Towards the Online Reconfiguration of a Dependable Distributed On-board Computer

Glen te Hofsté*, Andreas Lund*, Marco Ottavi‡, Daniel Lüdtke†
* Institute for Software Technology, German Aerospace Center (DLR), Weßling, Germany
{glen.hofste,andreas.lund}@dlr.de
†Institute for Software Technology, German Aerospace Center (DLR), Braunschweig, Germany
daniel.luedtke@dlr.de
‡University of Twente (UT), Enschede, The Netherlands
University of Rome Tor Vergata, Italy
m.ottavi@utwente.nl

***Abstract*—On-board Computers (OBC) are at the centre of space-faring systems. They provide computational performance to the system with high availability and dependability. However, these systems typically consist of expensive, slow, fault-tolerant hardware to cope with errors or failures during a mission. Commercial-off-the-shelf (COTS) components offer higher performance but do not provide the fault-tolerance mechanisms. The *ScOSA (Scalable On-board Computing for Space Avionics)* architecture uses COTS and rad-hard components as a distributed system, with the advantage of providing more computing performance than current OBCs while maintaining the dependability properties.**

ScOSA uses a middleware to manage the COTS components as a distributed system of nodes, which, in the event of a node failure, mitigates the effects by reconfiguring the system to a configuration that excludes the failed node using a predetermined configuration. These configurations are computed *offline* and have an exponentially growing memory usage depending on the number of nodes in the system, which limits the system's scalability. This paper presents an *online* reconfiguration algorithm as a solution to this scalability problem. Upon the occurrence of a node failure event, the online algorithm makes scheduling decisions at run-time, eliminating the need for predetermined configurations. A novel online scheduling mechanism, consisting of six phases, which includes a combination of fault-tolerance, parallelism, and the use of the real-time state of the system, is a step towards higher dependability in distributed on-board computing. The online reconfiguration is evaluated by comparing it to the offline reconfiguration in terms of *time* and *network traffic*, showing that it is not only capable of generating configurations dynamically but also provides a solution to the scalability problem.

***Index Terms*—Fault-Tolerance, On-board Computers, Embedded Systems, Reconfiguration, Middleware, Distributed Systems, Dependability, Self-Configuration, Self-Healing**

I. 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 computational performance to provide their service or fulfill 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 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.

To meet the increased performance requirements, NASA has developed a hybrid architecture [5]. Using an Ethernet communication medium, the Dependable Multiprocessors integrate dependable processors together 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]–[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).

A. 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. By using a middleware [9] that runs on all processors, the distributed complexity is abstracted for the application developer, facilitating the development process. This middleware also allows that if a node fails, the applications of that node are automatically migrated to other available nodes. This makes the system reconfigurable and dependable. 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 is the *Network Dispatcher*, 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.

B. Reconfiguration Services

The *Monitoring Service* of the coordinator monitors the state of other nodes through periodic heartbeat messages. If a node stops responding to heartbeat or to other 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 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 to reconfigure by first stopping all tasks on all nodes and then (re)starting the tasks based on the new configuration. Finally, the *Reintegration Services* 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, a.k.a. configurations, are pre-determined during the design phase. This means that all configurations must be stored on the nodes of the OBC. Problem is that as the system scales up, the number of configurations grows *exponentially* [10]. This consumes significant memory that is not available to the applications. 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 an online algorithm for the reconfiguration of the ScOSA OBC in this paper. The online algorithm can use information that is present during run-time, as 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.

II. Related work

Online scheduling algorithms for distributed systems are not a new phenomenon. Research on cloud scheduling [11], [12] proposes several path-searching algorithms, while the priority-based scheduling techniques [13]–[16] show how ranking functions can be applied. Multi-objective scheduling [17], [18] can be used to optimise for a set of objectives instead of one, increasing the balance in the system in terms of load and network usage, while other types of scheduling methods [19]–[22] show that several other scheduling 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, while providing important insights into on the convergence of an algorithm for its dependability.

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. No solution exists, however, that combines fault-tolerance, parallelism and the ability to integrate multiple-objective scheduling in one solution. The fault-tolerant works in particular do not include features such as parallelism or multi-objective scheduling. Even though multiple-objective scheduling falls outside the scope of this paper, a novel solution is needed that has the ability to integrate these features into one algorithm, while taking full advantage of the features unique to ScOSA. 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 OBCs 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.

The algorithm design is presented in Section III and evaluated in Section IV. The results are presented and discussed in Section V, followed by a conclusion of the work in Section VI.

III. Design of the Online Algorithm

This work presents an online algorithm that was designed specifically for ScOSA. Although related work exists, a gap was identified where fault tolerance, parallelism, 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 feasibility and scalability in a real non-deterministic system environment.

The temporal behaviour of the algorithm is evaluated by separating the time it takes to decide whereto schedule a task t, and the time it takes to apply these decisions by a single reconfiguration. A decision can either be 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*

making time in which it schedules a task to a node, in terms of clock $c \in \mathbb{R}_{\geq 0}$ is defined as: $decision_{tc} \in \mathbb{R}_{\geq 0}$.

When all scheduling decisions are made, they are applied by reconfiguring the system. The *reconfiguration time* in terms of clock c is defined as: $reconfiguration_c \in \mathbb{R}_{\geq 0}$.

The *total reconfiguration time* (trt) during which the system is in a "state of reconfiguration" is then determined by the sum of the decision-making times $decision_{tc}$ to schedule a set of tasks t_{set} and the reconfiguration time $reconfiguration_c$ as:

$$trt = \sum_{\substack{t_{set} \in T \\ t \in t_{set}}} decision_{tc} + reconfiguration_c \tag{1}$$

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* can take place or a switch to safe mode. 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 can be optimised by rebalancing 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.

A. 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 schedule 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*.

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*.

In **Phase 2**, the algorithm checks for a *cache entry*, which can provide a quick response if a scheduling situation has already occurred before. However, 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 locks all the scheduling decisions until it is filled. This allows the performance of a cache load to be evaluated in terms of the time taken to handle an event.

With no cache entry to load, in **Phase 3**, the tasks to schedule are prioritised to determine in which order they are

Algorithm 1 Scheduling procedure

Require: N ▷ Set of healthy nodes n in the system
Require: T ▷ Set of tasks t to schedule

Phase 1: Assign node roles
  if N does not contain a coordinator node **then**
    Assign new coordinator n in N
  if $isCoordinator == True$ **then**
    if N contains a node n without a role **then**
      Assign a role to n
    Move to **Phase 2**
Phase 2: Check cache
  if A cache entry exists for system N **then**
    Schedule tasks according to the cache entry
    Move to **Phase 6**
  else
    $toSchedule \leftarrow T$ ▷ Set list of unscheduled tasks
    Move to **Phase 3**
Phase 3: Prioritise tasks
  if length($toSchedule$) > 0 **then**
    Calculate priority value of $toSchedule$ tasks
    $priorityTask \leftarrow$ highest priority task id
    Move to **Phase 4**
  else
    Move to **Phase 6**
Phase 4: Prioritise nodes
  Advertise highest priority task to all nodes in N
  The nodes return a calculated normalised priority value
  Node responses are appended to $nodePriorities$
  Sort $nodePriorities$ in descending order
  Move to **Phase 5**
Phase 5: Schedule task
  $i \leftarrow 0$ ▷ Node priority index
  Schedule the $priorityTask$ to $nodePriorities[i]$
  if $isSchedulingSuccessful == False$ **then**
    Attempt to schedule to lower priority nodes
    if Attempt successful **then**
      Remove $priorityTask$ from $toSchedule$
  else
    Remove $priorityTask$ from $toSchedule$
  Move to **Phase 3**
Phase 6: Finish reconfiguration
  Nodes affected by scheduling stop executing
  Scheduling changes are applied on the affected nodes
  The nodes start executing and reconfiguration finishes

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 deadlines for tasks, they 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 by each node in the system individually. 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 due to limitations in the *SpaceWireIPC* layer. This will be addressed in future work. Each node returns its 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 goes back to Phase 3 to schedule any remaining tasks. If there are no more tasks to schedule, the algorithm moves to Phase 6 to finalise the changes. 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 will 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, sending a reconfiguration finish message to all nodes, notifying them of the changes that were made. At this point, the nodes start executing tasks again, resulting in the system being fully available again.

IV. 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. Both setups use four tasks, consisting of three receiver tasks receiving the output of one sender task. Three test programs are compiled for each test setup:

- Test program 1: Offline reconfiguration
- Test program 2: Online reconfiguration with caching
- Test program 3: Online reconfiguration without caching

The two online reconfiguration programs are used to determine the impact of caching on time and traffic. The offline reconfiguration program is used as a benchmark to compare with the online reconfiguration.

1) Test Setup 1: Time Analysis: The first test setup is used to analyse the scheduling procedure time in terms of $reconfiguration_c$ and $decision_{tc}$. The test programs are run on the target hardware, consisting of a network of three HPN nodes connected via Ethernet.

The $reconfiguration_c$ parameter tracks how long it takes for the system to reconfigure and apply a new configuration. In addition to $reconfiguration_c$, the online algorithm requires additional time for decision-making. The parameter $decision_{tc}$ keeps track of how long it takes the coordinator to decide and schedule a set of tasks to a node. Due to the dependency on the network during the decision-making process, the network delay is also part of the overall decision-making time. Finally, $reconfiguration_c$ and $decision_{tc}$ are used to calculate the total reconfiguration time trt (1) in milliseconds. An execution time of 5 hours was chosen for each time analysis test to collect enough data points for statistical analysis.

2) Test Setup 2: Network Analysis: The second test setup is used to analyse the network traffic regarding 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 an offline reconfiguration changed to a dependency on the network, it is essential to determine how the network traffic scales. With the ability to increase the number of virtual nodes, the network traffic of an 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. The accumulated reconfiguration traffic in bytes is then used to analyse the different test cases.

V. Results and Discussion

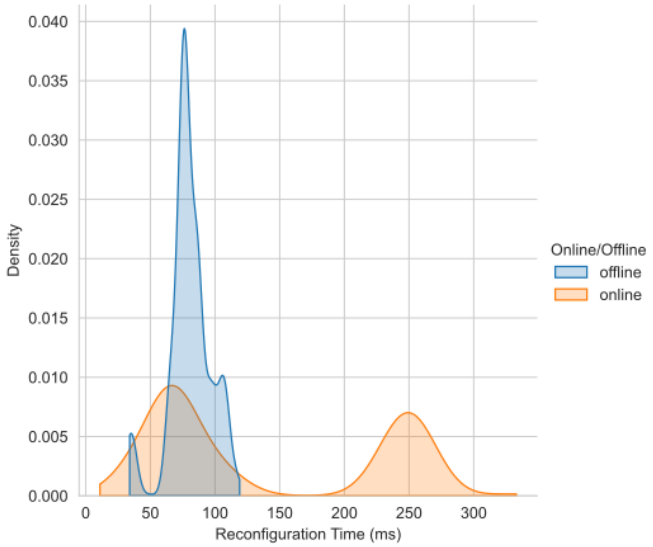
A. Timing Analysis

The timing analysis is performed on the results from test setup 1. The *reconfiguration times* of the three test programs can be seen in Table I. The results of test programs 2 and 3

are combined, as caching only impacts $decision_{tc}$, and not $reconfiguration_c$. The difference in the standard deviation between the offline and online cases can be seen in the distribution of Fig. 1, where a bimodal distribution can be seen. 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 reconfiguring, there is an increased reconfiguration time, resulting 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, similar to an offline reconfiguration.

TABLE I
RECONFIGURATION TIME (MS)

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



Kernel density estimation with Gaussian kernel
Smoothing bandwidth = 1, with independent function normalisation

Fig. 1. Reconfiguration time online vs. offline density plot

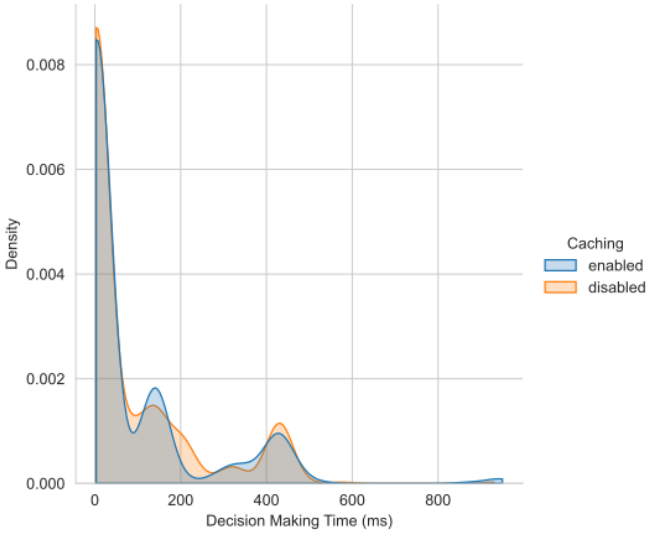
The $decision_{tc}$ of test programs 2 and 3 with cache *enabled* and *disabled* can be found in Table II. Note that these results also contain reconfigurations where only a node role was changed. In such a case, in Phase 3 of the algorithm, the size of the list of unscheduled tasks will be zero. This results in a direct transition to Phase 6 to finish the reconfiguration and a very low decision making time.

The two cases appear very similar, as supported by the distribution in Fig. 2, suggesting that caching does not have a noticeable effect. When cached decisions are separated from non-cached ones in Table III, caching appears to result in different decision times. For cached decisions, the average

TABLE II
DECISION TIME WITH CACHE ENABLED / DISABLED

Statistic	Reconfigs	Mean	Std Dev	Min	Max
Cache disabled	563	85.34	140.23	3	930
Cache enabled	570	84.92	155.91	3	950

is about half that of a non-cached decision. However, the number of cached decisions is only about a fifth of the total decisions. This is caused by the number of cache entries being fixed, therefore limiting the number of decisions it can store and load. The deviation of the cached decision is also smaller, with fewer outliers and a smaller max value. Rather than performing a full scheduling procedure for a situation that has already occurred, a quick load from the cache results in a reduced $decision_{tc}$. 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.



Kernel density estimation with Gaussian kernel
Smoothing bandwidth = 0.75
With independent function normalisation

Fig. 2. Decision making time caching influence density plot

TABLE III
DECISION MAKING TIMES (MS) WHEN LOADED FROM CACHE VS. NOT LOADED FROM CACHE

Statistic	Reconfigs	Mean	Std Dev	Min	Max
From cache	108	44.94	74.19	5	475
Not from cache	462	93.89	168.13	3	950

The trt outliers around max were found to be present only during the initial boot of the system. This happens when a (re-)starting node must completely set the system up from scratch. A changing system topology due to nodes failing or reintegrating can be handled quickly, but starting a (large) distributed system from scratch was found to take a long time.

One way to reduce the startup decision time is to increase the number of cache loads by implementing cache preloading. Like the offline algorithm, common system states can be predetermined and preloaded into the cache. The system can quickly load an optimised configuration for common situations, such as the nominal state when all nodes are healthy. This can make cache loads more frequent, resulting in a lower trt and increased availability.

The large standard deviation of the decision-making time (both with and without cache enabled) was found to be caused by the combination of reconfigurations with only a node role was changed, and the reconfigurations were also tasks were scheduled. This caused the results to be multi-modal, resulting in a large standard deviation.

B. Network Analysis

The test outputs from test setup 2 are used for the network analysis. Test programs 1 and 2 are used to compare the online and offline reconfigurations on a one-to-one basis for a three-node setup. The results are shown in Table IV. Here, we can see an increase of the average network traffic of about 3.3 times compared to the offline reconfiguration, which is well within the limits of the network.

TABLE IV
OFFLINE / ONLINE NETWORK TRAFFIC (BYTES)

Statistic	Count	Mean	Std Dev	Min	Max
Offline	23	491.43	33.67	436	511
Online	17	1592.94	41.80	1549	1680

Another important parameter is to see how the network traffic for the online algorithm *scales* as the number of nodes in the network increases. Fig. 3 shows the network traffic for an increasing number of nodes, both with caching enabled and disabled.

In both test programs 2 and 3, the network traffic increases linearly with a confidence interval of 95%, with the caching-enabled version requiring slightly less traffic. The largest system with 96 nodes had a worst-case network traffic of 139626 bytes, which is a large increase compared to the situation with three nodes but still within the acceptable limits of the network.

C. Memory analysis

When scaling up the number of nodes in the system using the network setup, the memory usage of test program 2, which uses the online algorithm, is analysed. The offline reconfiguration has been shown [10] to have memory consumption that scales exponentially. The online reconfiguration should solve this problem in particular. As nodes using the online reconfiguration node 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 on test program 2, it was discovered that the stack usage increases linearly with the number of nodes in the system, regardless of the node's role.

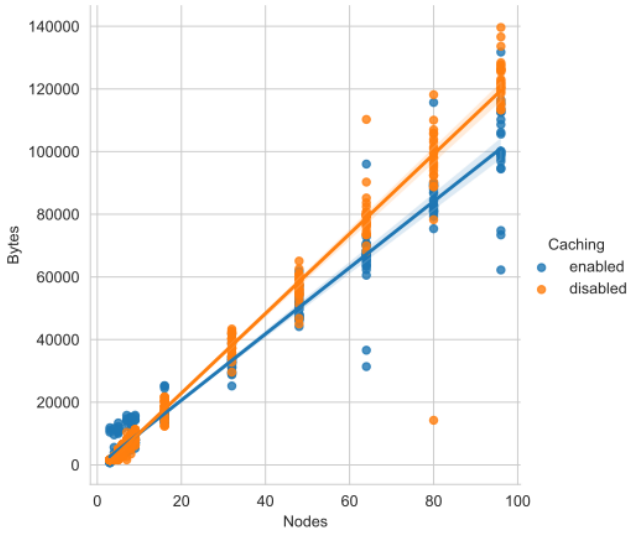


Fig. 3. Scaling network traffic with caching enabled / disabled

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 associated with 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. Online reconfiguration, therefore, allows larger system to utilise a distributed avionics middleware such as ScOSA.

VI. CONCLUSIONS

Dependability in spacecraft on-board computers remains a significant challenge. In this paper, we present 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 pre-determined 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. However, as the number of nodes in the system scales, the network traffic and memory usage increases linearly, compared to an exponential increase in

memory usage for the offline reconfiguration mechanism. From a maintainability perspective, changes to the scheduling phases can be easily made to further extend and optimise the decision-making process, paving the road for multi-objective scheduling in the future.

Enhanced with this online reconfiguration, ScOSA can be easily 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, i.e., constellations that dynamically distribute tasks among themselves.

As a first step, ScOSA will be demonstrated together with some typical space applications as part of a CubeSat mission in 2025 [25]. 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. [Online]. Available: https://www.baesystems.com/en-media/uploadFile/20210404045936/1434555668211.pdf

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

[3] A. N. Nikicio, W.-T. Loke, H. Kamdar, and 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.

[4] C. Wilson and A. George, "CSP hybrid space computing," *Journal of Aerospace Information Systems*, vol. 15, no. 4, pp. 215–227, 2018. [Online]. Available: https://doi.org/10.2514/1.I010572

[5] J. Samson, J.R., E. Grobelny, S. Driesse-Bunn, M. Clark, and S. Van Portfliet, "Post-TRL6 dependable multiprocessor technology developments," in *Aerospace Conference, IEEE*, 2010.

[6] A. Pawlitzki and F. Steinmetz, "multiMIND–high performance processing system for robust newspace payloads," in *2nd European Workshop on On-Board Data Processing (OBDP2021)*, 2021.

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

[8] P. Kuligowski, G. Gajoch, M. Nowak, and 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.

[9] A. Lund, Z. A. H. Hammadeh, P. Kenny, V. Bensal, A. Kovalov, H. Watolla, A. Gerndt, and D. Lüdtke, "ScOSA system software: The reliable and scalable middleware for a heterogeneous and distributed on-board computer architecture," *CEAS Space Journal*, Mai 2021.

[10] A. Kovalov, T. Franz, H. Watolla, V. Vishav, A. Gerndt, and 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), October 2020, pp. 55–62.

[11] L. Zohrati, M. Abadeh, and E. Kazemi, "Flexible approach to schedule tasks in cloud-computing environments," *Iet Software*, 2018.

[12] K. Karmakar, R. K. Das, and S. Khatua, "Resource scheduling for tasks of a workflow in cloud environment," *Lecture Notes in Computer Science*, 2020.

[13] W. Zheng, Z. Chen, R. Sakellariou, L. Tang, and 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.

[14] L. Liu, G. Xie, L. Yang, and 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.

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

[16] B. Hu, Z. Cao, and 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.

[17] M. N. Krishnan and R. Thiyagarajan, "Multi-objective task scheduling in fog computing using improved gaining sharing knowledge based algorithm," *Concurrency and Computation: Practice and Experience*, 2022.

[18] M. Chatterjee and S. K. Setua, "A multi-objective deadline-constrained task scheduling algorithm with guaranteed performance in load balancing on heterogeneous networks," *SN computer science*, 2021.

[19] L. Xu, J. Qiao, S. Lin, and W. Zhang, "Dynamic task scheduling algorithm with deadline constraint in heterogeneous volunteer computing platforms," *Future Internet*, 2019.

[20] L. Eskandari, J. Mair, Z. Huang, and D. Eyers, "I-Scheduler: Iterative scheduling for distributed stream processing systems," *Future Generation Computer Systems*, 2021.

[21] S. Ahmad, C. S. Liew, E. U. Munir, T. F. Ang, and S. U. Khan, "A hybrid genetic algorithm for optimization of scheduling workflow applications in heterogeneous computing systems," *Journal of Parallel and Distributed Computing*, 2016.

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

[23] J. Mei, K. Li, X. Zhou, and K. Li, "Fault-tolerant dynamic rescheduling for heterogeneous computing systems," *Journal of Grid Computing*, 2015.

[24] D. Feng, B. Liu, and J. Gong, "An on-board task scheduling method based on evolutionary optimization algorithm," *Journal of Circuits, Systems and Computers*, 2022.

[25] 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, and 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.