The online reconfiguration of a distributed on-board computer: The time and network behaviour of a dependable scheduling algorithm

Glen te Hofsté[a,*], Andreas Lund[a], Alexandra Coroiu[b], Marco Ottavi[c,d], Daniel Lüdtke[e]

[a]*Institute of Software Technology, German Aerospace Center (DLR), Münchener Straße 20, Weßling, 82234, Germany*
[b]*Institute for AI Safety and Security, German Aerospace Center (DLR), Wilhelm-Runge-Straße 10, Ulm, 89081, Germany*
[c]*Faculty of Electrical Engineering, Mathematics and Computer Science, University of Twente, Drienerlolaan 5, Enschede, 7522NB, The Netherlands*
[d]*University of Rome Tor Vergata, Via Cracovia 50, Rome, 00133, Italy*
[e]*Institute of Software Technology, German Aerospace Center (DLR), Lilienthalplatz 7, Braunschweig, 38108, Germany*

Abstract

On-board Computers (OBCs) are at the centre of space-faring systems. With the increasing demand for cost-effective computing power in space, using high-performance commercial-off-the-shelf (COTS) components for OBCs has gained significant traction. COTS components, however, do not provide the necessary fault tolerance mechanisms. The ScOSA (Scalable On-board computing for Space Avionics) architecture uses COTS components in a distributed system to provide more computing performance and dependability. The effects of node failures are mitigated by removing the failed node from the system through reconfiguration. A reconfiguration is performed by using a set of predetermined configurations, which hinders system scalability due to exponentially increasing memory consumption depending on the number of nodes.

This paper continues the work on the ScOSA online reconfiguration

*Corresponding author

Email addresses: glen.hofste@dlr.de (Glen te Hofsté), andreas.lund@dlr.de (Andreas Lund), alexandra.coroiu@dlr.de (Alexandra Coroiu), m.ottavi@utwente.nl (Marco Ottavi), daniel.luedtke@dlr.de (Daniel Lüdtke)

algorithm as a solution to this scalability problem. The online reconfiguration algorithm, which has been integrated into a scheduler, makes task scheduling decisions at run-time, eliminating the need for predetermined configurations. The six-phase scheduling mechanism uses the real-time state of the system and is a step towards higher dependability in distributed on-board computing. New test scenarios have been introduced to provide insight into the temporal and network behaviour of online reconfiguration. By evaluating in terms of *time*, *network traffic* and *memory usage*, it is shown that online reconfiguration is not only capable of dynamically generating configurations but also providing a solution to the scalability problem for systems with varying numbers of both nodes and tasks.

Keywords: Fault Tolerance, On-board Computers, Reconfiguration, Middleware, Distributed Systems, Dependability, Self-X

---

1. Introduction

Wildfire detection, autonomous missions on celestial bodies, and encrypted global communications are just a few examples of today's applications for space systems. They all have in common that they require a certain level of computing power to provide their service or fulfil their mission, and these performance requirements continue to increase. Typical on-board computer (OBC) architectures currently consist of a single, radiation-hardened, custom processing unit, such as the RAD750 [1] or the LEON5 processor [2]. These architectures are often unable to deliver the desired performance. For this reason, there is a trend towards using more Commercial-off-the-shelf (COTS) components in space systems. This saves cost, reduces time-to-fly, and simplifies application development. However, these components cannot withstand radiation in the same way as radiation-hardened parts [3, 4]. This leads to a trade-off between a high-performance but radiation-intolerant OBC and a radiation-tolerant but low-performance OBC.

1.1. Background

To meet the increased performance requirements, NASA has developed a hybrid architecture [5]. Using an Ethernet communication medium, the Dependable Multiprocessors integrate dependable processors with COTS processors. Other, more recent, solutions include the Xilinx Zynq Ultrascale+ System-on-Chip (SoC) as a high-performance processing unit, monitored by

rad-hard components [6, 7, 8]. Similar to these architectures, the German Aerospace Center (DLR) is working to overcome the aforementioned trade-off with the Scalable On-board computing for Space Avionics (ScOSA).

1.1.1. ScOSA - The Scalable On-board Computer Architecture for Space Avionics

ScOSA combines reliable, radiation-hardened components, with COTS components to form a distributed OBC. This creates an architecture that brings both worlds together. The reliable computing nodes (RCNs) execute the critical subsystems and act as a fallback for the high-performance nodes (HPNs). All nodes are interconnected via SpaceWire or Ethernet. Using a middleware [9] that runs on all processors, the distributed complexity is abstracted for the application developer, facilitating the development process. Applications range from earth observation and signal processing to remote sensing and artificial intelligence. An important feature of the middleware is that when a node fails, the node's applications, or tasks, are automatically migrated to other available nodes. This makes the system reconfigurable and dependable. An example of a reconfiguration due to a failing node, with tasks migrated to other nodes, can be seen in Fig. 1.

The middleware is implemented using a layered approach. The lowest layer is called *SpaceWireIPC*, a protocol that enables reliable communication over SpaceWire and Ethernet. On top of this, there is the *Network Dispatcher*, which is the intermediate layer that organises the messages to and from other nodes, and forwards them to the corresponding applications or services, i.e., the next higher layer. This next layer contains the *System Management Services* [9], which implement the fault tolerance mechanisms and the applications, which are implemented using the *Distributed Tasking Framework*. In these layers, the nodes in the system are also given *roles*, which can either be the role of *coordinator*, *observer*, or *worker*. At any time, there is only one coordinator and several observer nodes. The observer nodes monitor the "health" of the coordinator.

1.1.2. Reconfiguration services

The *Monitoring Service* of the coordinator monitors the state of other nodes through periodic heartbeat messages. If a node stops responding to a heartbeat or other types of messages, the service notifies the Reconfiguration Manager about the detected node failure. The *Reconfiguration Service* and *Reconfiguration Manager Service* allow the middleware to respond to node

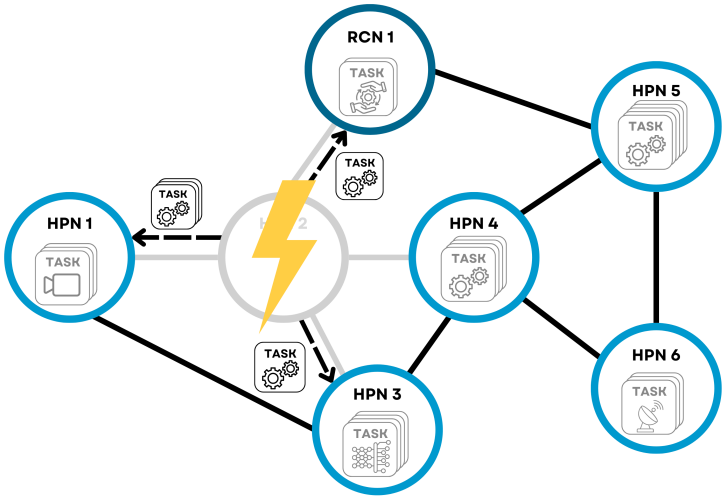

Figure 1: An example of ScOSA reconfiguration due to node failure

failures. To do this, upon an incoming *event*, the middleware looks up the current scenario, i.e., which nodes are still available and which are not in a tree structure. The configuration to be executed, i.e., the mapping of tasks to nodes, is stored there. The Reconfiguration Manager of the coordinator node then initiates the Reconfiguration Service on all available nodes. The reconfiguration is executed by first stopping all tasks on all nodes and then (re)starting them based on the new configuration. Finally, the *Reintegration Service* is used by starting or recovering nodes. It will request the Reconfiguration Manager to reintegrate it into the system. The node itself becomes the coordinator if no response is received within a timeout period.

Reconfiguration is implemented as an offline algorithm. This means that the responses to the possible failure scenarios are predetermined during the design phase and stored in configurations. A configuration holds information about node roles, task-to-node assignments, and network paths. All configurations are stored on all the nodes of the OBC as decision graphs. Problems

arise, however, when the system scales up. When there are more nodes in the system, the number of configurations grows exponentially [10], which consumes significant memory that is not available to the applications. Creating the configurations is furthermore an NP-complete problem, making the offline reconfiguration only feasible for small systems. In addition, the offline reconfiguration cannot react appropriately to unforeseen failure scenarios, instead, forcing the OBC to switch to a safe mode and wait for instructions from the ground.

To address these issues, we present the continued work on the online reconfiguration algorithm for the ScOSA OBC in this paper. The online algorithm uses information that is present during run-time, for example, *resources*, *suitability* of a task, or network *traffic*, to determine the next configuration. This enables the self-x properties of self-configuration and self-healing for the task-to-node mapping of the system.

1.2. Outline

This work is organised as follows: In Section 2, the related work is reviewed, followed by the design of the scheduling algorithm in Section 3. The design is implemented and incorporated in the ScOSA middleware, on which it has been evaluated in Section 4. The results are presented and discussed in Section 5, followed by a conclusion in Section 6.

2. Related work

Online scheduling algorithms for distributed systems are not a new phenomenon. There are several related works on desirable features for an online algorithm, such as fault tolerance, heterogeneity, parallelism, and multi-objective scheduling. Research on cloud scheduling [11, 12] proposes several path-searching algorithms with support for heterogeneity. Priority-based scheduling techniques [13, 14, 15, 16] show how ranking functions can be applied in combination with parallelism. Multi-objective scheduling [17, 18] can be used to optimise for a set of objectives instead of just one, increasing the balance in the system in terms of load and network usage. Other types of scheduling methods [19, 20, 21, 22] show that different approaches can also be feasible while being just as good, if not better, in some aspects. Because of the focus on dependability, the research on fault tolerant systems [23, 24] is particularly interesting, as it resembles the scheduling problem of ScOSA the

most. They provide important insights into the convergence of an algorithm in combination with heterogeneity and the ability to provide guarantees.

2.1. A new online scheduling algorithm

The literature indicates the diversity of heterogeneous distributed systems and their solutions. The field stretches from loosely coupled cloud systems to tightly coupled fault tolerant systems. Even though several fields cover features such as fault tolerance, parallelism, heterogeneity, and multi-objective scheduling, no solution exists that combines these features into one solution. The works on fault tolerance specifically do not include features such as parallelism or multi-objective scheduling. A novel solution is needed that integrates these features into one algorithm, while taking full advantage of the features unique to ScOSA. Multi-objective scheduling is outside the scope of this paper and will be part of future work. The other features of fault tolerance, heterogeneity, and parallelism, however, should be combined into a single algorithm. Therefore, this paper proposes a unique solution that combines the ScOSA middleware with the best of several scheduling techniques, resulting in the following contributions:

- A novel online scheduling algorithm design for reconfiguring dependable distributed on-board computers is described.
- The extendable algorithm provides a unique combination of fault tolerance mechanisms, extendability, caching, parallelism, self-x, and the usage of the real-time system state.
- The online scheduling algorithm is presented as a solution to the shortcomings of the offline algorithm based on an evaluation of its temporal, network, and memory behaviour.

2.2. Extension to the algorithm evaluation

This paper is an extension of the conference paper originally presented at the 37th GI/ITG International Conference on Architecture of Computing Systems (ARCS) held on May 14–16, 2024 in Potsdam, Germany [25]. The following contributions are introduced beyond the scope of the cited conference paper:

- The testing has been reworked and extended to provide more information on the time spent in each scheduling phase. Additionally, instead of

testing only for an increasing number of nodes, a new variation was introduced with an increasing number of tasks.

- A distribution of the total reconfiguration time was calculated based on the new test results. The calculated distribution is used to show the effects of caching.
- The network analysis has been extended with an analysis of the traffic that is generated during each phase. This is used to derive a formula for the network traffic that is generated during the scheduling cycles of the offline and online algorithms.

3. Design of the online algorithm

This work presents an online (scheduling) algorithm designed specifically for ScOSA. Although related work exists, there is a gap where fault tolerance, parallelism, heterogeneity, and multi-objective scheduling are combined into one solution. On the way towards implementing such an online algorithm, a solution is presented that focuses on fault tolerance and parallelism while providing extendability to implement multi-objective scheduling in future work. The algorithm is implemented in the ScOSA middleware as a part of its *System Management Services* to evaluate its scalability in a real non-deterministic system environment.

3.1. Definitions

The temporal behaviour of the algorithm is evaluated by separating the time it takes to assign the node roles, to decide where to schedule a task t to and the time it takes to apply these decisions by a single reconfiguration. The time it takes to assign the node roles is called the *role assignment time* and is defined in terms of clock c as:

$$roles_c \in \mathbb{R}_{\geq 0} \tag{1}$$

A task scheduling decision can be either that a task can be successfully scheduled or that no task mapping was found within a bounded period, resulting in a switch to safe mode, where it waits for instructions from the ground. The *decision time* in which it decides to schedule a single task t to a node, in terms of clock c is defined as:

$$decision_{tc} \in \mathbb{R}_{\geq 0} \tag{2}$$

Where:

$$c \in \mathbb{R}_{\geq 0} \tag{3}$$

When all scheduling decisions are made, they are applied by reconfiguring the system. The *reconfiguration time* in terms of clock c is defined as:

$$reconfiguration_c \in \mathbb{R}_{\geq 0} \tag{4}$$

The *total reconfiguration time* (trt) from the time a scheduling *event* arrives to the time the reconfiguration is finished is defined by the time it takes to assign the node roles, the total (sum of the) decision time $decision_{tc}$ to schedule a set of tasks t_{set} and the reconfiguration time $reconfiguration_c$:

$$trt = roles_c + \sum_{\substack{t_{set} \in T \\ t \in t_{set}}} decision_{tc} + reconfiguration_c \tag{5}$$

The maximum (trt) that the online scheduling algorithm shall not exceed is determined by the target hardware's limitations. A limit of 4 seconds was set by the ScOSA team, based on previous design objectives and internal testing. For the network traffic, the communication timeout was set to 200 ms as determined by internal testing.

3.2. Events

A reconfiguration is triggered in the Reconfiguration Manager on the arrival of four types of *events*. The *New Task Event* is generated when a new, previously unscheduled task needs to be scheduled. This task can, for example, be dynamically loaded during operation. The *Scheduling Failure Event* is generated when a task is unsuccessfully assigned to a node e.g., due to a severed communication. The task can be rescheduled to another node or, if this is not possible, *graceful degradation* or even a switch to safe mode can take place.

The *Node Recovery Event* is called when a node requests reintegration into the system. If a node has been in the system before, the system can recover to a state where the node was included or, as suggested for future work, the system should be optimised by re-balancing the tasks across its nodes. When a node failure is detected, the *Node Failure Event* is invoked. When a node fails, the running tasks are rescheduled to other nodes. The system is made aware of the failure so other nodes no longer attempt to engage with it.

3.3. Scheduling

The online algorithm's scheduling procedure starts when an event arrives. The algorithm's input is a data structure containing a set of tasks to be scheduled and a set of healthy nodes. The algorithm can be seen in Algorithm 1 as pseudo code and consists of six phases, starting at *Phase 1*.

Algorithm 1 Scheduling procedure

Require: N ▷ Set of healthy nodes n_{set} in the system
Require: T ▷ Set of tasks t_{set} to schedule

```
1:  Phase 1: Assign node roles
2:      if N does not contain a coordinator node then
3:          Assign new coordinator n in N
4:      if isCoordinator == True then
5:          if N contains a node n without a role then
6:              Assign a role to n
7:          Move to Phase 2
8:  Phase 2: Check cache
9:      if A cache entry exists for system N then
10:         Schedule tasks according to the cache entry
11:         Move to Phase 6
12:     else
13:         toSchedule ← T                    ▷ Set list of unscheduled tasks
14:         ScheduleTasks(N, toSchedule)      ▷ See function in Algorithm 2
15: Phase 6: Finish reconfiguration
16:     Nodes affected by scheduling stop execution
17:     Scheduling changes are applied on the affected nodes
18:     The nodes start executing
19:     Cache is updated and reconfiguration finishes
```

In **Phase 1** the *Coordinator*, *Observer 1*, *Observer 2* and worker roles are assigned to the healthy nodes in the system. If there is no coordinator in the system, one will be selected based on the lowest node id. If not already present, the (two) observer nodes are also assigned. All remaining nodes are then assigned the *Worker* role. If the coordinator or observer node roles change, an update is sent to all nodes in the system via a *partial reconfiguration*.

Algorithm 2 Scheduling Phase 3-4-5

```
    procedure ScheduleTasks(N, toSchedule)
2:      Phase 3: Prioritise tasks
            if length(toSchedule) > 0 then
4:              Calculate priority value of toSchedule tasks
                priorityTask ← highest priority task id
6:              Move to Phase 4
            else
8:              Move to Phase 6                    ▷ Break out of recursion
        Phase 4: Prioritise nodes
10:         Advertise highest priority task to all nodes in N
            The nodes return a calculated normalised priority value
12:         Node responses are appended to nodePriorities
            Sort nodePriorities in descending order
14:         Move to Phase 5
        Phase 5: Schedule task
16:         i ← 0                                  ▷ Node priority index
            Schedule the priorityTask to nodePriorities[i]
18:         if isSchedulingSuccessful == False then
                Attempt to schedule to lower priority nodes
20:             if Attempt successful then
                    Remove priorityTask from toSchedule
22:         else
                Remove priorityTask from toSchedule
24:         ScheduleTask(N, toSchedule)    ▷ Move to Phase 3 recursively
```

In **Phase 2**, the algorithm checks for a *cache entry*, which can provide a quick response if a scheduling situation has already occurred before. There is a limit to the number of cache entries that can be stored due to the limited memory and to improve the response time. A simplified version has been implemented that stores and maintains all the scheduling decisions until it is filled. This simplified version does allow the performance of a *cached decision* to be evaluated in terms of the time taken to handle an event.

With no cached decision to load, the transition is made to **Phase 3** by calling $ScheduleTasks()$, to which the set of healthy nodes N and set of tasks to be scheduled *toSchedule* is passed. As can be seen in Algorithm 2, the tasks to schedule are prioritised to determine in which order they are to be

scheduled. The prioritisation focuses on keeping as many tasks available in the system as possible. As the *Tasking Framework* does not currently provide mixed criticality or time-related parameters (such as arrival time, execution time, finish time, and task deadlines), the tasks are prioritised based on the number of successor tasks. If multiple tasks end up having the same priority, then tasks with a lower task id are currently prioritised.

In **Phase 4**, nodes are prioritised based on their ability to execute the highest priority task. The ability of a node to execute a task is calculated individually by each node in the system. The coordinator "advertises" the highest priority task to all healthy nodes over the network, which will individually and in parallel calculate a *normalised* priority value. Similar to the artificial hormone system in [22], the priority calculation is determined by factors such as:

- The availability of *resources* on a node (e.g., CPU utilisation, memory usage, temperature);
- The ability to execute a specific task, which may be different due to *heterogeneous* hardware in the system;
- The impact on the network by generating *increased traffic*;
- The *locality* to predecessor and successor tasks.

The priority calculation is currently based on the availability of resources and the ability of a node to execute a specific task, given the limitations in the *SpaceWireIPC* layer. These limitations will be addressed in future work.

Each node returns its calculation result to the coordinator. The coordinator creates a sorted priority list of the responses, limited by a timeout. If multiple nodes calculate the same priority, then nodes with a lower node id are prioritised.

In **Phase 5**, the highest priority task is scheduled to the highest priority node. This involves a single partial reconfiguration directed to the highest priority node with a request to execute this task. Using a *dynamic configuration*, the node stores the task change before applying it in Phase 6. When the coordinator receives the acknowledgement of the partial reconfiguration, the online algorithm removes the task from the set of tasks that need to be scheduled and recursively goes back to Phase 3 to schedule any remaining tasks. If there are no more tasks to schedule, the algorithm breaks out of the

recursion of $ScheduleTasks()$ and moves to Phase 6 to finalise the changes. Currently, if a partial reconfiguration cannot be applied due to a scheduling failure, the task should not be removed from the set. Instead, in future work, the Scheduling Failure event should be called, which will attempt to schedule the task to the second highest priority node.

Finally, in **Phase 6**, applying the scheduling changes will complete the reconfiguration. In this phase, the affected nodes will temporarily stop execution to reconfigure to the dynamic configuration, as created by the partial reconfigurations. Once applied, the nodes send the coordinator a "reconfiguration successful" message. The coordinator waits for all the successful reconfiguration messages from the nodes, after which it will finish the reconfiguration by sending a "reconfiguration finish" message to all nodes to notify them of the changes. At this point, the nodes start executing tasks again, resulting in the system being fully available again.

4. Evaluation

Two test setups are used to evaluate the online reconfiguration algorithm, a *time setup* on the target hardware and a simulated *network setup* on a Linux server.

4.1. Test Setup 1: time analysis

The first test setup is used to find the total reconfiguration time *trt*. The test programs are run on the target hardware, using three HPN nodes that are connected via Ethernet. Each HPN node runs Linux on a Xilinx Zync 7000 series system-on-a-chip with a dual-core ARM A9 processor with a total of 1GB of DDR3 memory and gigabit Ethernet.

The $reconfiguration_c$ parameter tracks how long it takes for the system to reconfigure and apply a new configuration. The decision time parameter $decision_{tc}$ keeps track of how long it takes the coordinator to decide and schedule a task to a node. Only the online algorithm has a decision time, as for the offline algorithm the decisions are predetermined offline using a decision graph. Due to the dependency on the network during the scheduling procedure, the network delay is also part of the overall decision-making time. Finally, $roles_c$, $decision_{tc}$, and $reconfiguration_c$ are used to calculate the total reconfiguration time *trt* (Eq. 5) in milliseconds.

Compared to the offline reconfiguration, the online reconfiguration depends heavily on the communication between nodes during phases 4 and 5 of the

algorithm. It is important to determine how the total reconfiguration time increases and how it scales for a larger set of tasks.

Four tests are performed on the target hardware to find $decision_{tc}$ and $reconfiguration_c$, using the following test programs:

- Test Program 1 (TP1): Offline reconfiguration (4 tasks)
- Test Program 2 (TP2): Online reconfiguration + cache (4 tasks)
- Test Program 3 (TP3): Online reconfiguration (4 tasks)
- Test Program 4 (TP4): Online reconfiguration + cache (20 tasks)

Two types of simple tasks are used in all tests: sender and receiver tasks. A sender task periodically sends a counter value, whereas receiver tasks receive the counter value and print it out. Besides incrementing the counter, the sender tasks do not perform any additional computations.

TP1 is used as the reference program to benchmark and compare the online reconfiguration programs with and without cached decisions. It is configured to use four tasks, consisting of three receiver tasks receiving the output of one sender task. The sender task sends the counter value every 500 ms to the receiver tasks. TP2 and TP3 run the same tasks as TP1 to make a direct comparison. The influence of the cache can then be evaluated between TP2 and TP3. The cache is disabled in TP3. The final program TP4 is used to evaluate the algorithm when there are more tasks in the mission configuration. It is configured to use five sender tasks with a send interval of 500 ms, each with a subset of three receiver tasks that receive and print the output of the sender task.

To automatically run each test program on the nodes, a test script is used, which starts and stops nodes at random time intervals, according to a uniform distribution between 20 and 60 seconds. The test script emulates the behaviour of nodes failing and reintegrating into the system.

4.2. Test Setup 2: network analysis

The second test setup is used to analyse the network traffic in terms of the number of bytes generated by a scheduling procedure. The network analysis tests are performed on a "worst-case" node failure with a failing coordinator node, as the coordinator selection combined with the task scheduling results in the largest amount of traffic. The tests are conducted on a server with

an x86_64 desktop processor, utilising internal loop-back routing for network traffic, allowing for the operation of more than three virtual nodes. As the dependency on memory for offline reconfiguration is effectively changed to a dependency on the network, it is important to determine how the network traffic scales. With the ability to increase the number of virtual nodes, the network traffic of online reconfiguration could be tested for a system with a higher number of nodes. Systems consisting of 3, 4, 5, 6, 7, 8, 9, 16, 32, 48, 64, 80 and 96 nodes are tested 25 times each. To analyse the different test cases, the accumulated reconfiguration traffic in bytes is filtered to only include packets from scheduling procedures. Then, each scheduling procedure, from the failure event to the finish reconfiguration message, is extracted from the filtered traffic to find the exact packets that are associated with a full reconfiguration. These packets are subsequently used to derive a formula that describes the network traffic.

Three different tests are performed on the server to find the worst-case network traffic, using the following test programs:

- Test Program 1 (TP5): Offline reconfiguration (4 tasks)
- Test Program 2 (TP6): Online reconfiguration + caching (4 tasks)
- Test Program 3 (TP7): Online reconfiguration (4 tasks)

These programs are essentially the same as in Test Setup 1, with the exception that they are compiled for x86_64.

5. Results and discussion

5.1. Timing analysis

The timing analysis is performed on the results from Test Setup 1 (see Section 4.1). It focuses specifically on the time it takes to execute the different phases of the algorithm. The timing analysis is performed with a 1 millisecond resolution. Throughout all tests, the time required to assign the node roles (Phase 1) was found to be no more than 1 millisecond. This value is used as a worst-case role assignment time $roles_c$ (Eq. 1) for finding the total reconfiguration time trt (Eq. 5).

5.1.1. Decision time

The time it takes to decide which node to schedule a (single) task to, is defined by $decision_{tc}$ (Eq. 2), which corresponds to phases 3, 4 and 5 of the algorithm. In case a cached decision exists, $decision_{tc}$ corresponds to Phase 2, determined by the time it takes to load and schedule a decision from cache. The results were then split between cached decisions and decisions based on node and task prioritisation. If only the node role was changed during Phase 1 of the scheduling procedure, zero tasks would be scheduled, resulting in a direct transition to Phase 6, and subsequent $decision_{tc}$ of zero for Phase 2 or Phase 3-4-5.

A distinction was made based on whether there were one or multiple healthy nodes in the system. If there is only one healthy node, the one running the scheduling procedure, then naturally all tasks are scheduled to that node, resulting in a reduced $decision_{tc}$.

The resulting $decision_{tc}$ of Phases 3-4-5 (without cached decisions) can be seen in Table 1. The data shows the difference in mean decision times between scheduling one or multiple nodes. The distributions of decision times from Table 1 can be seen in Fig. 2. Note that the results do not include experiments where a communication timeout occurred, which is set to 200 ms.

Multiple Nodes	Count	Mean	Std Dev	Min	Max
False	2815	34.50	4.57	33	199
True	9087	101.28	13.96	48	192

Table 1: Decision time (ms) per task, not cached

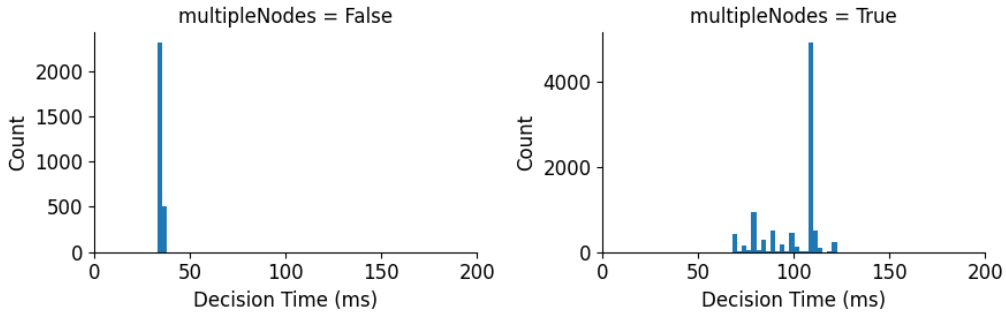

Figure 2: Decision time per task, not cached

When Phase 3-4-5 is executed with multiple nodes in the system, the mean stays around 101 ms, regardless of the number of nodes in the system. This is due to the parallelism of the algorithm, where all nodes calculate their own priority value in Phase 4. When there is only one node in the system, the coordinator schedules all tasks to itself if sufficient resources are available, resulting in a mean decision time of around 34 ms with a low standard deviation. The separation between multiple nodes and a single node in the system is therefore important to consider. It must be noted that when the system is in flight, a situation with only a single healthy node left is unlikely to occur when using larger system sizes than three nodes.

When decisions are loaded from cache in Phase 2, the $decision_{tc}$ for one and multiple nodes can be seen in Table 2 and Fig. 3. Notice that for both cases the mean is around 35 ms, with a very low standard deviation. Thus, when a cached decision can be loaded, the total decision time can be reduced.

Multiple Nodes	Count	Mean	Std Dev	Min	Max
False	32	35.16	1.61	33	38
True	4762	35.56	1.32	33	42

Table 2: Decision time (ms) per task, cached

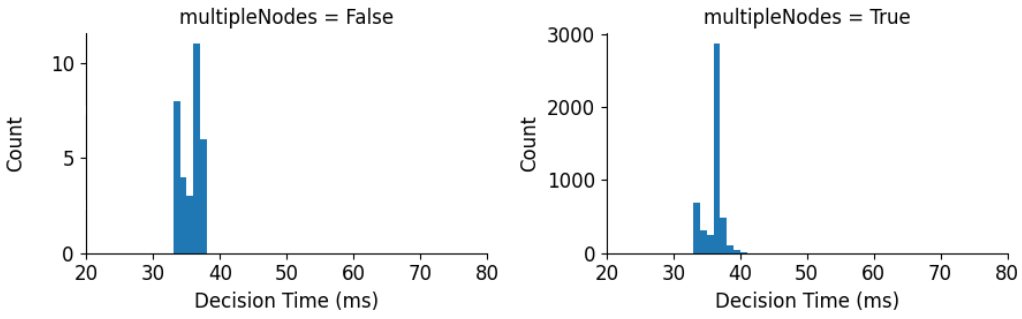

Figure 3: Decision time per task, cached

5.1.2. Total decision time

Knowing the decision times per task, the total decision time can be determined. Using TP4, the decision times to schedule a set of tasks without cached decisions are captured and can be seen in Fig. 4. The trend line

indicates that the total decision times increase linearly with the number of tasks that are scheduled, both for one and for multiple nodes.

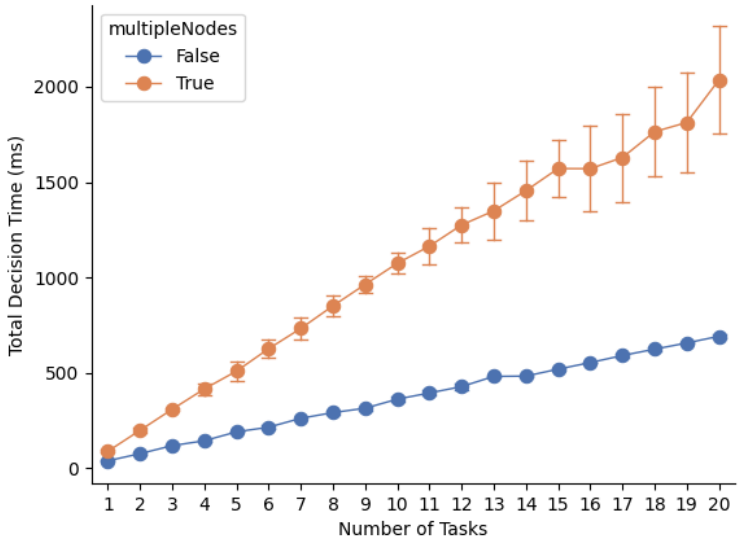

Figure 4: Total decision time (Mean and 1 Std Dev), 1-20 tasks, not cached

The effect of caching on the total decision time in the case of 20 tasks and multiple nodes can be seen in Table 3. Not only is the mean more than halved, but the standard deviation of cached decisions is also smaller, making them more predictable.

Cached Decision	Count	Mean	Std Dev	Min	Max
False	192	2035.08	280.80	701	2382
True	81	724.15	24.88	679	757

Table 3: Total decision time (ms), 20 tasks, with multiple nodes

5.1.3. Reconfiguration time

The *reconfiguration time* (Eq. 4) is the last missing value to find *trt* (Eq. 5), and is captured during Phase 6 of the algorithm. The reconfiguration time of TP1, TP2, and TP3 can be seen in Table 4. The results of test programs 2 and 3 are combined, as caching only impacts $decision_{tc}$, and not $reconfiguration_c$.

The difference between offline and online distributions can be seen in Fig. 5, and explains the higher standard deviation for online reconfigurations.

Statistic	Count	Mean	Std Dev	Min	Max
Offline	779	80.38	16.61	34	119
Online	2327	139.04	90.33	11	333

Table 4: Reconfiguration time (ms)

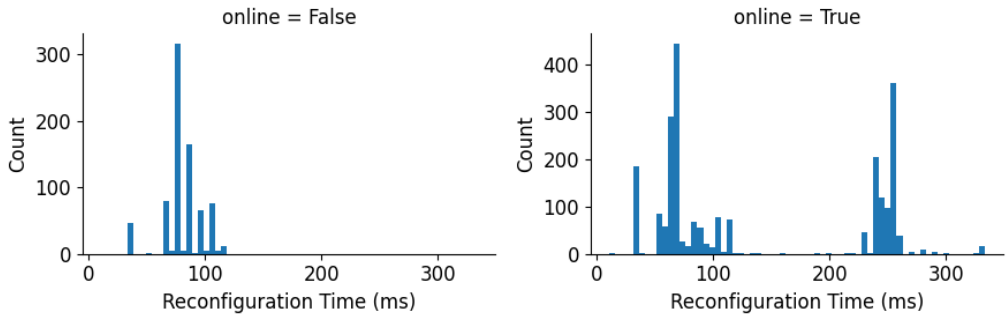

Figure 5: Reconfiguration time

The distribution of online reconfiguration times presents a bimodal distribution. The two modes are caused by *network delays* and the handling of *Node Failure* and *Node Recovery* events. When a node reintegrates into the system, it must be initialised. The initial reconfiguration of this node after a boot-up is time-consuming. Since the coordinator node has to wait for the reintegrating node to finish the time-consuming initial reconfiguration, there is an increased reconfiguration time, which results in the second mode. The first mode is caused by a reconfiguration after a node failure where no initialisation is required. This, therefore, results in a lower reconfiguration time, which is similar to a full offline reconfiguration.

5.2. Comparing to the benchmark

To be able to compare to the offline benchmark of TP1, the identical tasks are scheduled through an online scheduling procedure. Similar to what is displayed in Fig. 4, $decision_{tc}$ is examined per number of scheduled tasks in a scatter-plot in Fig. 6. It shows the decision times without cached decisions.

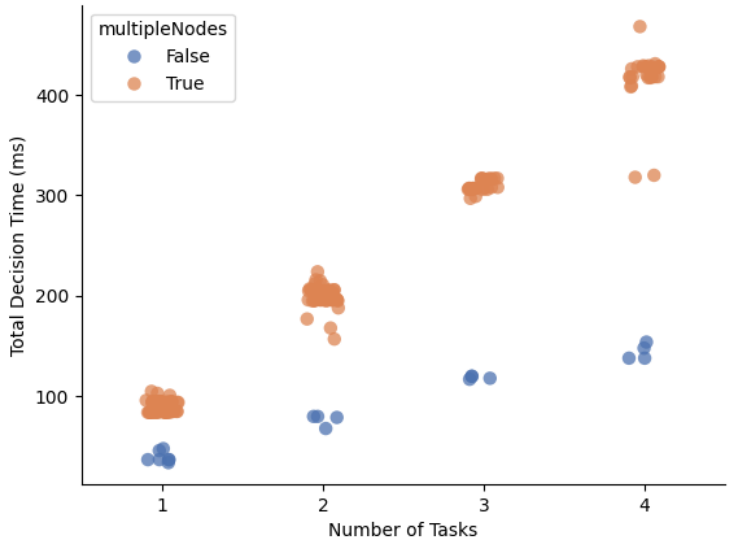

Figure 6: Total decision time, 1-4 tasks, not cached

A difference between offline and online reconfigurations is that an offline reconfiguration always re-loads and re-starts all tasks, contrary to the online reconfiguration where only the updated tasks are affected. Therefore, the total decision time can be any of the number of tasks of Fig. 6. The $decision_{tc}$ when explicitly considering the case where four tasks are scheduled can be seen in Table 5.

Multiple Nodes	Count	Mean	Std Dev	Min	Max
False	4	144.50	7.89	138	154
True	26	415.92	30.57	318	468

Table 5: Total decision time (ms), four tasks, not cached

Fig. 6 and Table 5 only show the $decision_{tc}$ that was not a result of a cache load in Phase 2. This is because there were insufficient occurrences of cached decisions where there is only one node in the system for statistical analysis. For cached decisions with multiple nodes in the system the mean $decision_{tc}$ is 151.63 ms for four tasks. This is an improvement over the mean value of 415.92 ms when executing Phase 3-4-5, indicating the effectiveness of

caching, even for a small set of tasks. If more cache loads can be achieved during scheduling, it is expected that the overall mean $decision_{tc}$ will decrease, resulting in a better *trt*.

5.2.1. Total reconfiguration time

With all the variables known to calculate *trt* (Eq. 5), the distribution of the total reconfiguration time was computed as the sum of two random variables: one drawn from the distribution of decision times, the other from the distribution of the reconfiguration times. This operation was performed for four tasks as seen in Fig. 7 and twenty tasks as seen in Fig. 8.

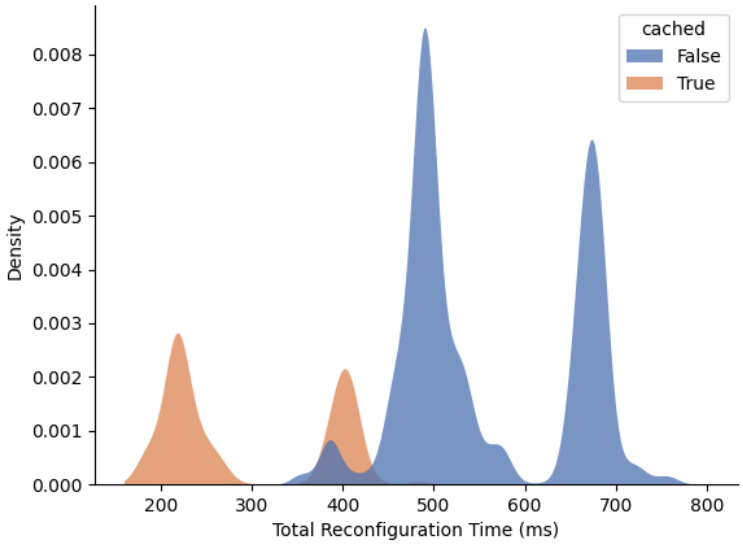

Figure 7: Computed total reconfiguration time, four tasks (kernel density estimate with bandwidth = 1)

Most importantly, the limit *trt* (4 sec) of the design objectives (as stated in Section 3.1) would never be exceeded (Max trt four tasks = 802, Max trt twenty tasks = 2716). Both figures also show the effect caching would have on the *trt*. In situations where cached decisions are loaded (indicated by cached = True) the *trt* would be lower than for the non-cached decisions. This difference increases even more when the number of tasks that need to be scheduled increases, as can be seen when comparing the *trt* distributions of four and twenty tasks. To extend on this, it is thus clear that during

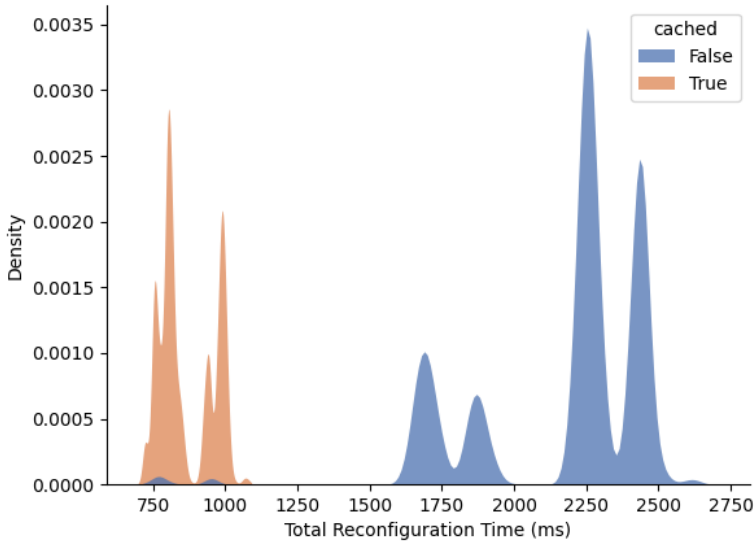

Figure 8: Computed total reconfiguration time, twenty tasks (kernel density estimate with bandwidth = 1)

boot-up the *trt* is expected to be high as well, since the full set of tasks needs to be scheduled, without the presence of any cached decisions. A changing system topology due to nodes failing or reintegrating can be handled quickly as usually not the full set of tasks needs to be scheduled.

One way to reduce the time during boot-up is to force a cached decision by implementing cache preloading. Common system states can be predetermined, similar to the offline configurations, and preloaded into the cache. The system can then quickly load an optimised configuration for a common situation, such as the nominal state when all nodes are healthy. This can make cache loads more frequent, resulting in a lower overall *trt*.

5.3. Network analysis

The test outputs from Test Setup 2 (see Section 4.2) are used for the network analysis. TP5 and TP6 are used to compare the offline and online reconfigurations one-to-one. The impact of caching on online reconfiguration is evaluated between TP6 and TP7. Since the offline and online scheduling procedures are always the same, the captured network traffic can be used to

find a formula that describes the traffic as a function of the number of nodes and tasks.

The total network traffic that is generated after a node failure by the offline reconfiguration is a function of the number of nodes that are in the system. After analysing the captured network traffic, it was found that the total network traffic can be calculated using Eq. 6, where n_{set} is the set of healthy nodes in the system. A Node Failure Event message (73 bytes) occurs only once and is represented as a constant. Traffic from the Reconfiguration Request, Reconfiguration Finish, and Reconfiguration Successful messages (217 bytes total) of the offline algorithm, is dependent on the number of healthy nodes (n_{set}) in the system. One is subtracted from n_{set}, since the coordinator is also counted in the set of healthy nodes. However, packets from the coordinator to the coordinator do not affect the network traffic, as they are handled internally by the middleware.

$$traffic_{offline} = (n_{set} - 1) * 217 + 73 \tag{6}$$

The total network traffic generated by online reconfiguration after a node failure is a function of the number of nodes in the system and the set of tasks that need to be scheduled. We found that the total network traffic can be calculated using Eq. 7, where n_{set} is the set of healthy nodes in the system, and t_{set} is the set of tasks to be scheduled. Node Failure Event and Reconfiguration Finish messages (121 bytes total) occur only once and are represented as a constant. Traffic from Phase 1 and Phase 6 (272 bytes total) depends on the number of healthy nodes (n_{set}) in the system, where again the coordinator is subtracted from n_{set}. Traffic from Phase 4 (233 bytes) depends on the set of healthy nodes and the set of tasks t_{set}, as for every task that needs to be scheduled, all nodes are contacted to advertise the task, to which they will all respond. Finally, traffic from Phase 5 Partial Reconfiguration Requests (100 bytes) depends on the set of tasks t_{set}, and are always sent to only one node.

$$traffic_{online} = (n_{set} - 1) * (272 + (t_{set} * 233)) + t_{set} * 100 + 121 \tag{7}$$

These two equations can be used to find the expected total network traffic for systems with a varying number of nodes and tasks. Table 6 shows the expected total offline and online network traffic. Note how the offline reconfiguration traffic is not dependent on the number of tasks involved in

a reconfiguration, since here, only the switch to the next predetermined configuration is handled over the network by the coordinator. For the online reconfiguration, the total traffic increases with the number of nodes and tasks. For large systems, where a large set of tasks needs to be scheduled, the amount of traffic will approach the limits of the network. However, as systems of 96 nodes are considered unrealistic, more realistic systems of e.g., 16 nodes were found to always be within the limits of the network.

Healthy Nodes	Offline reconfig. Tasks	Online reconfig. Tasks		
	Any	1	4	20
2	290	726	1725	7053
16	3328	7796	18581	76101
96	20688	48196	114901	470661

Table 6: Total traffic (bytes), offline vs. online (1, 4, 20 tasks) reconfiguration

The total network traffic has also been analysed in a virtual environment, to determine how non-determinism affects the behaviour of the algorithm, by e.g., introducing other miscellaneous network traffic or by simulating node failures while scheduling. It is furthermore important to see how the network traffic *scales* when the number of nodes in the network increases in a non-deterministic environment. Fig. 9 shows the captured network traffic of TP6 and TP7 for an increasing number of nodes with caching enabled and disabled.

For both TP6 and TP7, the network traffic increases linearly, with the caching-enabled version requiring slightly less traffic, since Phase 3-4-5 can be skipped. The largest system with 96 nodes had a worst-case network traffic of 139626 bytes. This is higher than the calculated traffic in Table 6, and occurs in failure situations where one or more network packet resends are necessary. This is, however, still within the limits of the network.

5.4. Memory analysis

When scaling up the number of nodes in the system using the network setup, the memory usage of the online algorithm was analysed using TP6 and TP7. Offline reconfiguration e.g., in TP6, has been shown to have memory consumption that scales exponentially [10]. Online reconfiguration should solve this problem in particular. As nodes using the online reconfiguration

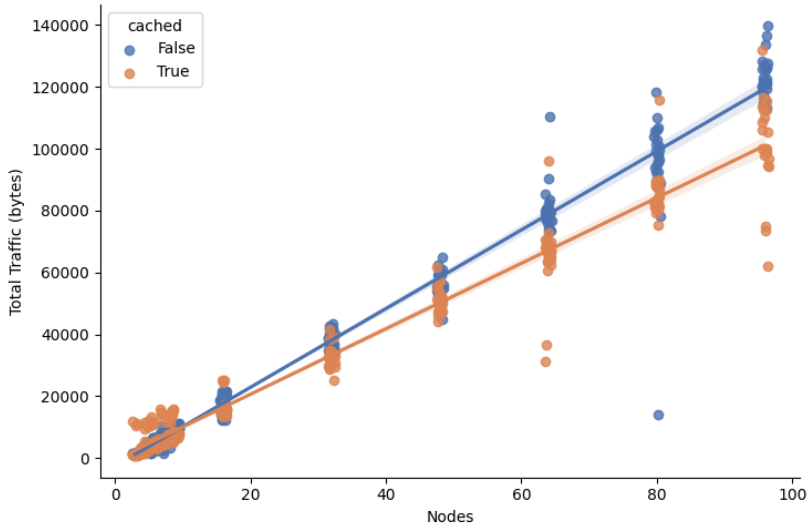

Figure 9: Total traffic for online reconfiguration (regression line CI=95%)

have to keep track of where tasks are running in the system, they are expected to consume more memory when the system size increases. After running a memory profiler, it was discovered that the stack usage increases linearly with the number of nodes in the system, regardless of the node's role. In fact, when the number of nodes doubles, the *stack* usage doubles as well, with the *heap* usage remaining stable for all system sizes. When the number of nodes increased from 3 to 96 (a 32 times increase), there is only a 28-29 times increase in stack usage. This shows the online reconfiguration's ability to solve the scaling problem, which is in contrast to the offline algorithm's exponentially increasing memory usage.

The results demonstrate that online reconfiguration is an effective solution to the scalability problem of the offline reconfiguration. Although runtime and network traffic increase, these do not increase exponentially and remain within the system's limits. The online algorithm provides a solution to the inability of offline reconfiguration to support systems with many nodes and to adapt to runtime behaviour. Online reconfiguration, therefore, allows larger systems to utilise a distributed avionics middleware such as ScOSA.

6. Conclusion

Dependability in spacecraft on-board computers remains a significant challenge. In this paper, we present the continuation of the work on a distributed system with the ability to self-configure based on the real-time state of the system. The novel online reconfiguration mechanism overcomes the scalability issues of offline reconfiguration by eliminating the need for predetermined configurations. By implementing online reconfiguration in the ScOSA middleware, the dynamic creation of configurations was evaluated in terms of time, network traffic, and memory usage. The online reconfiguration mechanism can schedule tasks to nodes at the cost of increasing the total reconfiguration time and network traffic, while remaining within the limits of the system. However, as the number of nodes and tasks in the system scales, the total reconfiguration time, network traffic, and memory usage increase linearly, in contrast to an exponential increase in memory usage for the offline reconfiguration mechanism.

There is thus a trade-off to be made depending on which aspects are more important. For smaller systems, where all configurations can still be predetermined offline and fit into memory, the increase in time and network traffic of the online reconfiguration may lead to an unacceptable increase in time and network usage. However, offline reconfiguration provides no resilience to contingencies and has no self-x properties.

If the best of both mechanisms can be brought together in a combined solution, a middle ground could be reached that would provide the greatest net improvement. It became clear that for the online reconfiguration, the total reconfiguration time increases with the number of tasks to be scheduled. Therefore, when a large set of tasks needs to be scheduled, online reconfiguration reduces the time in which all tasks are operational. Caching combined with cache preloading is proposed as a kind of hybrid solution that combines offline features with online reconfiguration to mitigate the shortcomings of both approaches.

Investigating this trade-off will be part of future work, while the development of the online reconfiguration algorithm continues. From a maintainability perspective, changes to the scheduling phases can be easily made to further extend and optimise the scheduling procedure, paving the road for multi-objective scheduling in the future. Enhanced with this online reconfiguration, ScOSA can be developed into a dynamic but dependable OBC architecture. This opens up new possibilities: from a power-aware system that adapts to

the available power to spacecraft-spanning systems e.g., constellations that dynamically distribute tasks among themselves.

As a first step, ScOSA will be demonstrated with some typical space applications as part of a CubeSat mission in 2026 [26]. This will initially include offline reconfiguration, but will be followed by an update to demonstrate online reconfiguration under operational conditions.

References

[1] BAE systems: Rad750 radiation-hardened powerpc microprocessor, accessed: 03-02-2023.
URL https://www.baesystems.com/en-media/uploadFile/20210404045936/1434555668211.pdf

[2] Frontgrade Gaisler: Leon5 processor, accessed: 03-02-2023.
URL https://www.gaisler.com/index.php/products/processors/leon5

[3] A. N. Nikicio, W.-T. Loke, H. Kamdar, C.-H. Goh, Radiation analysis and mitigation framework for leo small satellites, in: 2017 IEEE International Conference on Communication, Networks and Satellite (Comnetsat), 2017, pp. 59–66. doi:10.1109/COMNETSAT.2017.8263574.

[4] C. Wilson, A. George, CSP hybrid space computing, Journal of Aerospace Information Systems 15 (4) (2018) 215–227.
URL https://doi.org/10.2514/1.I010572

[5] J. Samson, J.R., E. Grobelny, S. Driesse-Bunn, M. Clark, S. Van Portfliet, Post-TRL6 dependable multiprocessor technology developments, in: Aerospace Conference, IEEE, 2010. doi:10.1109/AERO.2010.5446658.

[6] A. Pawlitzki, F. Steinmetz, multiMIND–high performance processing system for robust newspace payloads, in: 2nd European Workshop on On-Board Data Processing (OBDP2021), 2021. doi:10.5281/zenodo.5521502.

[7] R. Costa Amorim, R. Martins, P. Harikrishnan, M. Ghiglione, T. Helfers, Dependable MPSoC framework for mixed criticality applications, in: 2nd European Workshop on On-Board Data Processing (OBDP2021), 2021. doi:10.5281/zenodo.5521521.

[8] P. Kuligowski, G. Gajoch, M. Nowak, W. Sładek, System-level hardening techniques used in the COTS-based data processing unit, in: 2nd European Workshop on On-Board Data Processing (OBDP2021), 2021. doi:10.5281/zenodo.5521575.

[9] A. Lund, Z. A. H. Hammadeh, P. Kenny, V. Bensal, A. Kovalov, H. Watolla, A. Gerndt, D. Lüdtke, ScOSA system software: The reliable and scalable middleware for a heterogeneous and distributed on-board computer architecture, CEAS Space Journal (Mai 2021). doi:10.1007/s12567-021-00371-7.

[10] A. Kovalov, T. Franz, H. Watolla, V. Vishav, A. Gerndt, D. Lüdtke, Model-based reconfiguration planning for a distributed on-board computer, in: 12th System Analysis and Modelling (SAM) Conference - Languages, Methods and Tools for AI-based Systems, co-located with MODELS 2020, Virtual Event, Oct. 19-20, 2020, Association for Computing Machinery (ACM), 2020, pp. 55–62. doi:10.1145/3419804.3420266.

[11] L. Zohrati, M. Abadeh, E. Kazemi, Flexible approach to schedule tasks in cloud-computing environments, Iet Software (2018). doi:10.1049/IET-SEN.2017.0008.

[12] K. Karmakar, R. K. Das, S. Khatua, Resource scheduling for tasks of a workflow in cloud environment, Lecture Notes in Computer Science (2020). doi:10.1007/978-3-030-36987-3_13.

[13] W. Zheng, Z. Chen, R. Sakellariou, L. Tang, J. Chen, Evaluating DAG scheduling algorithms for maximum parallelism, 2020 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (2020). doi:10.1109/ISPA-BDCLOUD-SOCIALCOM-SUSTAINCOM51426.2020.00033.

[14] L. Liu, G. Xie, L. Yang, R. Li, Schedule dynamic multiple parallel jobs with precedence-constrained tasks on heterogeneous distributed computing systems, in: 2015 14th International Symposium on Parallel and Distributed Computing, 2015, pp. 130–137. doi:10.1109/ISPDC.2015.22.

[15] R. M. Sahoo, S. K. Padhy, A novel algorithm for priority-based task scheduling on a multiprocessor heterogeneous system, Microprocessors and Microsystems 95 (2022).
URL https://doi.org/10.1016/j.micpro.2022.104685

[16] B. Hu, Z. Cao, L. Zhou, Adaptive real-time scheduling of dynamic multiple-criticality applications on heterogeneous distributed computing systems, in: 2019 IEEE 15th International Conference on Automation Science and Engineering (CASE), 2019, pp. 897–903. doi:10.1109/COASE.2019.8842895.

[17] M. N. Krishnan, R. Thiyagarajan, Multi-objective task scheduling in fog computing using improved gaining sharing knowledge based algorithm, Concurrency and Computation: Practice and Experience (2022). doi:10.1002/CPE.7227.

[18] M. Chatterjee, S. K. Setua, A multi-objective deadline-constrained task scheduling algorithm with guaranteed performance in load balancing on heterogeneous networks, SN computer science (2021). doi:10.1007/S42979-021-00609-5.

[19] L. Xu, J. Qiao, S. Lin, W. Zhang, Dynamic task scheduling algorithm with deadline constraint in heterogeneous volunteer computing platforms, Future Internet (2019). doi:10.3390/FI11060121.

[20] L. Eskandari, J. Mair, Z. Huang, D. Eyers, I-Scheduler: Iterative scheduling for distributed stream processing systems, Future Generation Computer Systems (2021). doi:10.1016/J.FUTURE.2020.11.011.

[21] S. Ahmad, C. S. Liew, E. U. Munir, T. F. Ang, S. U. Khan, A hybrid genetic algorithm for optimization of scheduling workflow applications in heterogeneous computing systems, Journal of Parallel and Distributed Computing (2016). doi:10.1016/J.JPDC.2015.10.001.

[22] A. von Renteln, U. Brinkschulte, M. Pacher, The artificial hormone system—an organic middleware for self-organising real-time task allocation, Organic Computing—A Paradigm Shift for Complex Systems (2011) 369–384.
URL https://doi.org/10.1007/978-3-0348-0130-0_24

[23] J. Mei, K. Li, X. Zhou, K. Li, Fault-tolerant dynamic rescheduling for heterogeneous computing systems, Journal of Grid Computing (2015). doi:10.1007/S10723-015-9331-1.

[24] D. Feng, B. Liu, J. Gong, An on-board task scheduling method based on evolutionary optimization algorithm, Journal of Circuits, Systems and Computers (2022). doi:10.1142/S0218126623501001.

[25] G. te Hofsté, A. Lund, M. Ottavi, D. Lüdtke, Towards the online reconfiguration of a dependable distributed on-board computer, in: Architecture of Computing Systems, Springer, Cham, 2024, pp. 127–141. doi:https://doi.org/10.1007/978-3-031-66146-4_9.

[26] D. Lüdtke, T. Firchau, C. G. Cortes, A. Lund, A. M. Nepal, M. M. Elbarrawy, Z. H. Hammadeh, J.-G. Meß, P. Kenny, F. Brömer, M. Mirzaagha, G. Saleip, H. Kirstein, C. Kirchhefer, A. Gerndt, Scosa on the way to orbit: Reconfigurable high-performance computing for spacecraft, in: 2023 IEEE Space Computing Conference (SCC), 2023, pp. 34–44. doi:10.1109/SCC57168.2023.00015.