments remains uncertain due to inconsistent model performance. Furthermore, existing studies as in [11], and [12] do not incorporate deadline constraints, which are crucial for real-time systems. In contrast, our proposed incremental MOGA-based metascheduler explicitly integrates deadline awareness while ensuring predictability and stability in scheduling. By balancing makespan optimization and schedule adaptability, our approach provides a more reliable and scalable solution, making it better suited for dynamic, safety-critical applications.

In [13], the authors proposed a solution to address the state-space explosion issue in MSGs by introducing a path reconvergence algorithm. This approach minimizes the number of generated schedules by merging redundant transitions that appear along different paths within the scheduling tree. However, a key limitation is that if the new schedule only partially differs from the previous one, the entire new schedule must still be stored, leading to potential inefficiencies in memory usage.

Our proposed method effectively addresses the state-space explosion problem by utilizing a memory-efficient storage strategy. Instead of storing complete schedules, we use delta encoding [14], which only retains the differences between new schedules and their parent schedules. This significantly reduces memory requirements. By capturing incremental changes rather than duplicating redundant data, our approach minimizes the number of unique schedules that need to be stored, thereby shrinking the overall state space. In addition, our method explicitly accommodates incremental tasks by updating the MSG of the existing application to seamlessly integrate new tasks and their respective context events. This approach ensures minimal disruption to the current schedule while maintaining adaptability and scalability in dynamic environments.

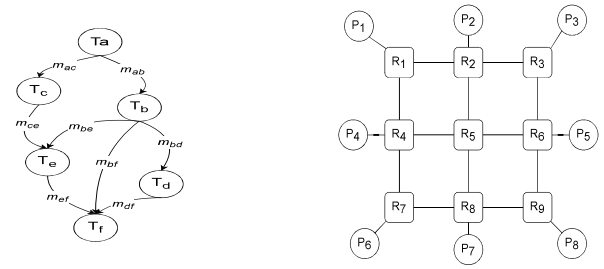
III. System model

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). In the following lines, we explain each of these entities.

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 as shown in Figure 1a refers to the computational tasks and their specific attributes, such as processing times and resource requirements. It 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. 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) [15] [16].



(a) Application Model (b) Platform Model

Figure 1. Comparison of the Application and Platform Models

2) Platform Model (PM): The platform model shown in Figure 1b provides information about the computational resources required to execute the application model. It can be represented as an undirected graph $\mathcal{P}(\mathcal{N}, \mathcal{L})$, where $\mathcal{N}$ denotes the set of routers and cores, and $\mathcal{L}$ represents the bi-directional links connecting them. A path within $\mathcal{P}$, starting from core P_i to $P_k \in \mathcal{N}$, is defined as a sequence $\langle R_i, ..., R_k, P_k \rangle$ of vertices linked by connections $l_{ik} \in \mathcal{L}$ for $i = 1, 2, ..., k$. In the figure, cores $P_i \in \mathcal{N}$ are labelled with the symbol P, while routers are labelled with the symbol R.

3) Context model: The context model encompasses events crucial for adaptation, including faults, dynamic slack, resource alerts, and environmental changes. Faults, such as permanent failures of cores or routers, require recovery actions like task and message reallocation. Another example is dynamic slack, where a task is completed earlier than its worst-case execution time (WCET). This results in messages being sent earlier than expected in the time-triggered schedule, creating opportunities to apply energy-saving techniques, such as Dynamic Voltage and Frequency Scaling (DVFS), without negatively impacting the rest of the system.

B. Output Model

1) Incremental Metascheduler: A metascheduler is an offline scheduling framework for computing time-triggered schedules while considering spatial, temporal, and contextual system dimensions at design time. The metascheduler utilizes the application, platform and context models as inputs for schedules S_i computation, which accumulate and construct the 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. 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. When a new application is introduced while the system is operating at one of the MSG nodes, the metascheduler is activated to adjust the schedule. It calculates new schedules that update the current MSG, integrating the context events of both the existing applications and the newly arrived one. This process involves incrementally updating the existing MSG rather than completely recomputing it, which helps preserve previously computed schedules and reduces computational overhead. The new schedules take into account the temporal and spatial requirements of the new application while ensuring compatibility with the current state of the system. By doing this, the metascheduler effectively refreshes the MSG to reflect the combined set of context events for all applications. This ensures system adaptability and maintains timing guarantees, even in the face of dynamic changes in workload.

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. An incremental schedule example can be seen in the Figure 2.

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. 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 introduces our innovative approach: an incremental genetic algorithm (GA)-based metascheduler designed to address the state explosion problem. Our method utilizes a Multi-Objective Genetic Algorithm (MOGA) to optimize

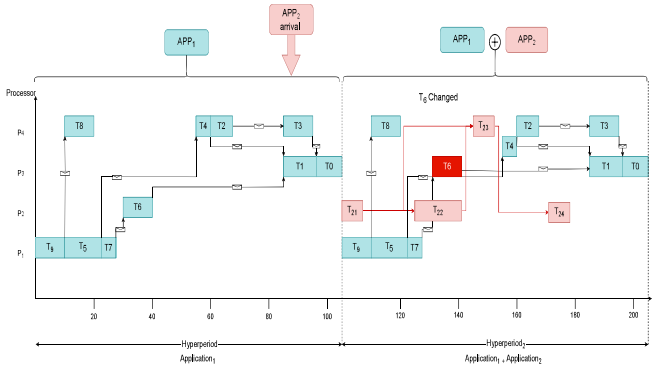


Figure 2. An example to an incremental time-triggered schedule, with minimum changes with respect to the base schedule

the schedule's makespan and memory utilization. The algorithm minimizes schedule deviations in response to context events—such as dynamic slacks or failures—and manages the arrival of new applications that require resource allocation, ensuring all applications meet their respective deadlines.

Additionally, our proposed metascheduler considers context events related to newly introduced applications, guaranteeing that the resulting Multi-Schedule Graph (MSG) integrates all possible scenarios for both existing and new applications. A key feature of our approach is the maximization of structural similarity between parent and child schedules within the MSG. This facilitates an efficient incremental storage mechanism, where only new information from the updated (child) schedule is saved. In contrast, unchanged portions from the previous (parent) schedule are retained, effectively minimizing redundant data storage and enhancing computational efficiency.

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 [16] [17]. Algorithm (5) provides a detailed overview of the genetic algorithm's working principle implemented in this study.

*2) **Genome description**:* In our approach, schedules are derived from the genomes within the population. Each chromosome represents a potential solution to the scheduling problem and consists of four genomes, each responsible for a specific aspect of scheduling: task ordering, processor allocation, message path indexing, and message priority ordering. The task ordering genome is initialized using a random permutation of task identifiers, ranging from zero to the highest task ID minus one. The processor allocation genome is generated by randomly assigning tasks to processing elements while ensuring that only valid computational units (excluding routers) are selected. This constraint guarantees that tasks are mapped exclusively to appropriate processing units within the system. The message path index genome defines the

Algorithm 1 GA for Task Scheduling Optimization

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

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

communication route from a source processor to one or more destination processors. This index represents the sequence of communication links used for message transmission. The paths are precomputed using the k-shortest path method [18], with each path assigned a cost based on the number of hops between the connected processors. All possible paths and their corresponding costs are computed in prior. The genetic algorithm prioritizes selecting communication paths that minimize overall transmission costs, thereby reducing latency and resource overhead while enhancing system efficiency. The message priority ordering genome determines the sequence of message exchanges between processing elements to optimize task execution. This ordering optimizes the overall makespan by prioritizing critical message transmissions, reducing communication delays, and ensuring effective task execution. It alleviates resource contention by managing message transmissions along overlapping paths according to priority levels. Implementing this structured prioritization scheme minimizes conflicts, streamlines communication flow, and enhances overall scheduling performance.

*3) **Chromosome selection**:* In the proposed MOGA-based metascheduler, the selection operator balances competing objectives by ensuring that application deadlines are met while minimizing schedule modifications due to context changes. Schedules that best satisfy both objectives receive higher fitness scores. Multi-Objective Evolutionary Algorithms (MOEAs) [19] [20] identify multiple Pareto-optimal solutions in a single run, enabling trade-off exploration between conflicting objectives. Our MOGA employs Pareto-based selection, such as non-dominated sorting, to maintain diversity and optimize makespan and memory utilization, improving system adaptability. To handle conflicting objectives, we use lexicographic ordering, prioritizing schedule similarity [21]. The Jaccard similarity metric measures the overlap between consecutive schedules after a context event, assessing how much of the original schedule remains unchanged. A higher similarity score indicates minimal deviations, ensuring stability while efficiently adapting to changes.

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)

B. Schedule reconstruction logic

The reconstruction logic presented in this paper aims to optimize task scheduling in a parallel computing environment. The genetic algorithm produces final genomes, which serve as input parameters for the reconstruction function. The *processing_times* attribute specifies the execution time required for each task, while the *message_list* contains details of inter-task communications, both derived from the application model. Additionally, the *Path_index_with_cost* provides routing information between processors via routers and the associated communication costs in the platform model. The reconstruction logic uses these inputs to assign tasks to processors and schedule their execution once dependencies

are resolved, including the receipt of necessary messages from other tasks. To ensure efficient resource utilization, the function accounts for communication delays by incorporating message transfer costs between processors. It dynamically adjusts task start times based on processor availability and the completion of preceding tasks. The pseudo-algorithm outlining the reconstruction process is detailed in Algorithm (2).

C. Incremental Metascheduler for memory optimization

The proposed metascheduler computes time-triggered schedules that adapt to various context events by precomputing schedules for all possible event sequences within a hyper-period during the design phase. These schedules are deployed at runtime at the beginning of each hyper-period. As context events occur, new schedules are generated, resulting in an MSG. However, as the number of context events increases, the number of schedules grows exponentially, making large-scale storage impractical. To address this challenge, the metascheduler employs two key strategies: first, it minimizes modifications to schedules in response to context events (as explained earlier in IV-C)), ensuring that changes from the original schedule are kept to a minimum; second, it uses an incremental storage mechanism that retains only the differences between successive schedules instead of storing entire schedules, which significantly reduces memory overhead. Additionally, the metascheduler considers scenarios where tasks are incrementally added to the system, along with their respective context events, ensuring that the resulting MSG adapts to both existing and newly introduced applications. This approach enhances adaptability while maintaining an efficient storage structure, optimizing both computational efficiency and resource utilization. The pseudo-algorithm for MOGA-based metascheduler is stated in Algorithm (3).

Algorithm 3 Incremental 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.(5)*
  If ($context\_event_{AM\_Base}$ = none):
    $S_{deployed} = S_o$
  ElseIf ($context\_event_{AM\_Base}$ = True):
    $AM_{updated}$, $PM_{updated}$ = *Apply_Context_event*
    S_{new} = Invoke GA // *Alg.(5)*
    Compute $\Delta_{schedules} = S_{new}$ - ($S_{new} \cap S_{predecessor}$)
    Add the node S_{new} to MSG // Δ information only
  ElseIf (AM_{new} = True &
  $context\_event_{AM\_new} || context\_event_{AM\_old}$ =True):
    $AM_{updated} = AM_{Base} \cup AM_{new}$
    $AM_{updated}$, $PM_{updated}$ = *Apply_Context_event*
    S_{new} = Invoke GA // *Alg.(5)*
    Compute $\Delta_{schedules} = S_{new}$ - ($S_{new} \cap S_{predecessor}$)
    Add the node S_{new} to MSG
return MSG

V. Experiments and Results

This section outlines the experimental procedures and their corresponding results. To create example graphs for both the application and platform models, we utilized the Stanford Network Analysis Platform (SNAP) [22]. SNAP is a powerful library designed for graph and network analysis, allowing us to generate a variety of scenarios by applying network (graph) theory. It provides essential functions for processing, analyzing, and manipulating large-scale networks. Using SNAP, we established key parameters such as the number of computational tasks in the application model and the structural composition of the platform model, which included processing elements and routers. These functions enabled us to create multiple configurations of application and platform models, ensuring a wide range of experimental conditions. To generate the application model, we executed the relevant code on the OMNI cluster at the University of Siegen [23], which allowed us to utilize its computational resources for efficient large-scale graph generation. Additionally, we implemented the genetic algorithm using the Distributed Evolutionary Algorithm in Python (DEAP) library [24], which provides a robust framework for evolutionary computation and multi-objective optimization in our scheduling approach. Using a random forest-fire model, we generated various task graphs for the application models, which defined structural properties such as the number of nodes (tasks), edges (dependencies between tasks), in-degree distributions, and out-degree distributions. The experimental setup included application models with task sizes ranging from thirty to a hundred tasks, allowing for an extensive evaluation of scalability and performance. We used application models with sizes of five, ten, and fifteen tasks for the incremental applications. The platform model consisted of eight homogeneous processing elements interconnected via a 3×3 mesh network of routers, as illustrated in Figure 1b. This network topology was chosen to simulate a realistic parallel computing environment with structured communication constraints. The incremented applications were considered to be independent from the base applications. We studied a scenario in which only one new application needs to run alongside the base application. To create the MSG for different test cases, we used fifty context events. The resulting MSG includes schedules for situations where the base application model initially utilizes all system resources. Additionally, it takes into account scenarios where the new application requires resource allocation, considering the relevant context events associated with both applications.

We compare our proposed incremental MOGA-based metascheduler to another GA-based metascheduler, optimizing the schedule's makespan. We assume different application deadline values, where the values were 20% or 30% or 40% of the base schedule makespan for each application model size. The genetic algorithm (GA) parameters remained constant across all application model (AM) sizes for the three metaschedulers. The population size was 200, ensuring a diverse set of candidate solutions in each generation. The

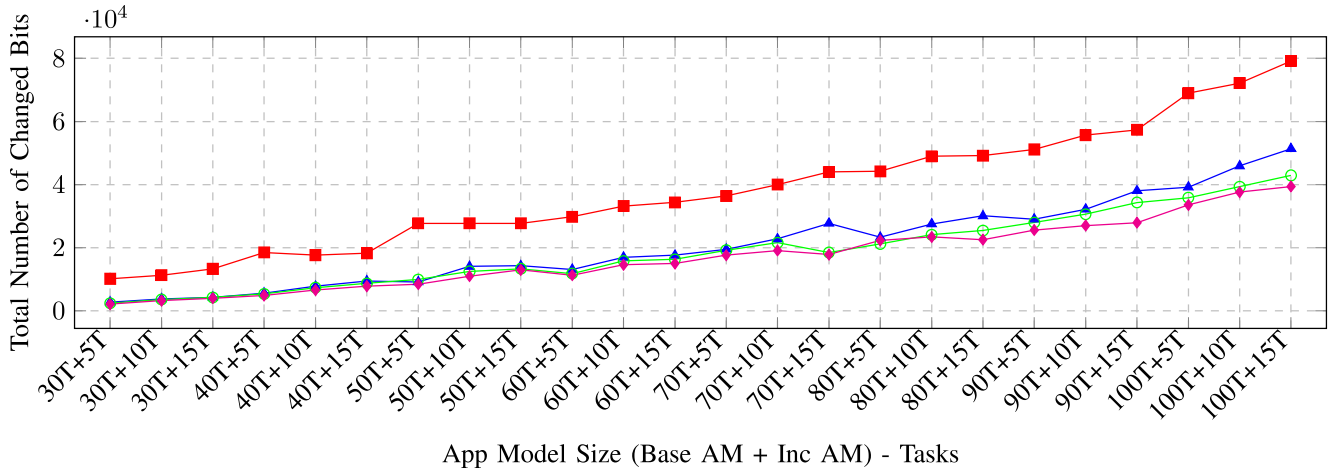


Figure 3. Total Changed bits throughout the entire MSG per one Base AM + one Inc.AM for different metascheduler variations

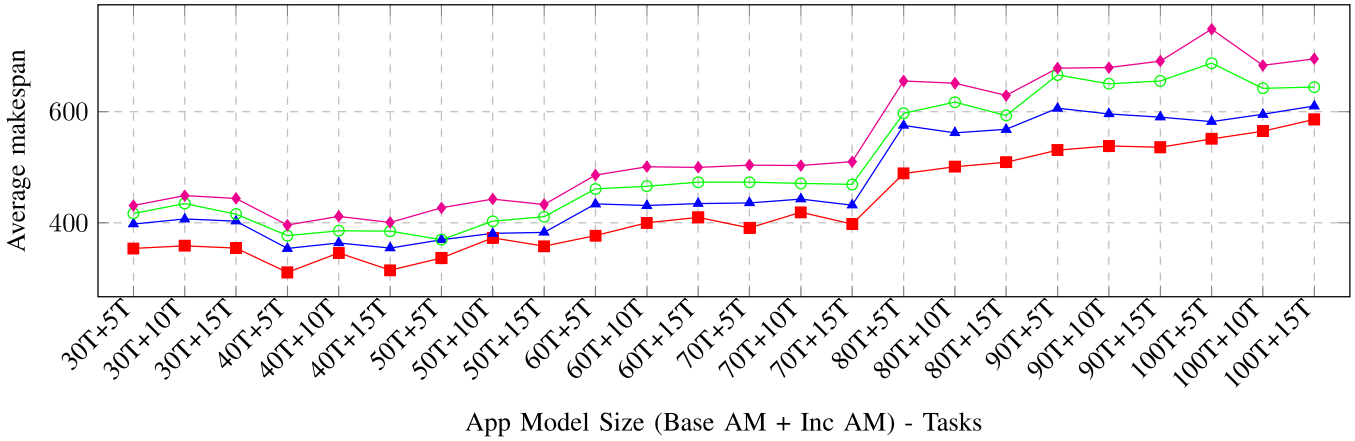
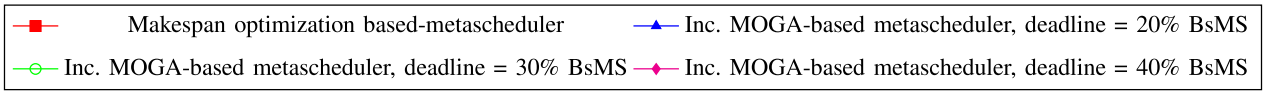


Figure 4. Average makespan throughout the entire MSG per one Base AM + one Inc. AM for different metascheduler variations



crossover probability was fixed at 0.4, allowing genetic material exchange between parent solutions to facilitate search space exploration. The mutation probability was set to 0.6, promoting diversity by introducing random solution variations. Lastly, the number of generations was set to 400, providing sufficient iterations for convergence toward optimized scheduling solutions.

The objective of this comparison is to evaluate the impact of the proposed method on the memory size required to store the MSG. The ”number of changed bits” metric indicates the amount of new information that needs to be stored after computing a new schedule in the MSG. Fewer changes are preferable, as reducing schedule modifications directly leads to a significant decrease in the memory space required for storing the MSG, thereby improving overall system resource efficiency.

As explained in Section IV-C, we only store the information that has changed about the schedule compared to its previous state in the MSG. Additionally, we compare the average makespan values across the entire MSG for both metaschedulers. The makespan represents the total time required to complete all application tasks.

The data presented in Figures 3 and 4 illustrates that the makespan optimization-based metascheduler leads to excessive schedule modifications, which increase as the application size grows. In contrast, the Inc. MOGA-based metaschedulers, which utilize different application deadline variations (20%, 30%, and 40% of the Base Schedule Makespan (BsMS)), consistently reduce the number of changed bits. Among these variations, those with application deadlines set at 30% and 40% of the BsMS demonstrate the best memory efficiency. This means that less new information needs to be stored, resulting in lower memory requirements and a more efficient, scalable scheduling process.

Our proposed incremental MOGA-based metascheduler effectively reduces memory modifications, though it does result in a moderate increase in makespan. The MOGA 20% BsMS leads to a slight increase in makespan while achieving notable memory savings. In contrast, the MOGA 30% BsMS option offers the best balance between memory optimization and execution time. Meanwhile, MOGA 40% BsMS maximizes memory efficiency but slightly increases makespan, making it more suitable for applications where memory optimization is prioritized over execution speed. These results show the effectiveness of our incremental MOGA-based approach in creating an adaptive scheduling strategy that balances memory optimization and execution performance, making it well-suited for dynamic environments.

VI. Conclusion

We introduce an incremental MOGA-based metascheduler designed to efficiently handle scenarios where tasks are added incrementally to the system. Our approach offers a trade-off between memory optimization and optimal makespan, ensuring efficient resource utilization without excessive execution delays. By updating schedules incrementally, we tackle the state space problem, minimizing unnecessary memory modifications while maintaining stable system performance. The results demonstrate that MOGA 30% BsMS achieves the best balance, making our method highly suitable for real-time computing, cloud scheduling, and HPC workloads. Future work will focus on enhancing adaptability through dynamic scheduling policies and extending applicability to heterogeneous computing 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] Jidé Odubiyi, David Zoch, A heuristic approach to incremental and reactive scheduling,Telematics and Informatics,Volume 6, Issues 3–4,1989,Pages 171-180,ISSN 0736-5853, https://doi.org/10.1016/S0736-5853(89)80014-0.

[11] 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.

[12] 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.

[13] 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

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

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

[24] 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