Article

Adaptive Scheduling for Time-Triggered Network-on-Chip-Based Multi-Core Architecture Using Genetic Algorithm

Pascal Muoka *,†,‡, Daniel Onwuchekwa †,‡ and Roman Obermaisser †,‡

Department of Electrical Engineering and Computer Science, University of Siegen, 57076 Siegen, Germany; daniel.onwuchekwa@uni-siegen.de (D.O.); roman.obermaisser@uni-siegen.de (R.O.)

* Correspondence: pascal.muoka@uni-siegen.de

† Current address: Hölderlinstraße 3, 57076 Siegen, Germany.

‡ These authors contributed equally to this work.

Abstract: Adaptation in time-triggered systems can be motivated by energy efficiency, fault recovery, and changing environmental conditions. Adaptation in time-triggered systems is achieved by preserving temporal predictability through metascheduling techniques. Nevertheless, utilising existing metascheduling schemes for time-triggered network-on-chip architectures poses design time computation and run-time storage challenges for adaptation using the resulting schedules. In this work, an algorithm for path reconvergence in a multi-schedule graph, enabled by a reconvergence horizon, is presented to manage the state-space explosion problem resulting from an increase in the number of scenarios required for adaptation. A meta-scheduler invokes a genetic algorithm to solve a new scheduling problem for each adaptation scenario, resulting in a multi-schedule graph. Finally, repeated nodes of the multi-schedule graph are merged, and further exploration of paths is terminated. The proposed algorithm is evaluated using various application model sizes and different horizon configurations. Results show up to 56% reduction of schedules necessary for adaptation to 10 context events, with the reconvergence horizon set to 50 time units. Furthermore, 10 jobs with 10 slack events and a horizon of 40 ticks result in a 23% average sleep time for energy savings. Furthermore, the results demonstrate the reduction in the state-space size while showing the trade-off between the size of the reconvergence horizon and the number of nodes of the multi-schedule graph.

Keywords: genetic algorithm; metascheduler; network-on-chip; MPSoC; adaptation; time-triggered systems

1. Introduction

The processing capacity of recent VLSI technology has considerably grown as several Processing Elements (PEs), tensor cores, memory elements and Intellectual Property (IP) cores are integrated onto a single chip. As a result, designers have shifted focus from traditional bus-based architectures where the different chip components are connected directly via dedicated links (point-to-point) to a Network-on-Chip (NoC)-based architecture where a switch-based network is used to route communication traffic between chip components. As more resources are added, the bus links become congested, to which scaling becomes a challenge. On the other hand, a Network-on-Chip indirectly connects chip components via a network of switches where each component interfaces the network through a network interface, combining the benefits of busses and point-to-point links. This architecture enables the NoC to be scaled by adding switches to the network.

Several application domains such as cloud computing, avionics, and multimedia now use NoC-based multi-core platforms [1]. Fault tolerance at a lower cost than active redundancy, energy efficiency, and adaptation to changing environmental conditions necessitates adaptation services to accommodate run-time changes. These run-time changes may arise from an execution slack (slack events), failures in system resources (failure events) or from a change in an operational mode requiring different application services [2]. These run time events are referred to in this work as context events. The adaptation to context events could be motivated by efforts to increase the reliability of a system through fault recovery.

It could also be motivated by ensuring a system's energy efficiency by providing an energy management service.

Current research such as in [3–5] promote time-triggered NoCs to fulfil determinism and safety requirements in time-triggered systems. However, the NoC-based platform requires a scheduling strategy for message exchange between connected PEs. Apart from using dynamic scheduling approaches, which is not suitable for safety-critical applications due to non-determinism, several scheduling-based approaches such as static scheduling have addressed run time changes via a hybrid approach. Static scheduling does not cover run time changes, but multiple potential allocation solutions can be obtained for several context events at design time. The context of hybrid scheduling discussed in this work is such that schedule changes are made at run time from multiple pre-computed static scheduling solutions obtained at design time, henceforth referred to as metascheduling. Metascheduling is also a solution to optimising pre-computed schedules. The pre-computed schedules are computed with worst-case execution times and accommodate other scenarios such as slack events. Therefore, the metascheduling approach provides a safe adaptation strategy as the run time changes involve transitions to an already verified schedule state.

Prior work has addressed computing a Multi-Schedule Graph (MSG) for time-triggered systems [6]. However, the increase in context events results in the exponential growth of the MSG. The MSG is a directed acyclic graph (DAG) of time-triggered schedules which form the vertices and context events that occupy the edges. The metascheduler pre-computes the MSG at design time using an application, platform and context model. A significant concern of the MSG is state-space explosion [2]. The exponential growth of the MSG caused by the increasing number of context events results in the state-space explosion. In addition, embedded devices often have limited storage space, which is challenging considering the memory required to store the generated schedules.

This work contributes the following.

- It introduces a Genetic Algorithm (GA)-based metascheduler for time-triggered NoC-based architectures.
- It presents a solution to combat the state-space explosion problem in the MSG using a reconvergence of paths algorithm.
- It implements a reconvergence horizon to maintain schedule generation within configured bounds.
- It further evaluates the reconvergence of paths algorithm using different input model sizes and configurations for the reconvergence horizon.

The remainder of this paper is organised as follows. Section 2 discusses the related work and Section 3 describes the input and output models. Section 4 discusses the GA-based metascheduling approach to manage state-space explosion. Experiments are discussed in Section 5 and results are presented. Finally, Section 6 concludes this paper.

2. Related Work

Metascheduling exploits the benefit of determinism in static scheduling and the flexibility of dynamic scheduling to achieve adaptation and efficiency. This scheduling technique is widely used in research for multi-core embedded systems to achieve reliability, efficiency, performance, fault tolerance, and lifetime extension.

Several scheduling algorithms have been used to generate task mapping solutions at design time. In [6], mixed-integer quadratic programming (MIQP) optimisation algorithm is used to generate schedules for time-triggered systems. Individual schedules are computed for relevant combinations of context events by applying application, platform, schedule, and context models to a metascheduler. After which, dynamic frequency scaling of hardware resources is used to optimise schedules for energy efficiency. This work did not consider the exponential growth of the schedule state space while finding optimal schedules with maximum energy efficiency.

In [7], a quantum-inspired evolutionary algorithm is used for static mapping of tasks to processing elements. The goal is to maximise the throughput of the target application.

This work determined a processor-to-processor mapping, and in the event of failure, the variation in task mappings for processor sets are used for migration. The generated schedules are encoded in a mapping table to manage the memory size. The temporal allocations of tasks are not considered. However, to avoid resource contention without dynamic arbitration, deadline misses, and race conditions, spatial and temporal allocations of tasks need to be considered, which is essential for time-triggered NoC. Furthermore, the size of the state space needed for adaptation to context events was not considered.

There is a linear relationship between the number of hardware resources or tasks and potential context events which could occur. In turn, computing a schedule for every context event increases the size of the MSG exponentially. With reconvergence of paths through a fixed reconvergence horizon, this relationship is polynomial [8].

In [9], an adaptive genetic algorithm is exploited for static scheduling such that physical process variations are considered for task mapping for a range of chips to generate schedules. An optimal schedule is then selected and mapped to the physical cores at run time. However, context events were not considered, such as the failure of these chips or energy efficiency, which is relevant for safety-critical systems.

A dark silicon/energy-aware core mapping technique is used for core mapping of application islands [10]. In this approach, mapping pools are combined into clusters and explored in a Pareto set, balancing energy with reliability. These clusters are then mapped to tiles on the die. A reliability prediction manager module within each NoC router is used at run time to toggle protection for reliability and energy efficiency. This work did not cover failures in hardware resources such as processing elements.

A population-based incremental learning algorithm is used in [11] for task mapping and scheduling at design time. The algorithm is also used to generate remapping schemas given failure events at run time. This work focused on the computational time without considering the size of the schedule state space.

The analysis of throughput and task-migration overhead in a Pareto space in response to different fault scenarios is used to construct a minimum-cost task mapping problem in [12]. An integer linear programming algorithm is then used to solve the task mapping problem. Finally, mappings are encoded in a look-up table and used to migrate tasks on fault occurrence at run time. In contrast, our work focuses on managing the number of states generated due to context events.

Most of the work discussed above predominantly solve task mapping problems without temporal allocation. In this work, a GA-based scheduler is used to compute schedules for adaptation at run time. It gives temporal, spatial, and contextual mapping solutions for time-triggered NoC-based multi-core platforms. A job allocation and job order problem is constructed in which GA is used to solve this problem while minimising the makespan of the schedule. The GA-metascheduler results in the generation of the MSG.

3. System Model

3.1. Input Models

To describe the scheduling problem of the metascheduler, we define an input model which consists of an application, platform and context model. The *application model* is a DAG, $\mathcal{G}(\mathcal{V}, \mathcal{E})$, where a vertex, $j_i \in \mathcal{V}$ represents job i, and the edge $m_{ik} \in \mathcal{E}$ is a message m representing the communication between the jobs $j_i, j_k \in \mathcal{V}$. Figure 1 illustrates an application model consisting of 10 jobs and 10 messages. The jobs are labelled 0–9 and messages a–j. A job $j_i \in \mathcal{V}$ has a computational cost which is the worst-case execution time $(wcet_i)$, and each message $m_{ik} \in \mathcal{E}$ has a message size ms_{ik}.

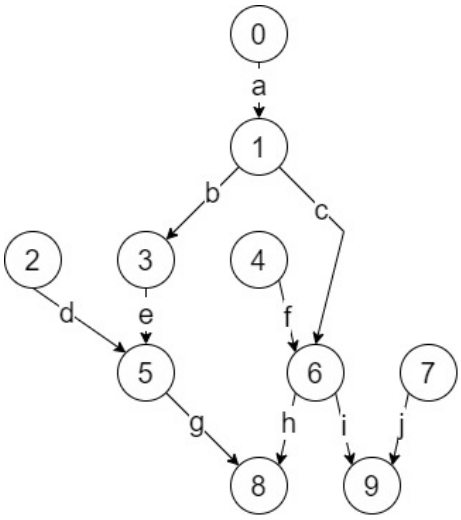

Figure 1. Illustration of an application model.

The *platform model* describes the computation and communication resources that are used to execute the application model. It is an undirected graph $\mathcal{P}(\mathcal{N}, \mathcal{L})$ where $\mathcal{N}$ is a set of routers and cores in the NoC topology, and $\mathcal{L}$ represents a set of bi-directional links connecting them. A path through $\mathcal{P}$ from a core es_i to $es_k \in \mathcal{N}$, is a sequence $\langle r_i, \ldots, r_k, es_k \rangle$ of vertices connected by edges $l_{ik} \in \mathcal{L}$ for $i = 1, 2, \ldots, k$ as shown in Figure 2. In this figure, the cores $es_i \in \mathcal{N}$ are indicated with the symbol *es* and routers with *r*. Two routers r_i, $r_k \in \mathcal{N}$ are linked by an edge $l_{ik} \in \mathcal{L}$. The number of hops of a path is the number of vertices in the path.

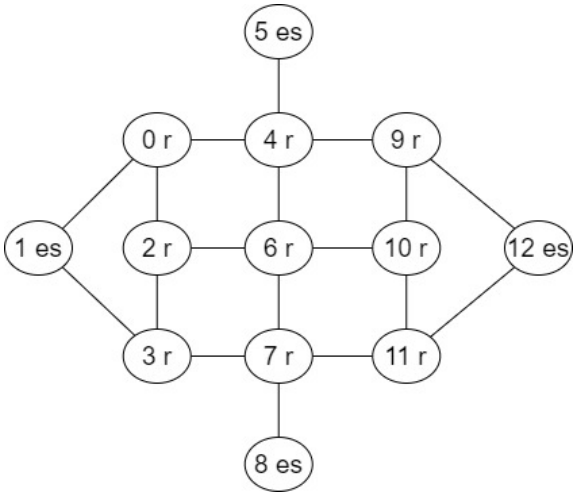

Figure 2. Physical structure of a platform model.

The *context model* is a set of context events $\mathcal{C}(\mathcal{S}, \mathcal{F})$, where $\mathcal{S}$ is a set of slack events $sl_{j,t}$ and $\mathcal{F}$, a set of failure events f_t. A slack event $sl_{j,t} \in \mathcal{S}$ of j_i is the difference between the worst case execution time $wcet_j$ and execution time et_j of j_i at time t, given by Equation (1). A failure event $f_t \in \mathcal{F}$ of node es_i, r_i or link l_{ik} is modelled as a permanent fault of a computation or communication resource. However, context events are constrained to independent time steps when observed through a calendar of events. Some events are mutually exclusive and a schedule is computed for each context event.

$$sl_{j,t} = wcet_j - et_j. \tag{1}$$

3.2. Output Model

The metascheduler generates an MSG, which is also a DAG of linked schedules. The schedule model describes the temporal and spatial allocation of jobs and messages such that resource restrictions and precedence constraints are observed. The schedules are non-preemptive static schedules that map the application model to the platform model ($map : \mathcal{G} \mapsto \mathcal{P}$), including the allocation of start time st_{ji} to job j_i. The start time st_{jk} of job j_k with preceding job j_i is given by Equation (2). Where t_f is the finish time of job j_i, which is equal to the arrival time of message m_{ik} at core es_k.

$$st_{jk}(es_k) \geq t_f(m_{ik}, j_i \mapsto es_i, es_k). \tag{2}$$

The communication cost of message m_{ik} is given by Equation (3).

$$mCost = \lceil ms_{ik} / packetsize \rceil * nHops * hoptime. \tag{3}$$

where $nHops$ is the number of hops between es_i and es_k, and $hoptime$ is the duration of one hop.

For every message $m_{ik} \in \mathcal{E}$, $j_i \in precedence(j_k)$ between j_i and j_k, a message path through the platform model $\mathcal{P}$ is computed. A message path $\langle r_i, \ldots, r_k, es_k \rangle$ that includes $j_i \mapsto es_i$ and $j_k \mapsto es_k$ is described in the schedule, which is the shortest path between es_i and es_k. To avoid resource contention, two messages with intersecting paths at a given time instant is handled via a priority scheme. A lower priority message is delayed for the transmission of a higher priority message.

4. Proposed Approach

A metascheduler computes an MSG at design time using the input models. The metascheduler repeatedly invokes a GA to solve a scheduling problem constructed by the metascheduler and computes the MSG. A detailed description of GA can be seen in [13]. The input to the GA-based scheduler is an application and platform model, from which a schedule is computed, fulfilling resource restrictions and precedence constraints. Several decision variables are considered in computing the schedule. These decision variables include job allocation $map : \mathcal{V} \mapsto \mathcal{P}$, job start $st_{jk}(es_k)$, message paths $\langle r_i, \ldots, r_k, es_k \rangle$, and message injection times.

Initially, a base schedule S_0 is computed from an initial $\mathcal{G}$ and $\mathcal{P}$, assuming no context event. After which the metascheduler applies the earliest event $e \in \mathcal{C}$. This results in the modification of $\mathcal{G} \; \forall \; sl_{j,t} \in \mathcal{S}$ or $\mathcal{P} \; \forall \; f_t \in \mathcal{F}$. In each case, the metascheduler invokes the GA, obtains a new schedule S_i and adds S_i to MSG. The context event e that results in S_i is represented as the corresponding edge in the MSG. The MSG computation also applies to mutually exclusive events. In this case, two slack events sl_{j_i,t_1} and sl_{j_i,t_2} are mutually exclusive as they both cannot occur within a time-triggered hyper period. If sl_{j_i,t_1} represents 50% slack and sl_{j_i,t_2} represents 20% slack, when a 50% slack occurs, a 20% slack event cannot occur for the same job within the same hyper period.

In the following subsection, the adaptation of the GA algorithm to solve time-triggered scheduling problems and the algorithm for the metascheduler are described.

4.1. Genetic Algorithm

The genetic algorithm (GA) is a search algorithm employing the principles of natural evolution to optimise a population of genomes. The initial population of genomes Pop_1 is randomly generated from an initial genome *prototype*, and the size of Pop_1 is a non-zero natural number. A solution to the scheduling problem is represented with a genome, which consists of the job allocation $map : \mathcal{V} \mapsto \mathcal{P}$ and the priority ordering of the jobs. Initially, a population of genomes is created, after which the population is evaluated, and the fittest genomes are selected for crossover and mutation to generate offspring and a new population.

Algorithm 1 describes the GA scheduler, adapted for a time-triggered NoC architecture (TTNoC). The algorithm uses the application model, platform model and a configuration file for the GA. The configuration file specifies the GA parameters, population size $popsz$, number of generation $ngen$, probability of mutation $pmut$, and crossover $pcross$. First, a prototype genome is constructed using $\mathcal{G}$ and $\mathcal{P}$. Next, the *prototype* genome is encoded with two sets of genes, job allocations and job weights. After which a population of genomes of size n is initialised from the prototype, where $n = popsz$. The *Evolve GA* function is then invoked, wherein the population is evaluated to obtain its fitness value, and the best genomes are retained in the next population. Each new generation is repeatedly evolved until a stop criterion is reached, after which a schedule from the fittest genome is converted and given as the optimal schedule.

Algorithm 1: Genetic Algorithm Adaptation for TTNoC Architecture.

```
Data: GA config = {popsz, ngen, pmut, pcross}, prototype
Input: G(V, E), P(N, L), GA config
Output: Single optimal schedule S_m
Function Evolve GA(){
    Evaluate Pop_i;
    Select parents;
    Crossover parents, produce offspring;
    Mutate offspring;
    Evaluate offspring;
    Insert offspring in Pop_k
    if termination criteria then
        Transform fittest genome to valid schedule;
        return valid schedule
    else
        Evolve GA
    end if
}
begin
    for j_i ∈ V do
        Add gene_alloc = es ∈ N to prototype
    end for
    for j_i ∈ V do
        Add gene_weight = [0, n − 1] to prototype
    end for
    Initialise Pop_1 = {genome_0, genome_1, · · ·, genome_n}; from prototype
    Evolve GA
end
```

Schedules are constructed from the genomes in the population using the genes for job allocation $map : \mathcal{V} \mapsto \mathcal{P}$ and the priority order. The objective score of each genome is the makespan of the schedule, which is used to compute the fitness value used by the GA for selection. The objective of the genetic algorithm is then to minimise this makespan, and the fittest genome at termination is chosen as the optimal solution. For a genome $genome_k \in Pop_i$, its fitness is determined by Equation (4), where i indicates the generation number.

$$Fit_{genome} = Makespan_{max} - Makespan_{genome}. \tag{4}$$

where $Makespan_{max}$ is the largest makespan of Pop_i, and $Makespan_{genome}$ is the makespan of $genome_k$ obtained by transforming $genome_k$ to a schedule.

Crossover and *Mutation* are operators employed by the GA to evolve any Pop_i. These operators create new genomes from existing genomes. In the case of crossover, two parents

are selected from Pop_i and offspring created for Pop_k by selecting a point in the parents where genes are swapped, creating the offspring. When selecting a parent from Pop_i, genomes with higher Fit_{genome} have a higher probability of selection. A probability of crossover is set where two offspring are created from two parents. The probability of mutation is less than that for crossover. The mutation operator is employed to avoid convergence of the search space at local minima. Unlike crossover, in applying mutation, a genome is copied from Pop_i to Pop_k, and a random gene is changed in the copied genome.

4.2. Metascheduler with Reconvergence of Paths in MSG

The metascheduler is a tool to compute schedules used by time-triggered NoC-based systems for adaptation to different context events. In Figure 3, an example of a time-triggered schedule is shown where its makespan is taken as the hyper-period. The time-triggered schedule has a makespan of 110-time units for ten jobs on four cores. Jobs $J_0 - J_9$ are assigned to cores $ES_1 - ES_4$, and arrows between jobs are messages representing job dependencies. A job allocation to a core is a time slot reserved for the execution of the job. The metascheduler computes schedules for all possible sequences of context events $(e1, e2, e3)$ within a hyper-period at design time. The schedules for a time-time triggered system are established a priori and repeatedly deployed after each hyper-period in a time-triggered system during runtime.

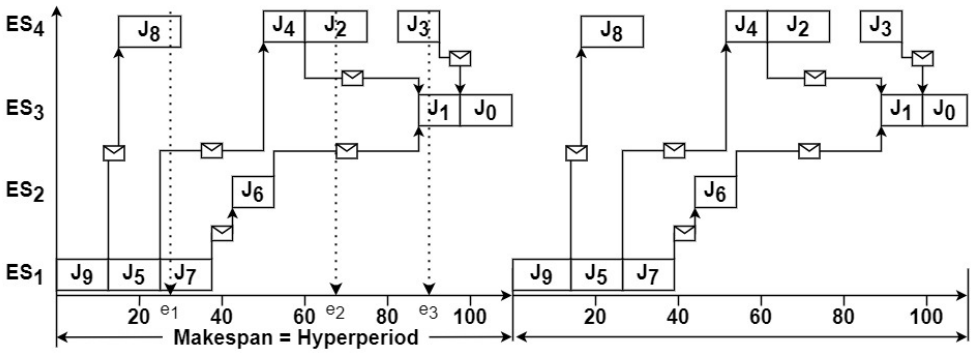

Figure 3. Example of a time-triggered schedule.

The proposed metascheduler performs adaptation at any particular instant in the schedule time-step, which is associated with context events $e \in \mathcal{C}$. For instance, at a given time step t, a slack event may exist for a particular job in which a new schedule is computed to adapt to the slack event while fixing events that have been considered in the past. These new schedules are computed by the metascheduler, which results in DAG as illustrated in Figure 4 to cover the occurrence of the given context events. At the deployment and operation of the time-triggered NoC-based system, the potential switching of schedules upon the occurrence of a context event is based on the MSG. Potential state traces in the MSG may differ in each hyper-period according to the occurrence of the context events. As a result, the number of schedules computed grows exponentially with increasing context events. It is thus a challenge for time-triggered NoC-based systems to utilise and store a reasonably sized number of schedules for adaptation at run time. Algorithm 2 is an algorithm to set a reconvergence horizon and execute the reconvergence of paths to solve the problem of the exponential growth of MSG.

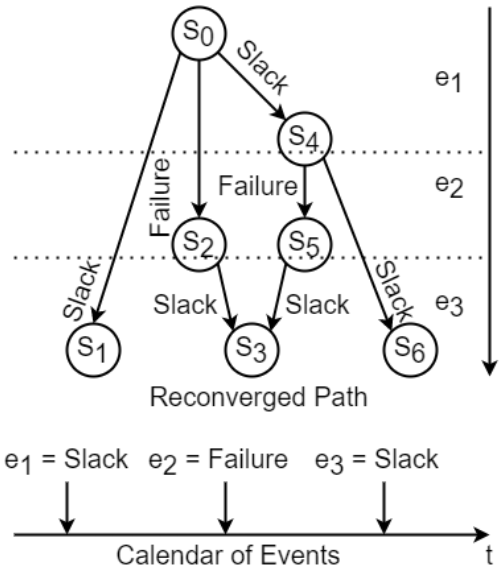

Figure 4. An example of a Multischedule Graph with reconverged path.

Algorithm 2: GA-based Metascheduler with reconvergence of paths in MSG.

Data: GA config = {popsz, ngen, pmut, pcross}, prototype
Input: $\mathcal{G}(\mathcal{V},\mathcal{E})$, $\mathcal{P}(\mathcal{N},\mathcal{L})$, $\mathcal{C}(\mathcal{S},\mathcal{F})$, GA config
Output: Multi-Schedule Graph

Function `metaschedule`$(\mathcal{G},\mathcal{P},Cal,node)${
  if $Cal = \{\}$ **then**
    return
  end if
  Take earliest $e_i \leftarrow Cal$;
  $Cal^{'}$ = *Remove* e_i from Cal;
  `metaschedule`$(\mathcal{G},\mathcal{P},Cal^{'},node)$;
  $(\mathcal{G}^{'},\mathcal{P}^{'}) = apply\_event(e_i,\mathcal{G},\mathcal{P})$;
  Set $horizon((e_i,t_i),t_k)$;
  Fix *decision variables* $\in$ *prototype*;
  S_i = Invoke *Genetic Algorithm*$(\mathcal{G}^{'},\mathcal{P}^{'})$;
  if $S_i \in MSG$ **then**
    Merge node and path
    return
  else
    Add S_i to *MSG* as *new_node*;
    Create edge from *new_node* to S_k;
    $Cal^{''} = update\_Cal(e_i,S_i,Cal^{'})$;
    `metaschedule`$(\mathcal{G}^{'},\mathcal{P}^{'},Cal^{''},new\_node)$;
  end if
}
begin
  S_0 = Invoke *Genetic Algorithm* $(\mathcal{G},\mathcal{P})$;
  Create *MSG*;
  Add S_0 to *MSG* as *root_node*;
  $Cal = create\_cal(S_0,\mathcal{C})$;
  `metaschedule`$(\mathcal{G},\mathcal{P},Cal,root\_node)$;
  return *Multi-Schedule Graph*
end

By reconvergence of path, the size of the multi-schedule state-space is reduced by merging paths that are precisely the same as any path already computed in the MSG. Furthermore, this approach eliminates redundant schedules that appear on different schedule tree paths by merging the transitions (see Line 13 in Algorithm 2). In setting the reconvergence horizon, new schedules computed by the metascheduler may only differ from the previous schedule within a limited time interval after the consideration of a context event. Thus, the metascheduler avoids state-space explosion by applying the reconvergence of paths in the MSG.

An MSG is generated using the metascheduler by invoking the GA repeatedly. The application of the GA to a time-triggered scheduling problem is implemented in Algorithm 1. The metascheduling algorithm repeatedly invokes Algorithm 1. In Algorithm 2, $\mathcal{G}$ and $\mathcal{P}$ are initially used to generate a base schedule S_0 which is added as the root node of the MSG. A calendar of events $Cal(e_i, t)$ is established from S_0 and $\mathcal{C}$, where all events at each time-step t is defined from S_0. Each event $e_i \in Cal$ is computed such that they are selected as discrete time steps in the metascheduling process.

The metascheduler implements a recursive function to traverse each time step. In each recursion of the metascheduler, the earliest context event is selected and removed from the calendar and used to modify $\mathcal{G}$ in the case of slack events $sl_{j,t}$ and $\mathcal{P}$ for resource failure $f_{es,t}$. Algorithm 1 is then invoked to solve the new scheduling problem. The resulting schedule is added to the MSG, and a path from the current node is defined by event e_i. Finally, the calendar Cal is again updated, and the *metaschedule* function is recursively invoked.

Metaschedules are generated recursively using $\mathcal{G}$, $\mathcal{P}$, S_i and Cal, where $\forall\ e_i \in Cal$, a reconvergence *horizon* is defined. The *horizon* is a time interval after event e_i, t in a schedule where the metascheduler is permitted to construct a new scheduling problem by modifying $\mathcal{G}$ or $\mathcal{P}$. The decision variables outside the *horizon* are fixed, enabling the reconvergence of paths in the MSG. In each case, the event e_i is removed from Cal, and the GA is invoked to solve the new scheduling problem. When a node is generated at each point in the metascheduler, it is checked against existing nodes in the MSG. In cases where duplicate schedules known as Deja Vu nodes exist in the MSG, the nodes are merged, and no further exploration is done. For each path in the MSG, non-mutually exclusive context events are explored, resulting in a sequence of events leading to multiple state traces. This approach leads to multiple branching of the MSG as shown in Figure 4.

5. Results and Discussions

5.1. Architectures and Applications

The GA-scheduler is implemented using the GAlib library [14] in C++. It uses the application and platform models to compute schedules, which are evaluated based on the makespan to optimise the job priorities and the job allocation. Experiments are conducted using the OMNI computing cluster of the University of Siegen for 10, 20, 40, 60, 80, and 100 jobs and reconvergence horizons set to 10, 20, 30, 40, and 50-time units. The OMNI computing cluster has 439 regular compute nodes, operated with Linux. Each regular node comprises two AMD EPYC 7452 CPU processors, with 32 cores per EPYC CPU and a 2.35–3.35 GHz CPU frequency. Each MSG is computed independently on a regular node, and results are compared.

The platform model is a 3x3 mesh NoC with homogeneous cores. Job graphs and platform models are generated using the SNAP library [15]. The Stanford Network Analysis Platform Library (SNAP Library) is a general-purpose, high-performance library that creates compact graph representations. This library is used to generate application models used to evaluate the metascheduler. Application models with various sizes were generated based on a random forest-fire model. The number of nodes, edges, indegrees, and outdegrees are specified in each case.

A total of six context models were generated with 0, 3, 5, 7, 9, and 10 context events (slack) and used to generate MSGs with reconvergence of paths. For each context model, a slack event is generated for a random job in the application model. The objective of further

experiments is to evaluate the impact of the reconvergence horizon and reconvergence of paths algorithm on the MSG state-space.

5.2. Selection of Genetic Algorithm Parameters

GA parameters were chosen based on a large population, motivated by high search space efficiency and a slow convergence rate. Very low probabilities of crossover and mutation normally prevent a random search by the GA. To find optimised parameters for the GA, *popsz*, *ngen*, *pmut*, *pcross* were initialised to 100, 1000, 0.01, and 0.4, respectively. A parameter tuning approach [16] was used to generate schedules repeatedly, and parameters resulting in schedules with the shortest average makespan was chosen to configure the GA, as presented in Table 1. The Application Model used to tune the GA parameters is a DAG of 100 jobs and 120 messages between the jobs. The platform model is a 3×3 mesh NoC with homogeneous cores. The AM has a maximum of 10 indegrees and outdegrees, with WCET of jobs ranging from 8 to 19 ticks. It is a challenge to optimise all tuning parameters; however, most researchers fix *pmut* to 0.01, and *pcross* to 0.4 [16]. Such parameter tuning with fixed mutation and crossover rates was applied to the metascheduler to minimise computational time while obtaining the least average makespan. The GA is then rigidly configured and the algorithm run using these parameters.

Table 1. Parameter tuning with fixed Mutation and crossover rates.

Generation	Population	Computational Time (s)	Makespan
1000	100	31	444
1000	100	31	466
1000	100	32	457
1000	100	31	459
1000	100	32	427
1000	500	160	550
1000	500	162	508
1000	500	160	505
1000	500	160	485
1000	500	161	476
3000	100	96	508
3000	100	94	484
3000	100	93	493
3000	100	95	427
3000	100	93	436
3000	500	474	458
3000	500	474	450
3000	500	473	463
3000	500	476	544
3000	500	483	445
5000	100	158	439
5000	100	160	443
5000	100	158	478
5000	100	159	453
5000	100	158	449
5000	200	326	419
5000	200	322	385
5000	200	321	387
5000	200	321	410
5000	200	325	427

From Table 1, a population of 200 and generation of 5000 results in schedules with the least makespan with low variations and adequate computational time, compared with other parameters. The proposed metascheduler computes multiple schedules for adaptation to context events. Child schedules differ from parent schedules only within the reconvergence horizon, so the schedules generated would benefit from parents with minimal makespan and variation. A population size of 100 with 1000 generations has a similar makespan as a population of 100 with 3000 generations, but computed in less time. The proposed metascheduler aims first to generate an optimal base schedule then minimise the computational time for child schedules. Decision variables outside the reconvergence horizon are fixed from parent to child, and that the child may only differ from the parent within the horizon. The GA is then initially configured with the number of generations set to 5000 and the population size set to 200 to obtain the base schedule. The number of generations for the GA is then set to 1000, and the population size to 100 to compute child schedules. Finally, the probability of mutation and crossover is fixed to 0.01 and 0.4, respectively, for the GA.

5.3. Evaluation of State Space Reduction

Figures 5–10 show plots of the MSG size against the number of context events for 100, 80, 60, 40, 20, and 10 jobs, respectively. All six plots show that the MSG size is reduced as the reconvergence of paths is applied. The reduction in MSG size is attained while maintaining a valid schedule. The makespan is maintained in each case by fixing the decision variables before and after the reconvergence horizon. The red plot in all Figures 5–10 is the same plot that shows the scenario when reconvergence of paths is not applied. It is seen that when ten context events are considered, a total of 1024 schedules are computed for the MSG. As the number of context events is increased from 0 to 10, MSG sizes increase exponentially. Existing metascheduling techniques [2,6] highlights this exponential growth of 2^n, where n is the number of context events considered. In [2], a reconvergence of paths in MSG is proposed but not evaluated. However, this work extends the earlier work in [2] by implementing and evaluating the reconvergence of paths algorithm for a metascheduler. However, the size of the MSG is reduced in all six cases (Figures 5–10) as reconvergence of paths is applied. This reduction can be seen in the plots attributed to different context events, which are all the plots other than the red plots in Figures 5–10.

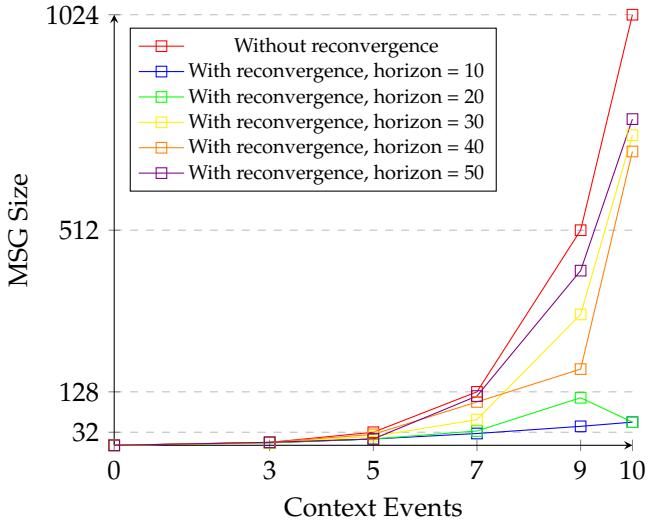

Figure 5. MSG with reconvergence for 100 jobs.

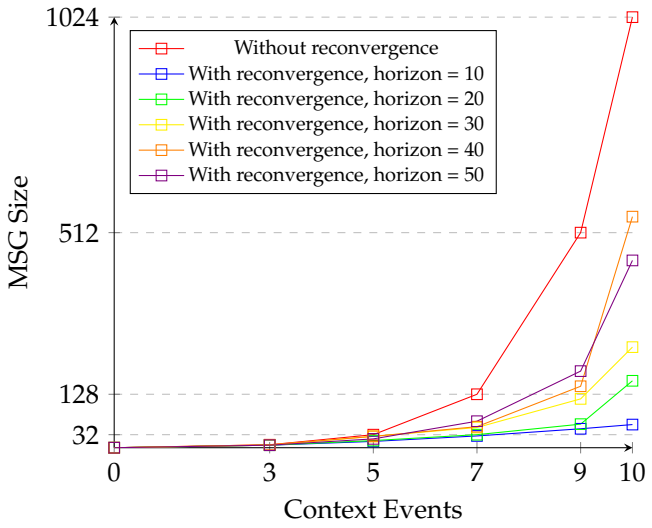

Figure 6. MSG with reconvergence for 80 jobs.

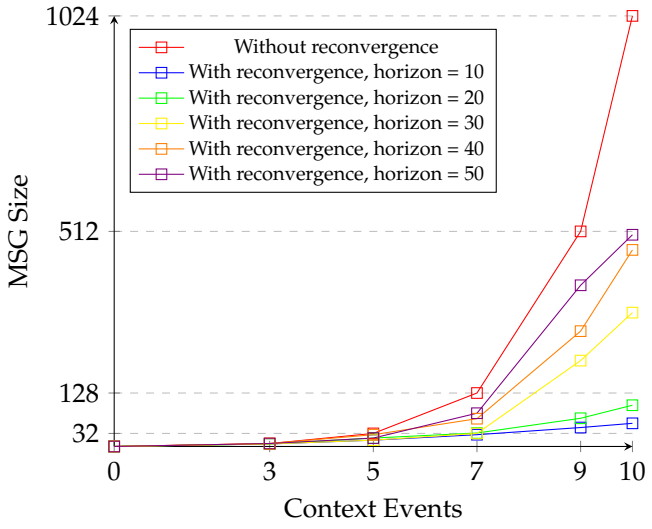

Figure 7. MSG with reconvergence for 60 jobs.

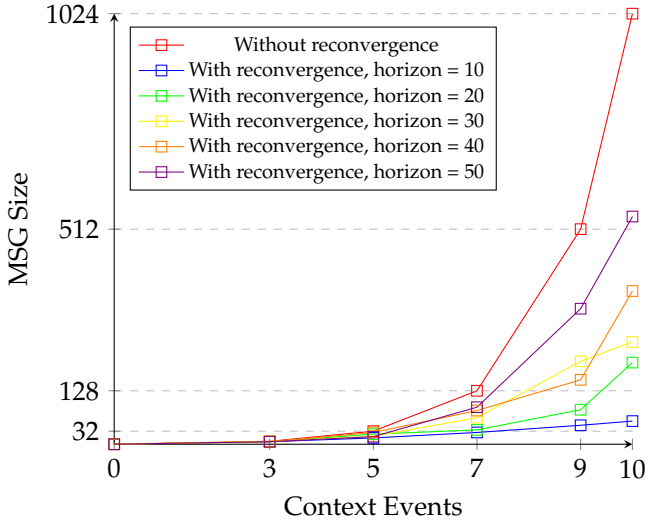

Figure 8. MSG with reconvergence for 40 jobs.

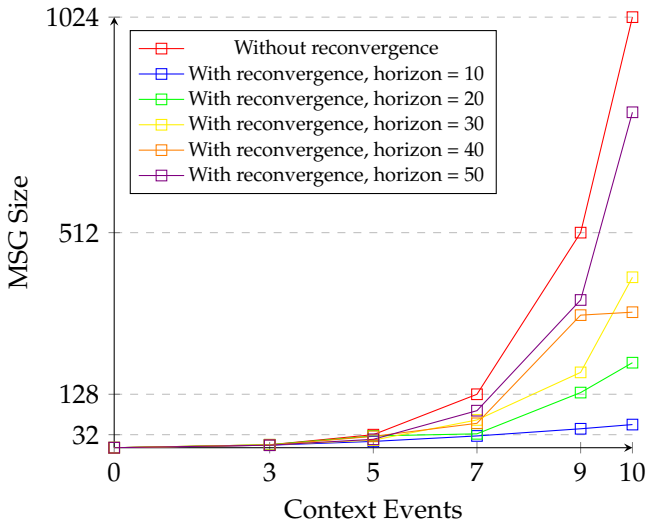

Figure 9. MSG with reconvergence for 20 jobs.

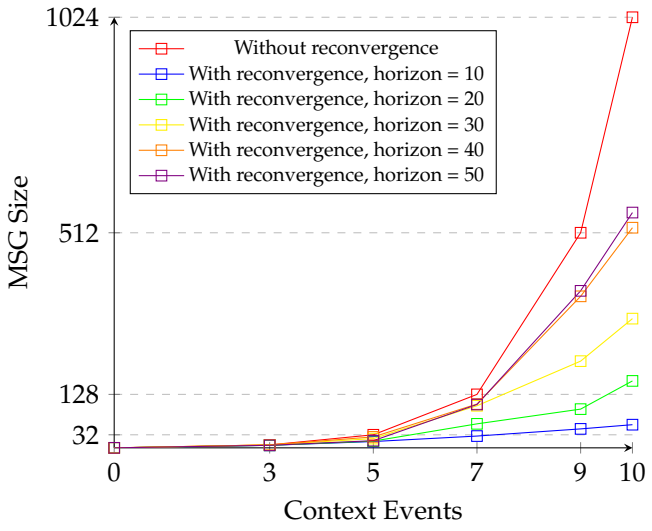

Figure 10. MSG with reconvergence for 10 jobs.

In Figures 7 and 10, the reduction of the MSG is proportional to the reduction of the horizon in a consistent way across all context models. However, this is not the case in Figures 5, 6, 8 and 9. In Figure 5, the plot for reconvergence horizon equal to 10 and 20 both present an output MSG size equal to 56 for 10 context events, respectively. The reconvergence of paths algorithm merges identical nodes when generated and terminates further exploration of the paths. This process results in cases where an increase in context events results in MSGs with smaller sizes if more "Deja Vu" nodes exist in the state-space, as observed by the green curve in Figure 5. In this case, for a horizon of 20, increasing context events from 9 to 10 results in an MSG of smaller size. Some reversed order in Figure 6 is also observed for 10 context events and horizon set to 40, which results in an MSG size of 550 being higher than when the horizon is set to 50 with MSG size equal to 446.

These results are based on the complex interactions that exists between adjusting a horizon that cuts across jobs with varying WCETs and changing scenarios that steer the paths in the MSG. Therefore, scenarios could exist where horizons with higher value results to lower MSG sizes than horizons with lower values, if in the computed paths more Deja Vu nodes exist. This scenarios can also be seen in Figure 8 for the pair ((horizon = 30, number of context event = 9, MSG size = 198), (horizon = 40, number of context event = 9, MSG size = 154)) and Figure 9 in pair ((horizon = 30, number of context event = 10, MSG size = 406), (horizon = 40, number of context event = 10, MSG size = 323)).

In all plots, Figures 5–10, it can be observed that for 10 events and a set horizon of 50 time units, there is at least a 22% reduction in MSG size (job size (100)= 24%, job size (80) = 56%, job size (60) = 50%, job size (40) = 47%, job size (20) = 22%, job size (10) = 45%).

5.4. State-Space Exploration Time

Algorithm 2 aims to first compute schedules with minimal makespan in the least computational time. In such computational time, Algorithm 2 generates an MSG for adaptation to a CM. Equation (5) gives an estimate of the computational time of a schedule computed for the MSG. The second is to fix past and future scheduling decisions outside the reconvergence horizon. Jobs in the parent node that start before or after the horizon have a fixed start time and allocation. Finally, Algorithm 2 manages the state space by merging identical nodes and paths (Deja Vu) as they are generated and added to the MSG. Deja Vu nodes are decided by considering future events in the current calendar of events at the instant such node is generated to those of nodes in the MSG at the same time instant.

$$Computational\ time\ per\ schedule = \frac{MSG\ computational\ time}{MSG\ size} \tag{5}$$

Table 2 shows the estimated computational time per schedule of Algorithm 2 referenced against the population-based incremental learning (PBIL) optimisation technique [11]. From the table, Algorithm 2 computes schedules in lower computational times than PBIL. Algorithm 2 minimises the average computational time per schedule by a minimum of 73%.

Table 2. Computational time (s) of existing and proposed metascheduling technique.

Setup	Scenario 1		Scenario 2	
	PBIL	**Proposed**	**PBIL**	**Proposed**
AM Size	20	20	35	40
NoC Architecture	3×3	3×3	3×3	3×3
Processors	9	4	9	4
Computational Time	30	8	110	22

Figure 11 reports the computational times for MSG generation of Algorithm 2 with design parameters described in Sections 5.2 and 5.3. MSG computational times are directly proportional to the average schedule generation time and the number of nodes in the MSG. However, as the number of scenarios increases, there is an exponential increase in the MSG size [2,6,8,11]. Existing techniques do not solve this exponential growth. Algorithm 2 computes MSGs within sufficient time, even for 100 jobs allocated to 4 homogeneous cores.

5.5. Sleep Time for Energy Saving

Sleep time is a time interval where no job is scheduled on any core. In such case, energy management techniques and services can be applied for energy savings. An example would be turning off an entire node. In each application of Algorithm 2, jobs after the reconvergence horizon are fixed and jobs within the horizon are optimised to finish earlier. This approach creates a time interval between the maximum finish time within the horizon and the earliest start time after the horizon as depicted in Figure 12.

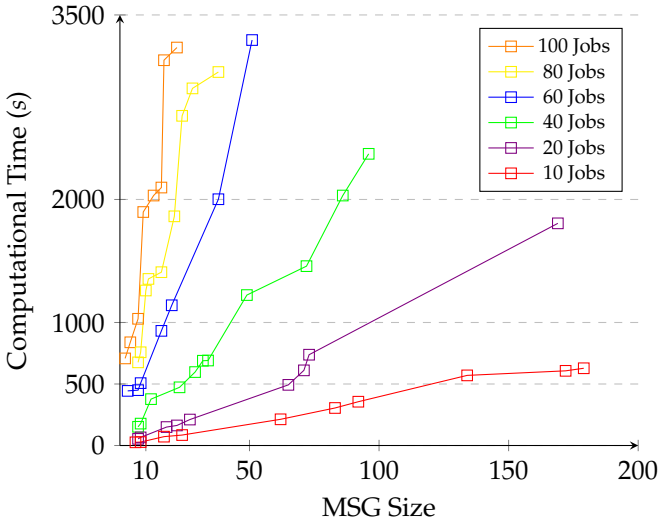

Figure 11. MSG computational time.

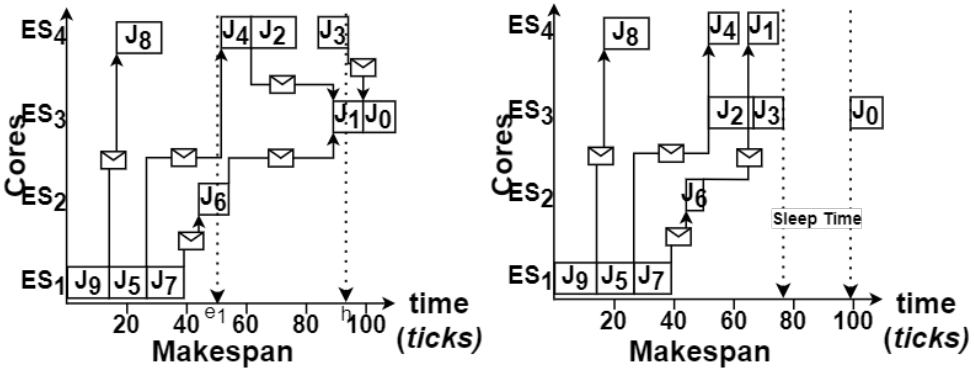

Figure 12. Algorithm 2 generation of sleep time.

In Figure 12, an example schedule is shown on the left. A slack event e_1 is generated by J_6 from finishing before its allocated slot. Algorithm 2 creates a recongvergence horizon between e_1 and h. J_9, J_5, J_8, J_7, J_6, and J_0 are outside the horizon and so are fixed. J_4, J_2, J_3, and J_1 are within the horizon and so are optimised to finish earlier, creating sleep time as shown on the right.

Figure 13 shows the average sleep time generation for different application models. In this figure, the average sleep time is shown as a percentage of the schedule makespan. Slack events are within 40–66% of the jobs WCET. AM, PM, and CM models are as described in Section 5.1. Within the horizon, jobs can be started earlier on an available core given the precedence constraints. It is observed that as the number of jobs is increased, the average sleep time is reduced. This is due to the higher number of jobs to be optimised within the horizon given its size. For a scenario of 10 jobs, the average sleep time is 23%. This corresponds to a 23% energy saving opportunity.

Compared with the conventional metascheduling technique of [2], such energy saving are comparable to power savings reported but Algorithm 2 achieves such savings using less number of schedules. Algorithm 2 computes these average sleep times by considering all combinations of slack events and creating paths in the MSG. The average sleep time is then the average sleep times for the leaf nodes in the MSG. Algorithm 2 is both scalable and achieves comparable results while generating smaller MSG sizes within acceptable computational time.

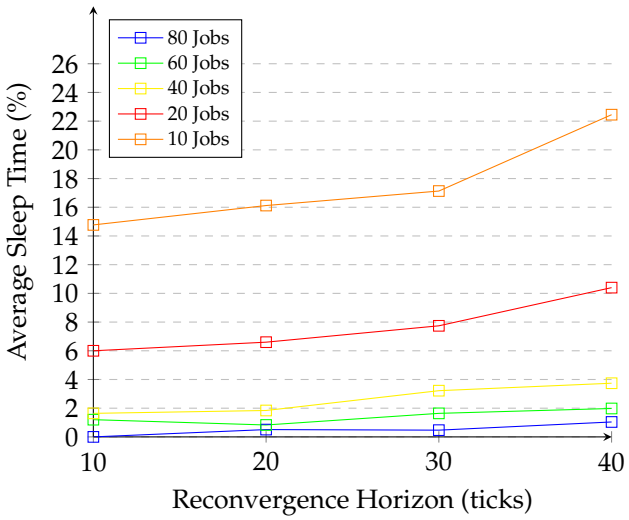

Figure 13. Average sleep time generation for 10 slack events.

6. Conclusions

In this paper, a metascheduling algorithm that implements a reconvergence of paths to manage the multi-schedule state-space problem of metascheduling, is presented. This algorithm repeatedly invokes a genetic algorithm to solve time-triggered NoC-based scheduling problems associated with different context events. The proposed algorithm establishes a horizon and merges Deja Vu nodes (schedules) to control the state-space explosion of the resulting multi-schedule graph for increasing context events. Results show that at least an MSG reduction of 22% is observed for 10 context events with the horizon set to 50, and as high as 56%. With the horizon set to 40, the results show a 23% average sleep time for 10 jobs with 10 slack events. The interaction between jobs and context events when generating schedules that can be merged on the occurrence of a Deja Vu node and how the choice of horizon size impacts the MSG size poses interesting future investigations.

Author Contributions: Conceptualization, P.M., D.O. and R.O.; methodology, P.M., D.O. and R.O.; software, P.M.; validation, P.M., D.O. and R.O.; formal analysis, P.M. and D.O.; investigation, P.M.; resources, D.O. and R.O.; data curation, P.M.; writing—original draft preparation, P.M.; writing—review and editing, P.M., D.O. and R.O.; visualization, P.M.; supervision, D.O. and R.O.; project administration, D.O. and R.O.; funding acquisition, R.O. All authors have read and agreed to the published version of the manuscript.

Funding: This research and APC was funded by ECSEL Joint Undertaking (JU) grant number 877056.

Conflicts of Interest: The authors declare no conflict of interest.The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

References

1. Paul, S.; Chatterjee, N.; Ghosal, P.; Diguet, J. Adaptive Task Allocation and Scheduling on NoC-based Multicore Platforms with Multitasking Processors. *ACM Trans. Embed. Comput. Syst.* **2021**, *20*, 1–26. [CrossRef]
2. Obermaisser, R.; Ahmadian, H.; Maleki, A.; Bebaway, Y.; Alina, L.; Sorkhpour, B. Adaptive Time-Triggered Multi-Core Architecture. *Designs* **2019**, *3*, 7. [CrossRef]
3. Ahmadian, H.; Obermaisser, R. Time-Triggered Extension Layer for On-Chip Network Interfaces in Mixed-Criticality Systems. In Proceedings of the Euromicro Conference on Digital System Design DSD, Madeira, Portugal, 26–28 August 2015; pp. 693–699.
4. Ahmadian, H.; Nekouei, F.; Obermaisser, R. Fault recovery and adaptation in time-triggered Networks-on-Chips for mixed-criticality systems. In Proceedings of the 12th International Symposium on Reconfigurable Communication-Centric Systems-on-Chip (ReCoSoC), Madrid, Spain, 12–14 July 2017; pp. 1–8.
5. Murshed, A. Scheduling Event-Triggered and Time-Triggered Applications with Optimal Reliability and Predictability on Networked Multi-Core Chips. Ph.D. Thesis, University of Siegen, Siegen, Germany, 2018.

6. Sorkhpour, B.; Murshed, A.; Obermaisser, R. Meta-scheduling techniques for energy-efficient robust and adaptive time-triggered systems. In Proceedings of the IEEE 4th International Conference on Knowledge-Based Engineering and Innovation (KBEI), Tehran, Iran, 22 December 2017; pp. 0143–0150.
7. Lee, C.; Kim, H.; Park, H.; Kim, S.; Oh, H.; Ha, S. A task remapping technique for reliable multi-core embedded systems. In Proceedings of the IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), Scottsdale, AZ, USA, 24–29 October 2010; pp. 307–316.
8. Lenz, A.; Pieper, T.; Obermaisser, R. Global Adaptation for Energy Efficiency in Multicore Architectures. In Proceedings of the 25th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), St. Petersburg, Russia, 6–8 March 2017; pp. 551–558.
9. Zhang, L.; Yang, J.; Xue, C.; Ma, Y.; Cao, S. A two-stage variation-aware task mapping scheme for fault-tolerant multi-core Network-on-Chips. In Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS), Baltimore, MD, USA, 28–31 May 2017; pp. 1–4.
10. Zou, Y.; Pasricha, S. HEFT: A hybrid system-level framework for enabling energy-efficient fault-tolerance in NoC based MPSoCs. In Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), New Delhi, India, 12–17 October 2014; pp. 1–10.
11. Bolanos, F.; Rivera, F.; Aedo, J.E.; Bagherzadeh, N. From UML specifications to mapping and scheduling of tasks into a NoC, with reliability considerations. *J. Syst. Archit.* **2013**, *59*, 429–440. [CrossRef]
12. Das, A.; Kumar, A. Fault-aware task re-mapping for throughput constrained multimedia applications on NoC-based MPSoCs. In Proceedings of the 23rd IEEE International Symposium on Rapid System Prototyping (RSP), Tampere, Finland, 11–12 October 2012; pp. 149–155.
13. Yang, X. *Nature-Inspired Optimization Algorithms*, 2nd ed.; Academic Press: London, UK, 2020.
14. Wall, M.; Galib, A. *A C++ Library of Genetic Algorithm Components*; Mechanical Engineering Department Massachusetts Institute of Technology: Boston, MA, USA, 1996.
15. Leskovec, J.; Sosič, R. SNAP: A General-Purpose Network Analysis and Graph-Mining Library. *ACM Trans. Intell. Syst. Technol.* **2016**, *8*, 1–20. [CrossRef] [PubMed]
16. Hassanat, A.; Almohammadi, K.; Alkafaween, E.; Abunawas, E.; Hammouri, A.; Prasath, V.B.S. Choosing Mutation and Crossover Ratios for Genetic Algorithms—A Review with a New Dynamic Approach. *Information* **2019**, *10*, 390. [CrossRef]