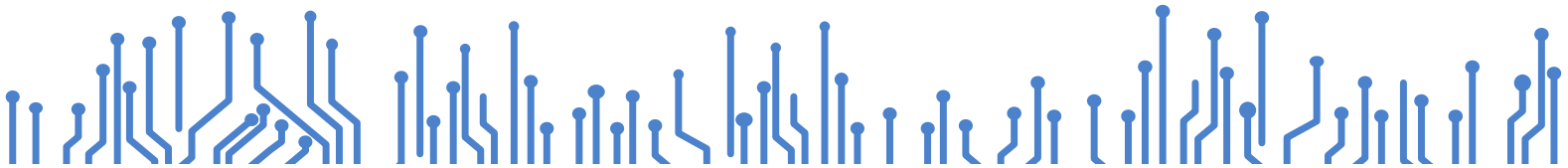
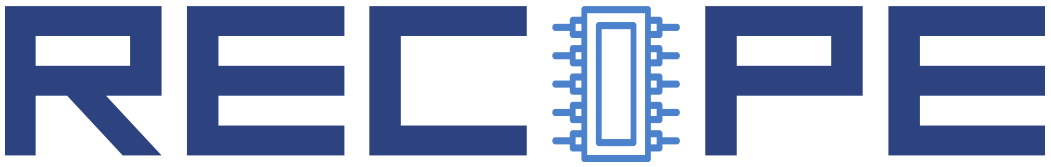
REliable Power and time-ConstraInts-aware Predictive management of heterogeneous Exascale systems





WP2 Runtime Resource Management Infrastructure

2.3 RECIPE Global Resource Manager Prototype

http://www.recipe-project.eu

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 801137

Grant Agreement No.: 801137
Deliverable: 2.3 RECIPE Global Resource Manager Prototype

Project Start Date: 01/05/2018 **Duration**: 36 months
Coordinator: *Politecnico di Milano, Italy*

Deliverable No:	2.3
WP No:	2
WP Leader:	Giuseppe Massari
Due date:	31/10/2019
Delivery date:	01/11/2019

Dissemination Level:

PU	Public Use	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

DOCUMENT SUMMARY INFORMATION

Project title:	**REliable Power and time-ConstraInts-aware Predictive management of heterogeneous Exascale systems**
Short project name:	RECIPE
Project No:	801137
Call Identifier:	H2020-FETHPC-2017
Thematic Priority:	Future and Emerging Technologies
Type of Action:	Research and Innovation Action
Start date of the project:	01/05/2018
Duration of the project:	36 months
Project website:	http://www.recipe-project.eu

2.3 RECIPE Global Resource Manager Prototype

Work Package:	WP2 Runtime Resource Management Infrastructure
Deliverable number:	2.3
Deliverable title:	RECIPE Global Resource Manager Prototype
Due date:	31/10/2019
Actual submission date:	01/11/2019
Editor:	M. Zapater
Authors:	M. Zapater, I. Penas, A. Iranfar, D. Atienza
Dissemination Level:	PU
No. pages:	32
Authorized (date):	31/10/2019
Responsible person:	W. Fornaciari
Status:	Plan Draft Working **Final** Submitted Approved

Revision history:

Version	Date	Author	Comment
v.0.1	01/10/2019	M. Zapater	Initial skeleton
v.1.0	15/10/2019	M. Zapater, I. Penas	Contributions to GRM
v.1.1	20/10/2019	M. Zapater, A. Iranfar	Reliability modeling
v.2.0	25/10/2019	M. Zapater	Improving GRM contributions
v.2.1	31/10/2019	M. Zapater	Addressing review comments

Quality Control:

	Who	Date
Checked by internal reviewer	POLIMI	31/10/2019
Checked by WP Leader	Giuseppe Massari	31/10/2019
Checked by Project Technical Manager	G. Agosta	31/10/2019
Checked by Project Coordinator	W. Fornaciari	31/10/2019

COPYRIGHT

ACKNOWLEDGEMENTS

RECIPE is a project that has received funding from the European Union's Horizon 2020 research and innovation programme under Grant Agreement No 801137. Please see http://www.recipe-project.eu for more information.

The partners in the project are Universitat Politècnica de València (UPV), Centro Regionale Information Communication Technology scrl (CeRICT), École Polytechnique Fédérale de Lausanne (EPFL), Barcelona Supercomputing Center (BSC), Poznan Supercomputing and Networking Center (PSNC), IBT Solutions S.r.l. (IBTS), Centre Hospitalier Universitaire Vaudois (CHUV). The content of this document is the result of extensive discussions within the RECIPE ©Consortium as a whole.

DISCLAIMER

Contents

1 Introduction

In this deliverable we introduce a prototype version of the Global Resource Manager (GRM) infrastructure of the RECIPE project, which comprises the GRM software architecture and implementation, and the policies for predictive reliability management, and power/performance/thermal awareness. The GRM initial software components were the outcome of the previous EU funded project H2020 MANGO. Within RECIPE, the GRM has been enhanced to improve the performance of the software itself, has been equipped with techniques for proactive reliability management, and novel multi-objective scalable and self-learning policies have been develop to enable power, reliability and performance aware resource management at the global level. This deliverable summarizes the results of T2.3 (WP2). The policies developed are based upon the models created as part of WP3.

In what follows we describe the structure of this document.

In Section 2 we provide an overview of the software architecture of the Global Resource Management framework, and its main components (SLURM, docker, kafka), in charge of performing workload dispatching among the multiple nodes of the HPC infrastructure, which will be extended by introducing reliability-awareness to properly implement proactive policies.

Section 4 describes our techniques for proactive reliability managemenr from the GRM perspective. In particular, we describe the thermal-aware reliability models used, together with the concept of *lifetime deposit* that allows us to increase the Mean Time To Failure (MTTF) of the system.

Section 5 summarize our efforts and current status towards the development of multi-objective policies able to incorporate power, performance, temperature and reliability awareness. We will present the current status of our multi-agent reinforcement learning (RL) based techniques.

Finally, in Section 6 we summarize the conclusions and some final remarks.

1.1 Overview of the Global Resource Manager of RECIPE

The main objective of the *Global Resource Manager (GRM)* in RECIPE is to act as a single-entry point for all the applications being run, to decide at runtime the most appropriate node to execute each new incoming application, and to enable proactive (i.e., model-based) thermal and reliability aware strategies. The GRM base architecture is built upon previous results of EPFL in the MANGO H2020 European project. Within RECIPE, the baseline GRM deployed in MANGO is extended in three different ways to pursue the following goals:

- Supporting the new heterogeneous resources in RECIPE (i.e., including GPUs), as well as the new software programming models proposed as part of the RECIPE project.
- Enabling the use of reliability models during runtime, to predict the changes in the MTTF of the hardware resources in RECIPE
- Proposing novel multi-objective resource management policies which include reliability,

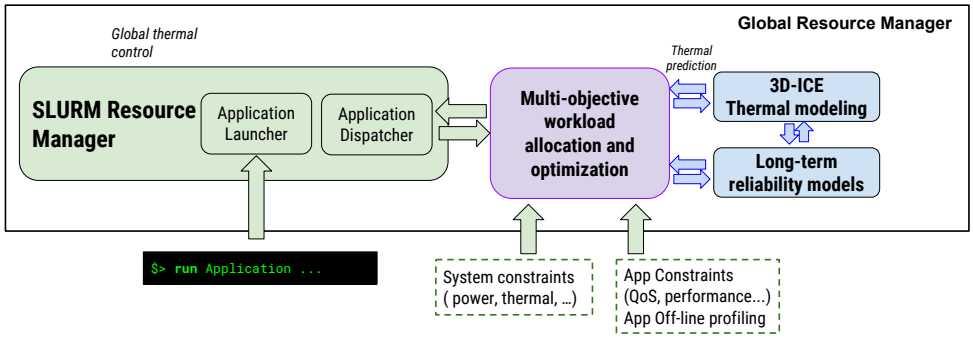


Figure 1: Overview of the BarbequeRTRM with the integration of RECIPE specific components.

temperature, power, performance (understood as both execution time and accuracy requirements) of the applications as their main goals.

An overview of the main blocks of the GRM is provided in Figure 1. Its main components are the following:

- The Resource Manager software, which is in charge of managing incoming workloads, scheduling (i.e., queuing them) and allocating them to the nodes. In our case, this RM software is SLURM. SLURM is complemented with the rest of the services developed in the GRM to adapt its functionality to the heterogeneity of the RECIPE hardware.
- The multi-objective workload allocation policies (in what follows, "GRM Allocator"), which take decisions on the specific allocation of tasks to nodes. The GRM allocator contains the reliability/thermal/ power/ performance/ - aware policies, which are in charge of improving the energy efficiency and performance of the system.
- The thermal models derived using the off-line version of the 3D-ICE thermal simulator, as well as the online prediction modules of 3D-ICE, which are in charge of predicting the thermal behaviour of the system during runtime.
- The long-term proactive reliability models, which include both the off-line version of the prediction models specifically developed for the RECIPE resources, together with the online prediction reliability modules, which will guide the GRM Allocator decisions.

From a purely implementation perspective, the GRM consists of a bunch of services working together in a coordinated way. The core functionality of the GRM, and the main contribution within RECIPE are the "GRM Allocator", which takes decisions upon how and where to allocate a workload (i.e., contains policies and algorithms), and "long-term reliability" module, which influences the Allocator decisions provided by the long-term reliability models, which in turn, are influenced by the 3D-ICE thermal models.

In what follows, we will provide further details on the GRM software architecture, and on how we have tailored the SLURM Resource Manager and the Allocator modules to the RECIPE infrastructure, and we will highlight the communication with the Local Resource Manager (LRM).

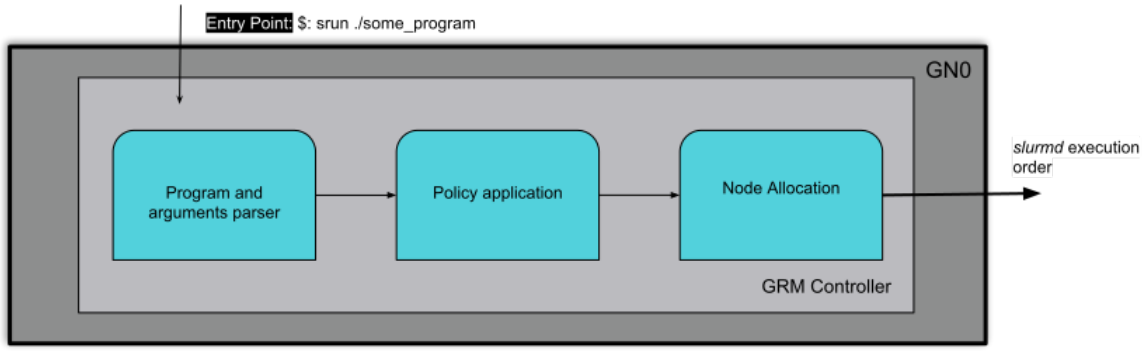


Figure 2: SLURM flow, application execution, policy and node allocation

2 Global Resource Manager Architecture

2.1 GRM Architecture

The GRM is in charge of performing workload allocation decisions, assigning each incoming task to a cluster in a temperature-, reliabilibity-, power- and performance-aware fashion. To this end, the GRM interacts with each of the instances of the LRM (BarbequeRTRM), which will be running in one of the Intel cores (on the server-side) of each of the heterogeneous clusters of RECIPE. Because of its hierarchical construction, as the global resource manager is the top-level service of the cluster, it will need to have visibility and full connectivity with all the resources available (and in particular of all the running instances of Barbeque RTRM). Only knowing their availability and load will suffice, as the LRM will be the one in charge or running the application on the accelerators.

When a new application is launched in the cluster via the entry point, the GRM Allocator will apply one upon the various policies developed ad-hoc for the RECIPE project, which will decide the node where the application will be executed, and will finally perform the allocation, by executing the SLURM Controller (in particular the slurmctld), which will, in turn, use the slurmd of the selected node, assign the task to a node, and pass its control to BarbequeRTRM.

To facilitate deployment and in order to achieve the objective of replicability and robustness, every service on the GRM controller and database services node will be running under docker containers[1]. By using Docker containers we are also reducing the overhead of software installation in the host machines that are needed in the Global Resource Manager.

However, because we want to also ensure the highest performance possible of the compute nodes of the cluster, specially ensuring that we can launch workload in both CPUs and GPUs, we do not dockerize the SLURM daemons (slurmd) components, which are natively run on the servers.

We modify SLURM to equip it with a number of features specifically targeted to the RECIPE platform. As a base installation, we used an already dockerized version of SLURM, which is publicly available on Github [2]. In this base docker installation, we can find a simulated cluster

[1] https://www.docker.com/
[2] https://github.com/giovtorres/slurm-docker-cluster

Figure 3: GRM Deployment Overview

prepared to be executed in only one host. However, for the RECIPE deployment we need to manage a real cluster with multiple nodes. Therefore, the following modifications were applied in order to adapt the base dockerized SLURM to the following requirements:

- Updating SLURM to the latest available version available in the Linux distribution of the container (Ubuntu Bionic Stable version 18.06.3 and SLURM 0.4.3-2build2, in our specific case).
- Changing SLURM compilation to force it to use an external allocator that will be capable of managing the heterogeneous cluster.
- Changing the deployment file named "docker-compose.yaml" as follows:
  - SLURMCTL container running on the master node (recipe0).
  - SLURMDBD container running on the master node (recipe0).
  - MySQL container running on the master node (recipe0).
  - SLURMD running natively on each of the slave nodes of the RECIPE prototype (recipe$i$, $i = 0..n$).

In order to store monitoring information about the current status of the cluster and history data, which will be provided by BarbequeRTRM, a Kafka server is used. Kafka was chosen as it can handle millions of messages per second and provides high reliability. Kafka is defined as a real time data Streaming publish/subscribe message broker redesigned as a distributed commit log. The main components of Kafka are messages, topics, Publishers and Subscribers. Topics work as distributors in the broker, so it can be shared among different applications. Messages compose the information transmitted through the broker being the publisher the instance that sends messages into the broker and the consumer the service that is subscribed to a specific topic. Also, in case of need, Kafka offers easy replicability. In summary, roles were changed so that daemons publish messages and servers subscribe to the topic where those messages are published. The Kafka server was taken from Apache Licensed Spotify docker repository[3].

The previous organization is depicted in Figure 3, where Docker containers are represented as boxes with rounded corners, whereas native software components are represented as squared boxes.

2.2 The SLURM Resource Manager tool

As described in the previous section one of the components of the GRM is the workload scheduler and allocator. Indeed, this is the most important component as it will allow to achieve the objective of having a single entry-point, and will also enable full control on the application execution status. SLURM was selected to accomplish this purpose as it is a highly scalable cluster management tool. However, because it is a general-purpose HPC cluster management tool, it does

[3]https://hub.docker.com/r/spotify/kafka/

not provide the heterogeneity-awareness required by RECIPE. However, by implementing the interaction between SLURM and BarbequeRTRM, we can enable the hierarchical approach and integration between GRM and LRM. Furthermore, because of its open-source nature, SLURM can be easily modified to suit the requirements of RECIPE, without requiring any kernel modification on the host system. Moreover, it enables replication and fault-tolerance, therefore being very suitable for single-entry point systems.

Regarding the architecture, SLURM works in a centralized way having a central manager called *slurmctld* (SLURM controller) responsible of monitoring the status of applications and the resource availability. On the other hand, each node will be running a SLURM daemon, or *slurmd*, which supervises applications running on each node and reports its status. In short, SLURM daemons work as a remote shell for the controller. Moreover, all the information gathered by SLURM, can be stored in a MySQL database, managed by the *slurmdbd* daemon, which runs in the controller and records accounting information, historical data and current status of the nodes, among others.

An overview of how SLURM works is depicted in Figure 4, taken from the SLURM official documentation.

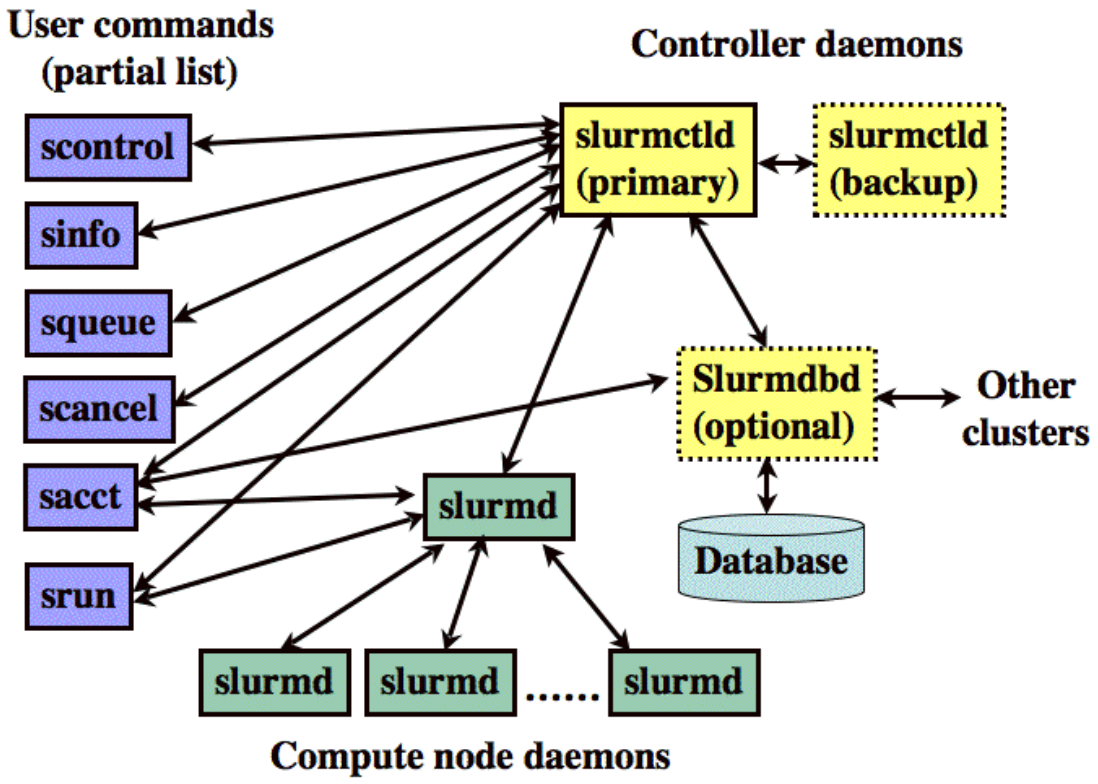


Figure 4: Overview of SLURM basic blocks

Regarding allocation, SLURM comes with a very basic set of scheduling and allocation policies, which consist on round-robin and "best fit / first fit" algorithms. However, as mentioned before, RECIPE requires more sophisticated allocation policies (capable of being aware of the hetero-

geneous resources) than the ones included in SLURM. Therefore, there is a need to include new policies in SLURM.

Due to the complexity of the RECIPE project the fully integrated plugin has not been possible to develop. There are two main reasons; the first is that in order to retrieve data from the LRM it is necessary a client permanently connected to it; and the second, the diversity of accelerators made impossible to integrate a plugin containing all these features. In consequence, we needed an external allocator which was capable of taking into account the diversity at the same time it receives updates from the LRMs. Thus, SLURM will work as log parser and executioner tool as the external allocator will order SLURM to execute a specific application in a specific node. From a purely implementation perspective, we are going to create two different instances, one running SLURM and other one running the external allocator daemon.

Therefore, the installation of SLURM will require the installation of the external allocator service adapted to RECIPE. The external allocator will be running in the same container as *slurmctld* because they need direct access in order to exchange information.

SLURM Basic Commands

This subsection shows the basic commands needed to execute, cancel and monitorize SLURM status.

- Running applications:
  - `srun`: run a parallel job on the cluster. After running this command, SLURM calls the custom allocator mentioned before where the optimal allocation (regarding temperature, power, performance, etc.) is performed.
  - `sbatch`: subscribes a batch script to SLURM. When the allocation is granted SLURM runs the script once.
- Management:
  - `scontrol`: monitorizes and allows the modification of SLURM configuration as job, nodes, partition.
  - `scancel`: signals and stops jobs under SLURM control
- Monitorization:
  - `sinfo`: displays the current status information of the cluster
  - `squeue`: shows information of the scheduled jobs

2.3 Docker as connectivity and deployment tool

Docker is a tool we can compare with a virtual machine. But, unlike a virtual machine docker shares the kernel of the host computer making the containers lighter and the deployment much faster. Containers are packages that envelops the code and all its dependencies which can be easily and quickly migrated among computers.

Docker Swarm is a cluster manager tool. In a Docker environment, a Swarm is a cluster composed by Docker daemons which can be virtualized or in actual machines. In our case, each General Purpose node will act as a Node(how Docker Swarm calls the components of the cluster) meanwhile GN0 is the Swarm Master.

Docker Swarm is a tool provided by Docker Engine which is specifically designed for cluster management, fast deployments, decentralized designs and scalability. Even more important for the GRM is the networking capabilities of Docker Swarm as it permits to create overlay networks and then attach the required container to that network. Also, all the containers attached to the same overlay network have full connectivity which means that the port bidding for connections inside the network is not necessary and solves many connectivity issues. This way we are parsing all the traffic that would go through the ports of SLURM and Kafka on the host machines converge over the ports opened between hosts also described in that section.

Deployment

The deployment of the GRM consists on the following steps:

1. Establish connectivity between hosts by by means of opening the ports needed by the `srun` daemon to execute applications. Each srun opens 3 ports by default plus 2 more for every 48 hosts[4]. Thus, the number of ports reserved must be estimated depending on the size of the size of the cluster and the predicted number of applications to be run simultaneously. The ports required by slurmctld are directly bind during the deployment of the docker containers.
2. Create Swarm stack on the node which will work as master. Note that is possible to have more than one master node in Swarm. For RECIPE, recipe0 is the master. The swarm network is initialised using the following command:

```
$: docker swarm init
```

3. After executing the previous command the following one will prompt which is the command to join the rest all the nodes to Swarm

```
$: docker swarm join --token XXXXXXXX 192.168.99.121:2377
```

4. Once Swarm is deployed, the next step is creating the overlay netwokrs:

```
$: docker network create -d overlay --attachable
    --subnet=20.1.0.0/16 slurm_overlay

$: docker network create -d overlay --attachable
    --subnet=20.2.0.0/16 kafka_overlay
```

5. Join all the nodes to the Network File System folder located on the master node.
6. Run under the folder DockerizedSlurm/compose_deploy the following command. This command will deploy all the containers required by the GRM and will connect them as specified in the configuration file *docker-compose.yml* to the networks created before and will attach

[4] https://slurm.schedmd.com/slurm.conf.html#OPT_SrunPortRange

the corresponding containers to the volume containing the applications and the necessary data.

```
$: docker stack deploy -c docker-compose.yml recipe_slurm
```

3 Plugging-in policies via the allocator module

As previously stated, for the GRM to accomplish the goals of the project we need to apply and test different power/performance/temperature/reliability aware workload allocation policies. The "GRM Allocator" module is in charge of this. Furthermore, all policies need to take into consideration the current usage of the cluster, which requires a close to real time synchronization between local and global RM.

Given that the RECIPE platforms is equipped with a wide range of heterogeneous resources (regular CPUs, FPGAs and GPUs) a node could be unavailable for one application (e.g. that requires one specific accelerator on an FPGA) but available for another with different requisites (e.g., which requests a mixture of CPUs and GPUs). Thus, the global resource manager requires an underlying flexible system that can provide a complete overview of the cluster which, at the same time, can be updated with the current status for each application requirements. To express these dependencies, the GRM Allocator will use a graph network which contains all the available nodes. This support will be implemented by using the Networkx [7] Python library. Networkx provides all the necessary tools to manipulate, study and distribute for complex networks graphs

To perform the above mentioned tasks, the GRM Allocator consists on the following modules, which will be next explained:

- Cluster architecture builder
- Cluster architecture update
- Workload allocation algorithms

3.1 Cluster Architecture Builder

This component is composed by a JSON parser which is configured to read an architecture configuration file.

The main function of this component is to provide a default network architecture that can be used by other modules. The architecture of the platform managed by the GRM should be provided following the following JSON format so that the `json_reader` block is able to successfully build the architecture. Then the JSON is converted to Python dictionary which facilitates iterating over it.

Listing 1: Example of initial architecture input file

```
{
  "name": "RECIPE Cluster",
  "version": "1.0.1",
  "master": "gn0",
  "architecture": {
    "recipe0": {
      "id": "recipe0",
      "master": 1,
      "components": {
        "fpga-acc1": 0,
        "fpga-acc2": 0,
        "fpga-acc3": 0
      }
    },
    "recipe1": {
      "id": "recipe1",
      "master": 0,
      "components": {
        "gpu": 0,
      }
    },
    "recipe2": {
      "id": "recipe2",
      "master": 0,
      "components": {
        "gpu": 0,
      }
    }
  }
}
```

Listing 2 is a key point of the process. With this initial configuration we provide to the allocator the number of accelerators contained in each node. Also, providing the number of nodes and the name by which they need to be identified. This last feature is critical as both, the name of the node and the name of the accelerators must match with the names by which the BarbequeRTRM-Slurm parser codes the information injected into Kafka. The architecture defined in Listing 2 is transformed into the network represented in Figure 5.

To ensure that the allocation is performed correctly, the *Networkx* object that is instantiated in this step has to be a DiGraph, i.e., a directional graph. *Networkx* networks are composed by two items: nodes and edges. As shown in Figure 5 there are paths joining together the different nodes. These paths (called edges in the *Networkx* documentation) could be forced to have only one direction by declaring the NetworkX object itself as DiGraph or DiMultiGraph. *NetworkX* offers four different types of Graphs depending on the edges characteristics: Graph, DiGraph, MultiGraph and MultiDiGraph. For the GRM Allocator current design we use a DiGraph type, strict single directional path. In the case of re-creating an edge between nodes, this new edge

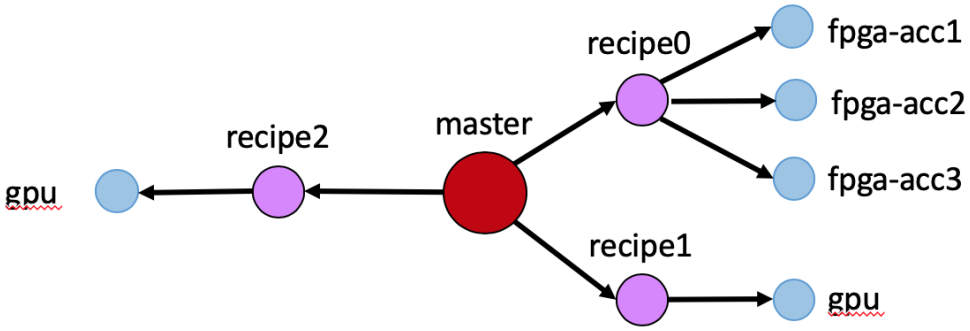


Figure 5: Network obtained after creating the initial network architecture given the configuration file shown in Listing 2

will substitute the previous which makes easier the process of updating the status of the cluster. On the other hand, to find the suitable allocation the shortest path is used as described bellow. Thus, directional paths assure that the allocation found is correct and avoid errors such as loops.

The edges described above, are the main component that would let us apply an arbitrary policy. Both edges and nodes can be assigned attributes. These attributes can be considered as a dictionary as each attribute is assigned a name and a value. However, one of them is mandatory, a predefined called weight, which can only be assigned a numeric value. This parameter is used by the path finding algorithms included in the library to find the shortest path between points. For simplification purposes, the Network architecture will be built like the one shown in Figure 5, then despite having more than one accelerator of each kind, every type will be represented by a single node. Then, this node contains two more attributes which are *#node* and *#node in use* representing the total number of hardware accelerators of the corresponding type node contains and the number of those currently in use respectively.

3.2 Cluster Availability Update

In order for the GRM Allocator to perform allocation decisions as closely to real-time as possible, it is necessary to have a real time "cluster overview". To accomplish this task, we use the Kafka Server (which will connect to BarbequeRTRM to gather data) and a Kafka Broker which consumes those messages, as we will describe next.

The "cluster overview" contains many data: availability, accelerator type, temperature, power consumption and errors report. Data from the nodes is gathered from the host system directly.

Apart from consuming Kafka messages, this block is also in charge of updating the DiGraph object. Firstly, a client which is registered as consumer in the Kafka topic *bbque_data* waits until a new message is received, then parses the data and calls the update function. Then, the algorithm weights the edges depending on the node availability. For instance, a node whose accelerators are fully busy is assigned a 1, while a node with free resources is assigned a 0. The same technique is applied at a local scale, meaning that the path joining a node (GN) and its accelerators (HNs) is set to 0 or 1 depending on their availability.

3.3 Algorithms Callback

This final block is called whenever a new allocation request is received and it is split into two sections. One in which the actual policy is applied and a second one where an auxiliary node is created.

Policies implemented for this project can be selected when launching an application, thus during run-time, which is very suitable on testing scenarios like ours. In consequence, if no policy is specified in the input command, the allocator will use whether the default one (Greedy) or other if it was previously modified. In order to make this situation possible the allocation is performed with a copy of the *NetworkX* object made at the time the new allocation request is received, then, depending on the algorithm selected the paths are weighted in different ways.

The copy of the main cluster overview object performed before, is in the second section used to include the auxiliary node thereby we keep the occupancy cluster clean so the availability update process is performed as fast as possible preventing thread block in the iterative daemon. The allocation decision is made automatically after the previous described process has been successfully executed. Afterwards we execute one of the most powerful tools of the library, the path finding algorithms. For this purpose, we chose Dijkstra algorithm which is widely used as it is one of the most efficient.

Dijkstra algorithm is an algorithm used for shortest path finding when the nodes (or as in our case, the path joining the nodes) are weighted. Thus, after the weighting process, we call this algorithm with the Master node as origin and the Auxiliary node as destination. Then, the algorithm finds the shortest path which corresponds with the allocation. Also, it provides the next two shortest paths so if after that process a problem is found, the next allocation is used.

Weighting the paths is a task that we leave for the specific policies. The algorithms callback assumes that the weighting has been performed by the policy called.

3.4 Entry Point

The fact of having an external allocator which will interact independently from SLURM implies the necessity of developing a custom entry point with high availability and reliability. In order to do so, a new Kafka topic has been added to the server. This Kafka topic will have 24 hours persistence to grant reliability and a historic of the previous hours requests. Availability is granted via multi-threading method where a new thread per request is created. Creating a new thread per request means we are able to handle a max of 64 per second (Safety time to generate the allocation and parsing the job to SLURM), good performance in terms of availability.

We need to take into consideration the collision between the updater and the entry point. As both threads require to access the cluster object it was necessary to establish dependencies among them so every time an instance needs to access the main object must call the mutex (MUTual EXclusion) [5] object related to the cluster. Thus only one thread is accessing it at a time.

Besides creating a new thread on demand, the entry point is located inside the cluster status update loop thus, assuring the last allocation policy is applied with updated information already

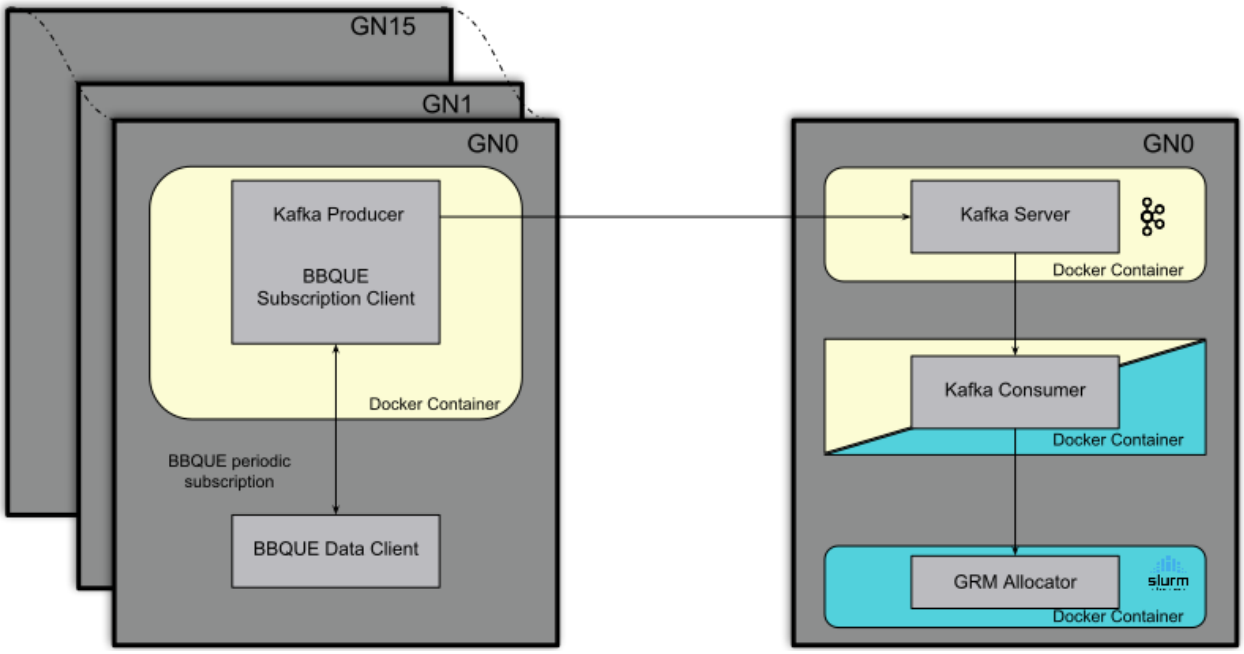


Figure 6: Flow overview, from BarbequeRTRM to SLURM

parsed.

3.5 Steps for enabling integration between LRM and GRM

The next steps for the GRM will be to accomplish the integration between the LRM and the GRM. In this section we provide further details about the procedure we will follow for integration of the GRM in the RECIPE software stack and the hardware. We split this explanation in two parts: (i) the explanation of how the master node (recipe0) acts as a single entrypoint for mapping applications, and (ii) how data is shared and used between the LRM and the GRM.

LRM-SLURM

As a GRM, SLURM requires having complete visibility upon the cluster, and in particular: mapping and resource usage for each application deployed in each cluster, utilization and load, and power and temperature of the available resources on the node and its accelerators. Data needs to be collected and used in (almost) real time, so that the allocator can efficiently allocate the workload, avoiding unnecessary queues at the LRM level.

Figure 6 illustrates all the components that perform the data gathering service from BarbequeRTRM to SLURM. In this figure, we can see two different components: (i) server and (ii) daemons, similarly to SLURM's original server-daemon architecture. On the server side, which always runs on GN0, we can find a broker and a database (Kafka and Allocator). These two components are mainly in charge of retrieving the status of the heterogeneous cluster and presenting the data to SLURM, so that it has an overview of the system to appropriately perform

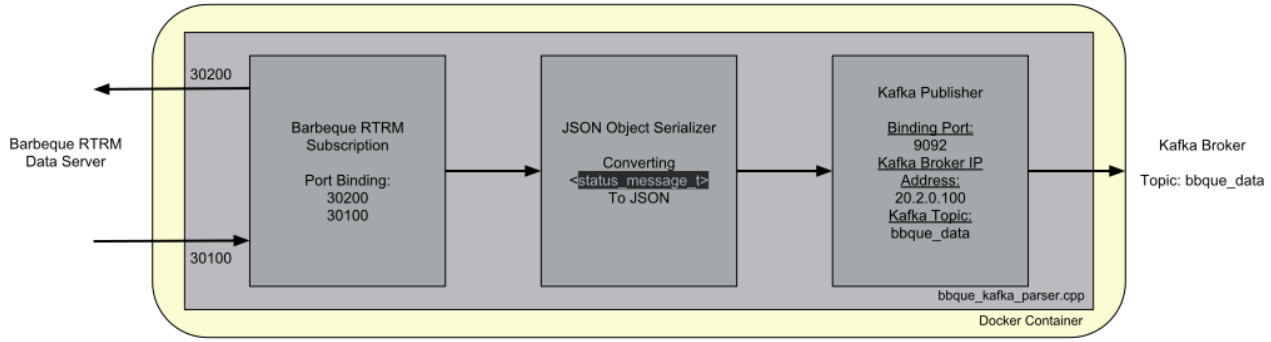


Figure 7: Data Parser work-flow from the Local Resource Manager to Kafka

the allocation decisions.

For SLURM, the most important feature is that the LRM monitoring implements a subscription-type server. This server provides the information of the node in which it is installed. Because of this, it was not possible to have only one instance on the main node (recipe0), as it would need to handle 16 subscriptions at the same time, jeopardising the cluster scalability and creating a single point of failure. Therefore, we decided to distribute the subscription servers, installing one in each slave node.

The daemonized part, illustrated by the recipe0 on the left in Figure 6, consists of a container that runs a C program composed by periodic a subscription to Barbeque data client that retrieves the corresponding data of that node, a Kafka Producer in charge of transmitting the information to the server, and an object serialization function which translates the object created on the subscription to the client with the status of the node to a more suitable JSON format. The flow is more precisely described in Figure 7

Regarding the Kafka Producer, every time a new message is received from the subscription (i.e, every 5 seconds), a new publication is pushed by this instance to Kafka Server. As shown in Figure 6 both subscriptions are running in the same service avoiding redundancy and the need of developing another service that connects both services.

The last step of this communication flow is the Kafka-consumer inside the external allocator. The external allocator will update the cluster status every time it consumes a message.

GRM-LRM

This section shows how the GRM can send an execution order to the LRM. In short, the user will try to execute an application from recipe0 (single entry-point) where the SLURM controller is running, and SLURM will be in charge of executing this application is the most suitable node depending on the objective of the allocation policy.

In order for this flow to work correctly, two issues had to be overcome: (i) how to execute a

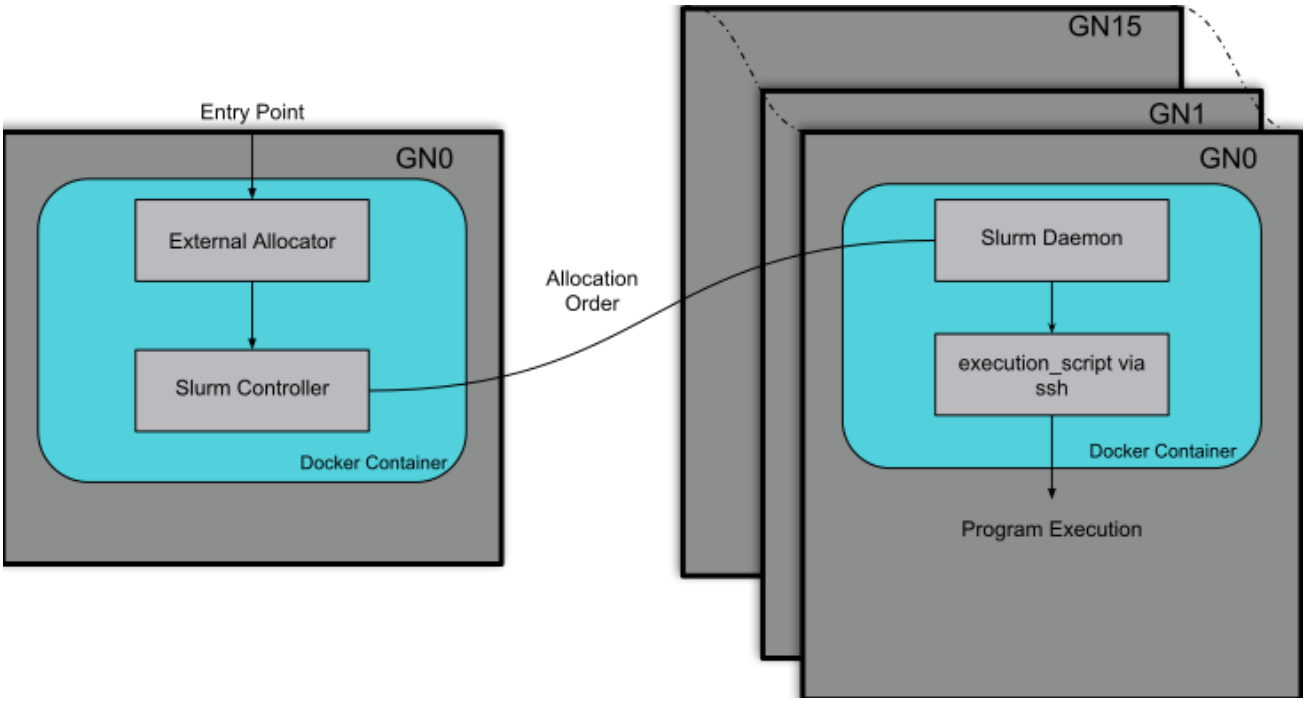


Figure 8: Application execution flow from the single entry-point on recipe0 to any LRM

program in the host computer from a docker container exporting all the libraries required by applications, and (ii) how to share that program/script to be run among the cluster so that the GRM controller does not need to handle sending the application to the actual node which will run the program.

In order to accomplish the first issue, the architecture illustrated in Figure 8 is developed. Then, with the next command, we can execute a script located in the docker container in the host machine: *"ssh user@172.17.0.1 'bash -h' < execute_app.sh"*

Thus, this *execute_app.sh* script can contain any export required by the applications.

Given the previous architecture, each application will require two execution scripts which contain all the library exports required by that application, its name, location on the shared folder, and some other parameters.

In short, the execution process requires two different scripts, one which will run the actual application and another which specifies the SLURM parameters required by that application. Initially, all the applications are executed under the same circumstances for testing and validation purposes.

The second issue can be overcome with a shared folder across the cluster, where all nodes have reading access permissions. To allow for this, we use SSHFS (Secure Shell File System) and docker volumes combined together. As a consequence, every host with a docker daemon is attached to the swarm network and all docker containers with the volume mounted will be able to see and execute that program.

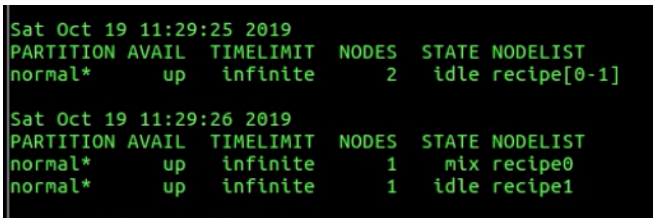


Figure 9: SLURM GRM nodelist and occupancy. Resources are idle right before launching workload (top) and become in mix state when workload is allocated (bottom)

3.6 Launching UC3 on CPU/GPU using the GRM

In what follows, we show an example of launching the inference of the CNN of Epilepsy application (UC3), on the recipe0,1 nodes in the cluster located at UPV, through the GRM. All components of the GRM, except the communication between GRM-LRM and LRM-GRM are used. Given that the integration between LRM-GRM for RECIPE has not yet been accomplished, the GRM launches the workloads directly into the servers, bypassing BarbequeRTRM. Furthermore, no policy is being used in the cluster, which therefore falls back to a greedy algorithm.

The cluster is therefore described as a JSON file, in the following way:

Listing 2: Example of initial architecture input file

```
{
  "name": "RECIPE Cluster",
  "version": "1.0.1",
  "master": "recipe0",
  "architecture": {
    "recipe0": {
      "id": "recipe0",
      "master": 1
   },
    "recipe1": {
      "id": "recipe1",
      "master": 0,
      "components": {
        "gpu": 0,
      }
    }
  }
}
```

Then we bring up the cluster with the two nodes. Before launching the workload, the GRM will show that both nodes are idle, as shown in the top part of Figure 9. The moment the workload is launched, the GRM will decide to allocate it to the node *recipe1*, as it contains a GPU. This node will switch to a mix state, in which part of its resources are allocated.

Once the workload is launched, the CNN will be allocated to both the CPUs and GPUs of the

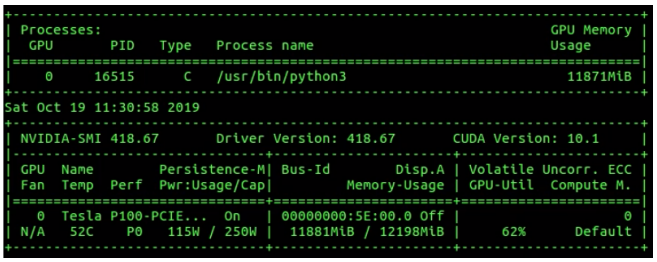


Figure 10: Status of the Tesla V100 GPU in node recipe1 once the workload of UC3 (CNN for epilepsy detection) is launched

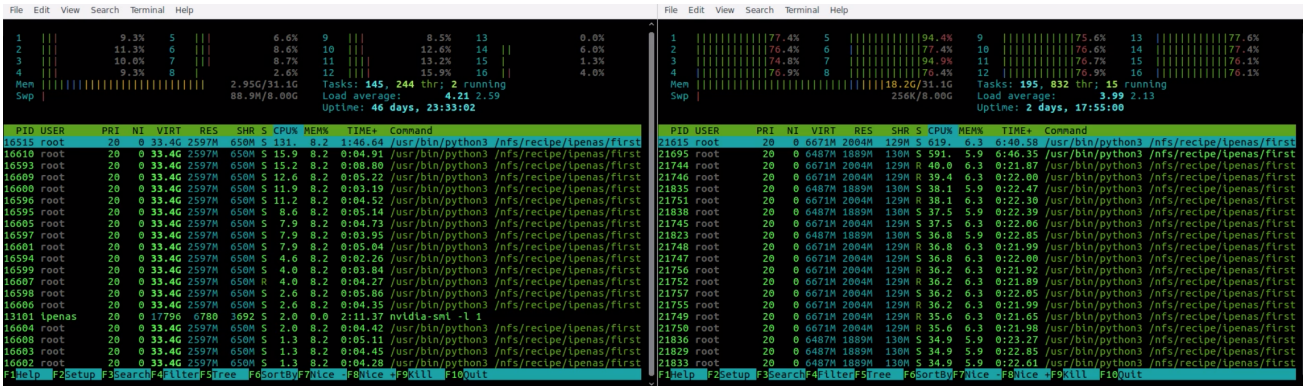


Figure 11: Occupancy of the recipe0 (left) and recipe1 (right) nodes, while running the CNN workload of UC3

recipe1 server, as shown on Figure 10. The workload will span all the CPUs of the system of recipe1 node, whereas the recipe0 node will remain idle, as no workload is allocated to it, as shown in Figure 11.

As shown the GRM is able to allocate both resources to CPU/GPU for the specific case of UC3. In future developments, we will enable the integration between GRM and LRM, incorporating monitoring data directly from BarbequeRTRM, which will enable to provide power/performance/reliability/thermal aware workload management and cooling control policies.

3.7 GRM enhancements in RECIPE and next steps

The current version of the GRM for RECIPE is able to provide workload management via two different allocators: (1) one for the FPGA resources (called external allocator) that connects with BarbequeRTRM; and, (2) a customized version of the consumable resources plug-in of SLURM for the CPU, memory and GPUs that controls the cluster status in a more general way.

In order to fully integrate the GRM with RECIPE, managing the accelerator heterogeneity and reducing the overhead, we will merge both allocators into a single one by developing a new custom plugin. This plug-in (`BBQUEparser`) should gather the FPGA status and update the SLURM database. On the other hand, we also propose the creation of an extension of the current *consumable resources*(cons_res) plug-in integrated already with the prototype that creates a new consumable resource for each FPGA RECIPE project accelerator and defines its

behaviour within the *cons_res* context.

The `BBQUEparse` will be able to read the information from the Kafka server and post it into the SLURM database. This functionally does not differ much from the current BarbequeRTRM parser integrated in the external allocator. The important point of creating a plug-in is to provide SLURM with tools to understand the status of the accelerators contained in the FPGA's. For this purpose, we need to create a standardized consumable resource.

The SLURM consumable resources plug-in already provides a template and instructions on how to create new resources. The most important point of this step is that the resources integrated in this plug-in should be synchronized with the information provided by BarbequeRTRM and, thus, their features standardized. Furthermore, these developments that we will undertake as next steps will enable us to use in a fully coordinated way all the heterogeneous resources (FPGAs and GPUs) of the cluster.

4 Proof-of-concept proactive reliability strategies

The goal of the proactive reliability policies developed within the GRM of RECIPE is to enable increasing the Mean Time To Failure (MTTF) of the system by monitoring temperature evolution through time, and changing the frequency (DVFS setup), the allocation of tasks to cores, or the cooling of the system (e.g., changing fan speed).

In this section, we provide an initial proof-of-concept of the capabilities of proactive reliability management. For this purpose, we first propose a reliability strategy based on the concept of *lifetime deposits*, and show how this would affect the reliability of a server system equipped with a fan-cooled Intel processor.

4.1 Thermal stress and reliability modeling

Thermal stress and spatial gradients are proven to have a negative effect on the long-term reliability of Multi-Processors System on Chip (MPSoCs) [2], impacting their Mean Time to Failure (MTTF) [9]. Reducing the thermal hot spots is not enough to achieve adequate thermal management for MPSoCs and increase MTTF. Any rapid temperature change, in either time or space, can be considered a thermal stress situation. Thermal issues due to temporal or spatial gradients and thermal cycling are among the most common temperature-induced reliability issues, and are therefore studies more in-depth in this project.

Temporal Temperature Gradient (TTG) can be defined as the rate of temperature changes over time. For a given time, the rate of the temperature changes from one point to another indicates the spatial temperature gradient (STG). Both STG and TTG pose a critical impact on the system lifetime reliability [3]. However, it has to be noted that STG is mostly affected by the power and thermal management techniques applied at the overall MPSoC system, i.e., the allocation and specific setup of all cores in the system need to be taken into consideration. In contrast, TTG is mostly affected by the core frequency and the workload running in each specific core.

Thermal cycling is the phenomena which takes place when the temperature rises up (or drops down) and goes back to the initial value (which can be defined as a thermal cycle) frequently [13]. Thermal cycling can be counted by Dowining simple rainflow-counting algorithms[6]. MTTF reduction due to thermal cycling occurs due to the mismatch on the expansion coefficient between the layers of the chip, which results in thermo-mechanical stresses. Thermal cycling (TC) tends to reduce the whole system MTTF as the number of cycles or amplitudes increases. Large amplitudes are normally induced due to improper task scheduling on a single core. Number of thermal cycles increases especially by the power management techniques which frequently turn cores on and off [3].

Even though several works consider thermal stress, they rarely provide a comprehensive solution to cope with all thermal stress mechanisms along with power constraints. Thermal management strategies, in fact, may exhibit contradictory goals between peak temperature reduction techniques and thermal stress reduction approaches. For instance, although in some works the trade-offs between temporal and spatial thermal gradient mitigation schemes are investigated [1],

power management and thermal cycling are not considered. In addition, other works [14] propose task scheduling methods for reducing temporal temperature gradients but disregard thermal cycling and spatial gradient.

Holistic policies, such as [8], are able to manage efficiently all thermal reliability aspects pave the way to collaborative hardware (e.g. DVFS) software (e.g. workload allocation and application configuration) techniques to enhance the reliability of the system while providing the adequate power/performance/QoS. However, the is still a lack of research works that tackle efficient thermal management with the goal of, not only controlling current thermal emergencies, but of preventing the consequences of thermal stress and that focus on long-term reliability both at the CPU and the the overall server level.

The increasing importance of thermal concerns, which impact reliability, in future HPC systems, call for appropriate task scheduling methods that consider all concerns holistically. Those approaches that mitigate ageing rather than simply predicting an imminent fault are of particular interest to minimise already high fault rates expected.

Together, performance and lifetime reliability issues make joint power and thermal management crucial and inevitable for modern MPSoCs, and also pose an important challenge from the energy efficiency perspective. Air-based cooling still remains the most common cooling solution, being present in above 80% of data centers[11], and therefore needs to be taken into consideration accurately to increase the overall MTTF of the system.

For the purpose of this work, and before adopting the final reliability models proposed in WP3 of RECIPE, we assume that the following MTTF model is able to capture most of the effect of temperature-induce reliability effects [12]. This model is in fact a simplified version of the models being currently developed in the project, but serves the purpose of demonstration.

4.2 Proactive reliability using lifetime deposits

To consider runtime reliability issues, apart from using an MTTF model, one critical aspect is to correctly estimate the thermal profile of the chip. For this, in the RECIPE project we make use of the 3D-ICE simulator [10] equipped with the models and enhancements developed as part of WP3 (Task 3.3) of RECIPE, and which are described in Deliverable 3.2.

To put both concepts together, while considering per-core spatial and thermal gradients, we leverage the concept of lifetime deposits. The lifetime deposit concept, instead of setting a pessimistic temperature threshold on the system, allows to dynamically adjust the lifetime of the system by either operating a lower temperature to increase overall lifetime, or by boosting performance for a period of time, violating reliability-safe temperature thresholds and *consuming* lifetime of the system. Because this period of high-temperature is compensated by the periods of low-temperature, if an adequate workload/system management is put in place, these policies will not degrade the overall design lifetime of the system. In summary, the reliability of the system can be depicted as in Figure 12.

The previous concept can be assessed during runtime execution of applications, by adjusting DVFS, which is one indirect control knob to change power and therefore temperature. Figure 13 demonstrates this concept. In this experiment we run two different workloads: (a) stressing the 8

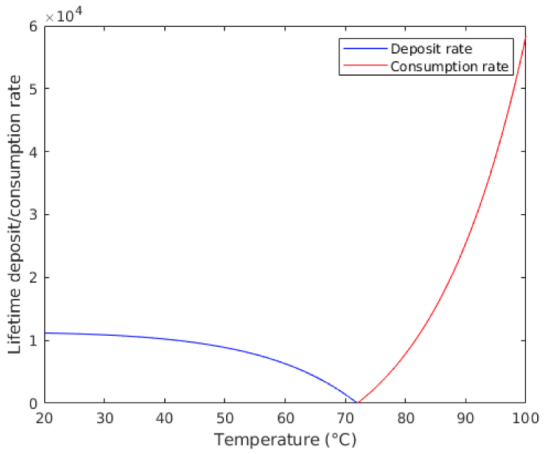


Figure 12: Lifetime deposit model considered in this work.

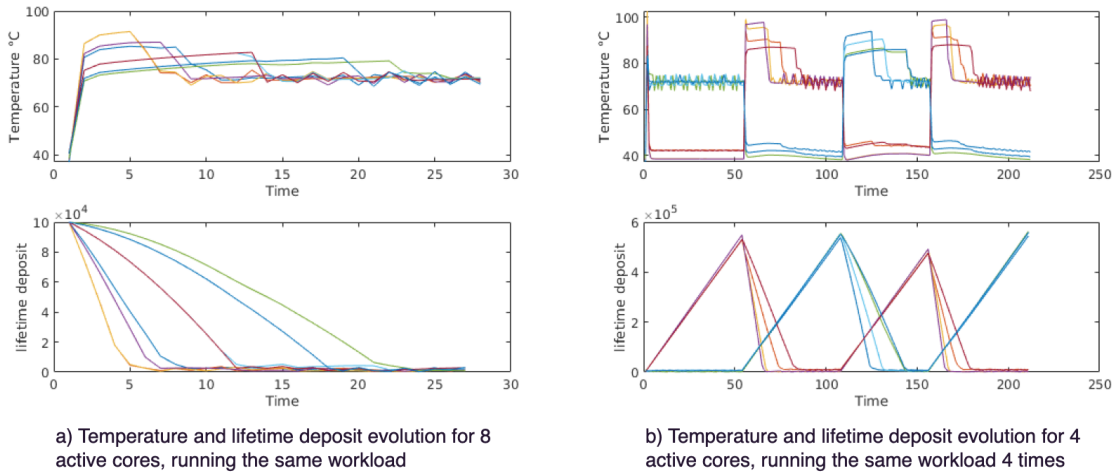


Figure 13: Proof-of-concept lifetime deposit evolution for a stress test.

cores of an Intel-based system with a CPU-intensive synthetic benchmark, and (b) stressing only 4 cores of the Intel system, but running the workload 4 consecutive times. Power measurements are obtained by real measurements using the RAPL tool on an Intel server similar to the ones deployed in the RECIPE platform. Temperature is estimated by using the 3D-ICE simulator and the models developed as part of WP3. Experiments assume a constant fan speed, where the only control knob of the system is DVFS. The figures show how by controlling both the workload allocation (4 vs 8 cores) and the frequency of the system we can consume the lifetime deposit at a different rate.

In order to control efficiently the evolution of the lifetime deposit, we need to find ways of adequately controlling the power consumption of the system via DVFS and task allocation. In Figure 14 we show a proof-of-concept control example, where we allow fine-grained temperature control on the processor. This work allows us to set the basis for runtime control of reliability due to thermal stress. These techniques will be further improved and integrated in the multi-objective multi-agent RM policies of the GRM.

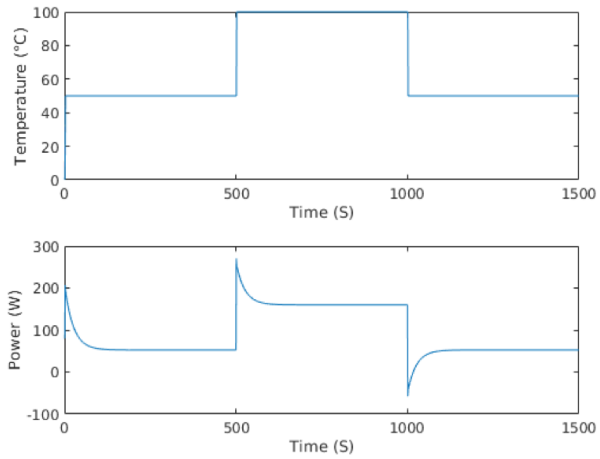


Figure 14: Proof-of-concept lifetime deposit evolution for a stress test.

5 Multi-objective policies at the global level

In this section we describe the multi-agent reinforcement learning infrastructure that will enable the development of runtime control of the power, performance, thermal and reliability of the RECIPE platform. The development of this infrastructure, named MAMUT, was started by the end of the MANGO project, but was not into the GRM, neither extensively tested with all use cases. Within RECIPE we aim at extending and improving MAMUT in order to enable flexible and scalable multi-objective multi-agent power, performance, thermal and reliability-aware resource management.

5.1 MAMUT: A Multi-Agent Machine Learning System for Performance- and Power-Aware Run-time Management

MAMUT is a multi-agent Q-Learning (QL)-based run-time management strategy for latency-constrained applications running in a multi-user multi-application environment.

In Multi-Agent Learning (MAL), multiple agents need to interact and behave cooperatively or competitively with some degree of autonomy. The problem domain may be decomposed into smaller sub-problems and each agent takes charge of one of them, while communicating and interacting with other agents. As a result of such *cooperative* and *concurrent* learning, if complexities arising from interactions between agents are managed well, cooperative multi-agent learning is a promising solution to explore larger design spaces with less computational complexity, leading to a faster learning phase compared to mono-agent learning.

In this work, we propose a concurrent cooperative multi-agent approach for run-time adaptation of CPU and memory-intensive applications, by changing the application configuration and system parameters. The environment is composed of two parts: application (one or more of the RECIPE UCs) and platform (i.e., servers). The action set $\boldsymbol{A}$ is split to three subsets A_1, A_2, A_3 such that: $\forall i \neq j$, $A_i \cap A_j = \emptyset$, and $\bigcup_{i=1}^{3} A_i = \boldsymbol{A}$. Agents can send messages such that each

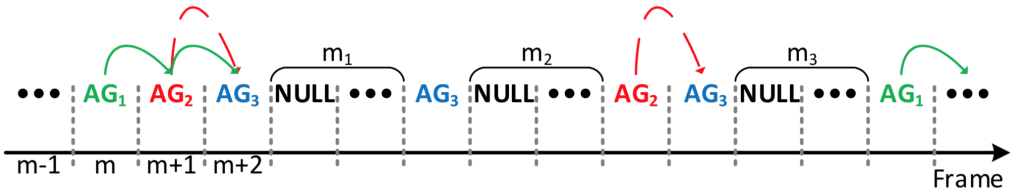


Figure 15: Agent sequence. Colored arrows show which agents need to look at the Q-table of the next agent.

agent accesses the Q-table of the others. The agents are equipped with the utility of passing messages such that each agent accesses the Q-table of the others. In addition, states and rewards resulting from one agent's action are observable to all agents.

In MAMUT, we consider three different agents:

- AG_1 for tuning one application configuration parameter,
- AG_2 for deciding the number of threads,
- AG_3 to set per-core DVFS.

Each of the agents takes care of one action. Furthermore, in a MAL approach we need to experimentally determine how frequently each agent should act, based on overhead, impact on our target objectives, and the number of parameter values to be explored as it is desirable that all agents finish the exploration phase at the same time, as shown in Figure 15.

5.2 Learning phases: Exploration and exploitation

Since each agent has its own action set, we let the agents explore only state-action pairs corresponding to their own actions. As we need to deal with a stochastic environment, applying action a_t^i by AG_i at state s_t may not always result in a particular s_{t+1}. The reason lies in the fact that (1) application input varies through time, changing behaviour, (2) other agents may apply an action that alters the next expected state to a different one, and (3) other applications existing in a multi-user multi-application platform with their corresponding agents can change the state unexpectedly. Thus, once a_t^i is taken at state s_t, all state transitions to new states need to be recorded during the exploration phase. Assume that $Num(s_t \xrightarrow{a_t^i} s_{t+1})$ shows number of times that applying a_t^i at s_t resulted in s_{t+1}, and $Num(s_t, a_t^i)$ represents total number of times that a_t^i was taken at state s_t. Then, the probability by which, after taking a_t^i at s_t, the agent observes s_{t+1} is $P(s_t \xrightarrow{a_t^i} s_{t+1}) = Num(s_t \xrightarrow{a_t^i} s_{t+1})/Num(s_t, a_t^i)$. This probability is updated throughout the learning process.

When the learning rate for each state-action pair drops below a threshold, α_{th1}, the agents start exploration-exploitation for that particular state. In this phase, agents do not take random actions, though after applying this particular action the Q-table is updated.

The exploitation phase starts when the learning rate drops below a threshold, α_{th2}. Entering the exploitation phase, however, does not mean that exploration is finished. In fact, whenever a new

state is observed by one agent, exploration phase starts for this particular state.

Although each agent learns separately and has its own Q-table, it needs to act in the exploitation phase cooperatively and in sequence. Consequently, the goal of each agent is not simply maximising the Q-value attainable for its own Q-table, but rather, maximising the expected Q-value after a sequence of actions taken by all agents. Imagine the sequence of agents shown as in Figure 15. Starting from the m^{th} frame, the first agent, AG_1, is followed by two different agents, AG_2 and AG_3. Thus, the action taken by AG_1 should consider the probable transitions from one state to the other throughout the entire chain, composed of these three agents, in order to maximize the Q-value. Indeed, AG_1 should select an action which ultimately moves the entire system to a state in which an action taken by AG_3 is capable of providing the highest Q-value. This is equivalent to considering the expected Q-value given that a particular action is selected by AG_1. Hence, the conditional expected Q-values should be computed for all available actions in the current state s_t, in the chain of $AG_1 \rightarrow AG_2 \rightarrow AG_3$.

One agent may move to the exploitation phase earlier than the others since the number of actions that belongs to each agent is different. In such a case, the first agent in the sequence cannot rely on the behaviour of the following agents. Hence, it only follows its own Q-table regardless of the expected Q-value that is achievable at the end of the sequence. This behaviour is not optimal as the whole system is not in the exploitation phase yet, and should be prevented.

5.3 Status and next steps in the development of MAMUT

MAMUT was initially developed by the end of the MANGO project to support the transcoding video application [4]. The agents were tailored to control the quality of the video (application parameter), the parallelization via threads and per-core DVFS. MAMUT exhibited better results both in energy consumption (up to 24% when compared with a heuristic approach, and 7% compared with a mono-agent approach) and less performance violations (up to 8x and 5x, respectively), while satisfying restrictions in power. In other words, MAMUT was able to simultaneously and transparently improve performance and power with no user intervention in real time scenarios, where multiple transcoding requests have to be served simultaneously.

The original MAMUT has been generalized for a wider range of applications, enabling the application configuration agent to tackle . The current version of MAMUT supports multi-agent reinforcement learning policies able to tackle two objectives at the same time: power and performance. For RECIPE, we aim to extend MAMUT to enable both reliability and thermal-awareness.

The next steps and MAMUT extensions envisioned are:

- Extension of MAMUT multi-objective policies from power and performance awareness to include also temperature and proactive reliability.
- Extension of MAMUT to work with multiple different applications, which might have different configuration knobs.
- Extension of MAMUT to tackle heterogeneous computing resources.
- Integration of the MAMUT infrastructure into the GRM.

6 Conclusions

In this deliverable, we have introduced the prototype version of the Global Resource Manager (GRM) infrastructure of the RECIPE project, which comprises the GRM software architecture and implementation, and the policies for predictive reliability management, and power/performance/thermal awareness.

We described the basic software architecture of the GRM, and how it interacts with SLURM. We have shown how dockers are used to enable isolation of the SLURM controller, while native *slurmd* daemons enable performant resource allocation. We have shown how the GRM is currently running in one of the setups of the RECIPE platform at UPV. The GRM is responsible for dispatching workload over multiple nodes, and cooperates with BarbequeRTRM, delegating into the local resource manager the runtime dynamic management of tasks allocated to a node.

Regarding proactive reliability, we have provided policies based on the concept of *lifetime deposit*, that enable us to manage the long-term temperature reliability of processors of the RECIPE platform. This is based upon the thermal modelling results of WP3, Task 3.3. Furthermore, we have also proposed multi-objective multi-agent power, performance, thermal and reliability-aware resource management policies. These policies are based on Reinforcement Learning (RL) techniques to enable scalability and flexibility.

Future developments within this task regarding the implementation of the GRM will enhance the monitoring of the platform to incorporate reactive reliability information provided by BarbequeRTRM, improving the communication between local and global RMs. From the reliability perspective, we will expand the models to more complex ones, including further components of the server, and we will incorporate reliability on the multi-objective strategies, while integrating them in the GRM of RECIPE.

References

[1] Jeonghwan Choi, Chen-Yong Cher, Hubertus Franke, Henrdrik Hamann, Alan Weger, and Pradip Bose. Thermal-aware task scheduling at the system software level. In *Proceedings of the 2007 International Symposium on Low Power Electronics and Design*, ISLPED '07, pages 213–218, New York, NY, USA, 2007. ACM.

[2] A. K. Coskun, D. Atienza, T. Simunic Rosing, T. Brunschwiler, and B. Michel. Energy-efficient variable-flow liquid cooling in 3d stacked architectures. In *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, pages 111–116, March 2010.

[3] Ayse Kivilcim Coskun, Tajana Simunic Rosing, and Kenny C. Gross. Temperature management in multiprocessor socs using online learning. In *Proceedings of the 45th Annual Design Automation Conference*, DAC '08, pages 890–893, New York, NY, USA, 2008. ACM.

[4] L. Costero, A. Iranfar, M. Zapater, F. D. Igual, K. Olcoz, and D. Atienza. Mamut: Multi-agent reinforcement learning for efficient real-time multi-user video transcoding. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 558–563, March 2019.

[5] Python documentation. Python mutual exclusion support. https://docs.python.org/2/library/mutex.html.

[6] S.D. Downing and D.F. Socie. Simple rainflow counting algorithms. *International Journal of Fatigue*, 4(1):31 – 40, 1982.

[7] Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using networkx. 01 2008.

[8] A. Iranfar, M. Kamal, A. Afzali-Kusha, M. Pedram, and D. Atienza. Thespot: Thermal stress-aware power and temperature management for multiprocessor systems-on-chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(8):1532–1545, Aug 2018.

[9] Clemens J.M. Lasance. Thermally driven reliability issues in microelectronic systems: status-quo and challenges. *Microelectronics Reliability*, 43(12):1969 – 1974, 2003.

[10] Arvind Sridhar, Alessandro Vincenzi, David Atienza, and Thomas Brunschwiler. 3d-ice: A compact thermal model for early-stage design of liquid-cooled ics. *IEEE Transactions on Computers*, 63(10):2576–2589, 2014.

[11] J. K. Matt Stansberry. Uptime institute 2013 data center industry survey, 2013.

[12] H. Wang, D. Huang, R. Liu, C. Zhang, H. Tang, and Y. Yuan. Stream: Stress and thermal aware reliability management for 3-d ics. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(11):2058–2071, Nov 2019.

[13] Yun Xiang, Thidapat Chantem, Robert P. Dick, X. Sharon Hu, and Li Shang. System-level reliability modeling for mpsocs. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES/ISSS '10, pages 297–306, New York, NY, USA, 2010. ACM.

[14] J. Yang, X. Zhou, M. Chrobak, Y. Zhang, and L. Jin. Dynamic thermal management through task scheduling. In *ISPASS 2008 - IEEE International Symposium on Performance Analysis of Systems and software*, pages 191–201, April 2008.