H2020-JTI-EuroHPC-2019-1

REGALE: An open architecture to equip next generation HPC applications with exascale capabilities

Grant Agreement Number: 956560

D2.3

Final integration of sophisticated policies in the REGALE prototype

Final

Version: 1.0

Author(s): Pierre-François Dutot, Francesco Antici, Eishi Arima, Andrea Bartolini, Yiannis Georgiou, Mohsen Seyedkazemi Ardebili, Mathieu Stoffel and Nikolaos Triantafyllis

Contributor(s): Lluis Alonso

Date: 12.04.2024

Project and Deliverable Information Sheet

REGALE Project	**Project Ref. №:** 956560	
	Project Title: REGALE	
	Project Web Site: https://regale-project.eu	
	Deliverable ID: D2.3	
	Deliverable Nature: Software	
	Dissemination Level: PU *	**Contractual Date of Delivery:** 31 / 03 / 2024
		Actual Date of Delivery: 12 / 04 / 2024
	EC Project Officer: Evangelos Floros	

* - The dissemination levels are indicated as follows: PU = Public, fully open, e.g. web; CO = Confidential, restricted under conditions set out in Model Grant Agreement; CI = Classified, information as referred to in Commission Decision 2001/844/EC.

Document Control Sheet

Document	**Title:** Final integration of sophisticated policies in the REGALE prototype	
	ID: D2.3	
	Version: 1.0	**Status:** Final
	Available at: https://regale-project.eu	
	Software Tool: Google Docs	
	File(s): REGALE_Deliverable_2.3.pdf	
Authorship	**Written by:**	Pierre-François Dutot, Francesco Antici, Eishi Arima, Andrea Bartolini, Yiannis Georgiou, Mohsen Seyedkazemi Ardebili, Mathieu Stoffel and Nikolaos Triantafyllis
	Contributors:	Lluis Alonso
	Reviewed by:	Lluis Alonso, Yiannis Georgiou
	Approved by:	Georgios Goumas

Document Status Sheet

Version	Date	Status	Comments
0.1	08.11.2023	Draft	Initial version
0.2	27.03.2024	Draft	Internal review
1.0	12.04.2024	Final	Corrections

Document Keywords

Keywords:	REGALE, HPC, Exascale, Sophistications, Performance, Energy Efficiency, Power Constraints

Table of Contents

Executive Summary

This final report details the REGALE project's achievements for optimised energy aware resource utilisation in HPC systems. The sophistications are designed with the project's three key objectives in mind: effective resource utilisation, broad applicability, and user-friendly supercomputing services.

Work Package 2 (WP2) "***Sophisticated resource allocation and management***" is the key focus of this report. WP2 subtasks explored various aspects, which are presented in this report with three main objectives, improving **performance and throughput**, providing **energy savings**, and achieving maximum efficiency **under power constraints**.

This final report provides a valuable resource for those interested in the REGALE project's achievements and how they contribute to improved resource allocation and management for High-Performance Computing systems.

List of Abbreviations and Acronyms

Abbreviation / Acronym	Meaning
AAPC	Application Aware Power Capping
CPU	Central Processing Unit
DVFS	Dynamic voltage and frequency scaling
EPI	European Processor Initiative
FW	Firmware
GPU	Graphical Processing Unit
HLC	High Level Controller
HPC	High Performance Computing
HW	Hardware
I/O	Input/Output
LLC	Low Level Controller
ML	Machine Learning
MPI	Message Passing Interface
OS	Operating System
OSPM	Operating System Power Management
PMI	Power Management Interface
RJMS	Resource and Job Management System
SCMI	System Control and Management Interface
SPM	System Power Manager
SW	Software

BDPO, BEO, EAR, OAR, Ryax are software names and not acronyms.

1 Introduction

The REGALE project aims to establish an open architecture, propose a prototype software stack, and integrate advancements into various components of the stack to optimize resource utilization. This final report details the project's improvements to its different components throughout its lifecycle.

The report outlines the sophistications developed in relation to the project's three strategic objectives, which can be simply summarized here:

- Strategic Objective 1 (SO1): **Effective Resource Utilization**
    - Improved application performance
    - Increased system throughput
    - Minimized performance degradation under power constraints
    - Decreased energy to solution
- Strategic Objective 2 (SO2): **Broad Applicability**
- Strategic Objective 3 (SO3): **Easy and Flexible Use of Supercomputing Services**

For more details on these objectives and for a description of the Regale Architecture, Prototype and Use Cases, readers are invited to read Deliverable 1.3. This report presents the progress made in Work Package 2 (WP2), which concentrates on sophisticated resource allocation and management. The various subtasks within WP2 address distinct aspects of resource allocation, including co-scheduling, moldability, and power management.

As this deliverable is meant for the general public, we will not present a simple collection of tasks, but rather structure the results along three main topics, in connection to the strategic objectives. The first of these topics, detailed in Section 2, is the improvement of application performances, and global throughput (this can be viewed as globally improving the number of jobs per second). The different subsections are different ways to improve this objective.

First, we explore *co-scheduling for enhanced throughput and application coupling at the node level*. In this, a machine learning-based approach is used to optimize resource allocation in HPC systems by co-scheduling multiple applications with different resource requirements on the same compute nodes. The method also involves determining the most suitable allocation policy (e.g. compact or spread allocation) for each application.

Then, we look at *co-scheduling within multicore processing units*. This subsection details a novel co-scheduling strategy that incorporates machine learning for automated resource allocation in heterogeneous HPC systems that include CPUs and GPUs. The aim is to assign resources based on specific job requirements, balancing individual job needs with overall system efficiency.

The next approach is *elastic resource management* with BeBiDa. This subsection explores BeBiDa, a system designed to enable elastic resource management for HPC systems, particularly beneficial for Big Data workloads with dynamic resource requirements. The core concept of BeBiDa revolves around leveraging the prolog/epilog mechanisms of HPC resource managers. This allows for dynamic integration and removal of HPC nodes from a Kubernetes cluster. When an HPC job is submitted, the HPC node it occupies is detached from the Kubernetes cluster, effectively making it unavailable for Big Data jobs. Once the HPC job completes, the node is seamlessly re-integrated into the Kubernetes cluster, enabling Big Data jobs to utilize its resources once again. In this context, specific contributions provided new optimization techniques to guarantee the successful termination of Big Data jobs in a timely manner.

Finally, *data-aware resource allocation* has been investigated and implemented as a hybrid workflow scheduling. The approach for handling hybrid workflows utilizes traditional HPC job submission queues, alongside additional resources requested in a flexible "best-effort" queue. This allows an improved QoS for workflow by allocating dedicated resources, while using all idle resources whenever possible.

In section 3, energy savings are considered. More broadly this can be seen as an improvement in jobs per watt. First we look at *moldability for energy efficiency*: This research investigates moldability as a tool for improving energy efficiency in HPC systems. Moldability empowers resource managers to consider multiple configurations for a single job submission, enabling exploration of energy-efficient resource allocation options. An initial integration of energy-aware moldability policies in a resource management system is proposed. This integration utilizes a score function to evaluate configurations and select the one with the most favorable energy efficiency.

A novel approach for real-time power management in HPC data centers is then presented. This approach leverages machine learning models to predict job power consumption, eliminating the need for complex job power characterization. The proposed technique utilizes user data such as username, job name, and requested hardware specifications to predict power consumption.

The report then details the integration of machine learning models into a comprehensive production system for HPC data centers. This system incorporates a multi-part framework with a monitoring subsystem that gathers sensor data to fuel the ML operation subsystem. Here, Docker containers and Kubernetes orchestrate the automated deployment, training, and utilization of the ML models. The trained models predict power consumption, enabling real-time optimization and resource management.

At the node level, optimizing GPU energy efficiency with BDPO integration is studied, exploring the integration of a runtime power management tool, BDPO, with GPUs for improved energy efficiency during HPC application execution. Similarly for CPUs, the utilization of on-chip controllers for thermal and power management within the REGALE prototype is detailed.

Section 4 investigates two techniques for managing power consumption in High-Performance Computing (HPC) systems under power constraints (often called power capping) while minimizing performance impact.

The first approach tackles power-capped situations by complementing the traditional First-Come-First-Served (FCFS) scheduling. It empowers users to designate jobs as EcoJobs, indicating their tolerance for running slower under power limitations. When power limitations are imposed, the resource manager prioritizes slowing down EcoJobs using Dynamic Voltage and Frequency Scaling (DVFS) techniques. This will reduce the power consumption of jobs under power caps, significantly reducing the number of jobs killed due to power limitations.

The second technique, Application-Aware Power Capping (AAPC), automatically addresses performance degradation caused by traditional power capping, which can significantly slow down compute-bound tasks. AAPC dynamically adjusts power budgets based on measured application characteristics like compute or memory intensiveness. By prioritizing compute-bound jobs (more susceptible to slowdowns) with efficient power allocations, AAPC aims to improve overall job throughput. The system integrates seamlessly with existing HPC software to leverage job information and manage power capping on individual compute nodes.

The power budget allocation algorithm prioritizes full power for memory-bound jobs, followed by mixed jobs. Compute-bound jobs have their power reduced only as a last resort. The study also addresses challenges associated with enforcing power caps due to delays between setting a cap and its actual application on compute nodes. To mitigate these delays, AAPC prioritizes stricter caps first and ensures enforcement only after reducing core frequencies.

Finally, in Section 5 we explore the relations between all the proposed sophistications and their position in the REGALE Architecture detailed in Deliverable 1.3.

While the presentation focuses on the measurable improvements, a strong commitment to applicability and ease of use as defined in strategic objectives 2 and 3 was always present during the project. As a result most of the processes detailed in this report are fully transparent to the users.

Note to the reviewers: As this report is reporting on work spanning the whole project duration, some parts were already complete for the intermediate report D2.2. Specifically, Subsection 2.4 is almost identical to Subsection 2.4 in D2.2, and Subsection 3.4 has a few common pages with Subsection 2.3.B in the same D2.2 report. Quantitative and qualitative evaluations of the sophistications are presented in Deliverable 1.4.

2 Performance and Throughput

2.1 Co-scheduling for throughput and application coupling at the node level

Motivation and problem definition

As we move towards exascale computing, the demands on process management in terms of flexibility and efficiency increase. Running multiple applications concurrently on a single node presents a potential solution to enhance resource utilization[1,2]. Moreover, today's processor manufacturers boost performance by increasing the number of cores within each CPU. However, not all high-performance computing (HPC) applications can fully leverage all cores on a single node, even when they scale across thousands of nodes. In such cases, resource sharing between applications on nodes can effectively distribute the load across various resources, improving overall utilization[3].

The typical and most straightforward approach to resource management in modern supercomputing systems is to allocate full compute nodes to the applications. This means that a job requesting *x* processes from a system with *y* cores per node will receive $\lceil x/y \rceil$ nodes to execute. While this scheme effectively ensures that tasks from diverse users do not negatively impact each other's performance, and additionally it is easy to implement, it comes at a substantial expense to system throughput and energy efficiency[4]. This is notably evident in the process of allocating a memory-bound application, which may face significant scalability problems. Quite importantly, several HPC applications are reported to have low operational intensity and thus are bound by the limited access to main memory. This family of applications would benefit from a resource allocation scheme that would provide more memory bandwidth, thus more memory links, i.e. in the concepts of resource management being spread in more nodes. Another study[5] emphasizes the redistribution of memory-bound applications across multiple nodes to alleviate performance bottlenecks and the co-location of jobs in a manner compatible with available resources, while another study[6] evaluates the co-allocation policy scheme in a simulated environment, indicating that co-scheduling has the potential to be a more efficient way to schedule jobs on high-end machines in both turnaround time and system and component utilization. On the other hand[7], applications that spend a significant part of their execution time in communication may be sensitive to

[1] Trinitis, C., and J. Weidendorfer. "Allocation-Internal Co-Scheduling." Co-Scheduling of HPC Applications 28 (2017): 46.
[2] Brinkmann, André. "Co-scheduling: prospects and challenges." Co-Scheduling of HPC Applications 28.1 (2017): 20.
[3] Frank, Alvaro, Tim Süß, and André Brinkmann. "Effects and benefits of node sharing strategies in hpc batch systems." 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE, 2019.
[4] Breslow, Alex D., et al. "The case for colocation of high performance computing workloads." Concurrency and Computation: Practice and Experience 28.2 (2016): 232-251.
[5] Tang, Xiongchao, et al. "Spread-n-share: improving application performance and cluster throughput with resource-aware job placement." Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. 2019.
[6] Hall, Jason, et al. "Evaluating the Potential of Coscheduling on High-Performance Computing Systems." Workshop on Job Scheduling Strategies for Parallel Processing. Cham: Springer Nature Switzerland, 2023.
[7] Bhatele, Abhinav, et al. "There goes the neighborhood: performance degradation due to nearby jobs." Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. 2013.

the number of nodes that they are allocated to, probably preferring to be mapped on a minimum number of nodes (although this may depend on the communication pattern and the underlying network setup). Finally, there exist applications that present no significant variation in their execution time due to different allocations in nodes. Figure 1 illustrates the recommended resource allocation strategy for the three application categories, specifically those that favor “spread”, “compact”, or “no preference”.

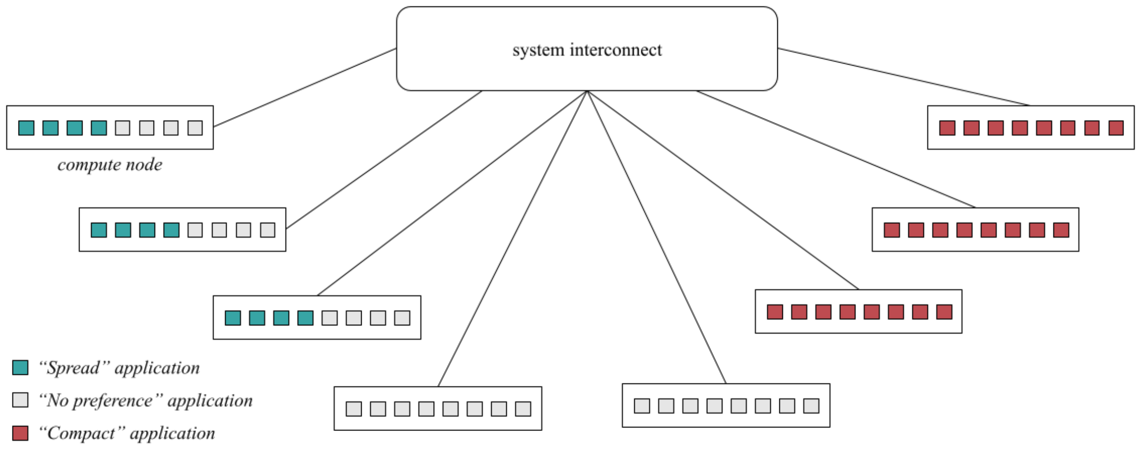


Figure 1: *Application categories based on resource allocation policy. In the “compact” category the application’s processes are ideally packed together. In the "Spread" category the application’s processes are assigned to half of the available cores in each CPU socket. In the “No preference” category the application’s processes are assigned in a best-effort concept across all available cores.*

Assumptions for a solution

The proposed approach revolves around a multi-step process aimed at optimizing resource allocation policies’ assignments for high-performance computing (HPC) systems. The solution assumes that the allocation preference of the application is known to the system. This can be achieved in two ways, either the user explicitly requests for an explicit scheme, or the system is able to collect information from prior, historical runs of the application, and is able to associate previous runs with the forthcoming one. For this second case, the strategy entails the collection of performance counters from actual HPC benchmarks (targeted to implementations under the MPI standard) operating within real-world HPC systems. These performance counters serve as crucial data inputs for the subsequent step, which involves training a predictive model. This model leverages the collected performance counter data to make informed predictions about the most suitable allocation policy for new, incoming applications with known performance counters. In cases where no pre-existing data is available for a specific application, a generic allocation scheme is applied.

Determination of the preferred resource allocation policy

Resource allocation policies ensure that computational resources are assigned optimally, maximizing the performance of HPC systems while minimizing operational expenses. By fine-tuning the allocation of resources, HPC facilities can reduce both power consumption and the associated cooling costs. This not only makes HPC operations more cost-effective but also aligns with the broader goals of sustainability in the face of escalating energy demands. Table 1 serves as an informative reference, offering comprehensive descriptions of the various resource allocation policies, thus providing a detailed understanding of how resource distribution is managed.

Table 1: Resource allocation policies definition.

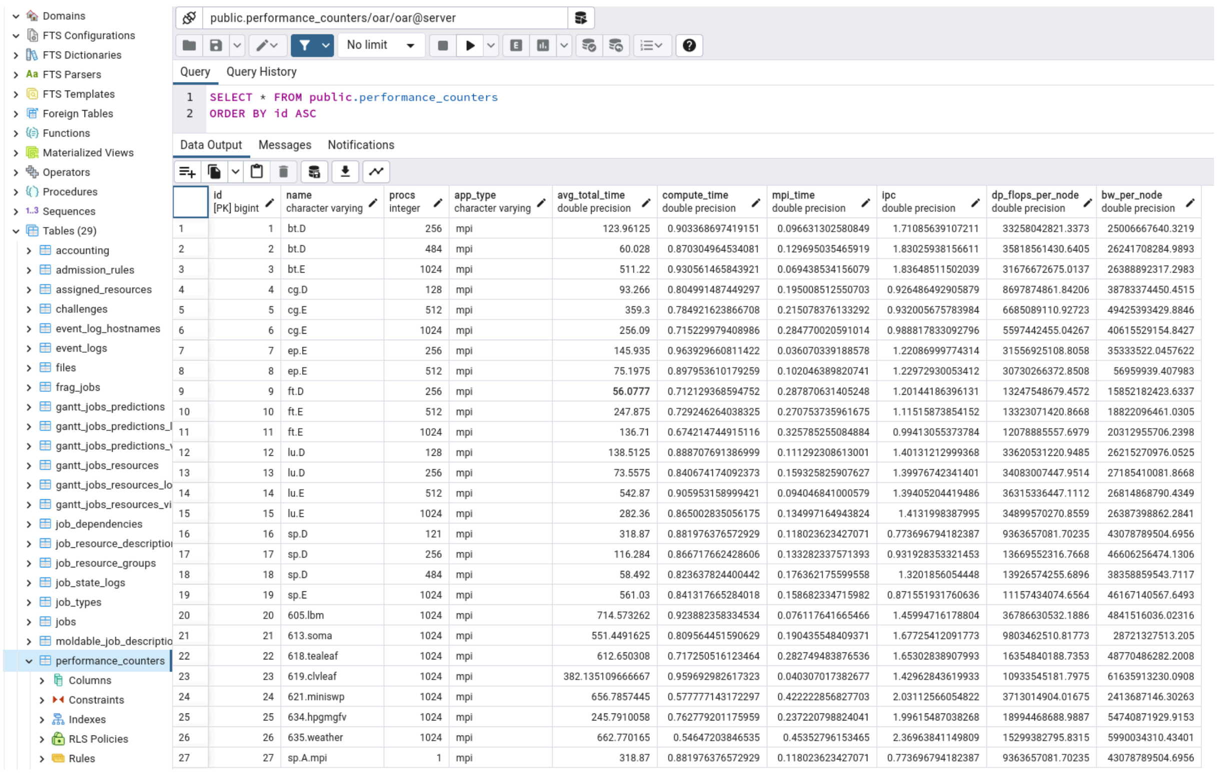
No Preference	The allocation strategy is designed to first fill non-empty nodes, prioritizing those with the fewest available cores, and it only resorts to occupying empty nodes when no other options are available. This approach may lead to a combination of both occupied and unoccupied nodes, ensuring hardware-independent compatibility across heterogeneous clusters. This policy applies to jobs with no significant performance dropoff (e.g. compute-intensive applications).
Compact	This policy ideally packs processes together and fills up the minimum required number of nodes, creating the least possible node fragmentation. This intends to improve the performance of applications that invoke a large number of message exchanges. The allocation policy places emphasis on occupying initially vacant nodes, with a primary focus on nodes that reduce fragmentation by offering the maximum available cores. Only after considering empty nodes does it proceed to allocate resources to non-empty nodes with the priority to those that further minimize fragmentation. This strategy can result in a combination of both occupied and unoccupied nodes, making it versatile for use across hardware-independent, heterogeneous clusters. It stands in contrast to the "No Preference" allocation policy by explicitly favoring nodes that reduce fragmentation.
Spread	This policy allocates only half of the available cores in each CPU. The subsequent larger number of used nodes (i.e. memory links) provides a higher memory bandwidth for the application. This sparse allocation scheme is suitable for applications with higher memory demands. This variant of the allocation policy mirrors the “compact” approach but incorporates a new constraint -each step accounts for only half of a CPU socket. In cases where the requested cores are fewer than half the cores available in a CPU socket, the policy avoids spreading them across multiple sockets. Note that as the number of cores in a typical HPC node rises, we expect that this policy can be further enhanced to assume other allocation portions (most probably, quarters, eights, etc). For the sake of simplicity, in our current solution we work with half CPU allocation and leave the other options for future work.

Performance counters and MPI time collection

Performance Counters (PCs) are a set of hardware-based or software-based metrics and counters that provide detailed information about the performance and behavior of a computer system, particularly during the execution of applications and workloads. These counters monitor various aspects of system performance, including CPU utilization, memory usage, I/O operations, cache hits, network traffic, and more. PCs are crucial tools for understanding and optimizing the performance of HPC applications and systems. Additionally, attributes allocated to MPI (Message Passing Interface) operations -which are essential for communication and coordination between parallel processes within the application's environment- can indicate a great impact on the overall performance of parallel applications. Due to their significant importance both metrics from PC and MPI Attributes are collected. The harvesting is systematically performed through the use of "perf" (a well-known and powerful performance analysis and profiling tool in Linux systems) and "mpiP" (a profiling tool designed for the MPI standard) utilities during runs under the "compact" resource allocation policy (default policy). The resulting data is conveniently organized and stored in CSV format file. The entire data collection and storage process is automated, sparing users from the intricacies involved. Users can effortlessly leverage these automated scripts by including the required file in their submission batch script, seamlessly integrating data collection into their workflow. These collected metrics serve as vital inputs for machine learning models, enabling data-driven insights and optimization in performance analysis. The metrics that are currently collected and used in the ML model training are the following:

- avg_total_time: Represents the average total time of the execution of a given application.
- compute_time: Signifies the time spent on computational tasks.
- mpi_time: Indicates the time allocated to MPI (Message Passing Interface) operations.
- ipc: Refers to the instructions per cycle, a measure of CPU efficiency.
- dp_flops_per_node: Represents the double-precision floating-point operations per second per node.
- bw_per_node: Denotes the bandwidth per node, typically a measure of data transfer rate.

In our OAR3 implementation, the PostgreSQL table -shown in Figure 2- includes a column named "type" that signifies the specific parallel standard used during application execution and the corresponding metrics gathered. This functionality is designed for future flexibility, allowing the referencing of metrics for alternative parallel standards, such as OpenMP or hybrid OpenMP-MPI configurations. In this example, the type of applications in the table is "*mpi*".



```
SELECT * FROM public.performance_counters
ORDER BY id ASC
```

	id [PK] bigint	name character varying	procs integer	app_type character varying	avg_total_time double precision	compute_time double precision	mpi_time double precision	ipc double precision	dp_flops_per_node double precision	bw_per_node double precision
1	1	bt.D	256	mpi	123.96125	0.903368697419151	0.096631302580849	1.71085639107211	33258042821.3373	25006667640.3219
2	2	bt.D	484	mpi	60.028	0.870304964534081	0.129695035465919	1.83025938156611	35818561430.6405	26241708284.9893
3	3	bt.E	1024	mpi	511.22	0.930561465843921	0.069438534156079	1.83648511502039	31676672675.0137	26388892317.2983
4	4	cg.D	128	mpi	93.266	0.804991487449297	0.195008512550703	0.926486492905879	8697874861.84206	38783374450.4515
5	5	cg.E	512	mpi	359.3	0.784921623866708	0.215078376133292	0.932005675783984	6685089110.92723	49425393429.8846
6	6	cg.E	1024	mpi	256.09	0.715229979408986	0.284770020591014	0.988817833092796	5597442455.04267	40615529154.8427
7	7	ep.E	256	mpi	145.935	0.963929660811422	0.036070339188578	1.22086999774314	31556925108.8058	35333522.0457622
8	8	ep.E	512	mpi	75.1975	0.897953610179259	0.102046389820741	1.22972930053412	30730266372.8508	56959939.407983
9	9	ft.D	256	mpi	56.0777	0.712129368594752	0.287870631405248	1.20144186396131	13247548679.4572	15852182423.6337
10	10	ft.E	512	mpi	247.875	0.729246264038325	0.270753735961675	1.11515873854152	13323071420.8668	18822096461.0305
11	11	ft.E	1024	mpi	136.71	0.674214744915116	0.325785255084884	0.99413055373784	12078885557.6979	20312955706.2398
12	12	lu.D	128	mpi	138.5125	0.888707691386999	0.111292308613001	1.40131212999368	33620531220.9485	26215270976.0525
13	13	lu.D	256	mpi	73.5575	0.840674174092373	0.159325825907627	1.39976742341401	34083007447.9514	27185410081.8668
14	14	lu.E	512	mpi	542.87	0.905953158999421	0.094046841000579	1.39405204419486	36315336447.1112	26814868790.4349
15	15	lu.E	1024	mpi	282.36	0.865002835056175	0.134997164943824	1.4131998387995	34899570270.8559	26387398862.2841
16	16	sp.D	121	mpi	318.87	0.881976376572929	0.118023623427071	0.773696794182387	9363657081.70235	43078789504.6956
17	17	sp.D	256	mpi	116.284	0.866717662428606	0.133282337571393	0.931928353321453	13669552316.7668	46606256474.1306
18	18	sp.D	484	mpi	58.492	0.823637824400442	0.176362175599558	1.3201856054448	13926574255.6896	38358859543.7117
19	19	sp.E	1024	mpi	561.03	0.841317665284018	0.158682334715982	0.871551931760636	11157434074.6564	46167140567.6493
20	20	605.lbm	1024	mpi	714.573262	0.923882358334534	0.076117641665466	1.45994716178804	36786630532.1886	4841516036.02316
21	21	613.soma	1024	mpi	551.4491625	0.809564451590629	0.190435548409371	1.67725412091773	9803462510.81773	28721327513.205
22	22	618.tealeaf	1024	mpi	612.650308	0.717250516123464	0.282749483876536	1.65302838907993	16354840188.7353	48770486282.2008
23	23	619.clvleaf	1024	mpi	382.135109666667	0.959692982617323	0.040307017382677	1.42962843619933	10933545181.7975	61635913230.0908
24	24	621.miniswp	1024	mpi	656.7857445	0.577777143172297	0.422222856827703	2.03112566054822	3713014904.01675	2413687146.30263
25	25	634.hpgmgfv	1024	mpi	245.7910058	0.762779201175959	0.237220798824041	1.99615487038268	18994468688.9887	54740871929.9153
26	26	635.weather	1024	mpi	662.770165	0.54647203846535	0.45352796153465	2.36963841149809	15299382795.8315	5990034310.43401
27	27	sp.A.mpi	1	mpi	318.87	0.881976376572929	0.118023623427071	0.773696794182387	9363657081.70235	43078789504.6956

Figure 2: A view of the PostgreSQL table integrated within the OAR3 database, depicting the collected Performance Counters and MPI time metrics. Data is produced by benchmarks' runs in actual HPC clusters. The implementation has a placeholder code to ingest new data to the table.

Speedup heatmap generation

Co-scheduling, as a technique, entails the placement of pairs of processes from different jobs on common CPU sockets, aiming to enhance system throughput and subsequently greater energy efficiency. Colocating pairs of different jobs on a shared set of nodes within a supercomputer environment was used to generate a Speedup heatmap. A heatmap is a data visualization technique that represents data values as colors in a two-dimensional matrix. Our experiments involve running applications both solo ("compact") and in pairs ("co-scheduled") repeatedly for 10 minutes to generate multiple run logs. Speedup is then calculated as the ratio of the execution time of the "compact" to the execution time of the application in a co-scheduled state. Co-scheduling is performed by applying the "spread" resource allocation scheme for both applications and concurrently co-located them under the same compute nodes subset. For instance, assuming two applications A and B of x MPI processes each. The applications will be submitted on the same compute nodes with the allocation scheme of the "spread" policy where half of the CPU sockets will be filled by each application. This will lead to a full CPU socket capacity allocation, by heterogeneous MPI processes. Concerning the Speedup_{AB} it would be the ratio of the execution time T_A of application A in a "compact" allocation policy to the execution time of application A co-scheduled with application B T_{AB}. A speedup_{AB} of 2 means that application A runs two times faster if it is co-allocated with application B instead of running under the "compact" resource allocation policy.

Speedup heatmaps are continuously populated with data collected from the ARIS supercomputer at GRNET[8], and the MARCONI supercomputer at CINECA[9], as depicted in Figure 3. It illustrates speedup heatmaps corresponding to individual supercomputers, showcasing results for identical pairs of the NAS Parallel Benchmarks[10]. The practical observation suggests that co-scheduling confers advantages to the HPC system, leading to a noteworthy 12% improvement in overall speed for both real-world HPC infrastructures, as opposed to the default allocation policy (“compact”). This performance gain is quite similar between these two HPC infrastructures, and it holds true despite variations in the characteristics of the compute nodes and the interconnections between the nodes of the clusters.

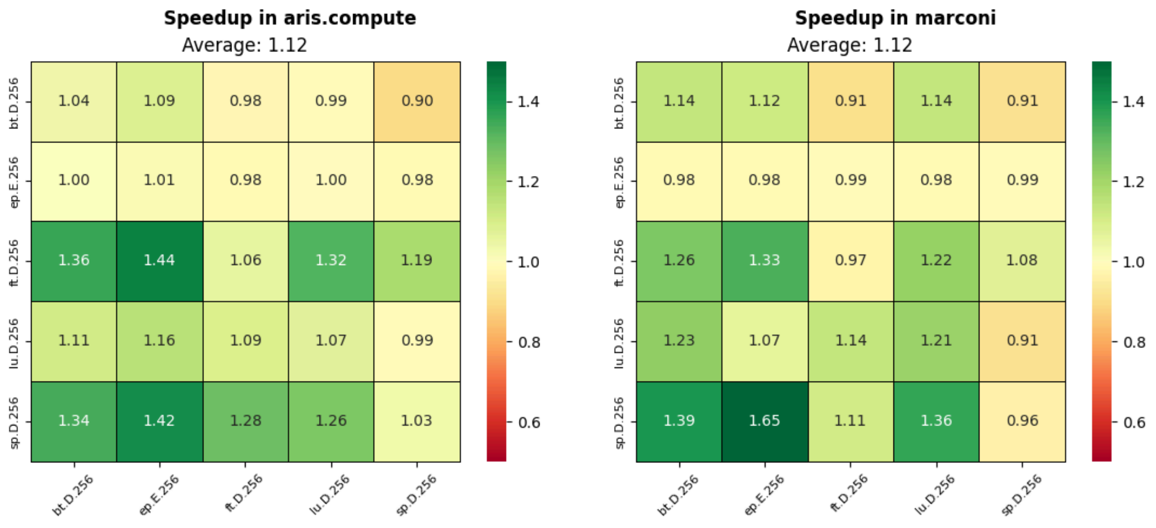


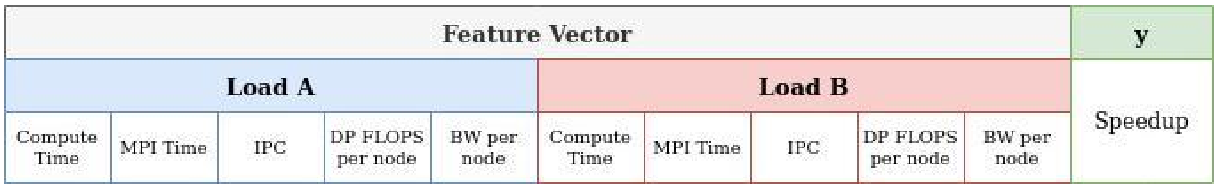
Figure 3: *A view of the Speedup Heatmap per supercomputer for the same benchmarks co-schedulings. The application name underlines the problem that it is solved (e.g. sp), the size of the problem (e.g. D) and the number of MPI processes spawned (e.g. 256). As a significantly performance-gained example the co-scheduling of sp.D.256 and ep.E.256 leads to an average of 54% faster execution of the sp.D.256 benchmark in both HPC infrastructures, without reducing the execution time of the ep.E.256 benchmark as it was run under the “compact” policy.*

Machine Learning model creation

A Machine Learning (ML) model is a computational system that has undergone a training process to learn patterns and relationships within data. During training, the model is exposed to a dataset containing input data and their corresponding outcomes, and it adjusts its internal parameters and algorithms to make accurate predictions or classifications. In our work, in order to build the ML model, a small dataset of performance counters and MPI attributes harvested from runs of the NAS Parallel Benchmarks and the SPEChpc 2021[11] applications under the “compact” allocation policy on the ARIS supercomputer, was used. Each row is a fusion of two application's 6-feature set (avg_total_time, compute_time, mpi_time, ipc, dp_flops_per_node, bw_per_node), forming a “Feature Vector”, and includes

[8] https://hpc.grnet.gr
[9] https://www.hpc.cineca.it
[10] https://www.nas.nasa.gov/software/npb.html
[11] https://www.spec.org/hpc2021

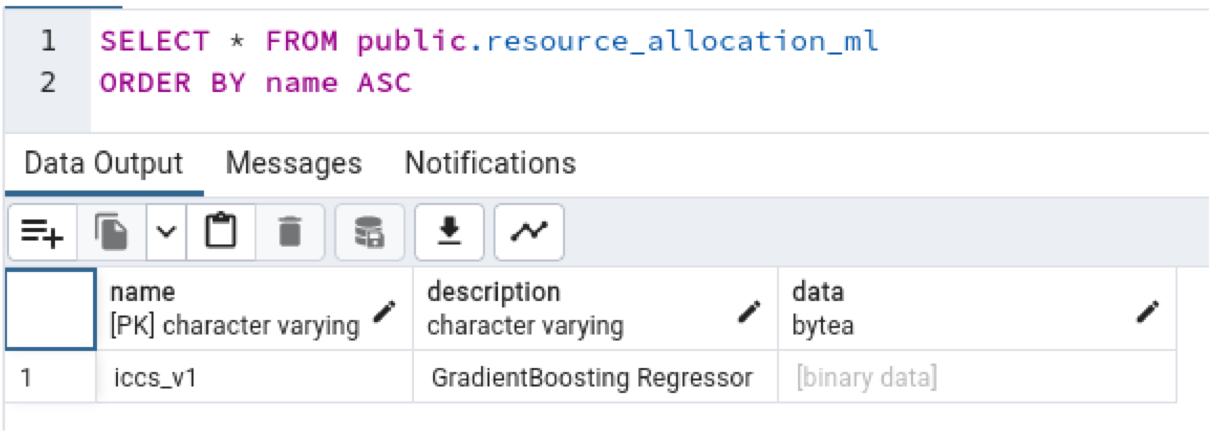
the corresponding co-scheduling speedup derived from the Speedup Heatmap, as shown in Figure 4.



Feature Vector										y
Load A					Load B					Speedup
Compute Time	MPI Time	IPC	DP FLOPS per node	BW per node	Compute Time	MPI Time	IPC	DP FLOPS per node	BW per node	

Figure 4: *A representation of a row of the dataset that is used to train the ML model. The "Feature Vector" was derived from runs under the "compact" allocation strategy, whereas the "y" is the speedup derived from a co-schedule run between the respective applications.*

A variety of linear, non-linear, and ensemble learning regression models from scikit-learn Python library were trained on the dataset. The data was split into 70% for training and 30% for testing. During the training phase, grid search was employed to optimize the model's hyperparameters, combined with a 5-fold cross-validation process. In the testing phase, the performance of these models was assessed and compared using metrics such as RMS and absolute error to evaluate their predictive accuracy. Our OAR3 implementation facilitates the ingestion of diverse models into the database, offering flexibility in choosing how to utilize them, as it is shown in Figure 5. Utilizing various ML models for predictions leverages their individual strengths, leading to improved accuracy and robustness in predictive tasks. It also allows for better adaptation to diverse data and problem scenarios.



```sql
SELECT * FROM public.resource_allocation_ml
ORDER BY name ASC
```

	name [PK] character varying	description character varying	data bytea
1	iccs_v1	GradientBoosting Regressor	[binary data]

Figure 5: *A view of the Machine Learning model as it is stored in the OAR3 PostgreSQL database. The implementation is able to store and load for use multi-purpose models.*

A captivating aspect of the ML model's predictions can be seen in the scenario illustrated in Figure 6. In the leftmost figure, you can observe a prediction generated by the ML model using performance counters and MPI time data ("Feature Vector") collected from the NAS Parallel Benchmarks executed under the "compact" allocation policy on the Marconi supercomputer. The figure on the right displays the real speedup heatmap achieved by co-scheduling these specific NAS Parallel benchmark cases, as previously presented in Figure 5. As previously indicated, the ML model has been trained using data from runs of the NAS Parallel Benchmarks and the SPEChpc 2021 applications conducted under the "compact" allocation policy on the ARIS supercomputer. Another intriguing aspect is that the ML model accurately predicted the same average Speedup, despite having no other information beyond the “Feature Vector” of the potential co-schedule.

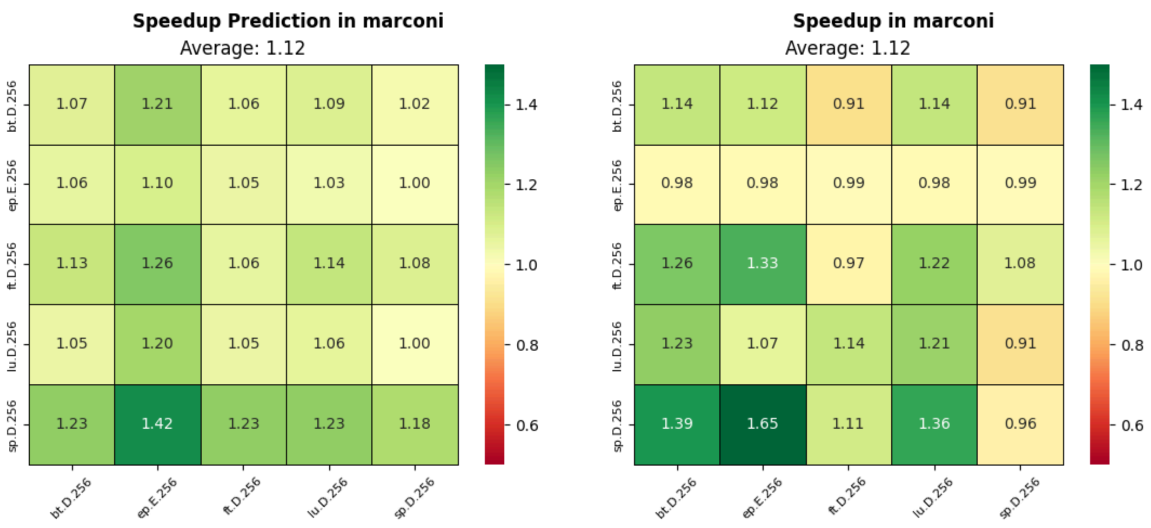


Figure 6: *A view of the predicted Speedup Heatmap vs the actual Speedup Heatmap in Marconi. The prediction is performed by the ML model based on the “Feature Vector” gathered from the NAS Parallel benchmarks on Marconi, under the “compact” policy.*

Integration within OAR3

A Resource and Job Management System (RJMS) is a software stack used in high-performance computing (HPC) to manage and coordinate the allocation of computational resources and the scheduling of jobs submitted by users. A typical workflow in RJMS consists of six core steps. Initially, it begins with job submission, where users provide job details and request to run a job in the cluster. Following, jobs are then queued based on various factors such as priority and fairness, and then resource allocation follows assigning resources to jobs. Afterward, the job is executed, which involves running the task while monitoring progress. Eventually, when the job is completed, the system updates the status and provides the respective results. Finally, in the job cleanup step, the system manages post-job tasks and resource releases.

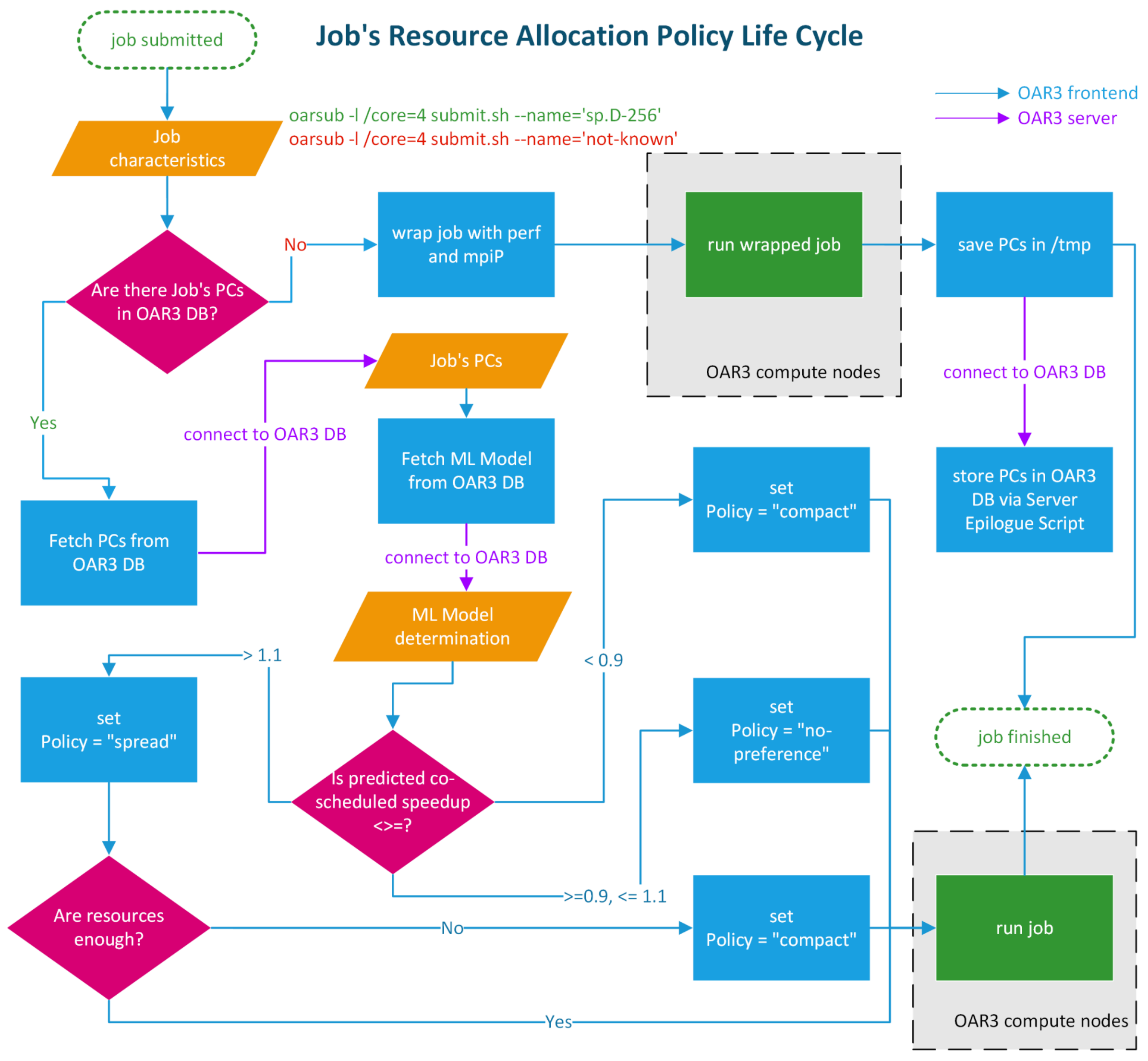

Job's Resource Allocation Policy Life Cycle
job submitted
OAR3 frontend
OAR3 server
oarsub -l /core=4 submit.sh --name='sp.D-256'
oarsub -l /core=4 submit.sh --name='not-known'
Job characteristics
Are there Job's PCs in OAR3 DB?
No
Yes
wrap job with perf and mpiP
run wrapped job
OAR3 compute nodes
save PCs in /tmp
connect to OAR3 DB
store PCs in OAR3 DB via Server Epilogue Script
Job's PCs
Fetch ML Model from OAR3 DB
Fetch PCs from OAR3 DB
ML Model determination
Is predicted co-scheduled speedup <>=?
> 1.1
< 0.9
>=0.9, <= 1.1
set Policy = "spread"
set Policy = "compact"
set Policy = "no-preference"
Are resources enough?
job finished
run job

Figure 7: *Life Cycle of job's resource allocation policy assignment in OAR3 integration. The Admission Rule reads the ML model with the corresponding Performance Counters and MPI time of the application and automatically assigns the policy (i.e. compact, spread, no_preference) based on its prediction design.*

Our work integration within OAR3 is covered in Figure 7, where the process flow of the co-scheduling functionality is visually represented and subsequently subjected to a step-by-step analysis. In the initial phase, a job is submitted by the user with the "oarsub" command. Afterward, an Admission Rule (AR) -an OAR3 mechanism to perform tasks between the user's input and the job's submission- analyzes the job's characteristics and inquires about the availability of Performance Counters (PCs) specific to the given job. In order to fulfil this task, the system establishes a connection with the OAR3 database to ascertain the presence of these counters. If they are available, the system retrieves both the PCs and a Machine Learning (ML) model from the OAR3 database. The PCs are then provided to the ML model, which, based on its predictions, determines the allocation policy for the job. This policy is set automatically by the AR as one of the following options: "compact", "spread", or "no-preference". In the case of the "spread" policy where double the resources are required, verification is conducted to ascertain whether the cluster possesses the capacity to accommodate this allocation. If the cluster lacks the necessary capacity, the allocation policy is converted to “compact” and the subsequent steps adhere to the standard workflow. In instances where PCs cannot be located for the job, the “perf” and the “mpiP” utilities are employed to encapsulate the job command and capture the relevant performance metrics. Following this, the job is submitted, and when it is executed, an epilogue script is executed on the server. This script collects the captured metrics and stores them within the PCs’ table in the OAR3 database for that particular job name. Throughout the entire process, all procedural steps are executed within the OAR3 frontend node. The interactions with the OAR3 database are facilitated through the server node. In contrast, the computational aspects are unequivocally executed on the compute nodes.

Within Table 2, we have encapsulated our contributions, showcasing the corresponding source code alongside comprehensive information pertaining to the associated branches and releases.

Table 2: Software implementation repositories.

Name	**Description**	**Repository**	**Branch**	**Release/ Tag**	**Based on oar-team**
Co-scheduling at the node level	Implementation of new resource allocation policies with auto-assignment using trained ML model	https://github.com/cslab-ntua/oar3	master	3.0.0.dev10	2023/10
Ecosystem of OAR3	All the necessary dependencies, libraries, and OAR3 setup’s characteristics and configuration	https://github.com/cslab-ntua/regale-nixos-compose	main	1.1	2023/10

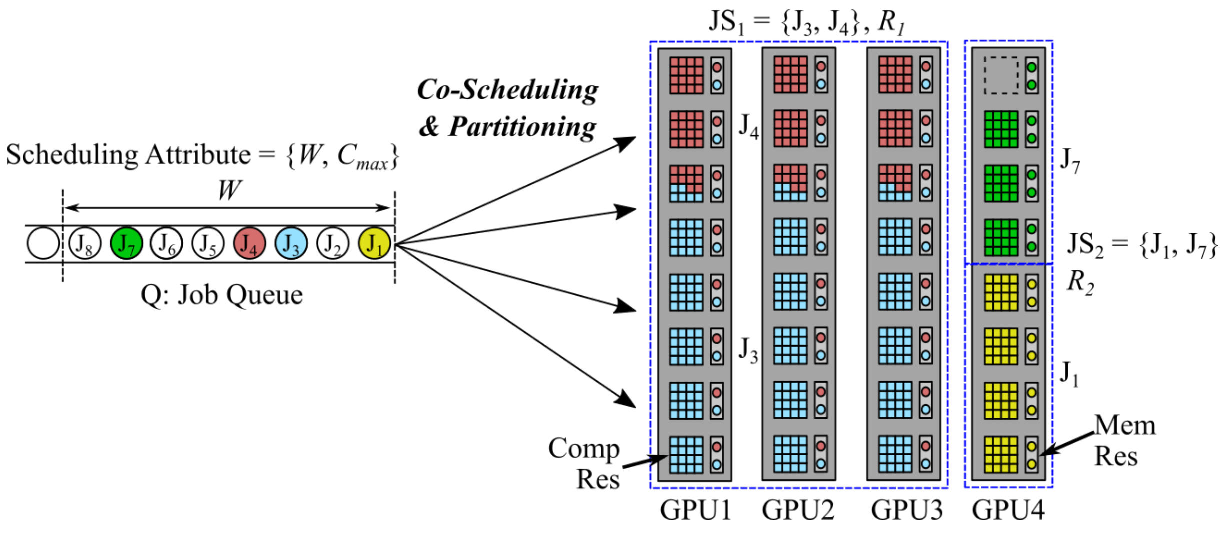
2.2 Co-scheduling within multicore processing units

Motivation

HPC clusters and supercomputers are becoming increasingly heterogeneous, consisting of CPUs and GPUs. In fact, around 190 of top-class supercomputers ranked in the Top500 list are GPU-equipped systems as of Nov 2023. Although exploiting GPU resources is indispensable on such systems, which is because they offer a large fraction of computational throughput and memory bandwidth, it is becoming more and more difficult to fully utilise the entire resources within a GPU chip by one single program. The first reason for this is not all GPU programs have sufficient parallelism to convert the available compute resources inside a GPU into speedup, which is governed by the well-known Amdahl’s law. The second reason is the throughput of memory intensive applications is limited by the available memory bandwidth, and thus increasing the compute resources does not contribute to the speedup for them, which is known as the memory-wall problem. The third reason is the compute resources inside a GPU are also becoming heterogeneous with different types of units (e.g., matrix engines, regular FP64 units, integer units, etc.), and depending on their usages, power can also be underutilised and wasted.

Problem Definition

Figure 8 illustrates our GPU co-scheduling and resource partitioning problem. We target W GPU jobs waiting within the window (size: W) on a given queue and attempt to minimise the total execution time of W different GPU jobs. Instead of executing them one by one, we consider dividing them into several sets of jobs and co-locate each set on the same GPU(s) in a space sharing manner while optimising the resource allocations to the co-located jobs. Here, we assume the concurrency of co-scheduling (or the number of co-located jobs) on each GPU is less than or equal to a given threshold (C_{max}) and the requested numbers of nodes/GPUs are the same for all jobs in a set, namely the same sized GPU jobs can be co-scheduled. For each set of co-scheduling jobs, we optimise the resource partitioning state on the allocated GPU(s), and to this end, we utilise several GPU partitioning features that recent commercial high-end GPUs support (e.g. NVIDIA MIG, NVIDIA MPS, etc.). With the MPS feature, one can partition/assign compute resources to co-located jobs at arbitrary rates in a fine-grained manner, while with the MIG feature, one can completely isolate multiple co-located jobs, resulting in coarse-grained but interference-free co-scheduling.



$JS_1 = \{J_3, J_4\}, R_1$
Co-Scheduling & Partitioning
Scheduling Attribute = $\{W, C_{max}\}$
W
J_8 J_7 J_6 J_5 J_4 J_3 J_2 J_1
Q: Job Queue
J_4
J_3
J_7
$JS_2 = \{J_1, J_7\}$
R_2
J_1
Comp Res
Mem Res
GPU1
GPU2
GPU3
GPU4

Figure 8: GPU co-scheduling and resource partitioning problem

Assumptions for a solution

Figure 9 demonstrates GPU throughput as a function of compute resource allocation to two co-located HPC benchmark programs across different program mixes. For this evaluation we used our testbed system equipped with an NVIDIA A100 GPU with the MPS feature enabled. The X-axis represents the ratios of compute resource allocation to the co-scheduled programs shown at the legend, while the Y-axis indicates the relative throughput normalised to that of a time-sharing scheduling, i.e., executing these two programs one by one without sharing the resources but with fully allocating the entire GPU resources. As illustrated in the figure, the optimal allocation of compute resources to the co-located programs depends highly on the given programs and their characteristics (e.g. compute/memory intensity). As we can observe in the third case, a balanced allocation achieves the best performance, while for the others, a skewed allocation has advantage over a balanced one with a unique optimal allocation point.

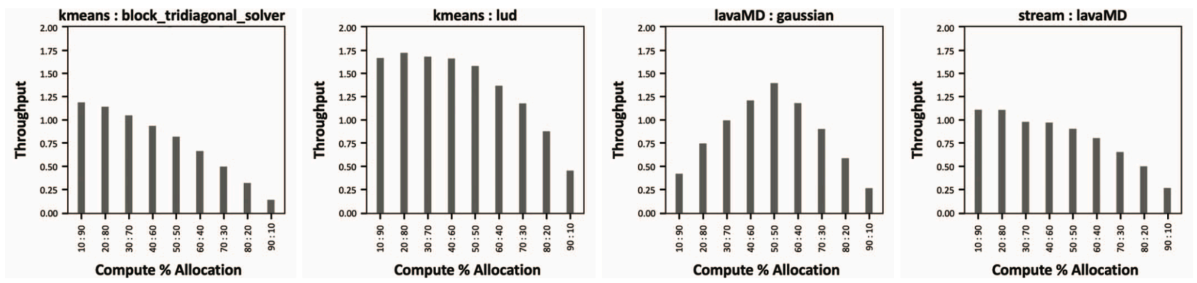
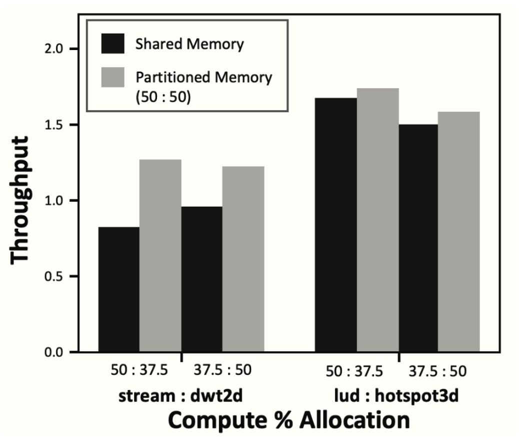


Figure 9: Co-scheduling Throughput as a Function of Compute Resource Allocations using MPS Partitioning

The left graph in Figure 10 presents the impact of memory bandwidth resource partitioning while using two different options (shared or private) offered by the NVIDIA MIG feature. The X-axis lists two different job mixes with two different compute resource allocation rates as well as two different memory options (shared or partitioned), while the Y-axis shows the relative throughput normalised to that of the time-sharing scheduling as mentioned above. To assess the impact of memory partitioning on performance, we set up exactly the same compute resource allocation for the shared and partitioned options. For these job mixes, we observe considerable speedup by partitioning/isolating memory bandwidth resources by mitigating the interference impact among the co-located programs. Therefore, depending on the given job mix, it is preferable to partition/isolate the shared memory resources in order to mitigate the interference impact. Finally, the right graph in Figure 10 compares multiple different partitioning options. The vertical axis indicates the relative throughput normalised to that of the time-sharing scheduling mentioned above. For this job mix, the MIG+MPS hierarchical option works the best.



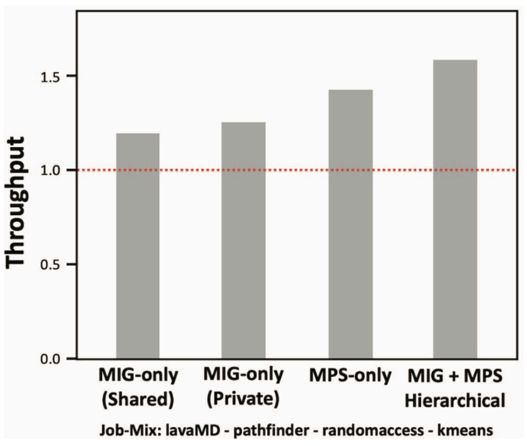


Figure 10: Performance Benefit of Bandwidth Partitioning (Left) and Performance Comparison for Different Partitioning Variants (Right)

Judging from the observations above, it is very important to choose the right combinations of jobs to co-schedule. At the same time, it is essential to have sufficient variants of resource partitioning and select the optimal one from them for a given set of co-scheduling jobs.

Solution Overview

Our approach requires (1) hardware support for enabling GPU partitioning; (2) software interface to interact with the hardware partitioning configuration; (3) RJMS support for job resource-wise scheduling; and (4) capability of RJMS node daemon to read/control the GPU partitioning via an associated software interface. So far the following selections meet the above requirements: NVIDIA Multi-Process Service (MPS) or NVIDIA Multi-Instance GPU (MIG); NVIDIA Management Library (NVML); and Generic Resource (GRES) plugin in the SLURM scheduler (partly supported). The default GRES plugin has several limitations to control the GPU partitioning. For MIG, it can recognize the device files of MIG partitions at the slurmd launch, however it does not control the MIG device setup nor reloads newly added device files generated by creating new MIG partitions. For MPS, the default MPS plugin does not offer per GPU / MIG partition MPS setting. Alternatively, we control MIG/MPS settings via our prolog script to offer a functionality to control MIG/MPS hierarchical partitioning as per request.

Figure 11 illustrates an overview of our approach that works over the GPU partitioning functionality offered by the above software/hardware mechanisms. Our approach optimises co-scheduling GPU job pair selections along with GPU partitioning configurations (MPS/MIG).

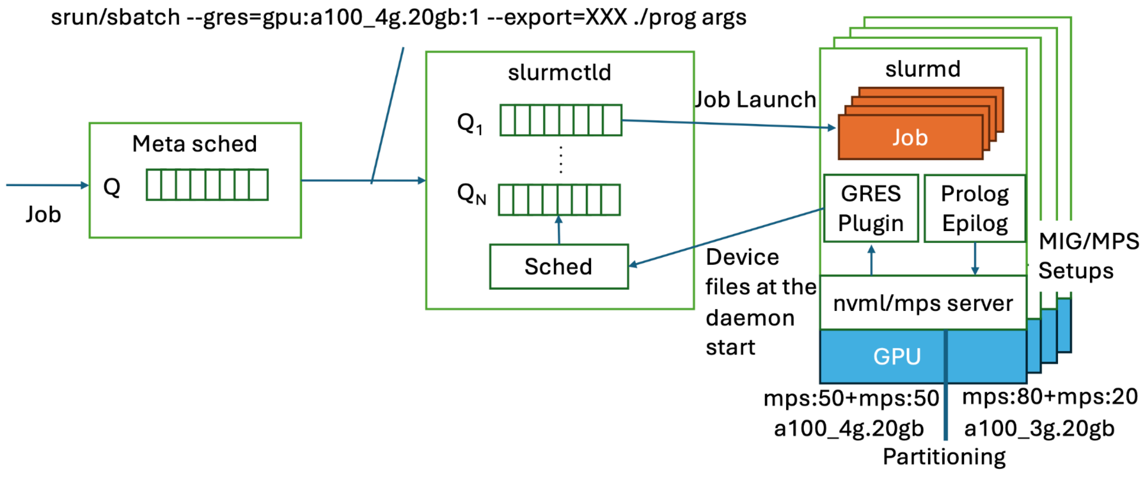


Figure 11: Overall Architecture

An ML-based Sophistication in the Meta Scheduler

Figure 12 illustrates the entire system architecture of our solution. As shown in the figure, the overall solution consists of three parts: (1) the offline profiling to collect application profiles; (2) the offline training to set up the coefficients of our agent; and (3) the online optimization to apply the trained agent to the decision making.

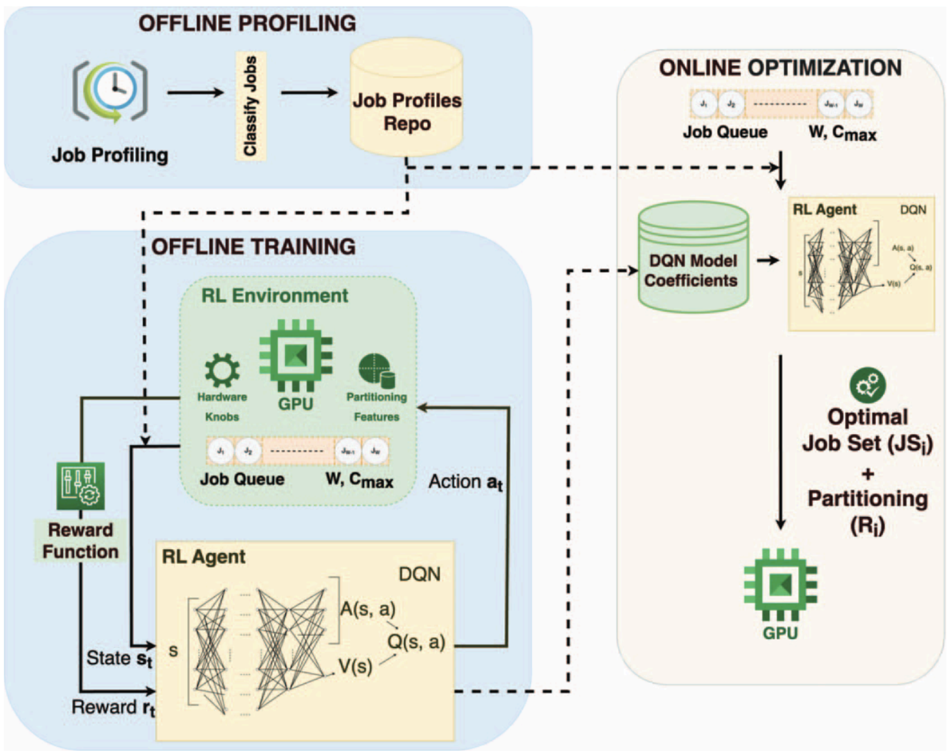


Figure 12: System Architecture of the Meta Scheduler

For the application profiling, we collect hardware performance counters to characterise the running jobs on the target system. The profiles need to be collected beforehand for any co-scheduling targets in both the offline and online phases. In the offline training phase, we collect the solo-run profiles for all the benchmark programs before the model training. In the online optimization phase, if no profile is available for a queuing job, it is excluded from the co-scheduling target and is executed exclusively using the entire GPU while collecting the profile that shall be stored in the Job Profiles Repository. If the application is executed again on the system, it is included in the co-scheduling target as the profile is available in the repository. This procedure requires a matching function to select a corresponding profile for each job based on its submission information (e.g., binary path, user ID, etc.). In the current version, we simply consider using the application binary path plus name as a key and checking if there is a profile associated with it in the repository. Note, developing an advanced way to generate the key from the job submission information, while taking a variety of aspects into account (e.g., input dependency), is an open problem for profile-based approaches.

For the offline model training, we create variants of benchmark program mixes to co-locate on the target GPU. For each program mix, we continuously examine the co-run throughput while changing the partitioning setup. This partitioning search is based on reinforcement learning, i.e., we update the partitioning and resource allocations accordingly when the next co-run (with the exact same program mix) based on the reward function output that takes the co-run throughput into account. During this procedure, the state-action table, which is approximated by a neural network in this study, is trained, and the model coefficients in the agent are eventually determined. The model coefficients are hardware specific and are not portable to different hardware, however the training procedure is required only once for a system though.

In the online phase, we deploy an optimization agent using the model generated in the offline phase. The agent regards the optimization as a classification problem and uses the model to choose sets of co-scheduling job mixes and corresponding resource allocations to maximise the GPU throughput.

2.3 Elastic Resource Management

Motivation and Background

HPC systems are by design rigid in the way resource management takes place. This is because in contrast with Cloud systems the HPC system managers consider executions with time constraints which enables them to have optimal control of how the computing resources are used, achieve higher system utilization, manage to serve higher demand of requests, provide higher scalability and minimize system fragmentation. In more detail the HPC system managers provide the ways to perform allocations of resources specifying both space and time requirements. They allow users to demand particular computing resources (such as number of nodes, cores, amount of RAM per core, number of GPUs, etc) for a certain amount of time. If the time limit for the job is not added and by default a small time limit is given by the system.

Another aspect related to the rigidness of HPC is their inability to support evolving jobs, which are those jobs that may need a dynamic scale-out (grow) or scale-in (shrink) of their amount of allocated resources. The typical but inefficient practice to satisfy evolving jobs with the traditional HPC resource managers is to allocate in advance the complete amount of resources which will be used during the whole execution of the evolving job which may mean that certain resources will be allocated but not utilized.

Evolving jobs were not the typical example of HPC jobs but with the increasing needs of Big Data and AI workloads to leverage HPC instruments we want to bring more elasticity in HPC in order to enable the efficient execution of these types of jobs and allow them to leverage the computing capabilities of HPC.

The time-constrained, non-dynamic mode of scheduling which fits very well to the High Performance Computing characteristics of these systems is not adapted for Big Data jobs which are more elastic by nature. In typical Big Data frameworks such as Spark and Flink the jobs are launched as Cloud services without time limits and have the ability to scale-out/in rapidly whenever they need.

In this task we want to bring more elasticity in HPC but without altering its resource manager’s internal aspects or losing in scheduling efficiency. We are convinced that each scheduling mode (HPC and Big Data) have their own advantages and disadvantages and they fit better to serve the needs of their typical use cases hence we do not want to change internals of any of them. For that, we study and extend techniques that enable the HPC and Big Data resource and job management systems to collocate with minimal interference on the HPC side but with acceptable and high guarantees for the Big Data jobs executions.

Towards Elasticity in HPC for Big Data jobs

HPC and Big Data resource and job management systems are complex pieces of software, and their interaction is not an easy task. In our case we are interested in the interoperability of Slurm and OAR HPC resource managers with Kubernetes as a Big Data resource manager. Big Data workloads that make use of ML frameworks such as PyTorch, Tensorflow or Horovod do not use a Big Data resource manager within their runtime and hence it is easier to allow an HPC resource manager to schedule these workloads and deploy them with singularity containers. However, in the case of Spark applications things are more complex. While the standalone mode can allow the simple static execution of jobs, which can easily be executed by Slurm and Singularity; In the cluster mode an external resource manager gives more intelligence and elasticity in the executions. In our case, Kubernetes can play the role of resource manager for Spark framework based on already existing developments[12] done by the community. However, these Spark workloads will need to collocate with other HPC

[12] Spark - Kubernetes integration: https://spark.apache.org/docs/latest/running-on-kubernetes.html

workloads on the HPC side, which is managed by another resource manager, Slurm or OAR. The simplest way is to give precedence to one workload over another. HPC jobs have tighter resource requirements, while the Big Data applications are designed to manage resource dynamicity. It is possible to combine the characteristics of each type of workload and achieve optimal behavior for both types of workloads by leveraging: the prolog/epilog mechanism that most of the HPC resource managers offer, along with the capacity of the Big Data resource manager to handle a dynamic number of nodes.

The work in this section makes use of the developments to support Kubernetes for Spark and combines it with previous works[13] upon collocation of Big Data and HPC jobs through the technique of prolog/epilog scripts. This software tool, named BeBiDa, is based on executing Spark applications upon unutilized resources by the HPC resource manager and whenever normal HPC jobs need the resources then Spark instances are removed, and they are restarted elsewhere when there is availability. In these previous works the developments and experiments have been done using Spark jobs with YARN resource manager on the Big Data side along with OAR resource manager on HPC side. In the context of REGALE, we have adapted the technique for Kubernetes on the Big Data side, Slurm and OAR on the HPC side and we have provided the support of the Singularity containerization platform in order to enable flexible environment deployment on HPC clusters.

Initially, in the context of REGALE, the BeBiDa technique has been enhanced for Kubernetes and Singularity, using only Singularity-CRI and Slurm resource manager. The initial version of the prototype implementation can be found here: https://github.com/RyaxTech/bigdata-hpc-collocation

This final version of the prototype implementation has enabled the tight integration of Kubernetes along with the support of both SLURM and OAR resource managers. The code of final version of BeBiDa for Kubernetes, Slurm, OAR and Spark can be found here: https://github.com/oar-team/regale-nixos-compose/tree/main/bebida

The integration of BeBiDa with Kubernetes goes a step further in enabling an elastic resource management for HPC systems, since it allows to bring Spark environment, or other Big Data related environments, on HPC systems.

In our setting we consider that there will be an HPC cluster managed by SLURM and a small Big Data cluster managed by Kubernetes. Big Data jobs launched on the Kubernetes side may have the ability to make use of the HPC cluster in best-effort mode. On the Big Data side we could just have 1 or 2 nodes with Kubernetes executors in order to host the Spark driver jobs. In case of total usage of resources on the HPC side the Big Data cluster resources could serve some minimum requirements for Spark jobs.

To better understand the resource sharing mechanism let us take a simple example of one HPC cluster shared between two applications: an MPI job and a Spark application. In this

[13] Michael Mercier, David Glesser, Yiannis Georgiou, Olivier Richard. Big data and HPC collocation: Using HPC idle resources for Big Data analytics. BigData 2017: 347-352

configuration, the Spark application is submitted to the Big Data resource manager, this application can take advantage of all the available resources of the Big Data cluster and the unutilized resources on the HPC cluster. When an MPI application is submitted to the HPC cluster, the resources previously used by the Spark application are removed from it and allocated to the MPI one. The Spark built-in resilience mechanism will handle this resource loss and split the work between the remaining resources. When the HPC jobs finishes, the freed resources will be allocated again to the Kubernetes cluster and the Spark application might use them again.

The advantage of this solution is that it is simple to implement – only based on configuration and integration – but still gives the flexibility to have HPC resources available for data analytics applications. As a comparison, in a classical setup of two clusters (HPC and Big Data), it is possible for a Spark application to run in stand-alone mode inside an HPC job, but then it cannot leverage resources from both Big Data and HPC clusters. Whereas if it is scheduled through Kubernetes, it can make use of both clusters dynamically and both HPC and Big Data workloads can co-exist efficiently, leveraging both clusters without keeping resources unutilized while others may be fragmented. This technique can be used in the context of Spark for batch executions, but can be also used, with minor changes, in the context of Streaming such as Spark Streaming [3] or Flink.

The main drawback of the solution is that it needs the installation of Kubernetes Node components on the HPC nodes participating in the Kubernetes cluster to be leveraged for Big Data jobs executions. Besides the installation of some additional software the main issue is the security concerns brought by the container runtime interfaces used by Kubernetes. However, this is continuously being improved by the community and since the beginning of REGALE project we have now effective and secure ways to deploy Kubernetes node components as a non-root user[14] helping to remove the security concerns.

Configuration and automation of BeBiDa tool for elastic executions on HPC
We hereby show some basic configuration steps, provided to enable the features to take place. The resource sharing between the HPC and the BD clusters is implemented by attaching the idle resources of the HPC cluster to the Kubernetes cluster. This mechanism is implemented using the HPC resource manager prolog and epilog scripts. Each HPC worker node is a Kubernetes worker which is decommissioned if an HPC job requires the node and re-attached to the pool of Kubernetes workers when the job finishes.

The following terminal details show the contents of the OAR prolog and epilog scripts. Similar scripts are used in the case of SLURM. The prolog script features the drain of the Kubernetes node and its removal from Kubernetes cluster available resources in order to be used by OAR HPC jobs. We can also see the definition of “BEBIDA_NOOP” job-name which, as we will see in the next section, is used for the optimization techniques.

```
#!/bin/bash
```

[14] https://kubernetes.io/docs/tasks/administer-cluster/kubelet-in-userns/

```
export OAR_JOB_ID=$1
export PATH=$PATH:/run/current-system/sw/bin:/run/wrappers/bin
(
echo Enter BEBIDA prolog

printenv
id

if [ "$OAR_JOB_NAME" = "BEBIDA_NOOP" ]
then
    echo BEBIDA_NOOP is set. Do not stop the kubernetes agent
    exit 0
fi

for node in $(oarstat -J -j "$OAR_JOB_ID" -p | jq
".[\"$OAR_JOB_ID\"][] | .network_address" -r)
do
  echo == Removing node $node
  oardodo kubectl --kubeconfig /etc/rancher/k3s/k3s.yaml drain
--force --grace-period=5 --ignore-daemonsets --delete-emptydir-data
--timeout=15s $node
  echo == Removed node $node
done
) > /tmp/oar-${OAR_JOB_ID}-prolog-logs 2>
/tmp/oar-${OAR_JOB_ID}-prolog-logs
```

The epilog script features the uncordon of the Kubernetes node and its to the Kubernetes cluster available resources for Big Data jobs executions.

```
#!/bin/bash
export OAR_JOB_ID=$1
export PATH=$PATH:/run/current-system/sw/bin:/run/wrappers/bin
(
echo BEBIDA epilog

printenv
id

for node in $(oarstat -J -j "$OAR_JOB_ID" -p | jq
".[\"$OAR_JOB_ID\"][] | .network_address" -r)
do
  echo == Adding node $node
  oardodo kubectl --kubeconfig /etc/rancher/k3s/k3s.yaml uncordon
$node
  echo == Added node $node
done
) > /tmp/oar-${OAR_JOB_ID}-epilog-logs 2>
/tmp/oar-${OAR_JOB_ID}-epilog-logs
```

Once these scripts are set in the OAR configuration of each HPC compute node to be added in the HPC-Big Data collocation pool of nodes, and the right changes are made in the OAR configuration files; the addition and removal of HPC nodes in the Kubernetes cluster will be done automatically.

Finally, Spark needs to be configured in order to have Kubernetes defined as its cluster manager. The main configuration change is the creation of a serviceAccount within Kubernetes which will grant kubernetes access required to Spark to spawn the workers across the desired Kubernetes nodes on the HPC side.

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: spark
  namespace: default
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: spark-role
rules:
- apiGroups: [""]
  resources: ["configmaps"]
  verbs: ["*"]
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["*"]
- apiGroups: [""]
  resources: ["services"]
  verbs: ["*"]
---
```

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: spark-role-binding
  namespace: default
subjects:
- kind: ServiceAccount
  name: spark
  namespace: ryaxns-execs
roleRef:
  kind: Role
  name: spark-role
  apiGroup: rbac.authorization.k8s.io
```

By doing that we give the ability to Kubernetes to perform the scheduling decisions for Spark instead of Spark standalone scheduler, hence taking advantage of the power of Kubernetes scheduling algorithms.

Experimenting the elastic execution of Big Data jobs upon HPC resources

Once the environment is configured, we can launch a Spark job to be deployed on HPC clusters leveraging on the dynamic nature of Spark jobs without interfering with the rigid HPC executions.

We provide here a simple example of experimenting with the BeBiDa system management tool to enable the elastic execution of Spark jobs on HPC resources. Once the environment is ready we will see that Spark will launch the driver which will be deployed on the Kubernetes side and then depending on the availability of Kubernetes workers it will deploy the Spark executors. Figure 13 provides a high-level view of how this deployment will take place among the available resources of Big Data and HPC clusters. In the figure we can see an execution with 3 executors where 1 lies on the Big Data cluster while the other 2 executors lie on the HPC cluster.

Now once the Spark job is deployed and we can see how the HPC nodes are used for its execution, we can then deploy an HPC job on the HPC side selecting the compute node 1 and observe how the Spark executor 2 running on the HPC cluster will be killed since the Kubernetes worker 4 will be stopped there so that the HPC job which has higher priority can be executed without interference. Nevertheless, the Spark job will deploy the Spark executor 2 and resume execution (without needing to restart from the beginning) on other available nodes and in particular it will try to use the HPC compute node 3 if it is not used by an HPC job. This will allow both the higher priority rigid HPC Slurm job and the dynamic Spark one to be run successfully.

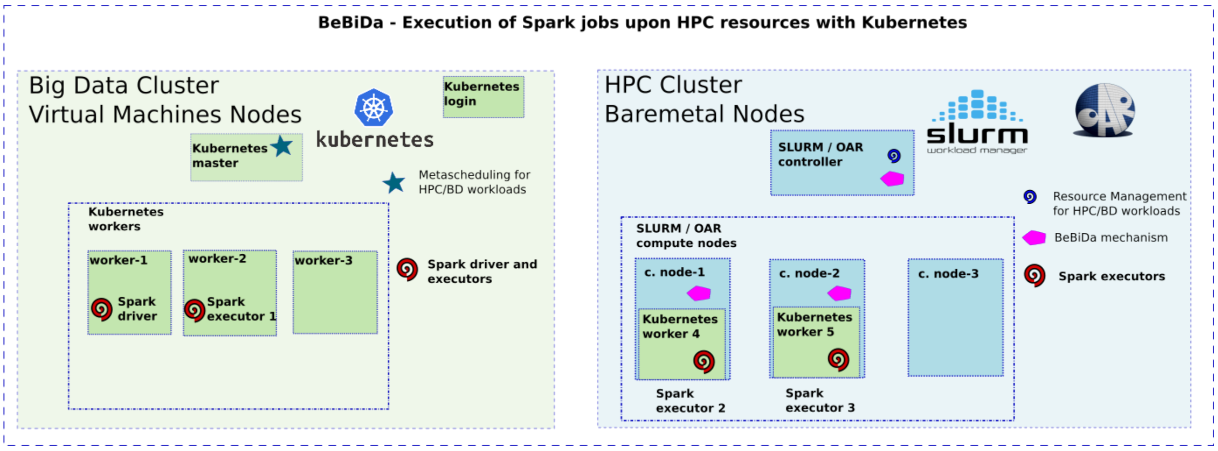


Figure 13: The high-level view of a Spark job execution on HPC resources elastically through BeBiDa

In our procedure, we have made use of NixOS-Compose tool[15] to provide a reproducible methodology to seamlessly prepare the environment on a PC, using VMs, and on Grid5000, using baremetal machines. The procedure is detailed here https://github.com/oar-team/regale-nixos-compose/tree/main/bebida but we provide an example right beneath for a VM based experimentation.

We can either deploy BeBiDa environment with OAR or SLURM resource manager.

```
$cd regale-nixos-compose/bebida
# Build the environment with OAR
$nxc build -C oar::vm
# OR use the following for SLURM nxc build -C slurm::vm
export MEM=2048
nxc start
```

We can wait for the VM to start, and then in another terminal (in the same directory), you can connect to the frontend with:

```
$nxc connect frontend
```

We can check that OAR has two nodes alive with oarnodes:

```
[user@frontend:~]# oarnodes -s
node1:
        1: Alive
node2:
        2: Alive
```

[15] https://github.com/oar-team/nixos-compose

We can then connect to the server to check that Kubernetes is also seeing the nodes as Ready:

```
$nxc connect server
```

```
[user@server:~]# kubectl get nodes
NAME      STATUS    ROLES                     AGE     VERSION
server    Ready     control-plane,master      2m48s   v1.23.6+k3s1
node2     Ready     <none>                    2m36s   v1.23.6+k3s1
node1     Ready     <none>                    2m36s   v1.23.6+k3s1
```

Now we can use OAR and Kubernetes with Bebida enabled. On the server we can watch the Kubernetes nodes state with kubectl get nodes -w, where we can see:

```
[user@server:~]# kubectl get nodes -w
NAME      STATUS    ROLES                     AGE     VERSION
server    Ready     control-plane,master      7m35s   v1.23.6+k3s1
node2     Ready     <none>                    7m23s   v1.23.6+k3s1
node1     Ready     <none>                    7m23s   v1.23.6+k3s1
```

Then we can deploy a simple Spark job execution based on the Spark-pi example here:https://github.com/oar-team/regale-nixos-compose/blob/main/bebida/scripts/spark-pi.yaml

```
[user@server:~]# kubectl apply -f scripts/spark-pi.yaml
```

We can then observe how the Spark executors are deployed across the available Kubernetes resources on either the Big Data or the HPC cluster (through BeBiDa tool)

While this Spark job is running we can go on the frontend node and we can create a simple OAR job

```
[user@frontend:~]# oarsub -l nodes=1 hostname
# INFO:  Moldable instance:  1  Estimated nb resources:  1
Walltime:  3600
```

```
OAR_JOB_ID=2
```

We will then observe in the server terminal that the node allocated to the OAR job becomes unavailable for the Kubernetes workload (SchedulingDisabled), during the OAR job execution and then after some seconds it comes back in a Ready state:

```
node1    Ready,SchedulingDisabled   <none>                8m11s
v1.23.6+k3s1
node1    Ready,SchedulingDisabled   <none>                8m11s
v1.23.6+k3s1
node1    Ready                      <none>                8m15s
v1.23.6+k3s1
```

In the meantime we can obslerve that the Spark executor which was running on the particular node has been removed, since the node was no longer part of the Kubernetes nodes, and it has been placed upon a different available node.

An important aspect that needs to be improved is the execution guarantees we can give on the Big Data jobs when executed elastically on the HPC side through BeBiDa. The current problem is that since HPC jobs have priority and Big Data jobs are always launched on the unutilized resources of the HPC system, it is possible that we do not manage to finalize Big Data jobs in a timely manner. This means that we cannot easily use this technique for real-time Big Data streaming cases. Optimizations are needed to the BeBiDa technique in order to address this issue. This is the subject of the following section.

Improving Big Data jobs turnaround time through new BeBiDa optimization techniques

The BeBiDa system management tool will enable the effective execution of Big Data jobs upon HPC resources with no interference of typical HPC jobs. This functionality comes with the drawback that in case there is a nearly 100% utilization of the system through the typical HPC workloads then the Big Data jobs will suffer from high turnaround times. To improve this we proposed optimization techniques which make use of simple mechanisms that allow the system to go through the typical HPC usage to get resources which will be used by the Big Data side elastically.

We have implemented the improved BeBiDa guarantees in a way that Big Data jobs can respect deadlines and serve time-critical applications..

A) Deadline-aware

In this technique we create empty jobs which do not trigger the prolog/epilog scripts to leave room for Big Data applications to be executed in the space of HPC jobs. The idea is to prepare holes on the HPC schedule plan to guarantee a fixed pool of resources for the Big Data workload.

For that we have set-up a Big Data queue of jobs and when the job queue grows above a certain threshold we submit some HPC jobs with the particular name of `BEBIDA_NOOP`. which as we also mentioned during the configuration of BeBiDa in the previous section does not remove the Kubernetes node from the available resources hence removing the possibility for HPC jobs to kill the Big Data job which is running on that resource.

For this implementation we have created a new service named “BeBiDa Shaker”, hosted on Kubernetes, which has access to the pods list and status. This service looks at new submitted applications and gets the deadline and resource needs using Kubernetes annotations. In particular for Spark applications, these needs are provided as parameters of the Spark driver which demands resources for the spawned Spark executors. Using annotations, the service tries to reserve resources for the application in a way to be sure that it will finish before its deadline. This is done by submitting a HPC job -hence higher priority than the typical Big Data jobs- with prolog and epilog scripts deactivated which will leave the resources for the Big Data workload. In SLURM this has been implemented using the “–begin” parameter. For OAR the advanced reservations functionality has been used to cover this need. This service provides a simple interface to retrieve the following details for each application, its deadline, its estimated resource usage, the associated HPC jobs and the estimated completion time. Figure 14 shows a high-level view of the technique and how the different internal processes are related. In the exceptional case where the deadline cannot be fulfilled an alert is automatically sent to the user by email.

B) Time-critical

In this technique we make use a dynamic set of resources to serve applications immediately and scale them out and in (grow and shrink) when necessary, inspired by the work of Liu et al[16].

The technique makes use of the same core mechanisms implemented for Deadline-aware: the “BeBiDa Shaker” service which tracks the submitted Big Data applications on Kubernetes side and if a particular application is labeled as “Time-Critical” then particular resources are taken out of the HPC pool to serve directly the urgent “Time-Critical” demand coming from the Big Data applications. Furthermore this demand can evolve dynamically based on the Big Data application needs.

[16] Feng Liu, Kate Keahey, Pierre Riteau, Jon B. Weissman. Dynamically negotiating capacity between on-demand and batch clusters. SC 2018: 38:1-38:11

The implementation of this feature made use of the specific feature of OAR named quotas[17] which allows the administrator to limit the amount of resources used by users or the whole system. In our case we configure it for the whole system. This means that the particular resources will be removed from the typical pool of HPC resources for HPC jobs and will become available as Kubernetes resources to be allocated by the urgent “time-critical” Big Data jobs. The following procedure gives the different steps taking place to address the needs of both “deadline-aware” and “time-critical” techniques of managing elastic Big Data jobs on HPC infrastructures

1. A user may submit an elastic Big Data (HPDA) application using Kubernetes.
2. The New Bebida optimization service watches the HPDA submission
   - If the job is a deadline aware job then execute it with Slurm or OAR jobs (no prolog/epilog) setting the right walltime.
   - If it is a time critical job run it directly on the HPDA reservation
3. The New BeBiDa optimization service will regularly increase or decrease the on-demand machine pool through Slurm or OAR allocation, reservations or quota
4. The New BeBiDa optimization service will also watch HPDA application status: if the job finishes before its deadline delete the associated deadline aware job.

The following Figure 14 sketches the design of executing jobs using the new BeBiDa deadline-aware and time-critical techniques.

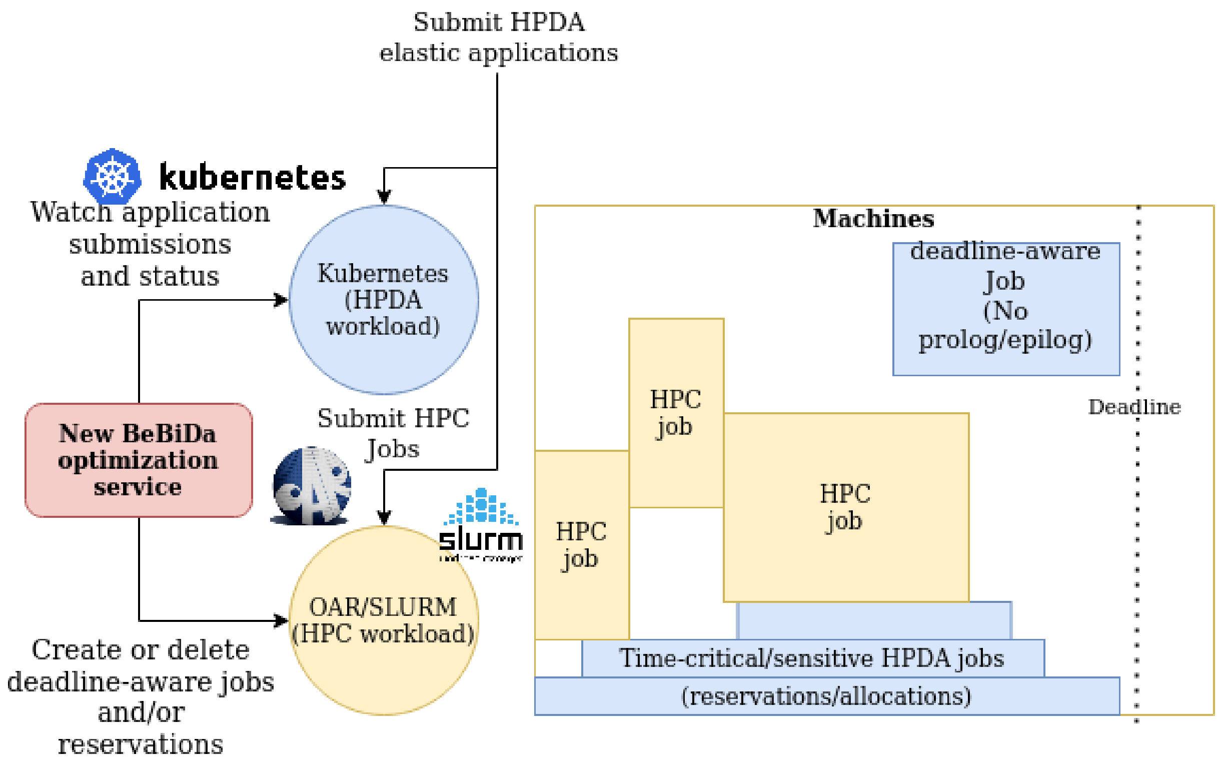

Submit HPDA elastic applications
kubernetes
Watch application submissions and status
Kubernetes (HPDA workload)
New BeBiDa optimization service
Submit HPC Jobs
slurm
OAR/SLURM (HPC workload)
Create or delete deadline-aware jobs and/or reservations
Machines
deadline-aware Job (No prolog/epilog)
Deadline
HPC job
HPC job
HPC job
Time-critical/sensitive HPDA jobs
(reservations/allocations)

Figure 14: High-level view of the deadline-aware and time-critical BeBiDa mechanisms

[17] https://oar-3.readthedocs.io/en/latest/user/mechanisms.html?highlight=quota#quotas

The code of the New BeBiDa sophistications is provided as open-source software in this github repository[18]. The experimentation and evaluation of the new BeBiDa optimization techniques are provided in the evaluation deliverable D1.4.

Conclusions and Perspectives

This section presented the contributions made upon BeBiDa software to be used effectively in the context of REGALE. Furthermore we have proposed new optimization techniques to improve the turnarnound times of Big Data jobs and introduced ways to bring elasticity in the ways to control the traditionally rigid mechanisms of HPC resources management. This goes through combining the power of the traditional HPC resource managers OAR and SLURM with the more elastic Cloud resource manager Kubernetes. The proposed techniques demand the administrator configuration and privileges but there are already ways to install Kubernetes in the user-space minimizing the security risks.

In terms of perspectives we believe that more fine-grained integration and direct communication between the HPC resource managers and Kubernetes will allow the solution to reach higher levels of efficiency, performance and scalability.

2.4 Data-aware resource allocation

Motivation

Current resource allocation strategies as deployed on supercomputers focus on fulfilling user requirements regarding compute and memory needs, globally optimizing machine usage and user response time. But as applications evolve towards more complex codes, the applications :

1. become more data intensive either because larger applications produce more data or applications need to read and process more data (large scale neural network training for instance)
2. are now built by assembling different codes into complex workflows where components exchange data, each component having different needs regarding resources and the different components need to be scheduled within a given order imposed by dependencies.

These evolutions are increasing the pressure on the shared components of the supercomputer (I/O, network), users experiencing more and more often performance impact (interference) due to the concurrent execution of other applications. The machine capabilities are also not evolving into a direction that would soften these issues. For instance, in 10 years the compute power made a leap by a 134× factor, from 1.5 PFlop/s peak on Roadrunner (TOP500 #1 in 2008) to 201 PFlop/s peak on Summit (#1 in 2018), the I/O throughput for the same machines only increased by a 12× factor, from 204 GB/s to 2,500

[18] https://github.com/RyaxTech/bebida-optimization-service/tree/main

GB/s. To prevent this I/O bandwidth shortage to have a catastrophic performance impact on applications, solutions are emerging like adding persistent storage capabilities on each node or group of nodes using SSDs or NVRAM. But they make the architecture more complex, impacting the different software components as well from the system up to the applications. Today's system managers are missing capabilities to control a fair sharing of these I/O and bandwidth resources, impacting application execution, overall machine performance and leading users to find workarounds to compensate for the weakness of the underlying software components.

Assumptions for a solution

Extending resource allocation algorithms to become data-aware requires both to have a better and more complete model of the machine architecture, and to have more information about the application resource needs and execution behavior. Often, user-provided information is limited to compute and memory needs, and a very approximate expected run time. A solution will require to address these different issues:

1. Better model of the machine, integrating networking and storage capabilities;
2. Richer information about application resource needs and consumption. Two options that can be mixed: i) providing the user with a language to describe the application profile/needs; ii) compute an application profile without user input relying on machine learning techniques.
3. Resource allocation algorithms capable of leveraging these data to better fulfil the application needs, reduce the interferences on shared resources, and so ensure low execution times as well as high machine usage.

An important identified issue is the capacity to study harmful resource interferences due to excessive activities (I/O, network, computing) observed on real infrastructures. In particular, we need to be able to reproduce them realistically in simulation in order to be able to replay scenarios with different interferences so that we can fairly compare the different solutions. This raises the question to define and build good models (jobs, I/O, file systems, etc.) and workloads. This is challenging because the interactions we have to take into account between types of resource and systems are numerous and reveal complex behaviors. To tackle this, we consider a coarse-grain modelization in resource usage and time to cope with the trade-offs between complexity and accuracy. Empirical studies imply a lot of evaluations and experiments of distributed systems. As an important by-product we are concerned by reproducibility issues and we are investigating methodologies to offer strong guarantees.

Methods and Algorithms

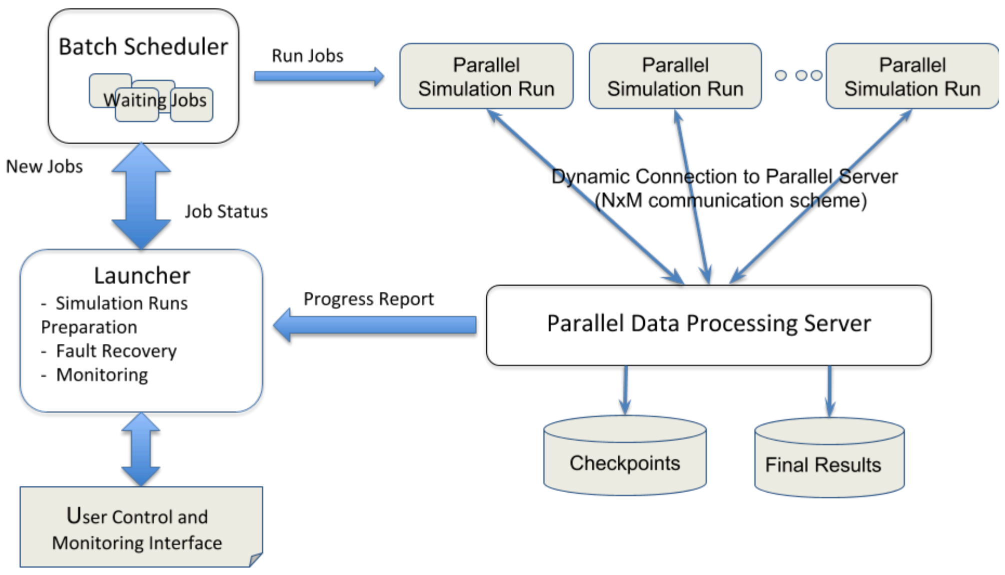


Figure 15: Melissa architecture

The first step to tackle these complex data movement interference problems was the development of a new simulation framework named Batsim. As large scale computation systems are growing to exascale, system managers need to evolve to manage this scale modification. Batsim is an extendable, language-independent and scalable system manager simulator. It allows researchers and engineers to test and compare any scheduling algorithm, using a simple event-based communication interface, which allows different levels of realism. We have demonstrated that Batsim's behavior matches the one of the real system manager OAR.[19]

As a use-case we will rely on the ensemble run workflow Melissa and associated Pilots. Melissa (Figure 15) executes several instances of the same simulation with different input parameters (the members of the ensemble). These members produce data that is directly sent to a Parallel Data Processing Server which aggregates and incrementally processes the data received.

Melissa relies on a flexible architecture where each member and the server are different executables that connect dynamically. Melissa is fault-tolerant and elastic (resources can be added/removed on-the-fly). Thus, Melissa offers an advanced scenario of workflow application with I/Os, multiple jobs, communication intensive, and various degrees of flexibility that make it a good choice for testing data aware resource allocation strategies.

The regularity and predictability of the workflow jobs can hopefully be leveraged to avoid I/O bottlenecks. This ideal placement and execution of workflow jobs can only result in a close interaction between the system manager (who allocates resources), the job manager (who monitors execution and communications) and the workflow engine which produces the jobs and synthesizes the results of the ensemble run. There are needs to enable more

[19] Dutot, P.F., Mercier, M., Poquet, M., Richard, O.: Batsim: A realistic language-independent resources and jobs management systems simulator. In: Desai, N., Cirne, W. (eds.) Job Scheduling Strategies for Parallel Processing. pp. 178–197. Springer International Publishing, Cham (2017)

interactions between the system manager with the global view of the system, and workflow manager with the global view of the workflow application, to optimize the mapping of applications and minimize the data movement between workflow components.

The Melissa Launcher interacts with the system manager (OAR or SLURM in the REGALE project) to submit as many jobs as necessary for the parameter sweep of the ensemble run. Even without considering communication, when using simple naïve queue management algorithms there can be important interferences with the other users on the cluster. For example if a single job of the ensemble run fails, it will be resubmitted and could delay the result of the whole ensemble run by being executed much later. Similarly, the Launcher and the Parallel Data Processing Server components of Melissa only need to use resources when a non-trivial amount of resources are dedicated to workers executing simulation runs.

Even though each simulation run can be associated to a single job in the batch scheduler, this often leads to performance degradation at the system manager level. Clustering jobs and allocating resources in sets reduces the number of calls to the system manager, at the price of having to schedule the jobs through Melissa on the resources provided by the system manager. We will extend this approach by studying the interactions between the system manager scheduler and the workflow manager scheduler.

In another approach, we are exploring a control-based approach (feedback loop) to prevent I/O congestion on shared file systems in HPC platforms due to bag-of-task workloads.

Integration in the REGALE architecture

Melissa and the OAR system manager are already used jointly on several production systems. The main difficulty for this approach is to define precisely the possible interactions, firstly between the job manager which will report job metrics and the system manager, and secondly between the system manager and the workflow manager.

A first approach is to provide to the workflow manager information on the system manager's provisional schedule, to anticipate resource requirements and jobs running times for upcoming simulation runs. For example, when the scientific workflow in execution is about to have a large pool of tasks to process (when one or several of the tasks currently in execution will spawn many tasks requiring a large amount of additional resources), providing the resource manager wait time estimates for any job with specific resource requirements can help the workflow manager to anticipate when to ask for resources and how many resources to ask for. Depending on admission rules, queue structure and the general concurrent workload, some job sizes can have much shorter waiting time as they will be easily backfilled in the current provisional schedule.

A second approach is to provide information to the system manager from the workflow manager, for example giving more information on tasks. In our case, there are tasks which can be cheaply stopped and restarted like clients and essential tasks which should not be interrupted like the server. In addition, although the server can be executed on simple resources (e.g. cores for sensitivity analysis), it can also require specific devices (e.g. GPUs for deep learning). Without communication between workflow manager and system manager, there could be situations where resources for clients are assigned to the scientific workflow without the necessary resources to run the aggregation server, or conversely resources to run the server without enough clients running to avoid idleness. In this regard, specific submission requests formulated by Melissa and addressed to the system manager make it possible to initiate the execution of the server and a few clients only once a specific set of predefined resources becomes available.

Since the sophistication of Melissa was part of this task's objectives, emphasis was placed onto adding new features to Melissa's launcher in order to improve its relation with OAR. The second approach was hence investigated with the strategy described in the next section.

Implementation strategy

The strategy presented herein leverages both Melissa and OAR functionalities like elasticity and fault-tolerance for the former, and job container[20] as well as best-effort[21] jobs for the latter. Thus, the second approaches discussed in the previous section was tackled with the following integration scheme:

1. A job container is allocated for the server and a few clients.
2. Complementary clients are submitted on the best-effort queue outside of the container.
3. The study is stopped as soon as the targeted number of clients is executed or when a convergence criteria is met.

Such implementation is illustrated on Figure 16 and details are discussed hereafter.

[20] https://oar.imag.fr/wiki:job_types
[21] http://oar.imag.fr/docs/latest/user/usecases.html

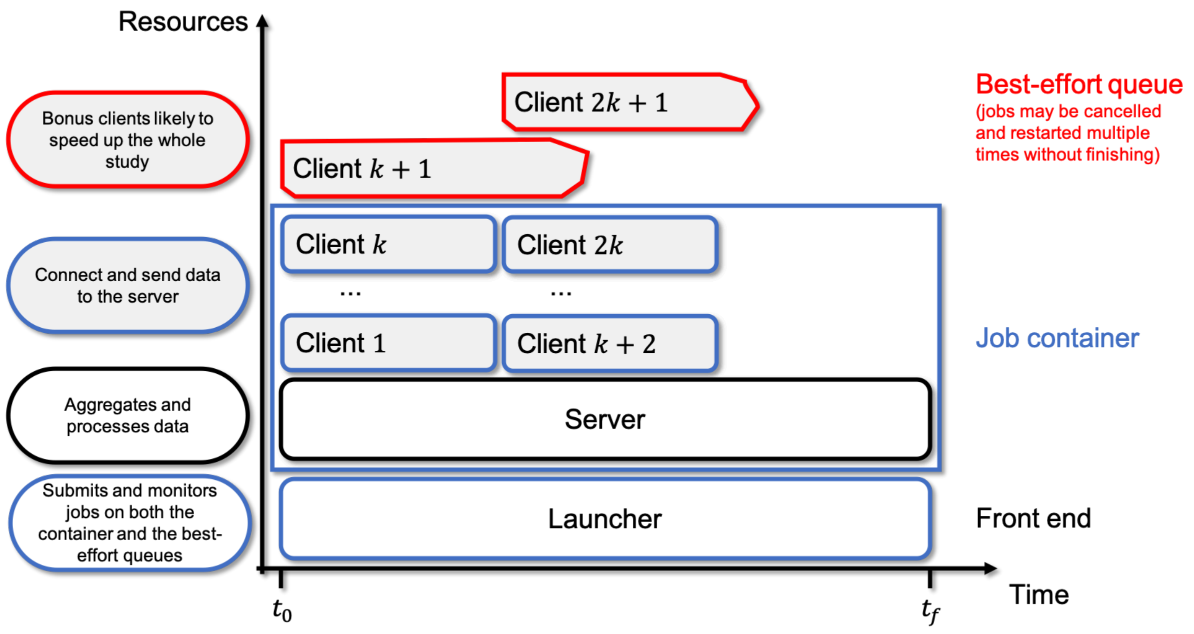

Resources
Bonus clients likely to speed up the whole study
Connect and send data to the server
Aggregates and processes data
Submits and monitors jobs on both the container and the best-effort queues
Client $2k+1$
Client $k+1$
Client k
Client $2k$
Client 1
Client $k+2$
Server
Launcher
Best-effort queue
(jobs may be cancelled and restarted multiple times without finishing)
Job container
Front end
Time
t_0
t_f

Figure 16: Melissa/OAR scheduling strategy

First, Melissa's launcher is executed on the frontend node with a command of the form :

```
python3 -m melissa.launcher vvv --output-dir=/path/to/dir
                  scheduler=oar_hybrid\
                  --oar-hybrid-parameters=<client_freq>\
                  --oar-hybrid-parameters=<size_factor>\
                  --num-server-processes=<server processes>\
                  --scheduler-arg-server=<server option>\
                  --num-client-processes=<client processes>\
                  --scheduler-arg-server=<client option>\
                  path/to/server.sh
```

which selects the newly developed allocation strategy and specifies various resource related options.

From these options, the launcher does the following:

1. The container's *resources* parameter is inferred by combining the server's resources with the clients' multiplied by the *size_factor* parameter.
2. The container's *walltime* is computed from the largest specified *walltime* plus an extra minute.
3. A dummy *walltime* consistent sleeping shell script is written to the working directory.
4. A request for the container is finally made with the following core command:

```
oarsub -t container -t <types> -p <properties> -l <resources>
./dummy_script
```

Once the container's request has been submitted, its *job id* is read from the standard output.

The launcher then submits the server's job inside the container with the command below:

```
oarsub -t inner=<job id> -t <types> -p <properties> -l <server
resources> oarsub.<server id>.sh
```

Then, as soon as the server has processed the user's configuration file defining the study's size and parameters, it will connect to the launcher and ask it to submit as many clients as specified. In this context however, the launcher will switch between the container's and the best-effort queues.

Note that the best-effort queue is requested with the option below:

```
oarsub -t besteffort -t <types> -p <properties> -l <client
resources> oarsub.<client id>.sh
```

The *client_freq* parameter regulates the frequency of jobs submitted and is equivalent to *k+1* with Figure 16 notations.

With such an allocation paradigm, specific care must be taken to make sure that the container's job complies with Melissa's fault-tolerance logic. This means that the container's job state must be monitored so that appropriate measures can be taken in response to abnormal behaviour (*e.g.* an unexpected interruption).

In addition, complementary features were implemented to enhance this strategy's robustness and benefits. Indeed, although the main point of best-effort jobs is that they are likely to start quickly if resources are available anywhere, such jobs can also be arbitrarily cancelled. Thus, despite the fault-tolerance protocol which automatically resubmits such jobs, since Melissa is mostly concerned with numerical solvers, checkpointing is not necessarily an option. As a consequence, every time a job is restarted, it must start over from zero. In this regard, it is totally plausible that some jobs never finish as in case of container or server *walltime* exceeding for instance. For the DeepLearning version of Melissa, missing timesteps can jeopardise the batch construction ultimately resulting in a failure. A way to reduce the likelihood of this issue is to make sure that any best-effort cancelled job is resubmitted inside the container if there is room in it. The strategy can be illustrated as follows:

1. During the initial clients launching phase, the submission queue depends on whether the number of submitted clients is a multiple of the *client_frequency* parameter or not.
2. Failed clients are detected after the first round of submission and are resubmitted, either in the container if there are available resources or otherwise in the best-effort queue.

The proposed strategy offers several advantages. First, the job container ensures a constant stream of data to the server which will in turn avoid any susceptible idle state. Then, as highlighted earlier in the discussion, the procedure relies on a minimal set of extra-parameters related to the size of the container and the proportion of jobs submitted in the best-effort queue. Depending on the cluster's occupancy, this strategy could significantly reduce the computation time and optimize the cluster's throughput. Finally in the worst case scenario, all clients would be run inside the container which would be equivalent to Melissa's standard functioning.

Development steps

In an effort to harmonize the different Melissa flavors, a significant amount of work was put into building a generic launcher compatible with the entire Melissa suite. The full description of the modifications made to the launcher in order to achieve this would however be overly technical for the present report. The reader is then invited to directly go through the documented code[22] to learn more about this aspect. In the following we will only focus on the scheduling management policy.

From a general perspective, schedulers can be divided into two main categories: direct and indirect schedulers. Examples of such schedulers are given in Table 3 below.

Table 3: direct/indirect scheduling examples

Schedulers	Kind	Submission command	Job monitoring command
SLURM	indirect	`sbatch some-script.sh`	`sacct --job=<job-id>`
OAR	indirect	`oarsub --scanscript some-script.sh`	`oarstat -j <job-id>`
OpenMPI	direct	`mpirun -np ntasks --some-options some-executable`	

It may be noted that in case of direct scheduling, there is no specific command to monitor the job state. In Melissa, Python *subprocesses* are used to process the submission command and the associated *poll* function can then be used to monitor its state from each process' *returncode*.

It can also be noted that both SLURM and OAR can be turned into direct schedulers when jobs are submitted interactively respectively with *srun* and *mpirun* inside an allocation.

Thus, Melissa's launcher gives the possibility to work with all kinds of schedulers by defining a specific *scheduler object* coming with its own set of methods to submit, monitor and cancel jobs. In the frame of the REGALE project and WP2, the focus is mostly given to OAR which

[22] Melissa's new launcher has not been made publicly available yet but access can be granted upon demand at https://gitlab.inria.fr/melissa.

was already supported in an indirect fashion. The object of our work towards sophistication was then to extend this module in an alternative version denoted *oar_hybrid* comprising new options related to the container/best-effort job submission. In doing so, specific treatment was needed for instance to:

- Infer the container's dimensions and adapt all submission commands (*container*, *inner*, *besteffort*).
- Keep track of the queue each job was submitted to.
- Reassign the best-effort jobs into the container depending on its availability and on the job statuses.
- Monitor the container's job state.

Testing process

Although definite conclusions regarding this strategy can theoretically only be drawn through the use of rigorous platform simulators, this sophistication task constitutes a proof of concept whose feasibility can be assessed with basic testing. In this logic, grid5000 platform[23] and OAR version 2.5.10 were chosen to validate our implementation. Even though these tests are preliminary, we still chose to run them on a real cluster in Nancy. The queue used is not as filled as it would be on a real production cluster but still allows us to test submitting workflows with some concurrent workloads and some reservations.

In addition, since the DeepLearning version of Melissa was recently updated to work with the new launcher, it was the considered Melissa flavor for this experiment. The specificity of this version is that the DeepLearning framework relies on GPU computing for the server and Core usage for the clients while the other Melissa flavors (*i.e.* Melissa-SA and Melissa-DA) only have use cases involving Core resources.

Regarding the clients, the 2D heat-equation problem solved with *finite differences* on a cartesian grid is extremely flexible since it can be tuned to match any level of computational cost either by extending the grid size or by extending the simulation's duration. A C based heat PDE solver was then selected as our demonstration use case with the following parameters:

- grid size of 32 400 elements (180 x 180),
- a total of 6000 time steps and 100 messages sending per simulation,
- average execution time around 54 - 70 seconds,
- each client was executed on a single core.

This last point is paramount since experiments involving clients with execution time smaller than a minute are likely to be tainted by OAR's side effects. Finally, all runs were dimensionalized according to grid5000's policy and studies of 30 clients were performed. The plan of experiment used to evaluate this implementation is summarized in Table 4. In the

[23] https://www.grid5000.fr/w/Grid5000:Home

second column are listed the Hybrid parameters as a couple of numbers *(x,y)*. In this notation *x* is the *"client frequency"*, that is how often a client job is submitted to the best effort queue. In the extreme case the values are '1' (all client jobs are submitted as best effort), or '31' (all 30 jobs are executed locally with the server). The second parameter *y* is the *size_factor*, that is the number of resources dedicated to client jobs added to the server requirements in the server container. The last column in this table also gives the preliminary results for the server execution time. Note that these values are extremely skewed by the scarcity of concurrent workload on the test platform. Indeed, in the control experiment (-,-), all the client jobs are submitted individually and scheduled immediately along the server. In comparison, the other *(x,y)* experiments all suffered from the additional cost of executing more than one client job on each resource of the server container.

Table 4: computational plan of experiment

Experiment Id	**Hybrid parameters (*client_freq, size_factor*)**	**Number of clients**	**Server execution time (sec)**
1	(-, -)	30	93.246
2	(31, 30)	30	170.156
3	(2, 10)	30	197.919
4	(5, 10)	30	298.081
5	(10, 10)	30	332.776

Scaling tests were performed with a number of clients up to 500, similar results were obtained. As the test platform is mostly idle, best effort jobs are scheduled immediately and rarely interrupted. Another caveat of this toy experiment is that client execution times are small, leading to a disproportionate impact of the resource manager overheads. The next step is to design a more realistic experiment either in concurrence with a real production workload, or at least with synthetic workloads to generate the cluster loads as witnessed in production environments.

Conclusion and perspectives

The targeted scheduling strategy was successfully implemented and validated. Although these preliminary tests are limited by the low machine workload, the robustness of this scheduling technique has been confirmed and many complementary experiments are in preparation.

OAR also offers the possibility to submit moldable jobs[24], this feature could be added to the newly designed scheduling technique. This only requires to add the comma separated syntax below:

```
oarsub -l nodes=4,walltime=2 -l nodes=2,walltime=4
```

and can be easily added to Melissa launcher. We can even extend this possibility by executing workflows on heterogeneous clusters as the Melissa server supports heterogeneous clients running with different configurations (number and heterogeneity of cores used).

[24] http://oar.imag.fr/docs/latest/user/usecases.html

3 Energy Savings

3.1 Moldability for energy efficiency

Motivation and problem definition

The concept of performance and energy efficiency tradeoffs is well-established in the HPC community; a large class of applications cannot take full advantage of added resources in terms of performance, as their scalability is limited by data movement across hierarchies and/or the interconnect. In literature, this concept is examined as a multi-objective optimization problem and often regarded as a Pareto frontier [25,26]. Modeling approaches [27,28] have been proposed to construct this Pareto frontier and assist in the selection of optimal resource allocation. In the scheduling literature, applications for which the number of resources can be decided between runs are called *moldable*, in opposition to applications requiring the same number of resources for every run which are *rigid*. Furthermore, *malleable* applications can even change their resource requirements at runtime. While most parallel applications are in principle *moldable,* and even *malleable* if the underlying programming model supports it, the current state of practice in resource management regards application resource requirements as *rigid*, leaving no flexibility to the system manager to take advantage of this Pareto frontier of possible allocations. A number of works consider the concept of moldability and/or malleability in the resource manager for power-aware or power-bound job management [29],[30]. While existing techniques that employ the concept of moldability may be sufficient to operate a system under a power budget, moving away from the current state of practice also requires shifting from the current paradigm, where only the user is responsible for performing the necessary estimation of resources for their jobs, while maintaining high user satisfaction (or else, Quality of Service). User satisfaction metrics include waiting time, turnaround time, and slowdown, which can be conflicting goals for a resource manager attempting to maintain high throughput, at a given power budget, with a specified energy footprint.

[25] Balaprakash, P., Tiwari, A., & Wild, S. M. (2013, November). Multi objective optimization of HPC kernels for performance, power, and energy. In International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (pp. 239-260). Springer, Cham.

[26] Gschwandtner, P., Durillo, J. J., & Fahringer, T. (2014, August). Multi-objective auto-tuning with insieme: Optimization and trade-off analysis for time, energy and resource usage. In European Conference on Parallel Processing (pp. 87-98). Springer, Cham.

[27] Endrei, M., Jin, C., Dinh, M. N., Abramson, D., Poxon, H., DeRose, L., & de Supinski, B. R. (2018, November). Energy efficiency modeling of parallel applications. In SC18: International Conference for High Performance Computing, Networking, Storage and Analysis (pp. 212-224). IEEE.

[28] Endrei, M., Jin, C., Dinh, M. N., Abramson, D., Poxon, H., DeRose, L., & de Supinski, B. R. (2019). Statistical and machine learning models for optimizing energy in parallel applications. The International Journal of High Performance Computing Applications, 33(6), 1079-1097.

[29] Sarood, O., Langer, A., Gupta, A., & Kale, L. (2014, November). Maximizing throughput of overprovisioned hpc data centers under a strict power budget. In SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (pp. 807-818). IEEE.

[30] Patki, T., Lowenthal, D. K., Sasidharan, A., Maiterth, M., Rountree, B. L., Schulz, M., & De Supinski, B. R. (2015, June). Practical resource management in power-constrained, high performance computing. In Proceedings of the 24th international symposium on high-performance parallel and distributed computing (pp. 121-132).

In fact, to move away from the existing paradigm, the resource manager needs to be enhanced with three features: a) the ability to handle moldable applications, i.e. pick from a set of available options for each job submitted, b) the incorporation of energy optimization targets beyond the traditional optimization targets, and c) the ability to predict the execution behavior of an application based on previous runs in order to utilize the aforementioned moldability features, even when the user does not provide the relevant information. In this way the RJMS will be able to assume the responsibility of selecting the resources of applications within their performance/energy/power Pareto frontier, to optimize its target metric during operation.

Assumptions for a solution

Currently, extensions in resource management software allow users to pass a set of allowed configurations of resources to the resource manager. Ultimately, we assume that i) the RJMS will be able to construct the Pareto frontier for a job, based on historical data on performance, energy consumption, and power consumption, from previous executions, existing in its database or ii) the RJMS will be able to receive information on the Pareto frontier of a job from the user (either from experience or by code analysis) and explore possible configurations. We will rely on existing modelling approaches to build analytical or ML models for the construction of the Pareto frontier, either by the user or by the resource manager.

Moldability as the state-of-practice

The introduction of moldability in state-of-practice system managers requires modified scheduling algorithms, which will be able to select an appropriate configuration of resources for an application. The goal of the scheduling algorithm will be to encapsulate the following two optimization strategies:

- A local optimization strategy, which will be able to trade the resources of a particular job, for energy efficiency, based on its Pareto frontier, without reducing user satisfaction. For example, if a job is running on many resources to be completed quickly at a high energy cost, to achieve this local optimum the resource manager will have to consider how scaling down the resources of a job to achieve better power/energy efficiency would affect the waiting time, turnaround time and slowdown for the job.
- A global optimization strategy, which will be able to scale up or down the resources of a batch of jobs (a workload), to meet the operation targets of the system (throughput, power budget, energy efficiency, user satisfaction). For example, if the energy provider of the system wants to globally scale up or down the power input of the system, the power share of each job has to be reevaluated to match the new power budget.

Integration in the regale architecture

Our work targets the Resource and Job Management System (RJMS). RJMS constitutes a software framework employed within the realm of high-performance computing (HPC) to oversee the distribution of computational resources and the orchestration of user-submitted job scheduling. We foresee integration with OAR3, which already supports moldability; instead of a single resource allocation, users can request a number of possible allocations. OAR3 is a flexible resource and job manager (also known as a batch scheduler or job scheduler), that is tailored for high-performance computing clusters and diverse computing infrastructures. Its capabilities encompass the management and scheduling of computing resources and tasks across a range of environments, including HPC clusters and distributed computing experimental testbeds.

Moldability feature in OAR3

Primarily, the OAR3 infrastructure already supports the ability to provide users with the capability of setting multiple resource allocation preferences, during the submission time of a job. Both infrastructure and time parameters determine a resource allocation preference. When a user submits a moldable job (i.e., at least two resource allocation preferences), OAR3 will choose the preference based on the fact that:

- resources could be fulfilled (now or in the future)
- the shorter end-time would occur

It is worth mentioning that the end-time is not always analogous to the wall-time since the requested resources could lead to different resource slots that could be already occupied, leading to a waiting time before they are free.

A user is able to submit a moldable job via the OAR3 frontend by executing, for instance, the command: *oarsub -l /nodes=1,walltime=2 -l /nodes=2,walltime=1 /scripts/run.sh*. This submits a batch job with the corresponding script and raises two different resource preferences/configurations. The first preference requires the allocation of one compute node for two hours of usage, while the second preference requires two compute nodes for one hour of usage. Assuming an empty cluster, OAR3 will choose the second option (i.e. -l /nodes=2,walltime=1), since it will lead to a shorter end-time than the first one. On the technical level, the implementation of this behavior is executed as a subtask under the scheduler module.

Initial integration of energy-aware moldability

The new implementation of the energy-aware moldability feature, currently constructs a simple algorithm as its score function, with a global impact, that is:

$$score = \min\{f(1), \ldots, f(n)\},\ where f(i) = cores_i * endtime_i$$

More extensively, the flow of the process is:

1. The user passes various resource allocation configurations at job submission.
2. For each of the received configurations, the new implementation calculates the partial score function by multiplying the number of requested resources -at the granularity level of the core- by the time in seconds of the estimated end-time.
3. The qualified resource configuration is the one with the minimum score function *f*.

The modifications have been accomplished inside the OAR3 scheduler module. Specifically, the responsible statement for choosing the foremost resource configuration has been substituted by the score function (see Figure 17).

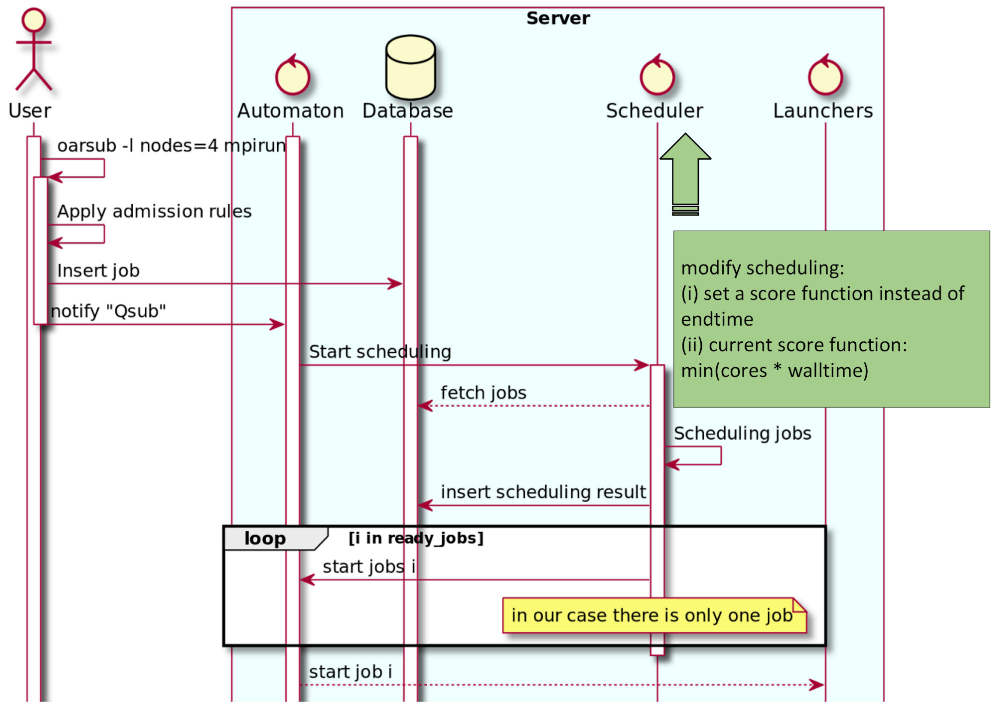


Figure 17: *Sequence diagram of the lifecycle until the job starts: The scheduler module has been altered and modified accordingly to support the new score function's features (as described in the green box). Note: "User" corresponds to the user client of OAR.*

Moldable energy-aware plugin

The score function is indicated to be transformed into a moldable energy-aware separate plugin to the OAR3 infrastructure by developing an encapsulation mechanism for the score function. Consequently, the OAR3 operator will be able to provide its own Python implementation of the score function. The idea is similar to the current mechanism that OAR3 already provides with the plugins. Additionally, as already mentioned, the resource qualification mechanism currently has a global impact without considering the knowledge of the job's affiliations such as tagging, behavior, previous runtimes, etc. Thus, a local optimization target could be implemented, where the energy-aware plugin will have access to jobs' characteristics through previous runs (e.g., through the OAR3 database) or indications specified by the operator, job tagging (i.e., user-labeling of the job at submission time e.g., compute, memory or communication -intensive).

Within Table 5, we have encapsulated our contributions, showcasing the corresponding source code alongside comprehensive information pertaining to the associated branches and releases.

Table 5: Software implementation repositories.

Name	Description	Repository	Branch	Release/ Tag	Based on oar-team
Moldability for energy efficiency	Implementing moldability with a score function that is based on requested core-level resources and estimated end-times, choosing the configuration with the lowest score as the qualified resource option.	https://github.com/cslab-ntua/oar3	energy_aware	3.0.0.dev8	2022/07
Ecosystem of OAR3	All the necessary dependencies, libraries, and OAR3 setup's characteristics and configuration	https://github.com/cslab-ntua/regale-nixos-compose	main	1.1	2023/10

3.2 ML-based User-Labelling of the Job

Accurate prediction of workload power consumption plays an important role in achieving efficient power management in HPC systems, as it allows to design strategies to forecast and control the system's power consumption (e.g. power capping at the workload manager level). Predictive models need quality data, which is often limited due to the inherent complexity of collecting structured data for job power characterization in a production system.

At UNIBO, to fill the lack of resources for job power prediction, we provided (i) a methodology to create a job power consumption dataset from workload manager data and node power metrics logs, and (ii) PM100[31], a novel and large dataset comprising around 230K jobs and the corresponding node, CPU and memory power consumption values recorded during their execution. The dataset is derived from M100[32], a holistic dataset extracted from a production supercomputer hosted at the HPC centre CINECA in Italy.

Furthermore, we designed a lightweight non-parametric technique combining Machine Learning (ML) algorithms with Natural Language Processing (NLP) tools to predict maximum and average users' jobs power consumption. Given that in HPC systems the jobs belonging to the same user tend to be similar, we perform job power prediction by considering each user's data individually, to improve the prediction performance. Our solution employs SBert to encode the job features, and then it performs the prediction through a k-nearest neighbors algorithm (KNN). In Figure 18**,** we present a high-level scheme of the functioning of the prediction model.

The algorithm incurs a negligible overhead on the system's operations, as it does not require any training phase. Moreover, the algorithm leverages only the information available at the

[31] Antici, F., et al. "PM100: A Job Power Consumption Dataset of a Large-scale HPC System". Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis, Zenodo, 24 novembre 2023, doi:10.5281/zenodo.10127767.

[32] Andrea Borghesi, et al. "M100 Dataset 1: From 20-03 to 20-12." 1.0.0, Zenodo, 31 gennaio 2023, doi:10.5281/zenodo.7588815.

time of the job submission (e.g. username, job name, and amount of hardware requested), since in a real system those are the only data available before the job is scheduled and executed. We considered, as the target of the prediction, the maximum/average job node power consumption, normalised on the number of nodes allocated to the job during its execution. We do that to predict each job power consumption as if it was running on a single node, thus making the prediction task less error-prone and more useful in the context of workload scheduling[33] [34]. The proposed algorithm works in an online fashion, meaning that jobs are treated as continuous streaming in time and the models are re-trained periodically to adapt to the change of workload in the system.

We test our solution on the job power data comprised in PM100, and we obtain a Mean Absolute Percentage Error around 20% on both the average and the maximum job power consumption prediction. We further estimate the whole system power consumption in a certain timeframe by summing the power consumption of the jobs running concurrently in the system, in that specific timeframe. When using the predictions obtained with our solution instead of the actual job power consumption, we are able to reconstruct the whole system maximum/average power consumption with an error $\lesssim$ 10%.

We also explored the prediction of the job power consumption at the CPU and memory level, obtaining similar results.

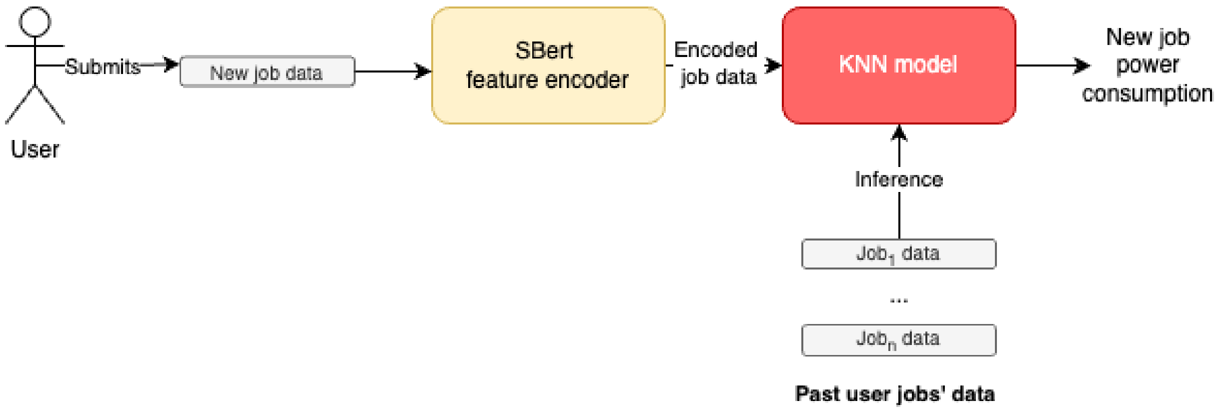


Figure 18: High-level scheme of the user based power prediction algorithm for a new job.

3.3 Integration of ML Models in Production System

At UNIBO, a Machine Learning (ML) production framework has been developed for integrating ML models with production systems. This framework consists of three main subsystems: the **monitoring subsystem**, the **ML operation subsystem**, and the **ML**

[33] Qureshi, Basit. "Profile-based power-aware workflow scheduling framework for energy-efficient data centers." *Future Generation Computer Systems* 94 (2019): 453-467.

[34] Khaleghzadeh, Hamidreza, Ravi Reddy Manumachu, and Alexey Lastovetsky. "Efficient exact algorithms for continuous bi-objective performance-energy optimization of applications with linear energy and monotonically increasing performance profiles on heterogeneous high performance computing platforms." *Concurrency and Computation: Practice and Experience* 35.20 (2023): e7285.

anticipation/prediction model (e.g., anomaly prediction/detection models, forecasting of node's/job's characteristics) (See Figure 19).

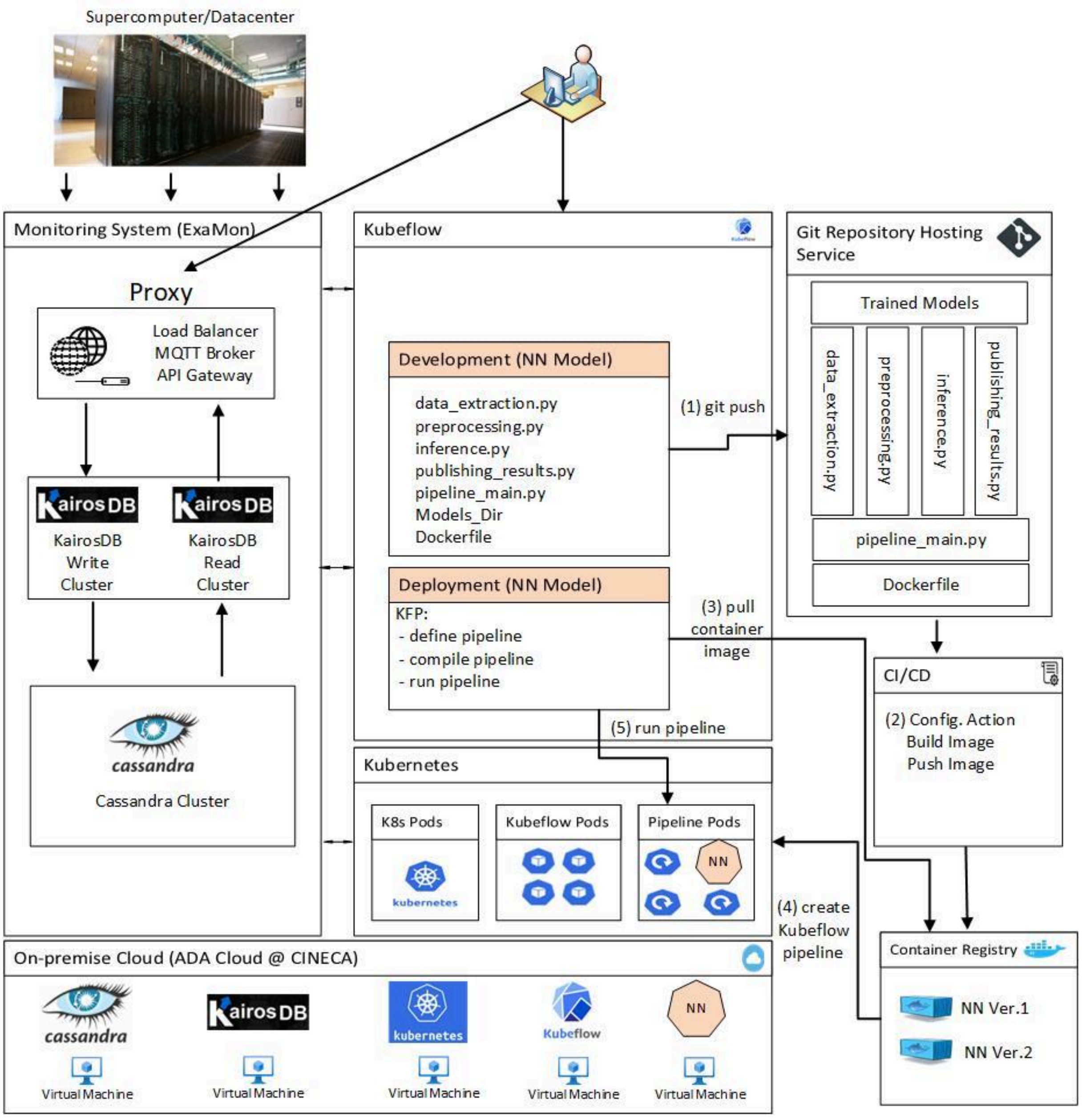


Figure 19: HPC monitoring and Machine Learning Operations framework.

Monitoring System:

The CINECA datacenter features a holistic monitoring framework, ExaMon, which aggregates a wide set of telemetry data collected via a set of plugins (one for each monitored component) that read the sensor and communicate to the ExamonDB via MQTT messages.

ExamonDB uses Cassandra and KairosDB technologies. The different monitored components are at the system level, the job scheduler, the cooling and power provisioning equipment, while at the compute node level, Nagios and Ganglia for in-band telemetry and IPMI for out-of-band telemetry. The complete list of collected metrics is described in M100 ExaData[35]. The ExaMon monitoring system collects sensor data and stores these data in its internal KairosDB database as time traces and is remotely accessible through REST APIs.

Machine Learning Model:

This subsystem can contain different machine learning models used by the sophistication algorithms. Several models have been discussed in the previous sections with inherently different trade-offs between network complexity (number of parameters), number of input features, and accuracy. The framework will query the input features from the monitoring system, and will compute the inference. In the next section, we will analyze the performance of the proposed framework in terms of throughput (inference/s) and latency breakdown, as well as its associated overheads and computational costs (CPU, network, and memory usage). We report three types of model results: the model described above, which uses job information before submission, which consists of a limited number of input features, the models using a medium number of input features, which is the case of rack-level forecasting (power, availability), this model may use as input subset of the node's features and models using large number of input features which are the case of forecasting of node's characteristics (power, availability), this models may use as input all the node's features (~hundreds x time-slice x # of nodes). These three different model types will vary in the computational cost associated with data retrieval and inference computation.

Machine Learning Operations

Figure 19 illustrates the architecture of the HPC monitoring and ML operations framework. Horizontally, it consists of three main components: the monitoring system, the ML models, and the git repository and container registry. The abstraction layers of the ML component are summarized from bottom to top as follows: (i) The on-premises cloud layer hosts all the tools needed on the higher layers. (ii) The software platform for automating the deployment, management, and scaling of containerized applications, namely Kubernetes. (iii) The software tools for ML and (Machine Learning Operations) MLOps on top of Kubernetes. This layer provides an environment for the user to develop, test, and deploy the ML models, namely Kubeflow.

The monitoring system runs in the HPC cloud infrastructure and is not based on Kubernetes. Kubeflow provides a flexible framework for data analytics development and deployment, consisting of dashboards, JupyterLab, and Kubeflow Pipeline. This is implemented in micro-services using the Kubernetes container orchestration framework.

[35] Borghesi, Andrea, Carmine Di Santi, Martin Molan, Mohsen Seyedkazemi Ardebili, Alessio Mauri, Massimiliano Guarrasi, Daniela Galetti et al. "M100 ExaData: a data collection campaign on the CINECA's Marconi100 Tier-0 supercomputer." Scientific Data 10, no. 1 (2023): 288.

The proposed framework's key steps are depicted in Figure 19. The data analytics pipeline for deploying the trained ML models is designed using the JupyterLab services provided by Kubeflow. The pipeline consists of a set of Python scripts responsible for (1) extracting data from the monitoring system, (2) preprocessing and transforming data to a format useful for the ML model, (3) making predictions based on input samples (inference), and (4) publishing the results to the monitoring system via the Examon plugin and MQTT protocols.

A Git repository is used to store these scripts, in conjunction with the trained ML models (with its weights/parameters), and a Dockerfile, which outlines the necessary packages and actions for containerization (step 1 in Figure 19). We configured the Git repository to automatically build a Docker container and push it to the container registry after each update (CI/CD) (step 2). With this final step, the development phase is concluded as we created a Docker container with all the dependencies and scripts needed for the data analytics pipeline to execute and automatically push the image to the Docker registry. Further updates in the git repository will automatically lead to an update of the container image and deployment environment.

The deployment phase consists of executing the containers developed above in Kubernetes containing the data analytics pipeline. The first step consists of creating a pipeline and running it as pods, which will be the target of the deployment. To implement the scripts and container developed in the previous step into a production system, we need to create a pipeline. This involves determining the necessary container images, specifying their runtime behaviour, and configuring them accordingly. It is also necessary to define the inputs and outputs of the pipeline. Once the pipeline is defined, it can be run in Kubernetes.

The pipeline can be created in Kubeflow using Kubeflow Pipeline Python SDK. After defining the pipeline, it is executed. This pulls the required container images from a container registry (step 3 in Figure 19) and runs the pods in Kubernetes (steps 4, 5). When running as a pod in Kubernetes, the pipeline generates failure prediction signals for the supercomputing nodes. The ExaMon plugin and MQTT protocol automatically transmit these signals to the monitoring system. The monitoring system visualizes the results of ML model inference, such as the failure probability, node characteristics (power, etc) in a dashboard. This allows the system administrator or software to take appropriate countermeasures. While the ML pipeline handles the in-production model inference, the training is deployed via SLURM jobs on the Marconi 100 HPC system. ML models need significant computing resources for their training. Supercomputers are particularly suited to meet this requirement, while a cloud environment may not be ideal as it may lack dedicated accelerators. Consistency between the trained model version and the model used in inference is guaranteed by storing the model parameters as part of the data analytics pipeline repository and its integration with CI/CD, which will generate new containers upon repository updates.

Deployment Evaluation

To implement the proposed framework, we employ a cloud system hosted in the CINECA supercomputing facility (on-premise) without creating any overhead on the HPC nodes. This cloud infrastructure is based on the OpenStack version of Wallaby. The nodes of this cloud system are composed of Dual-Socket Dell PowerEdge servers, 2xCPU 8260 Intel CascadeLake processors (24 cores, 2.4GHz), 48 cores per node, hyperthreading x2, 768GB DDR4 RAM, and an internal network of Ethernet 100GbE. The OpenStack virtual machine executes the ExaMon production, which we extended with additional ones for the Kubeflow and Kubernetes pods needed by the MLOps. The computational resources available for the ExaMon monitoring systems are 300GB of RAM and 40 vCores. We also collect standard Kubernetes metrics. For implementing the MLOps framework, we used Kubernetes version 1.24 in our framework for automated deployment, scaling, and management of containerized applications. Our Kubernetes cluster has 48 vCPUs and 360 GB of RAM available. For Kubeflow, we used the canonical Charmed Kubeflow version 1.6.

We analyzed three types of neural networks, each requiring varying amounts of input data or data points. The data points must be extracted from the database. As will be indicated in Table 6, the data extraction phase is the most time-consuming part of the pipeline. Following this, the pipeline includes steps like data preprocessing, inference, and reporting results back to the monitoring system. These steps may vary in latency or computation overhead depending on the size of the data points and the NN model. These three profiles represent different classes of the Neural Network (NN) model. They require less than 1K, approximately 100K, and around 5M data points. For these three sizes, we evaluated the inference pipeline’s computing time, network, memory, and computing cost.

We collected several metrics to evaluate our pipeline, including data extraction latency, preprocessing time, inference computing time, and publishing results latency. These metrics are measured in seconds and presented in Table 6. We also monitored resource usage, including CPU and memory usage and the number of pods used. The corresponding metrics are summarized in Table 7, which shows resource usage for the baseline setup and the three different types of NN models (in view of input data size); the results are grouped into five sub-tables, reporting the resource usage for the monitoring system/ODA (”ExaMon” sub-table), Kubernetes management, Kubeflow management, user workload not including the NN inference (”User Namespace” sub-table), and the workload due to the NN models pipeline (”ML Production pipeline” sub-table).

Computational Resources Overhead

Table 6 shows the latency for different pipeline parts. The last column reports the inference rate, measured as the number of inferences per hour each NN model type achieves. In pipelines, the inference rate depends on the processing time and latency of the pipeline. Data extraction is the most time-consuming step in the pipeline. Pre-processing only takes up 1% of the data extraction latency, and inference time is less than 1%. As evident in Table 6, when scaling the pipeline from the limited number of data points (e.g., from one rack to all the racks of the Marconi100 supercomputers), we notice that the data extraction time scales sublinearly - while increasing the data request of 50x the query time to the ExaMon monitoring system increases only by ~ 5x. After extracting the data, the data preprocessing step requires the second most computing time, while the inference and result publishing steps take negligible time. This result indicates that the proposed framework can scale to exascale system requirements. Moreover, being the pipeline bottleneck, the data extraction of more complex models can be afforded with the current system at a negligible cost.

To better understand the implication and cost of the proposed MLOps framework in conjunction with ODA, we collected resource usage data for different parts of the monitoring system, Kubernetes, and Kubeflow without running any pipelines to determine the base load of the framework (as shown in Table 7). By looking at the baseline case (Baseline in Table 7), we can notice that the ExaMon ODA framework under normal operations (continuous data collection from the different sensors and dashboards) consumes 3 virtual cores (vcores) and 190GB of memory, while the MLOps framework while not processing any data analytics pipeline uses 75 pods (13 used by Kubernetes, 59 Kubeflow, 3 user namespace), 0.66 vcores and almost 7GBs of memory for its micro-services – almost the 22% more vcores and 4% more memory than the pure monitoring framework. Interestingly, when a real-time ML model pipeline is performed (for ~5M input data points in Table 7) for all the nodes of the Marconi100 supercomputer, the ExaMon load increases from 3.08 to 3.41. And the MLOps load increases, from 0.66 vcores to 0.96 vcores with a relatively negligible cost for real-time inference (0.03 vcores). As a result, supporting a real-time ML model in production on the Marconi100 supercomputer requires 30% more vcore resources than merely monitoring it. Of this 30% increase, 11% is attributable to the increased load on the monitoring system, while the remainder is associated with the MLOps component. The ML inference pipeline accounts for less than 1% of the entire overhead, making it ready to scale to larger supercomputers, like exascale systems.

Table 6: Processing time and latency of different deployment configurations.

MLOps Pipeline Stage Execution Time [s]						
Data Points	Data Extraction [s]	Preprocessing [s]	Inference [s]	Publishing Results [s]	Total [s]	#Inference /Hour
~1K	4.2	0.11	0.013	0.002	4.325	832
~100K	10.33	0.15	0.014	0.002	10.496	343

~5M	50.238	4.6	0.337	0.06	55.235	65

Table 7: HPC monitoring and MLOps framework computation resource requirements and ML model pipeline deployment overhead; the 5 main sub-tables indicate the different framework's components.

Data Points	ExaMon					Kubernetes				
	#vcores		Mem [GB]	Net in [kB/s]	Net out [KB/s]	pods	#vcores	Mem [GB]	Net in [kB/s]	Net out [KB/s]
-	3.08		189.5	6670	6739	13	0.31	0.63	1350	864
~1K	3.35		189.5	6680	7334	13	0.31	0.63	1430	870
~100K	3.59		189.5	9588	7732	13	0.31	0.63	1780	880
~5M	3.41		189.5	8686	7975	13	0.31	0.63	1910	880
Data Points	Kubeflow					User Namespace				
	pods	#vcores	Mem [GB]	Net in [kB/s]	Net out [KB/s]	pods	#vcores	Mem [GB]	Net in [kB/s]	Net out [KB/s]
-	59	0.22	5.44	23	28	3	0.13	0.47	7	1
~1K	59	0.22	5.41	24	30	3	0.2	0.5	8	1
~100K	59	0.23	5.41	26	32	3	0.2	0.91	8	1
~5M	59	0.22	5.41	31	32	3	0.4	1.63	21	1
Data Points	ML Production Pipeline									
	pods	#vcores	Mem [GB]	Net in [kB/s]	Net out [KB/s]					
-	-	-	-	-	-					
~1K	1	0.01	0.4	1	1					
~100K	1	0.01	0.44	1	1					
~5M	1	0.03	1.07	14	1					

3.4 Node level power controls (BDPO)

Bull Dynamic Power Optimizer (BDPO) is a lightweight tool whose goal is to optimise at runtime the energy-efficiency associated with the execution of an HPC application. To do so, one standalone instance of BDPO runs in parallel with the target HPC application on each involved compute node. It implements system-wide monitoring of processor-centric performance metrics through the Performance Monitoring Units (PMU) embedded in the compute cores of the CPUs. Indeed, using those metrics, such as the number of Instructions retired Per reference Cycle of the processor (IPC), BDPO can identify phases of low computational intensity and apply Dynamic Voltage and Frequency Scaling (DVFS). In a few words, DVFS consists in switching at runtime the (frequency, voltage) operating point of a compute core of the processor. When compared to its nominal frequency, operating a processor at lower frequencies makes it exhibit less computational power and draws less electrical power. As a result, by adapting the frequency of the CPU to its computational load, it is possible to decrease its average power consumption while inducing only a limited, nay negligible, impact on the time-to-solution of the optimised HPC application. Hence, optimising the energy-efficiency associated with the execution of the target HPC application. As a concluding note regarding this short description of BDPO, let's highlight that its approach makes it completely agnostic of the optimised application. Indeed, since the implemented monitoring and DVFS are system-wide, the target application remains a black box to BDPO. Thus, there is no need to modify or even rebuild an application before trying to optimise its energy-efficiency with BDPO.

This section describes the work associated with BDPO carried out in the context of the REGALE project. To begin with, the first subsection is about the integration of BDPO with OAR, which is the Resource and Job Management System (RJMS) of the REGALE toolsuite. Then, an evaluation of the portability of the approach of BDPO to GPU-accelerated compute nodes is presented in the second subsection. Finally, the third and last subsection gives the rationale underlying the complete refactoring of BDPO to support next generations of CPU-centric compute nodes.

3.4.1 Integration with OAR

To begin with, BDPO had to be integrated into the REGALE software stack so that it should be possible for the end-users to use it to optimise the energy-efficiency associated with the execution of their HPC applications.

In order to execute, BDPO requires elevated privileges on the compute nodes of the HPC cluster to be able to enforce DVFS. As a result, it is necessary to provide end-users, who usually do not have elevated privileges, a means to start and stop BDPO to optimise the energy-efficiency associated with their jobs.

To do so, prolog and epilog scripts for OAR, the RJMS integrated in the REGALE prototype, were written. Indeed, OAR has elevated privileges, and can, hence, start and stop BDPO for the job launched by the end-user when required to do so thanks to the associated command line option:

```
$ oarsub -t bdpo=monitor  -l nodes=4/walltime=2 user_job
$ oarsub -t bdpo=optimize -l nodes=4/walltime=2 user_job
```

Note that it is possible to specify which execution mode of BDPO is to be used: either only as a monitoring tool, or as a power-optimizer through DVFS enforcement.
Finally, on top of that, an entry is added to the `eventLog` table of the database associated with OAR, so as to log the use of BDPO, and its execution mode, for the concerned job.

3.4.2 *Porting the approach of BDPO to GPU*

To optimise the energy-efficiency associated with the execution of an HPC application, BDPO enforces DVFS on the processors of the compute nodes. Hence, even if it can have an effect on the power consumption of other components, the frequency scaling performed by BDPO solely affects the power consumption of the processors. On CPU-centric compute nodes, the latter usually represents between 60% and 80% of the total power consumption of the compute node, which makes DVFS on CPUs the most significant lever for energy-efficiency on this type of compute node.

However, with the shift toward GPU-accelerated compute nodes, the contribution of the processors to the total power consumption of the compute node tends to become marginal. For instance, a BullSequana XH2415 compute node (the nodes of JUWELS[36], currently 18th of the TOP500 ranking) features two AMD EPYC 7402 (Rome) processors, and four NVIDIA A100-40 (Ampère). The power consumption of the CPU part of the node is less than 15% of the total one, while the GPU part is accountable for roughly 80% of the latter. It means that reducing the energy consumption of the processors by a third is equivalent to reducing the energy consumption of the accelerators by only a sixteenth. It thus clearly appears that porting the approach of BDPO to GPUs would be of uttermost importance for the ExaScale era.

Context of the exploration work

To begin with the definition of the context of the exploration work to port the approach of BDPO to accelerators. The first thing to highlight is the fact that there is not common and open-source management and monitoring interfaces for GPUs. Each vendor implements and supplies its own set of tools to do it. As a result, only one vendor, namely NVIDIA, was selected for this initial investigation step. In the remainder of this section, all the presented results stem from experiments carried out on NVIDIA Ampère A100-80 GPUs.

Then, let's precise that three GPU-ready HPC applications were used to evaluate the prototypes porting BDPO approach to accelerators:

- HPCG[37], the well-known HPC performance benchmark built upon the conjugate gradient computing recipe;
- PW from QuantumEspresso[38] (QE-PW), whose goal is to compute molecular electronic-structure properties. It was applied to the small UEABS test case named AUSURF[39], with 112 atoms;
- EasyWave[40], which is an application which simulates tsunamis given coast topology. Originally written for CPU, it was ported to GPU with CUDA in 2014. It was executed

[36] TOP500 system information for JUWELS: https://www.top500.org/system/179894/
[37] HPCG home webpage: https://hpcg-benchmark.org/
[38] QuantumEspresso home webpage: https://www.quantum-espresso.org/
[39] AUSURF test case: https://repository.prace-ri.eu/git/UEABS/ueabs/-/tree/master/quantum_espresso
[40] EasyWave git repository: https://git.gfz-potsdam.de/id2/geoperil/easyWave

with one of the datasets provided with the application, namely the one labelled "large".

Final point regarding the experimental context associated with the exploration work: each experimental condition was executed at least 11 times, and the presented results are average values with 95%-confidence intervals.

Additional constraints imposed by the GPUs

As explained in the introductory section, for processors, BDPO relies on system-wide fine-grained monitoring to detect phases with low computational needs, during which it applies DVFS at the scale of a compute core. However, accelerators exhibit some constraints which make it impossible to simply duplicate the approach of BDPO.

First, to monitor refined metrics such as the IPC during the execution of an application on an NVIDIA GPU, the CUPTI Profiling API must be initialised and used. The only way to do so is by annotating the source code of the target application, which is intrinsically in opposition with the application-agnosticity of BDPO. Thus, another metric representative of the computational intensity of the executed GPU kernel had to be designed using the NVML API, which is accessible from both the host and the GPU (that is to say from both inside and outside of the target HPC application). After several tries, the metric which seemed the most representative of the computational intensity was the occupancy rate of the Streaming Multiprocessors (SM). Yet, it was far less accurate and slightly less relevant than IPC was on the CPU side.

Second and last additional constraint to be mentioned when working with accelerators: DFVS is system-wide and the range of availables frequencies is narrower[41]. The first part means that the frequency can only be scaled for the whole GPU, and not for a specific SM for instance. And the second part means that the maximum achievable decrease of the average power consumption is lesser than on an average CPUs (since the dynamic part of the power consumption depends on the operating frequency of the computing element).

With the above constraints in mind, two different adaptations of the approach of BDPO for accelerators were designed.

First prototype: API interception

The CUPTI CallBack API offers a way to intercept CUDA functions provided by both its RunTime and Driver APIs. Among those functions, two are specifically of interest: (1) `cudaMemCpy` which is called when a data transfer between the host RAM memory and the accelerator vRAM memory has to be performed, and (2) `cuKernelLaunch` which is called when the execution of a kernel should start on the GPU. The rationale of this prototype, named "API interception prototype" is the following:

[41] As an example: [1005MHz, 1410MHz] for an A100-80 NVIDIA GPU versus [1200MHz, 3200MHz] for an Intel Xeon 'Skylake' Gold 6146 CPU.

- When a data transfer is detected, and the SM occupancy rate is below a certain threshold, the computational load of the GPU is most probably low. Thus, the frequency of the GPU can be decreased, following the same logic as the approach of BDPO;
- When the execution of a kernel is to be started on the GPU, the frequency of the latter is scaled up (if necessary), in order to avoid performance degradations. Once again, this follows the same logic as the approach of BDPO.

Three additional notes. First, since scaling the frequency of the GPU is a relatively tedious and time-consuming operation, it should not be performed too often. Thus, to prevent small data transfers from triggering consecutive and close in time frequency scalings, which would most likely be heavily counterproductive regarding performance degradations, only *large enough* data transfers are considered. Empirically, it was determined that 10 GB was a decent threshold to regard a data transfer as *large*. Second, based on experimental observations, the threshold on the SM occupancy rate was set to 10%. Third, and this is true for the remainder of the document, scaling down the GPU frequency means setting it to its minimum value, which is 1005MHz for the considered accelerator. In the same way, scaling up the GPU frequency means setting it to its maximum and nominal value, which, here, is 1410MHz.

To conclude this section, let's note that this prototype is referred to as "proto 10" in the upcoming figures, as a reference to its two inner thresholds being set to 10.

Second prototype: BDPO-alike daemon

The second prototype, named "daemon prototype", consists of a daemon running on the host part of the compute node. It monitors the occupancy rate of the SM through the NVML interface, from the host processors. With the same rationale as the first prototype, if this rate goes under a 10% threshold, it is considered that the GPU is under a low computational load, which then triggers a downscaling of the frequency of the GPU. On the contrary, when the occupancy rate is back above the 10% threshold, the frequency of the accelerator should be scaled up to avoid performance degradations. This daemon should be started before the execution of the target HPC application runs on the GPU, and stopped after the latter ends. This workflow is similar to the one of BDPO, and was a good candidate for the implementation of a Slurm SPANK plugin so as to be able to automatically handle its lifecycle at the time of job submission. Integration with OAR is also technically possible, but since the technical details are slightly different it was not pursued.

Just as for the first prototype, let's conclude this section by noting that this prototype is referred to as "proto dmon" in the upcoming figures.

Results for HPCG

Figure 20 shows that executing HPCG while letting the daemon prototype manage the frequency of the GPU yields the same performance as the "1410 MHz" configuration, which represents executions at the constant maximal frequency of the GPU. In the remainder of the experimental discussions, this configuration acts as the reference to which others are compared, since it is the most commonly encountered default configuration for GPUs in an HPC production environment. However, the "proto dmon" configuration entails a 8% decrease of the energy consumption, thus significantly improving the energy-efficiency of the execution of HPCG.

In the same way, letting the firmware of the GPU manage its frequency (which is referred to by the label "no constraints" on the figures) yields the same performance as the "1410 MHz" configuration while saving 6% of energy. From those two observations, it can be gathered that the firmware of the GPU tries to adapt the frequency of the latter to its load.

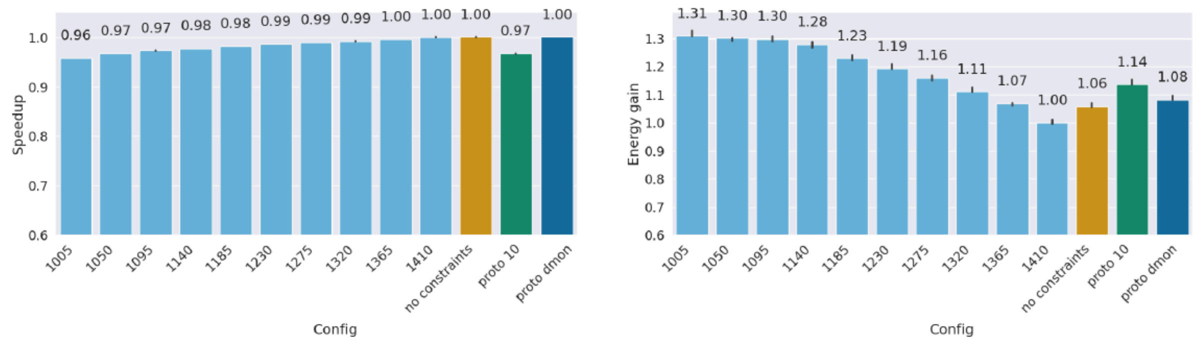


Figure 20: Speedup and energy savings for HPCG with 95%-confidence intervals. For both, higher values are better.

Regarding the "API interception" prototype, its use results in even more energy savings (14%), while the induced slowdown remains acceptable (3%).

Thus, for HPCG, both prototypes yield satisfying results with significant energy savings and no significant slowdown.

Results for QE-PW

Figure 21 shows that executions of QE-PW with the "no constraints" configuration on average yields a 3% slowdown for no energy gain when compared to "1410 MHz".

Now looking at the two prototypes, it appears that "API interception" one induces a 2% increase of the energy consumption accompanied by a 6% slowdown. Regarding the daemon prototype, it manages to maintain an acceptable 3% slowdown for only a 1% energy gain.

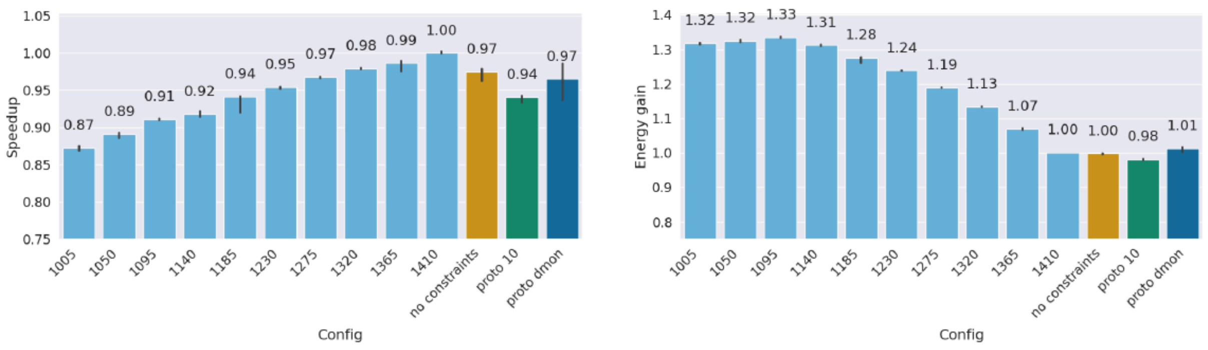


Figure 21: Speedup and energy savings for QE-PW with 95%-confidence intervals. For both, higher values are better.

The speedup plot shows that the Times-to-Solution (TtS) of the executions of QE-PW seem to depend linearly on the frequency of the GPU. This kind of trend characterises heavily compute-bound executions of HPC applications. Therefore, it is not surprising that the results yielded by the two prototypes are deceiving, since it is not an easy task to improve the energy-efficiency of a compute-bound application through DVFS.

Results for EasyWave

Figure 22 shows that both the "no constraint" configuration and the daemon prototype do not induce any significant slowdown while saving energy significantly (respectively 5% and 11%). However, due to an incompatibility between the way the API calls are intercepted by the other prototype and the way EasyWave was ported to GPU, it was not possible to use the former to try to optimise the energy-efficiency of the latter.

It can also be observed that downscaling the GPU frequency entails little to no slowdown. This behaviour is typical of memory-bound HPC applications, which are usually good candidates for the approach of BDPO. This tends to explain the encouraging results exhibited by the daemon prototype.

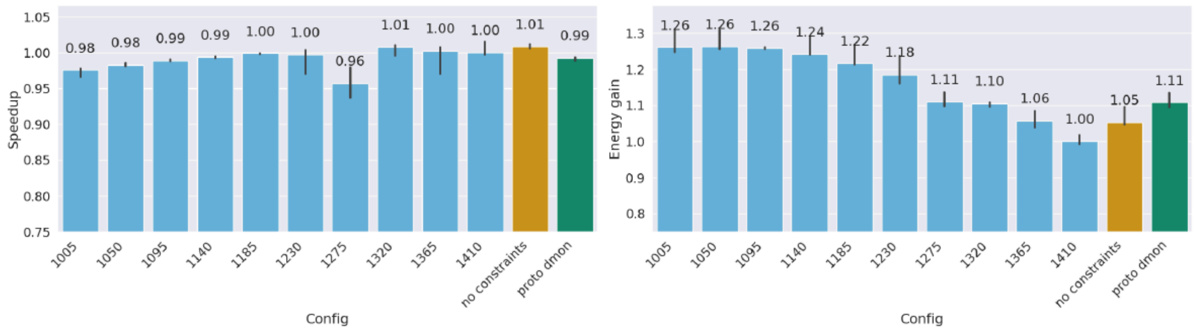


Figure 22: Speedup and energy savings for EasyWave with 95%-confidence intervals. For both, higher values are better.

Figure 23 demonstrates that the "1095 MHz" configuration seems to be the best from the energy-efficiency point of view, since it minimises the Energy Delay Product (EDP). Looking back on Figure 23, the 1095 MHz frequency actually provides an energy reduction of 26% for only a 1% slowdown.

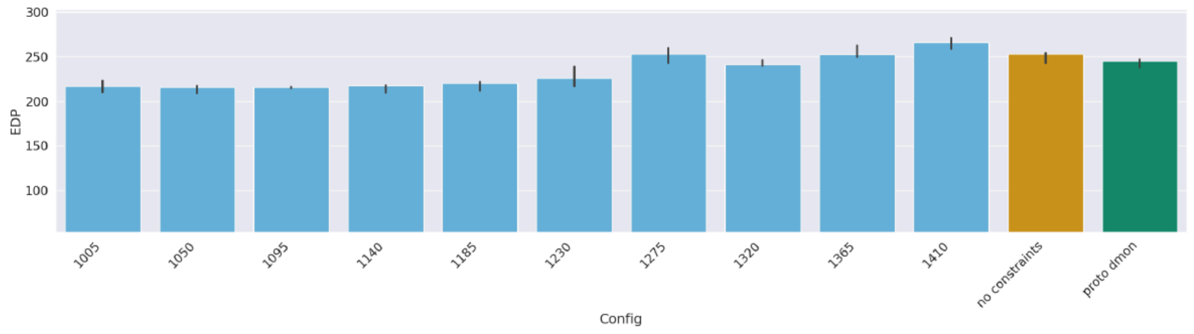


Figure 23: Energy Delay Product (EDP) for EasyWave with 95%-confidence intervals. Lower values are better.

However, finding out that this exact frequency is the best one would require executing the application several times to roam the range of available frequencies. On the contrary, the daemon prototype while providing more modest but still solid energy savings (11%) for negligible performance degradations (1%° did it without any prior execution. And the EDP associated with the execution for the "proto dmon" configuration is lower than the ones of the "no constraints" and "1410 MHz" configuration, indicating a better energy-efficiency.

Conclusions and feedback from NVIDIA

Let us now summarize what was observed in the above experimental discussions, and draw some conclusions. One the one hand, for the selected panel of three HPC applications the daemon prototype globally exhibited promising results, being able to yield reductions of energy consumption without significant performance degradation, even for QE-PW which is compute-bound. On the other hand, the results associated with the “API interception” daemon were mixed, and it could not be used with EasyWave. As a result, from this experimental exploration it appeared that the way to port the approach of BDPO to GPUs implemented by the daemon prototype was the one to be favoured.

However, the occupancy rate of SM seemed less relevant than the IPC for CPUs to estimate the computational load of the GPUs. On top of that, the firmware of the NVIDIA A100-80 accelerator seemed to be managing its frequency based on the aforementioned computational load. Those two observations lead to a series of discussions with NVIDIA to ask for their feedback on our approach, and to try to gain some knowledge of what the firmware is doing regarding frequency management. The key points of those discussions are the following:

- The occupancy rate of SM was considered not representative enough of the computational load of the GPU by the NVIDIA engineers, who recommended not using it to decide when to enforce DVFS;
- The hardware of the accelerator exposes more than 300 metrics to the firmware, among which only a handful are exposed through the CUPTI Profiling and NVML APIs. The decision made by the firmware regarding power management are mostly based on metrics not exposed through the APIs;
- Starting with the Hopper generation of GPUs, Machine Learning (ML) and Neural Networks (NN) models are built based on the aforementioned set of metrics exposed to the firmware to improve notably the power management of the GPU;
- More metrics should be exposed to end-users of the accelerators through the CUPTI Profiling and NVML APIs in the next generation of GPUs (coming after the Hopper generation and to be released in 2025).

As a result, it was decided to put the porting of the approach of BDPO to GPUs on hold, and to wait for the next generation of GPUs which, hopefully, will expose more metrics and fine-grain control knobs for power management.

3.4.3 Complete refactoring of BDPO

Initially, it was planned to work on several new features for BDPO. The latter included for instance the integration of performance models to make BDPO capable of predictive DVFS rather than only resorting to reactive frequency scaling.

However, those plans had to be revised. Indeed, major technological ruptures in the architecture and microarchitecture of the compute cores of modern processors occurred, starting with Intel Ice Lake and AMD Milan. Among those changes, the fact that the Performance Monitoring Unit (PMU) of the compute cores are no longer shared by logical cores associated with the same physical core greatly impacted BDPO. As a matter of fact, the latter resorted on the fact that all the logical cores associated with the same physical cores could be monitored through the same PMU. As a result, on those latest generations of processors, one monitoring thread must be spawned for each logical core, instead of one per physical core. Additionally, the number of cores of modern processors is ever increasing, nearing 400 logical cores for high-end dual socket AMD Genoa compute nodes. However, the legacy implementation of BDPO features several blocking synchronisation points in its main loop. Consequently, on those latest generations of processors, the processor time taken by BDPO skyrocketed, mainly due to time wasted in the aforementioned synchronisation. It made its action far less efficient, nay counterproductive.

Thus, any new feature would have had little to no value with BDPO inherently degrading the performance of the target HPC application by the sole effect of its monitoring loop. That is what motivated a deep refactoring of BDPO internals, which was soon extended into nearly a complete rewrite. Indeed, this refactoring was seen as the perfect opportunity to rethink the whole structure of BDPO to pave the way for the implementation of its future features.

Now at the end of the REGALE project, the refactoring of BDPO is nearing completion, and its core features were ported to the aforementioned new generations of processors. According to the roadmap of BDPO, the features still to be ported should be completed by the end of 2024.

3.5 Thermal and Power control on a node level (ControlPULP)

In this section we will first discuss the sophistication for integrated node-level power and thermal control, then we will discuss ML models which can be used to detect node-level critical thermal conditions which can lead to room-level thermal hazards.

Until two decades ago, power management (PM) was an exclusive responsibility of the Operating System (OS) running on top of application-class processors (AP). The OS agent handling PM in the OS is known as operating system-directed configuration and power management (OSPM). Bearing all the power policy responsibility on the OSPM brings several drawbacks. Firstly, the complex interaction between power, temperature, workload, and physical parameters in an integrated system on chips (SoCs), coupled with additional safety and security requirements, might be too complex for the OS to manage while simultaneously optimizing workload performance. Secondly, the OS does not have introspection on the application behaviour and events, but can only operate on timer-based tick instants, or slower. Finally, in a central processing unit (CPU) thermal constants, power and current can vary so quickly that OSPM software (SW) is unable to deal with them timely. For instance, the timer resolution in Linux (jiffy) is in the range 1 − 10ms while cores' thermal constants are in the order of ms.

Historically, these issues called for a paradigm shift in PM responsibilities within high-performance computing processors. The new paradigm transitions from an OS centric to a delegation based model where user-space power management runtimes (Node-manager and Job Manager in REGALE/ HPC PowerStack nomenclature) collaborate with general purpose, on-chip embedded HW power controllers units. In this schema, the OS has to provide a power management interface (PMI) between these two components. This communication layer could potentially add significant delay to the process, increasing the risk of deadline misses and introducing performance penalties to the overall end-to-end power management service.

In REGALE we target the use of the ControlPULP IP developed in the european-processor-initiative (EPI) project which provides an open-source HW implementation of on-chip embedded power controller units with associated power management firmware. At the time of the deliverable preparation, the ControlPULP IP is integrated into the Rhea design by SiPEARL. Since there is not yet Rhea processors available the EPI project has released together with the ControlPULP IP an FPGA emulation platform based on the Xilinx Ultrascale+ ZCU102 SoC which integrates the ControlPULP HW/FW IP in the Programmable Logic (PL) with a Processing System (PS) composed of two Arm A53 processors.

The on-chip power controller periodically interfaces with Process, Voltage, Temperature (PVT) sensors and actuators and responds to Job Manager and Node Manager directives from the OSPM and user's applications through dedicated HW and SW PMIs.
These on-die interactions are collectively known as in-band services, and are the main focus of the present integration. Finally, the on-chip power controller interfaces with the BMC on the motherboard to support off-chip system services, also called out-of-band. These comprise fine-grain telemetry on the chip power and performance status, chip-level and system-level power capping, and reporting errors and faults in the chip and central processes.

The Job Manager and Node Manager are agents that control the node's and job power policy by interfacing with Linux OSPM. Usual tasks subsumed by the OSPM are Application Processors (AP) idle and performance PM, device PM, and power monitoring. These tasks are managed by OSPM governors, routines tightly coupled with the OS' kernel scheduling policy. An HPC workload consists of parallel applications distributed across all PEs of a set of processors, with one process per core. Under these conditions, the performance, powersave, and userspace governors exhibit the same behaviour — selecting the highest operating frequency. However, state-of-the-art processor PM strategies often leverage application PM run-time, which isolates different phases of the application and requests a new dynamic voltage and frequency scaling (DVFS) level from the OSPM at phase transitions. Two major approaches exist for identifying application phases: (i) timer-based (or periodic), which involves periodically reading performance counters to identify computational bottleneck regions (i.e., memory- or- CPU-bound), and (ii) event-based, which uses application code instrumentation (which can be programmer-driven, parallel programming-driven, or compiler-driven) to flag the entry into different code regions.

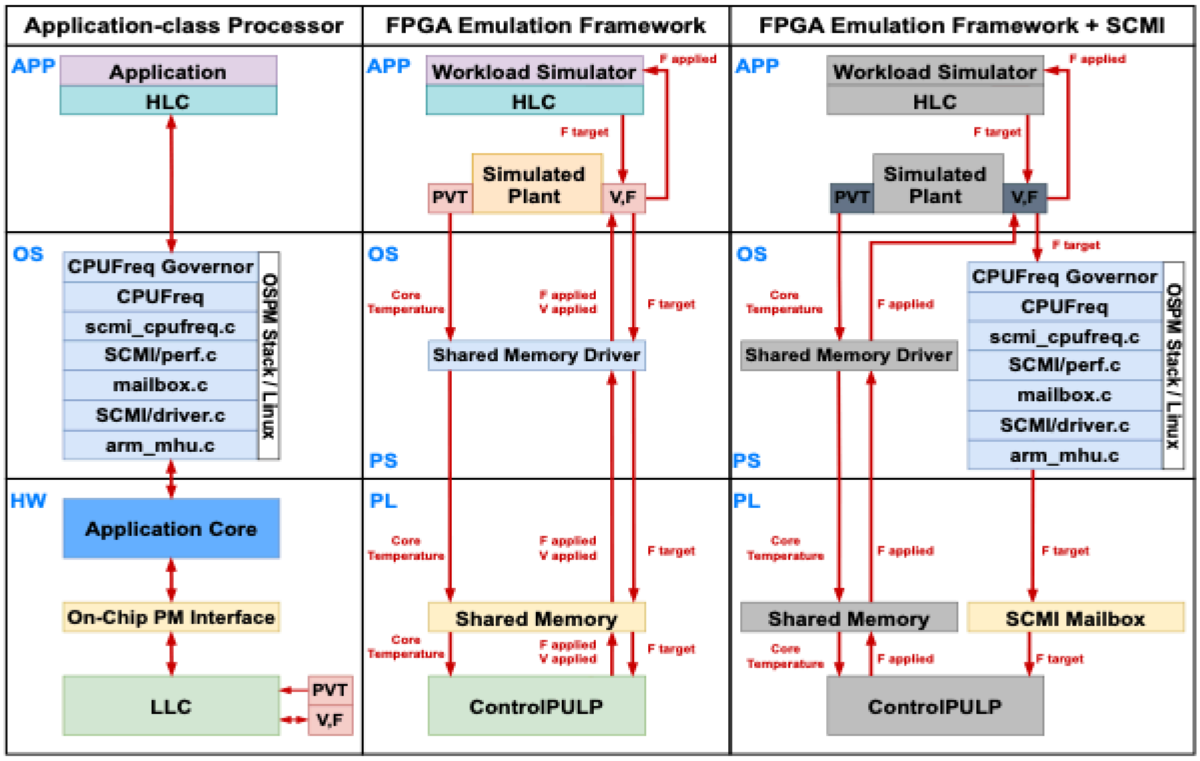


Figure 24: (a) Overview of the main PM components of modern HPC socs. (b) ControlPULP co-simulation platform; (c) Extensions for modelling workload and SCMI Power Management interfaces

On-chip power controllers are usually 32-bit microcontrollers with optional general-purpose or domain-specific modules, from efficient data-moving engines to microcode-driven co-processors or programmable many-core accelerators (PMCAs) to accelerate the PM policy. The LLC is subject to soft and hard real-time requirements, thus demanding streamlined interrupt processing and context switch capabilities. Moreover, its I/O interface has to sustain out-of-band communication through standard HW PMIs such as Power Management Bus (PMBUS) and Adaptive Voltage Scaling Bus (AVSBUS).
The PM policy running on the on-chip power controller can be scheduled as a bare-metal FW layer, or leverage a lightweight real-time OS (RTOS) (e.g., FreeRTOS), as in the open-source ControlPULP IP used in the project. The latter is based on RISC-V parallel cores, and an

associated cascade control PM FW. It supports the co-simulation of the design with power and thermal simulators in Xilinx Zynq Ultrascale+ ZCU102 FPGA as reported in Figure 24 (b)).

PMI interface

Industry-standard FW layer: ACPI is an open standard framework that establishes an HW register set (tables and definition blocks) to define power states. The primary intention is to enable PM and system configuration without directly calling FW natively from the OS. ACPI provides a set of PM services to a compliant OSPM implementation: (1) Low-Power Idle (LPI) to handle power and clock gate regulators; (2) Device states; (3) collaborative processor performance control (CPPC) for DVFS enforcement; (4) Power meters for capping limits. CPPC allows the OSPM to express DVFS performance requirements to the LLC, whose FW makes a final decision on the selected frequency and voltage based on all constraints.

The communication channel between userspace power management policies and on-chip power controller ones is formally defined as a platform communication channel (PCC) by ACPI; it can be a generic mailbox mechanism, or rely on fixed functional hardware (FFH), i.e., ACPI registers hardwired for specific use.

OS-agnostic FW layer: Most industry vendors rely on proprietary interfaces between HLC and LLC. Intel uses model-specific registers (MSRs) mapped as ACPI's FFH to tune PEs' performance. Performance requests are handled through the intel_pstate governor in Linux, while capping is enforced through the Running Average Power Limit (RAPL) framework. In IBM systems, the OpenPower abstraction layer (OPAL) framework relies on special purpose registers (SPRs) and a shared memory region in the On-Chip Controller (OCC) to interact with the LLC. In the RISC-V ecosystem, the recent efforts in designing HPC systems have led to specific ACPI extensions and PMIs, known as RPMI. Arm does not rely on FFHs, which limits flexibility and proposes the SCMI protocol to handle power management performance, monitoring, and low power regulation requests. SCMI involves an interface channel for secure and non-secure communication between an agent, e.g., a PE, and a platform, i.e., the on-chip controller. The platform receives and interprets the messages in a shared memory area (mailbox unit) and responds according to a specific protocol. The design of the SCMI protocol reflects the industry trend of delegating power and performance to a dedicated subsystem and provides a flexible abstraction that is platform-agnostic.

To integrate the REGALE node-manager and job-manager with the ControlPULP IP we extended the FPGA emulation platform with the SCMI HW, SW and FW support. Making it possible for the application processors in the PS part of the FPGA to exchange SCMI power management requests to the Linux operating system toward the ControlPULP HW emulated on-chip power controller synthetized in the PL part of the FPGA. This allowed us to measure and evaluate the cost of the SCMI PMI in terms of power management efficiency loss when serving the request coming from userspace-level control policies (job manager and node manager). To ease the notation will refer to these as High-Level Controller - HLC, and to on-chip power controllers as low-level controllers - LLC. For this aim, we created in the system two dummy job managers who were selecting the optimal energy-efficiency operating point following the workload phases. One controller was based on periodic performance monitoring of the application properties through performance counters (timer-based), and the other was based on instrumentation of the workload phases (event-based).

We observed that the default power control policy implemented in ControlPULP due to its feedback nature was introducing a delay in applying the requested set-point causing a loss of efficiency. We proposed an optimized policy which bypasses the power-capping and thermal-capping optimal internally computed operating point if the requested one has lower power. This optimized firmware leads to an application performance improvement up to 3%. Moreover, we measured a degradation in the performance of the power management algorithm of 0.2% when the SCMI communication is used w.r.t. an ideal zero-latency PMI interface based on shared memory.

Figure 24 (c) shows the block diagram of the proposed HIL emulation framework to model and evaluate the PMI in the end-to-end PM HW/SW stack. The framework consists at the lower layer by the RTL implementation of ControlPULP emulated in the Programmable Logic (PL) of the FPGA-SoC to which we added a SCMI mailbox unit (SCMI mailbox unit (SCMI-MU)) to provide the HW transport for SCMI protocol. At the same time, the Linux OS image running on the Processing System (PS) has been modified to propagate OSPM requests to the ControlPULP LLC via the SCMI-MU. Additionally, a shared memory interface is in place to emulate PM virtual sensors and actuators. Indeed the simulated plant (at the top of Figure 24 (c)) provides a thermal, power, performance, and monitoring framework to simulate the power consumption and temperature of a high-end CPU. The plant simulation is programmed in C and runs on the PS's Arm A53 cores.

SCMI Mailbox:
We implement a hardware mailbox unit called SCMI-MU according to Arm's SCMI standard. In the following, we compare the two.
Arm MHU-v1: In an Arm HPC processor, a communication channel is typically implemented using two HW units: the shared memory and the message handling unit (MHU). The former provides a shared storage for SCMI message data. The latter handles channel arbitration through (i) an interrupt generation logic, which notifies the platform about new messages dispatched to the shared memory, and (ii) a set of registers storing the unique identifier of the transaction associated with the message (the LLC is agent-aware). The format for messages encoded in the MHU's channel registers is user-programmable.
SCMI-MU: We designed the SCMI-MU to be compatible with high-level SCMI drivers in the Linux kernel. As shown in Figure 24, the SCMI mailbox is accessible from both the PS and the LLC through a 32-bit AXI Lite frontend.

The shared memory implements 33 32-bit registers for SCMI compliance and serves as shared spaces for storing host messages from multiple agents. The interrupt generation logic comprises two 32-bit registers, named doorbell and completion, respectively, and an interrupt generation logic. Both registers generate a level-sensitive, HW interrupt to the LLC/HLC when set, and gate the interrupt when cleared. On the LLC side, the doorbell interrupts are routed through a RISC-V Core-Local Interrupt Controller (CLIC), which handles ControlPULP hardware interrupts and improves real-time performance. On the HLC side, the completion interrupts are routed through the Arm GIC-V2. When a message is fetched by the LLC, the MHU register is cleared to not block subsequent writes. This mechanism allows the sender to know if the message has been fetched by the receiver.

Linux SCMI SW stack:

As mentioned earlier, application-level PM policies (HLC) require the *cpufreq_userspace governor*. In SCMI this is composed of the following main routines: *cpufreq*, *scmi_cpufreq*, *perf*, *mailbox*, *arm_scmi driver*, and *arm_mhu*. The HLC writes the new operating point value for core i in an interface file in the system virtual filesystem in Linux which automatically calls the *cpufreq* driver methods. We use release v4.19.0 of the Linux kernel, compatible with Arm MHU-v1, where the *cpufreq* and *scmi_cpufreq* are tightly coupled for message transmission and reception. The scmi_cpufreq driver directly calls perf, which prepares an SCMI message through the *perf_level* set command. The drivers in the bottom layers transmit the message through the SCMI-MU.

Once the message is sent, ControlPULP processes the request, applies the target frequencies to the cores, and triggers the completion signal. On the OSPM, such signal is mapped to an interrupt service routine (ISR) handled by the arm_mhu driver, which reads the transaction identifier, decodes the message in shared memory, and resets the completion register.

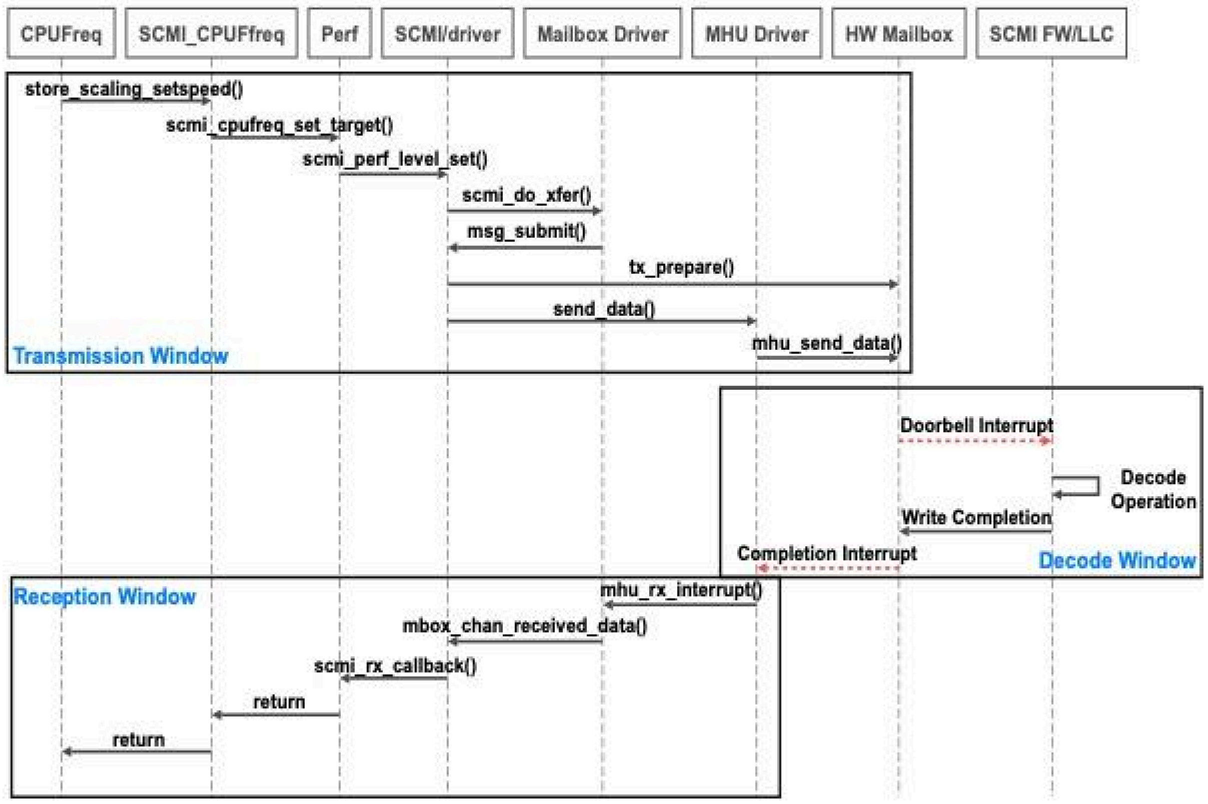


Figure 25: Sequence of function calls within the Linux CPUFreq stack, for a perf_level_set SCMI request.

Figure 25 reports the call stack for SCMI and it is split into three parts: the transmission window **T1** on the agent side, ranges from *store_scaling_setspeed()* in cpufreq to message delivery in the shared memory by arm_mhu; the decode window **T2** on both platform and agent sides, ranges from the doorbell-triggered ISR hook in the LLC to the completion-triggered ISR hook in the HLC via arm_mhu; the reception window **T3** on the agent side, covers function calls until return of cpufreq.

SCMI FW module:

The SCMI FW module executes on the ControlPULP LLC. Its primary purpose is to enable the exchange of SCMI messages between the governor, running on the HLC, and the power policy FW executing on the LLC.

It must be noted that SCMI channel management operations are subject to timing constraints imposed by the Linux SCMI drivers through software timers tasked with detecting channel congestion. Upon expiration, the ongoing transaction may be cancelled. The native FreeRTOS-based SW stack on top of the LLC is leveraged to ensure fast platform-agent reaction time in compliance with these timing constraints.

The FW comprises two sections: (1) a low-level, SCMI-MU HW management layer, and (2) a high-level decoding layer for the governor’s command. The management layer contains methods to access the SCMI-MU shared memory, as well as populating its registers related to interrupt generation. The decoding layer, on the other hand, is embedded within ControlPULP’s power control firmware (PCF) as FreeRTOS task, called decoding task, which leverages the methods exposed by the hardware management layer to access messages from/to the SCMI-MU. After decoding the values of the header and payload fields of the message, the FW maps the SCMI message into a command. The SCMI specification supports a wide range of message types; the firmware implemented in this work focuses on perf level set commands, to handle new frequency setpoint values for a given performance domain, common to all cores in our simulation setup.

On a new message arrival, a doorbell interrupt signal triggers an ISR, which saves the message waiting for the decoding task to execute. The latter is queued for execution until scheduled by FreeRTOS, having lower priority compared to the power and thermal policy tasks. The decoding task reads the transaction identifier from the doorbell register, and the rest of the message from the shared memory. It then clears the doorbell register to signal message reception to the sender and performs decoding. After decoding, it populates the shared-memory with the response and rings the completion interrupt.

ControlPULP PM policy optimized for latency

In REGALE we propose to extend the ControlPULP PM policy to be optimized for latency. The default ControlPULP PM policy (PCF) relies on a periodic control task executing every 500µs (configured to account for ms-scale temperature time constant and µs scale PLL locking time); it executes a cascade of a model-based power capping algorithm and PID thermal capping algorithm. The SCMI new frequency setting is read by the algorithm during the 500µs period to compute the novel power management settings (PLL frequencies and Voltage level) which is then applied in the next task period. This induces, as best case, two period latency between a receipt of an incoming SCMI message and a change of the HW power management interface. This is intrinsic of the control scheme stability and thermal and power capping capabilities. We propose to bypass this latency and directly apply the new SCMI setting if the new operating point has a lower frequency, and yet power, w.r.t. the PCF's computed one for the previous two cycles. This allows to reduce the latency in case of HLC requests which reduces the actual power consumption. We named this policy LLC Optimized.

To evaluate the effect of the proposed sophistication we evaluate its behaviour under two different HLC configurations. A timer-driven one and an event-driven one. The first reads the workload properties with a constant period and assumes with a last-value prediction that in the next period, the workload will share the same properties. Based on that, it computes the optimal operating point for energy and performance and requests the change of operating point through the PMI (SCMI or ideal zero-latency shared memory) - this would be the case of a node manager. The latter one assumes perfect phase instrumentation and selects the optimal operating point using an oracle at phase transition - this would be the case of a job manager like COUNTDOWN or EARlib. This HLC policy emulates the best-case scenario for instrumentation-based power management policies.

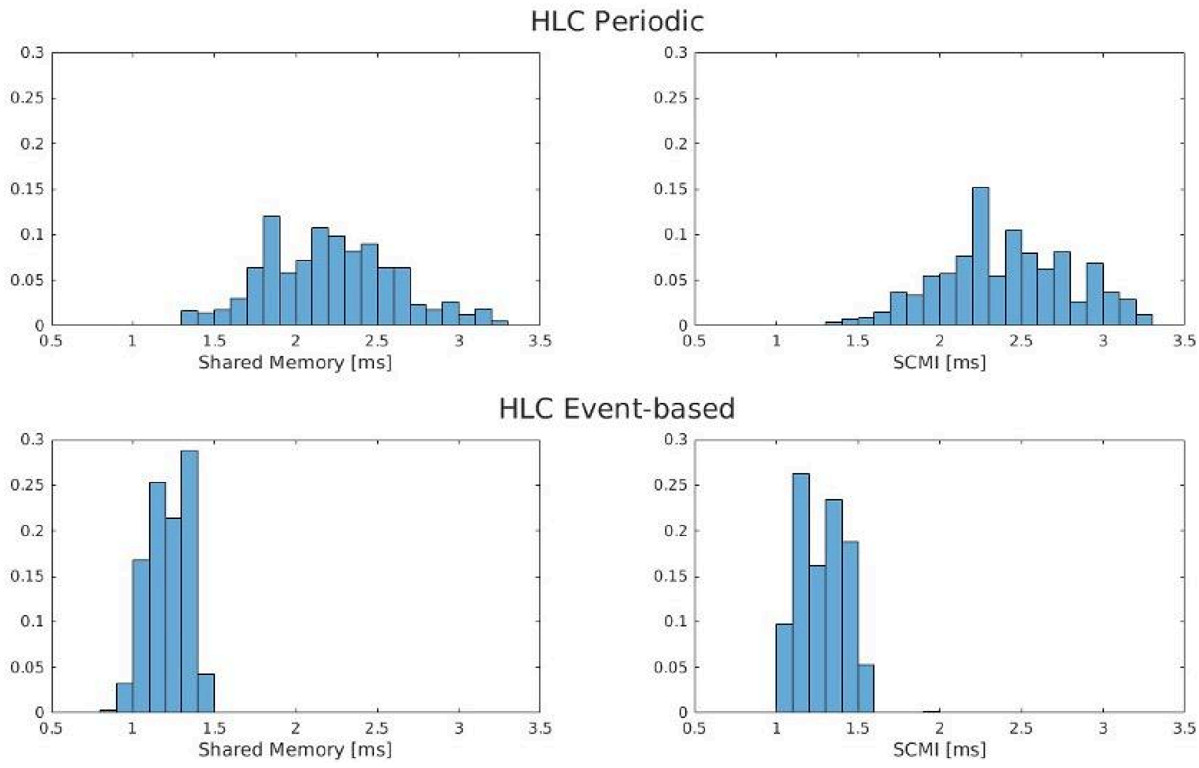


Figure 26: Distribution of Command Delay over two different HLC configurations, for cores controlled through shared memory and SCMI Mailbox interfaces.

Figure 26 reports the command delay (CD) distribution measured as the time elapsed from a workload phase front to an actual change in the applied frequency. We perform three tests: (i) with periodic HLC, (ii) with an event-based HLC, and (iii) with the proposed LLC optimized FW. The average CD for (i) is 2.20ms for SCMI communication and 1.94ms for the shared memory case. Test (ii) results in an average CD of 1.35ms with SCMI communication and 1.16ms for the shared memory case, showing an improvement thanks to the responsiveness of the event-driven HLC. Finally, in the proposed optimized LLC FW (iii) the average CD reduces to 0.77ms.

	Execution Time Speedup [%]	
Configuration	Shared Memory	SCMI
(i) HLC periodic	0.00	-0.02
(ii) HLC event-driven	2.75	2.71
(iii) HLC event-driven, Opt. Control	3.18	2.89

Figure 27: Distribution of Command Delay over two different HLC configurations, for cores controlled through shared memory and SCMI Mailbox interfaces.

Finally Figure 27 reports the control performance for the same three tests measured as the application speedup w.r.t. HLC periodic and baseline ideal shared memory PMI. From it, we can notice that the event-driven HLC (ii) leads to a speed-up in the application (+2.75%) which further increases with the proposed LLC-optimized FW (+3,18%). Moreover, the command delay introduced by a real PMI, the SCMI protocol, leads to a marginal reduction of the attained speedup (from -0.02% to -0.3%). These application speedups are due to better energy efficiency which translates to higher frequencies selected by the LLC FW while enforcing thermal and power caps.

Machine Learning Model for Thermal Anomaly Detection[42]

To predict thermal hazard anomalies, it is necessary to define a labeling approach and NN model that have the potential to predict future thermal anomalies. To do this, it is necessary to first analyze thermal hazard periods to identify relevant patterns and characteristics. Based on this study, we then define a rule-based statistical method to classify the monitoring signals collected from the nodes' sensors into binary categories. Next, we apply this method to generate ground-truth labels and create a dataset for training the ML model. Therefore, we propose a framework for thermal hazard prediction that consists of various capabilities and subcomponents to create a complete pipeline. This pipeline starts with collecting or extracting monitoring data, followed by data preprocessing, implementing a machine learning (ML) model, making inferences, publishing the inference results, and more. The framework also addresses the training requirements of the ML model and handles imbalanced datasets, which are common when anomalies are rare. To build this framework, we compare different machine learning and deep learning models, such as Last Value Prediction, Linear SVM, RBF-SVM, SGD Classifier, LSTM, and TCN, which serve as the "brain" of the framework. Additionally, we studied different data structures to preserve the spatio-temporal information of the monitoring signals.

Thermal Hazard Analysis

To begin our thermal hazards analysis, we studied the temperature distribution of the nodes during two reported thermal hazards as identified by human experts. We then compared this distribution to the temperature distribution during non-thermal-hazard periods. For example, Figure 28 shows the temperature distributions of the inlet (top) and outlet (bottom) for one node in three cases: non-hazard, hazard on 28th June, and hazard on 1st July. Comparing the two figures, we can observe that the inlet temperature distributions for non-hazard and hazards are distinguishable, but this is not the case for the outlet temperature distributions. We also assessed this property for other randomly selected nodes and different monitoring signals. Figure 28-top reports with a dashed black line the quantile 0.95 of the node's non-hazard distribution. As it is visible from the figure, this quantile (0.95) can be used as a threshold to separate the non-hazard and hazard node temperatures. Based on this analysis, we concluded that the quantile 0.95 of the inlet temperature of each

[42] Seyedkazemi Ardebili, Mohsen, Andrea Acquaviva, Luca Benini, and Andrea Bartolini. "HazardNet: A thermal hazard prediction framework for datacenters." Future Generation Computer Systems (2024).

compute node during non-hazard periods is a good parameter to discriminate between hazard and non-hazard. This approach can be expanded to include other monitoring signals. However, in this report, we have only focused on this specific metric.

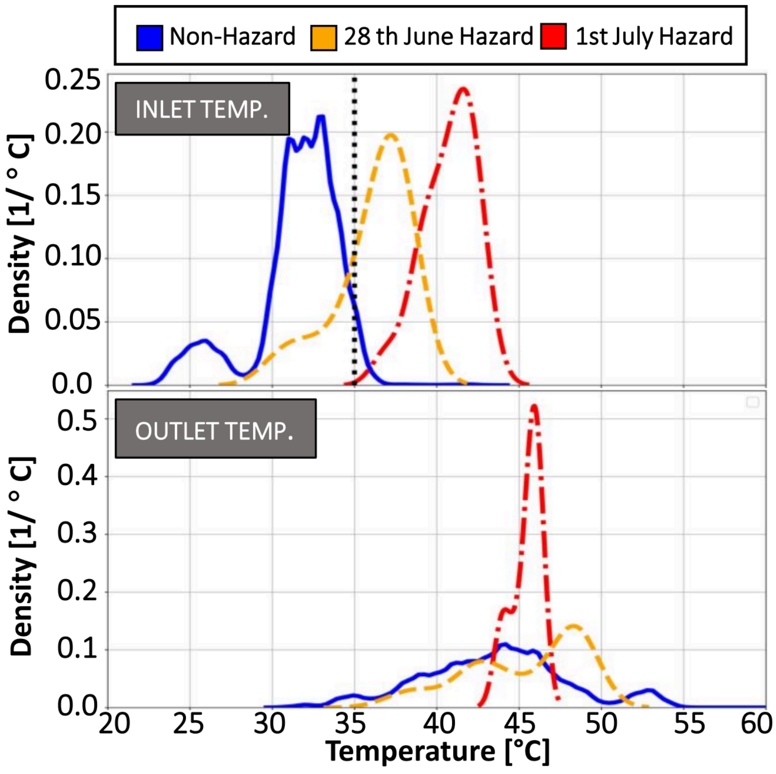


Figure 28: Temperature Distributions for One Compute Node in CINECA's HPC Cluster, June-July 2019.

Thermal (Hazard) Anomaly Labelling Method

We propose a rule-based statistical method to label (detect) thermal anomalies (hazards) by expanding the statistical analysis of thermal hazards.

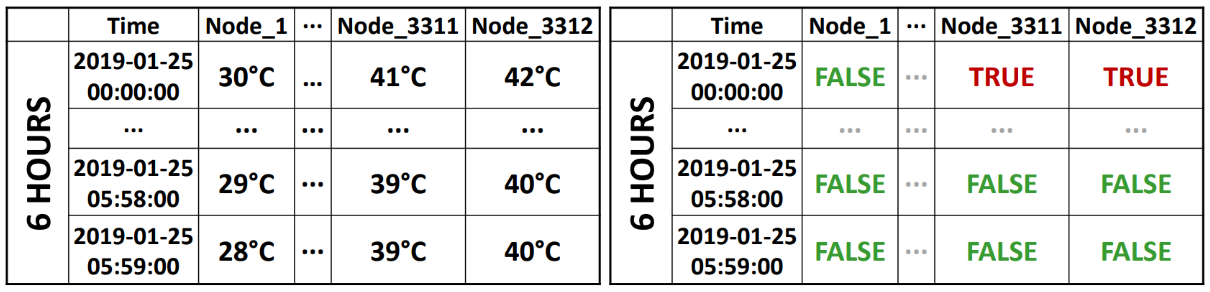
Node-threshold (NT)

Based on the characterization of thermal hazards described, we introduce the node-threshold to assign a binary thermal hazard label for a specific compute node and timestamp. (if a node features a thermal hazard? True: If a node in a timestamp experiences a temperature greater than the node threshold, False: otherwise). We defined the node-threshold individually for each compute node as the 0.95 quantile of its temperature distribution over the entire dataset (one year). Therefore, different nodes can have different node-threshold. Figure 29(a) summarizes a 6-hour time window (TW) of the inlet temperature dataset. We applied the node-threshold to assign to each (node, time) cell a True/False label indicating sample-by-sample thermal trouble, as shown in Figure 29(b). We chose TW = 6 hours which is equal to the prediction horizon.

Spatio-temporal-impact-threshold (STIT)

To assign hazard/non-hazard labels to Time Windows (TW)s (Figure 29(a)), we introduce a spatio-temporal-impact-threshold that takes into account the spatial and temporal continuity of thermal hazards. A TW with 3312 nodes and a duration of 6 hours, with a sample rate of 1 sample per minute, would have a total of 1192320 true/false values (as shown in Figure 29(b)). The spatio-temporal-impact-threshold determines the minimum percentage of ”true” values required within the TW to classify the datacenter as being in a thermal hazard. A higher threshold will result in the selection of thermal hazards that are more widespread, i.e. that involve more nodes for a longer period of time. The spatio-temporal-impact-threshold is a general answer to the following question. How much thermal hazard spread in time and different nodes in the datacenter (in a TW)?

It is essential to note that although this statistical labeling approach is based on real information extracted from the reported thermal hazard distribution, this statistical labeling approach is artificial and must be confirmed by comparing it with the reported thermal hazards. After reviewing the results of the statistical labeling process using human-identified thermal hazards, we discovered that by setting the spatio-temporal impact threshold at 5%, our statistical approach successfully identifies the reported thermal hazards. Additionally, it detects additional thermal hazards that were overlooked by the human expert. These are conditions in which the compute nodes’ temperatures have drastically increased without causing immediate damage but still potentially damaging the nodes. Our statistical labeling approach can capture these events which are unnoticed by humans. If we increase the spatio-temporal-impact-threshold (STIT) quorum to 25%, the statistical labeling approach could only detect the very severe hazard, thus making it too restrictive in identifying abnormal states. For the selected STIT of 5%, the datacenter is labeled as being in thermal hazard for 19.5% of the time (we checked for one year). When we raise the threshold to 15%, the thermal hazard category reduces to 3.8%, while still detecting all hazards. This quantifies the rarity of extensive thermal hazards compared to narrow ones. Both of these thresholds can correctly capture real thermal hazards, but with different levels of sensitivity. Since, in the production scenario, there will be an operator/software that will react to the alarm, we prefer to train the model to be skewed toward higher sensitivity (5% threshold). Still, the operator/software can change this threshold with conditions (See Table 8).



	Time	Node_1	···	Node_3311	Node_3312
6 HOURS	2019-01-25 00:00:00	30°C	···	41°C	42°C
	···	···	···	···	···
	2019-01-25 05:58:00	29°C	···	39°C	40°C
	2019-01-25 05:59:00	28°C	···	39°C	40°C

(a) Inlet Temperature dataset

	Time	Node_1	···	Node_3311	Node_3312
6 HOURS	2019-01-25 00:00:00	FALSE	···	TRUE	TRUE
	···	···	···	···	···
	2019-01-25 05:58:00	FALSE	···	FALSE	FALSE
	2019-01-25 05:59:00	FALSE	···	FALSE	FALSE

(b) True-False table

Figure 29: Time Windowing and Labeling.

Table 8: Thermal Hazard Percentage.

	Spatio-Temporal-Impact-Threshold		
	5%	**10%**	**15%**
Node-threshold 95%	19.5%	8.0%	3.8%

Machine Learning Tools

To determine the most appropriate ML/DL model for the thermal hazard prediction framework, we evaluated different classical ML and DL tools in predicting thermal hazards in CINECA's HCP system.

Classical ML-Learning Tools:

0. Last Value Predictor (LVP): a minimum baseline for any time-series task; the prediction ŷ is simply a copy of the present observation ytrue : ŷ(t + 6H) = ytrue (t), with 6 hours (6H) prediction horizon.

1. Support Vector Machine (SVM): SVM with either linear or Radial Basis Function (RBF) kernels. SVMs produce decision boundaries with margins to improve generalization.

2. Stochastic Gradient Descent (SGD)-classifier: linear SVM trained with SGD instead of convex optimization, enabling larger train set size. SVMs and SGD-classifier were implemented in Scikit-learn 0.23.

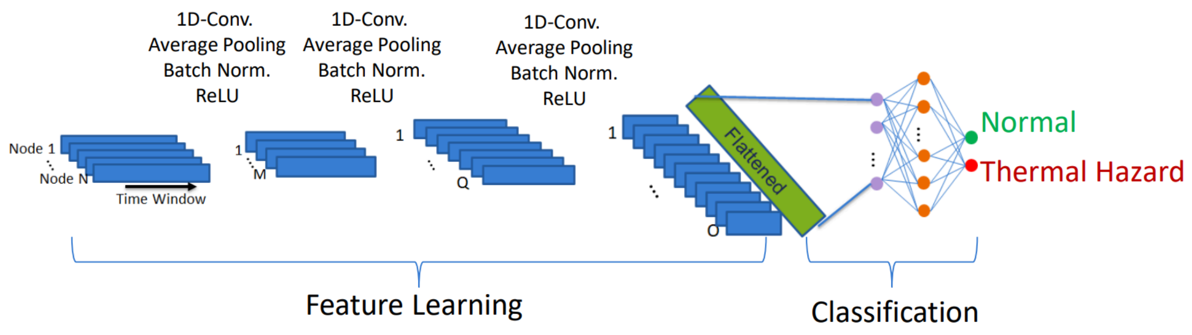
Deep Learning Methods

1. Long Short-Term Memory (LSTM): a type of Recurrent Neural Network (RNN) that can learn long-term dependencies. Our LSTM has 2 layers of hidden and output size 16, followed by a dense layer. The LSTM model was implemented in Keras 2.4.

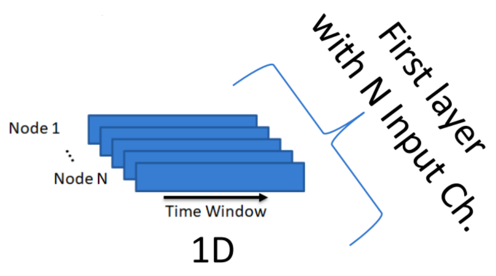
2. Temporal Convolutional Network (TCN): The common TCN is a sequence to a sequence modeling tool. However, our framework requires a classification tool, so we modified the TCN to suit our needs by adding a classification block at the end of the model. Figure 30(a) depicts the proposed TCN model. We propose the different architectures of the TCN model by modifying the convolutional layers and adjusting the input data structure (see Figure 30(a) - 30(e)). While keeping the model's size appropriate, using complex models will allow for expanding the number of input features. Additionally, a more complex architecture is expected to better learn the complex spatio-temporal relations of the monitoring signals. The input data structure also plays a crucial role in maintaining the spatio-temporal relations of the monitoring signals. By using complex TCN architectures, it becomes possible to use more efficient input data structures that still retain the spatio-temporal information of the monitoring signals.

We evaluated the proposed framework in two different scenarios. In the first scenario, we evaluated the model's performance over the entire study period, resulting in an F1-score of 0.98. In the second scenario, we enforced causality in the collected data by training and testing the model in two disjunct and consecutive periods, resulting in an F1-score of 0.87.

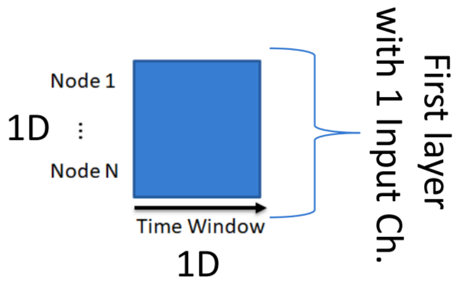
When integrated with the Machine Learning Operations framework described above ”*Integration of ML Models in Production System*” the data extraction time depends on the data points needed to compute the input feature of the model and accounts for 34s which lead to inference latency of ~37s which is negligible with respect of the prediction horizon of the model (6 hour).



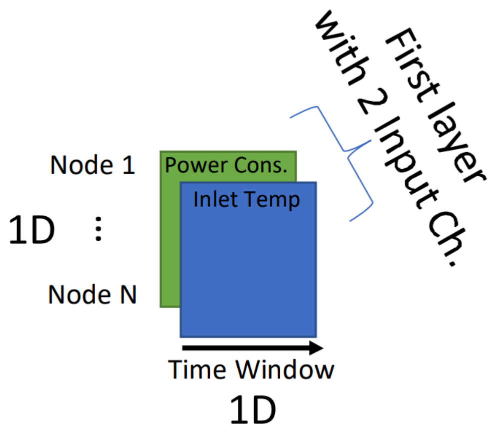
(a) TCN Model with 1DConv. Layers.



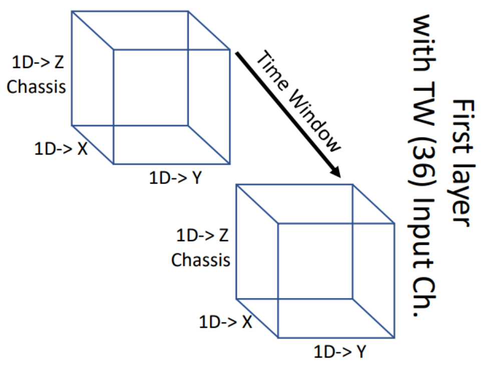
(b) Input Data Structure of the TCN Model with 1DConv. Layers.



(c) Input Data Structure of the TCN Model with 2DConv. Layers.



(d) Input Data Structure of the TCN Model with 2DConv. Layers Power Consumption as a Second Input Channel.



(e) Input Data Structure of the TCN Model with 3DConv. Layers.

Figure 30: TCN Model's Architecture and Input Data Structures for Different Types of Convolutional Layers (1DConv., 2DConv., and 3DConv.).

4 Under Power Constraints

4.1 System Level Power-Capping with OAR

*4.1.1 **Related Work***

In this subsection, we briefly present recent research trends and related works relevant to power/energy-aware HPC resource monitoring and management. We invite the reader to consult Maiterth et al[43]. and Kocot et al[44]. for a more detailed survey on the subject.

Researchers have been proposing energy/power aware scheduling methods for HPC platforms by employing a large variety of methods, ranging from integer programming[45] to heuristics[46,47], and Machine Learning[48]. Most of these works calculate the power consumption of an application relying on the Thermal Design Power (TDP) of the processors/accelerators, and a mix of the application's processing time and resource utilization.

Several works also use energy measurements from interfaces such as RAPL[49,50], but in aggregations such as average energy/power consumption. In contrast, our work relies on real-measured data consisting of fine-grained time series of the power consumption of the applications to show the efficiency of our method.

Few works rely on time-series data of the power consumption, and it is mainly exploited in studies that analyze the behavior of the platforms[51]. This work is a step towards exploiting power consumption time-series data for power-aware HPC resource management.

In summary, the majority of works present sophisticated methods for energy-aware resource management. However, it is widely known that sophisticated methods hinders their deployment in real-world production platforms[52].

Furthermore, these sophisticated methods may require additional a priori characterizations or predictions of the power consumption of the applications.

[43] Maiterth, M., Koenig, G., Pedretti, K., Jana, S., Bates, N., Borghesi, A., Montoya, D., Bartolini, A., Puzovic, M.: Energy and power aware job scheduling and resource management: Global survey—initial analysis. In: 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). pp. 685–693. IEEE (2018)

[44] Kocot, B., Czarnul, P., Proficz, J.: Energy-aware scheduling for high-performance computing systems: A survey. Energies 16(2), 890 (2023)

[45] Pierson, J.M., Baudic, G., Caux, S., Celik, B., Da Costa, G., Grange, L., Haddad, M., Lecuivre, J., Nicod, J.M., Philippe, L., Rehn-Sonigo, V., Roche, R., Rostirolla, G., Sayah, A., Stolf, P., Thi, M.T., Varnier, C.: Datazero: Datacenter with zero emission and robust management using renewable energy. IEEE Access 7, 103209–103230 (2019). https://doi.org/10.1109/ACCESS.2019.2930368

[46] Chasapis, D., Moretó, M., Schulz, M., Rountree, B., Valero, M., Casas, M.: Power efficient job scheduling by predicting the impact of processor manufacturing variability. In: Proceedings of the ACM International Conference on Supercomputing. pp. 296–307 (2019)

[47] Hu, Q., Sun, P., Yan, S., Wen, Y., Zhang, T.: Characterization and prediction of deep learning workloads in large-scale gpu datacenters. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–15 (2021)

[48] D'Amico, M., Gonzalez, J.C.: Energy hardware and workload aware job scheduling towards interconnected hpc environments. IEEE Transactions on Parallel and Distributed Systems (2021)

[49] Khan, N.K., et al.: Energy measurement and modeling in high performance computing with intel's rapl (2018)

[50] Saurav, S.K., GL, G.P., Chauhan, M.: Adaptive power management for hpc applications. In: 2016 2nd International Conference on Green High Performance Computing (ICGHPC). pp. 1–7. IEEE (2016)

[51] Patel, T., Wagenhäuser, A., Eibel, C., Hönig, T., Zeiser, T., Tiwari, D.: What does power consumption behavior of hpc jobs reveal?: Demystifying, quantifying, and predicting power consumption characteristics. In: 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 799–809. IEEE (2020)

[52] Feitelson, D.G., Rudolph, L., Schwiegelshohn, U., Sevcik, K.C., Wong, P.: Theory and practice in parallel job scheduling. In: Job Scheduling Strategies for Parallel Processing: IPPS'97 Processing Workshop Geneva, Switzerland, April 5, 1997 Proceedings 3. pp. 1–34. Springer (1997)

Predicting the power consumption can be quite hard to achieve. For instance, in the case of GPUs, some applications can result in high power consumption variability when running in distinct GPUs of the same model and vendor.

Our work presented in this subsection is in the category of power capping for power-aware resource management in power constrained platforms. Power capping consists in adjusting the computing nodes to consume less power than its allowed maximum by employing, for instance, Dynamic Voltage and Frequency Scaling (DVFS).

We distinguish ourselves from previous works in power capping because we rely on the users' engagement to run applications under power cap. Engaging users results in a simple power-aware resource management method that does not need any a priori characterization or prediction of the power consumption of the applications, making it easier to deploy in production platforms. The principle is straightforward: we can comply with the power cap if enough users volunteer to run their applications under slight levels of power cap. No need for complex power-capping decision-making from the side of the platform.

*4.1.2 **Preliminary definitions***

We consider an HPC platform as a computing cluster with computing nodes that are connected by a certain interconnection topology. Each computing node has one or more CPUs and also accelerators (e.g., GPUs). In the context of this paper, we consider the accelerators as only GPUs. The computing clusters are homogeneous in the context of the computing nodes. That is, all computing nodes have the same number and model of CPUs and GPUs.

During the platform's operation, several users submit applications, hereafter referred to as jobs. To deal with the jobs' submission and processing on the platform, we use OAR, a management system called Resources and Jobs Management System (RJMS) that runs on the platform. OAR is the main interaction point between the users and the HPC platform. Users can submit applications to OAR at any point in time, and OAR has no ahead information about which jobs will arrive. The literature refers to this job submission characteristic as online job submission. We configure OAR to assign the processing order of the jobs in First Come First Served (FCFS) order. FCFS is the baseline job queue ordering heuristic in the popular backfilling scheduling algorithm.

An electricity provider powers the platform with electricity. We consider that there are certain time periods where the electricity provider is not able to fully power the platform, resulting therefore in power capped periods during the platform's operation. In the real-world, this cap can come from sources such as (i) general electricity grid overdemand due to an external event, and (ii) temporary reduction of the renewable power capacity of the electricity provider. The electricity provider informs the platform maintainers in advance about when and how much the HPC platform will be power capped to accommodate the peaks of demand.

OAR is also responsible for managing the jobs that are currently running in the HPC platform. Therefore, once a power capped period arrives, it is up to OAR to deploy measures to comply with the power cap. We elaborate on these measures in the section below.

*4.1.3 **Proposed methodology***

Methods to comply with a power cap

In production clusters such as Marconi100, the scheduling algorithm used is usually a variant of FCFS, or EASY-bf. As such, in the need to set a power cap, production clusters can adopt a variant of their preferred scheduling algorithm that would either kill jobs arbitrarily, or make a reservation for machines to shut down, such that the maximum power draw of the remaining machines is below the powercap. We investigate in this subsection two scheduling algorithms:

FCFS_killer (Baseline): kills jobs until the power cap is reached, newest job first. Upon job termination, OAR can shut down the computing nodes previously allocated by the killed job, thus saving power. This method is arguably the easiest to be deployed in practice, but it can be very intrusive, affecting the Quality of Service (QoS) of the platform, since FCFS_killer risks losing all the computation that was being performed by the killed job. We decided to select the newest job first in an attempt to reduce the wasted energy caused by killing.

FCFS_eco_mode (Our contribution): on the job submission, users may flag their jobs, indicating that the job can be run in a slowed down state. We hereafter refer to this flagged job as EcoJob. During the power-cap period, OAR looks for flagged jobs that are running, and uses DVFS to proportionally slow down the nodes assigned to these flagged jobs, until the power cap is reached. OAR slows down the assigned nodes up to a lower limit of 50% of the maximum power of the node. If all possible nodes are slowed down to this limit and the power cap is still not satisfied, a FCFS_killer routine starts, which will kill jobs until the power cap is satisfied. Non-flagged jobs will be prioritized to be killed by the FCFS_killer routine.

There are 4 events that the schedulers need to react to in a power-capping setting. FCFS_eco_mode reacts to these events as follows:

- set DVFS to the lowest mode kill jobs, non-EcoJobs first, newest first increase DVFS until powercap is met
- set DVFS to the highest mode
- Liberate resources Reset corresponding machines' DVFS state Execute jobs from queue as long as they fit in power and resources
- Add to queue Execute jobs from queue as long as they fit in power and resources

The idea behind FCFS_eco_mode is to let the user decide to trade-off performance (i.e., let their jobs run slower) to increase the chances that their jobs will not be killed. FCFS_eco_mode tries to engage users as a front line measure to comply with the power cap, potentially avoiding degrading the QoS with job kills. It is important to emphasize that this idea behind FCFS_eco_mode is agnostic for any kind of scheduling algorithm, since it concerns the jobs that are already running.

To go beyond FCFS, we also investigated a modified implementation of EASY backfilling with job power prediction. The difference with the standard EASY backfilling algorithm is that we only backfill a job if its power consumption will not go beyond the power budget at any time during its execution. Our focus with this modified implementation was not to react in real time to the real power consumption at a fine grained level as we did in the rest of this subsection, but to assess the precision of different power estimations for jobs and the power consumption achieved during the execution, compared to the predicted available power. Results on this are presented in Deliverable 1.4.

Understanding the effects of FCFS_killer and FCFS_eco_mode by experimenting them in real-world platforms is risky, onerous, and time-consuming. We address this hindrance by performing simulation experiments of operating an HPC platform. In the next section, we present the details to achieve in this simulation.

Simulating a supercomputer under power cap

Data sources for simulation

It is a common practice from supercomputer maintainers to register information about the operation of the platform, notably details about the jobs in the form of workload traces (e.g., number of requested processors, arrival time, run time). An example is the workload traces present in the parallel workloads archive[53]. However, such traces do not contain information about the power consumption of the computing nodes during the platform's operation, which hinders the task of simulating the workload traces taking into account the energy consumption and power cap.

To overcome this hindrance, we exploit a recent dataset from the Marconi100 supercomputer. This dataset not only contains the aforementioned information about the jobs, but it also contains time-series data about the power consumed by the computing nodes.

On top of the common data collection for job information in a Job Table, two plugins were used to collect energy information, IPMI and Ganglia. The IPMI (Intelligent Platform Management Interface) plugin serves as a data collection tool, retrieving information from the Out-of-Band (OOB) management interface, specifically the Baseboard Management Controller (BMC) of the computing nodes. The BMC is a hardware component embedded in servers or cluster nodes that facilitates remote monitoring and management of the system. The IPMI plugin gathers sensor data – notably the power consumption – from the BMC installed in the nodes. The Ganglia plugin serves as an energy monitoring tool for most components of each node. In particular, it recorded the energy use of each GPU for every node, with a similar frequency to IPMI on the CPUs.

[53] Feitelson, D.G., Tsafrir, D., Krakov, D.: Experience with using the parallel workloads archive. Journal of Parallel and Distributed Computing 74(10), 2967–2982 (2014)

To establish at which points in time the HPC platform will be power capped, we considered a use-case from the available electricity demand data from RTE (Réseau de Transport d'Electricité). RTE's data gives us real-time data and history on the electricity demands in France. With this data we can observe regular daily peaks around 6PM to 8PM. Therefore, in our use-case we suppose that the power cap periods happen every day between 6PM and 8PM.

Preparing data for simulations

By crossing the timestamp information about the start/end time of the jobs (Job Table data) and the time-series data of the computing nodes' power consumption (IPMI Data and Ganglia Data), we can extract time-series data of the power consumption of the jobs. Figure 31 illustrates a few examples of the extracted jobs' power consumption data. From this per-job power consumption data, we extract the power consumed by the CPUs and GPUs during the jobs' execution as a time-series data, sampled once every 20 seconds.

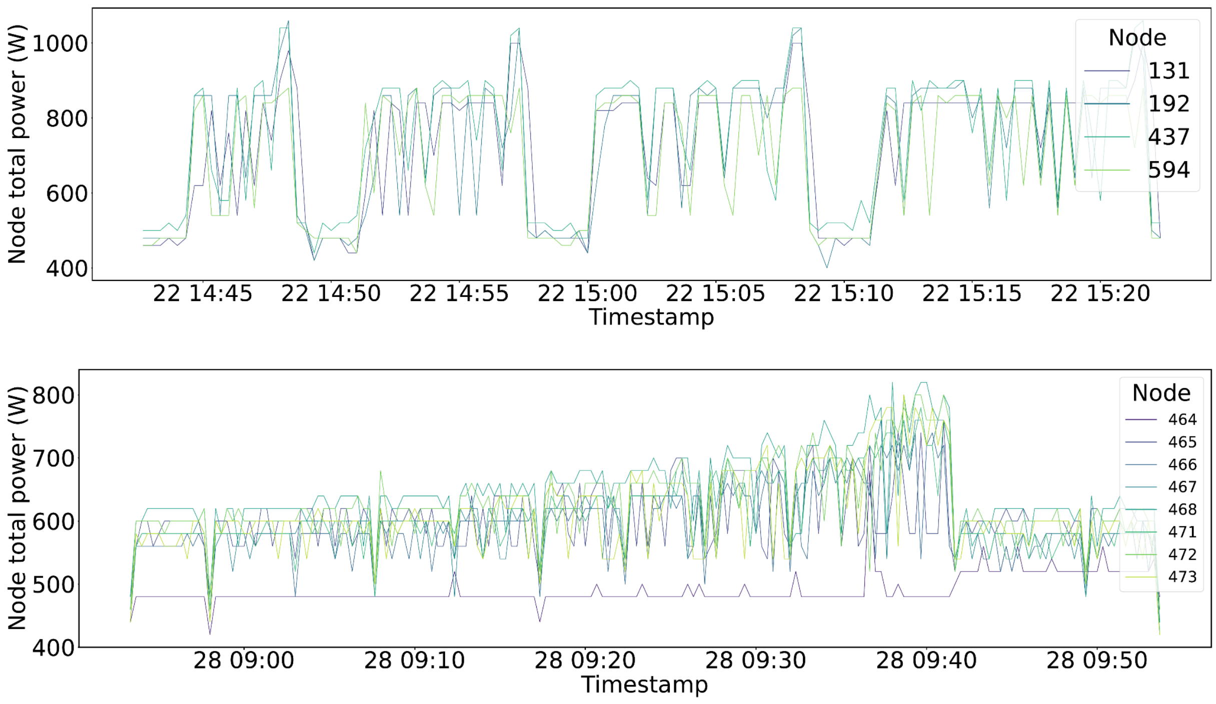


Figure 31: Examples illustrating the power consumption profiles of nodes of two jobs, one job per graph, obtained from the Marconi100 dataset. Each job is a multi-node application, i.e., four nodes in the top graph and eight nodes in the bottom graph.

Those power consumption and processor utilization time series are then used in simulation. As such they allow us to simulate DVFS in a non-trivial fashion. The jobs are more than the simplistic rectangular job of constant resource use, and the machines have a more realistic behaviour thanks to the non-linearity of their model.

After this conversion, the jobs are now modeled as a sequence of amounts of computing to be processed for every 20 seconds. This new modeling of the jobs constitutes a more fine-grained computing profile of jobs, which gives us flexibility to simulate the DVFS impact in the jobs in precise time windows during the jobs' execution, and therefore to simulate the effects of a power cap on the jobs. The power state employed for task execution is contingent upon its inherent characteristics. Tasks characterized by significant input/output (I/O) operations exhibit differing power state requirements compared to those involving intensive numerical computations. Consequently, the execution time of I/O-bound tasks experiences less pronounced sensitivity to reduced clock frequencies, in contrast to computationally intensive tasks.

Configuring the simulations

We use Batsim for all our simulations. As a simulator, Batsim is able to modulate parameters such as frequency and power during simulation, mimicking a DVFS behavior. Batsim takes as inputs a description of the targeted platform, a workload and a scheduler process.

The platform has been made to resemble Marconi100. As such, it has 5 batsim computing nodes, 1 representing the CPUs and 4 representing the GPUs, for each real Marconi100 nodes. The lack of publicly available studies on Marconi100 components and the inherent limitations of production environments in revealing job details necessitate assumptions for converting Marconi100 traces to a batsim workload. As we lack most of the specifics of each job, we decided to ignore communications, and focus only on computations. Fortunately, analysis by Zacharov et al. enables direct conversion of power consumption to GFlop/s for GPUs. However, no such straightforward method exists for CPUs. Therefore, we scaled a CPU with known DVFS power consumption to match the idle and maximum power of Marconi100 CPUs. This approach yields a batsim platform emulating Marconi100, featuring an approximation of DVFS for both CPUs and GPUs.

In our scenario, we imagine that the electricity provider imposes a power cap on the platform at specific times. This power cap is given as dates in a configuration file, specifying for each change the time and the new power cap. This power cap information is then used by the scheduler to place, slow or kill jobs when needed.

We exploit Batsim's composed jobs representation to use the calculated computing profile time-series data as workload inputs for the job submission into the simulation. To fairly compare between all the schedulers and parameters, we dynamically submitted jobs to the platform to simulate users behavior for a set amount of time (10 days in our experiments). Submitting jobs dynamically instead of replaying a static workload with fixed submission times is important, as it allows us to regulate the job queue, preventing some algorithms from having the unfair advantage of being able to select from a large pool of jobs simply because they mismanaged the beginning of the schedule and created a large backlog.

Additionally, comparisons between different power cap settings would yield limited insights if the workload remained static across all scenarios. Replaying the same job submissions under varying power caps would primarily reflect the available energy, rather than provide meaningful differentiation based on the evaluated parameters. Results are fully detailed in the evaluation deliverable.

4.2 Application level Power-Capping with BEO

Most future ExaScale supercomputing centres, not to say all, will heavily depend on power capping functionalities. Indeed, with the rising energy costs and the threat of frequent energy shortages, the power consumption of exaflopic supercomputers (22.7 MW for Frontier[54]) makes it nearly impossible to operate them at their peak levels of performance on a daily basis.

Thus, having the possibility to limit the power consumption of some of the components of HPC systems, for instance by enforcing power caps for both processors and accelerators, is becoming a must-have. However, enforcing a power cap on a computing component has a significant impact on the level of performance it exhibits: the lower the power cap in Watts (and hence, the more constraining the power cap), the greater the reduction of its computing power. Yet, the main goal of supercomputing centres is to reach the highest job throughput possible, which is at odds with enforcing power capping since it may induce severe performance degradations. That is why several efforts have emerged, notably from the academic world (for instance GEOPM[55] and PShifter[56]), to make power capping smarter and reduce the performance penalty it may entail.

At the beginning of the REGALE project, Bull Energy Optimizer (BEO) implemented the enforcement of basic power capping on the compute nodes manufactured by Atos/Eviden, and offered several additional features revolving around this functionality, notably including:

- The possibility to create collections of power capping rules which can later be activated or deactivated;
- The possibility to give a power budget for a group of components, which is then automatically translated to a set of individual capping rules for the components, with a fair distribution of the power budget between the latter (i.e. a percentage of their nominal power consumption);
- For a subset of equipment, an experimental determination of the effective lowest power cap, that is to say the strongest power constraint which still allows the capped component to operate. Indeed, some technically achievable values of power capping can make the target component unresponsive.

However, EO did not offer any mechanism to mitigate the potential performance degradation of the execution of an HPC application induced by the enforcement of a power cap.

That is why we designed a power capping mechanism, called “Application-Aware Power Capping” (AAPC) to specifically address this issue.

[54] TOP500 system page for Frontier: https://www.top500.org/system/180047/

[55] J. Eastep et. al., Global Extensible Open Power Manager: A Vehicle for HPC Community Collaboration on Co-Designed Energy Management Solutions. In Proceedings of the 32nd ISC High Performance Conference, 2017

[56] N. Gholkar et. al., PShifter: feedback-based dynamic power shifting within HPC jobs for performance. In Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '18), 2018.

The remainder of this section presents the latter mechanism in depth, starting from an overview of its architecture and of the hardware and software ecosystems in which it integrates. It also contains the description and specification of all the algorithmic elements required to implement the AAPC mechanism.

Rationale of AAPC

To begin with, let's specify the use case to be addressed by the Application-Aware Power Capping (AAPC) mechanism. In a few words, it is based on the following empirical observation: HPC applications exhibit a wide range of workloads, which all have different sensibilities to power capping regarding performance. Indeed, on the one hand, some applications/use-case pairs, for instance NEMO TOP/PISCES solver applied to GYRE[57], tend to be heavily memory-bound. As a consequence, they do not constantly require the computing cores to consume their whole nominal power budgets to execute the workload associated with the application. Thus, when considering the average behaviour of the application, the nodes which execute it can be constrained by a power cap with only moderate impact on the performance of the application. On the other hand, the execution of applications which are much more compute-bound, such as HPL[58], highly and almost permanently stresses the computing cores. As a result, decreasing the raw computational power exhibited by the nodes, for instance by enforcing a power cap on them, induces a significant performance degradation. That is why compute-bound applications tend to be greatly sensitive to power capping.

Based on those experimental observations, the rationale associated with the AAPC mechanism consists in leveraging knowledge about the applications executed on a partition of compute nodes at a given time to dynamically redistribute the power budgets allocated to the jobs. The goal is to favour compute-bound jobs so as to increase the job throughput of the considered partition, when compared to the standard "Fair Sharing Power Capping" (FSPC) strategy, which is described in depth later on. Indeed, by trying not to power-constraint the nodes executing compute-bound jobs, heavy performance degradations for the associated applications could be avoided, which should tend to increase the job throughput for the considered partition. On the contrary, the shift of power budget from nodes allocated to memory-bound jobs to nodes allocated to compute-bound jobs should tend to increase the performance degradations for the applications associated with the former jobs. This will translate to a decrease of the job throughput for the considered partition. However, as explained beforehand, the avoided performance degradation for compute-bound jobs should positively counterbalance the induced performance degradation for memory-bound jobs. Hence, on average, at the scale of a power-constrained partition of nodes, the job throughput should be increased by the AAPC mechanism when compared to the FSPC strategy.

[57] NEMO home webpage: https://www.nemo-ocean.eu/
[58] HPL home webpage: https://netlib.org/benchmark/hpl/

Integration of AAPC in an HPC software architecture

An overview of the software architecture associated with the implementation of the AAPC mechanism, and of how it integrates in the management stack of a supercomputer is presented by Figure 32. At the heart of the AAPC mechanism lie two main components, which interactions are described by the sequence diagram presented by Figure 33:

- An AAPC extension plugin for the Resource and Job Management System (RJMS);
- An AAPC module for Bull Energy Optimizer (BEO).

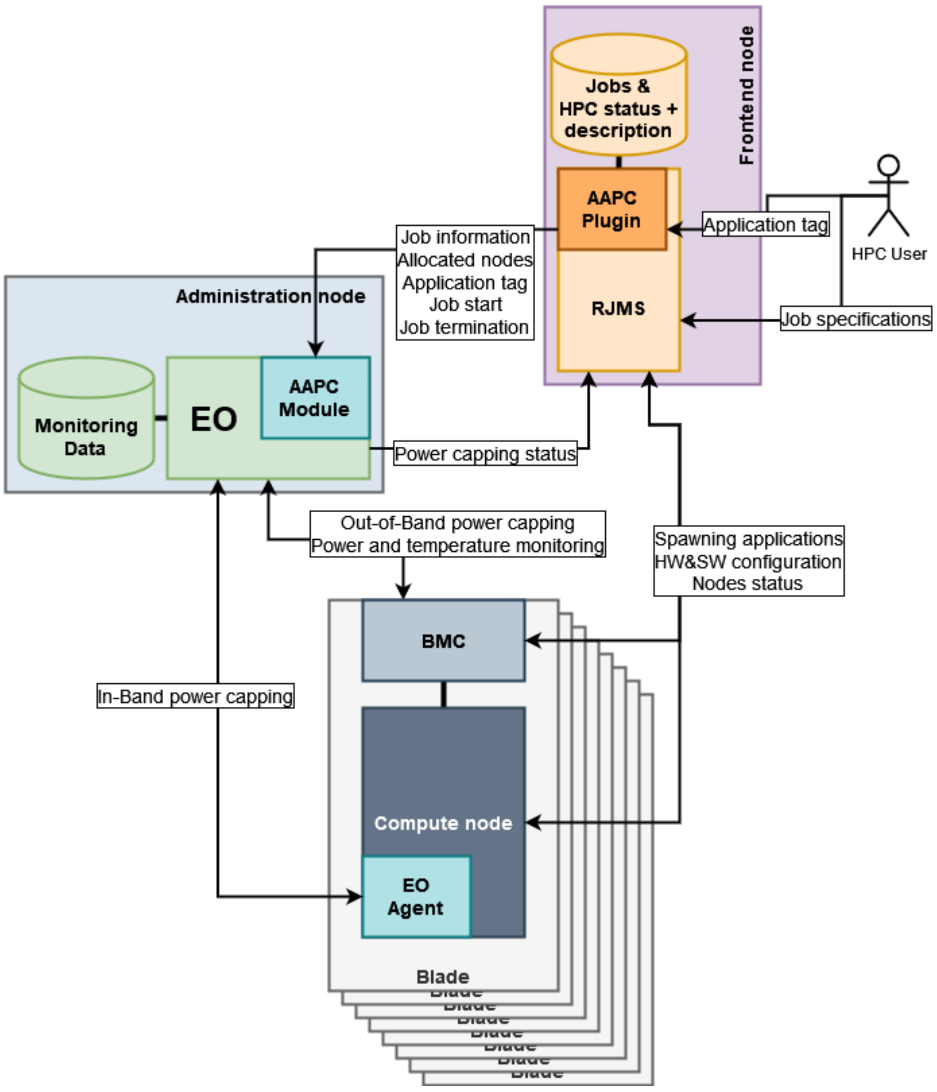


Figure 32: Overview of the architecture of the Application-Aware Power Capping (AAPC) mechanism, and of its integration in the management software stack of a supercomputer.

The role of the former is to notify the latter of two kinds of events, namely the start and the termination of jobs. The notifications are accompanied with several information about the concerned job: its name and ID, the list of nodes allocated to the job, and the tag of the application associated with the job (more details about the tags later on). The AAPC module for EO can thus build and maintain an internal representation of the state of the partition of compute nodes it manages regarding power capping. Incidentally, the power budget for the aforementioned partition is specified by the administrator of the supercomputer and is used as input by the AAPC module for EO. Thanks to those pieces of information, it is possible for the AAPC module for EO to compute a sharing of the power budget between the nodes, and to update it at each job start or termination.

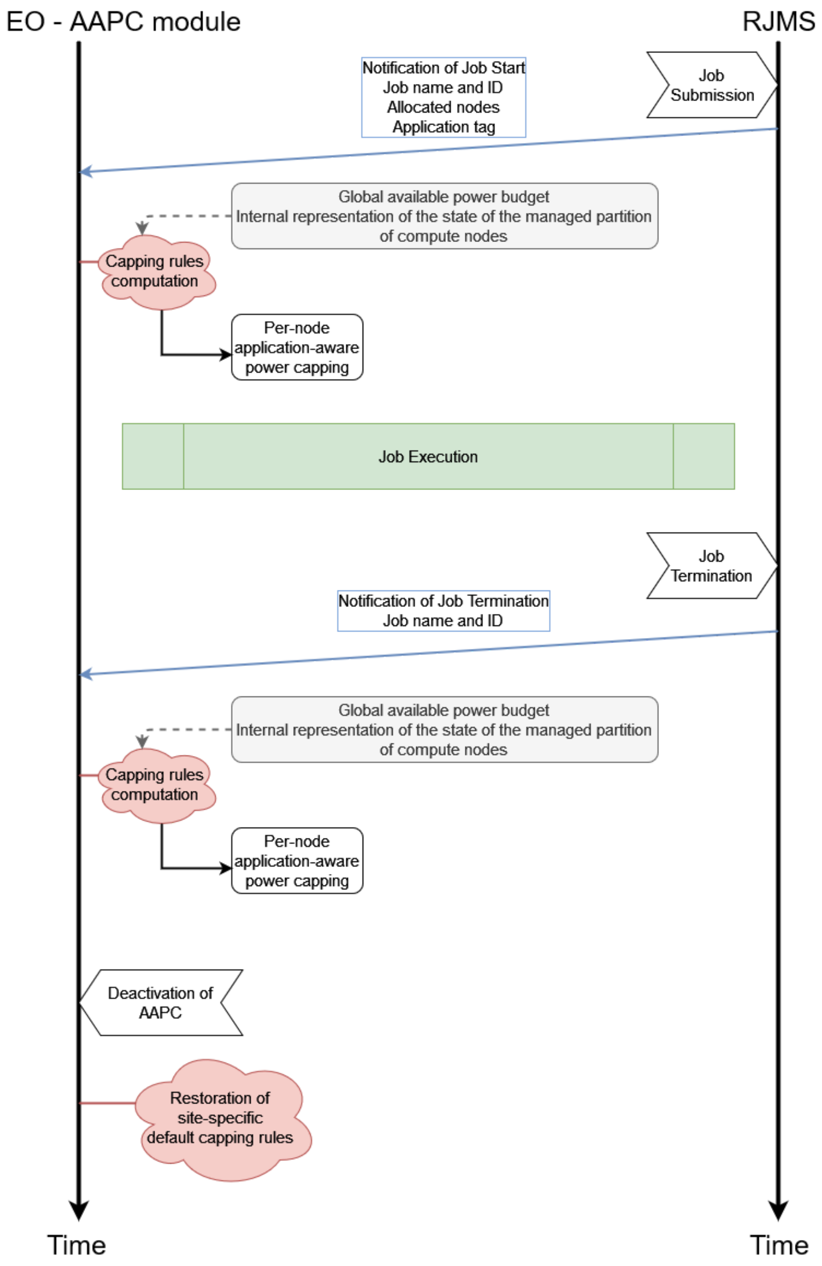

EO - AAPC module
RJMS
Notification of Job Start
Job name and ID
Allocated nodes
Application tag
Job Submission
Global available power budget
Internal representation of the state of the managed partition of compute nodes
Capping rules computation
Per-node application-aware power capping
Job Execution
Job Termination
Notification of Job Termination
Job name and ID
Global available power budget
Internal representation of the state of the managed partition of compute nodes
Capping rules computation
Per-node application-aware power capping
Deactivation of AAPC
Restoration of site-specific default capping rules
Time
Time

Figure 33: Sequence diagram describing the interaction between the Resource and Job Management System (RJMS), and the Application-Aware Power Capping (AAPC) module of Energy Optimizer (EO).

Using the features implemented by the core engine of BEO, it is then possible to dynamically create, update and enforce power capping rules on the compute nodes. It should also be noted that in case of deactivation of the AAPC mechanism, the site-specific default power capping rules should be enforced again.

Algorithmic elements about AAPC

This third subsection presents a subset of algorithmic elements related to the design of the AAPC mechanism, starting with the specification of a prerequisite regarding the power budget allocated to the managed partition of compute nodes. Then, a focus will be performed on application tags, and some elements revolving around it. Finally, the algorithm underlying the AAPC mechanism will be detailed.

Prerequisite: minimum required power budget

Application profiles and tags

Let's define what application tags are. In a few words, as explained beforehand, HPC applications exhibit a wide range of workloads which can be classified according to several different taxonomies. The one used in the context of this work about the AAPC mechanism defines three categories of applications:

- COMPUTE tag: The application is mainly compute-bound, and hence its performance tends to be heavily degraded under power cap;
- MEMORY tag: The application is mainly memory-bound, and hence its performance tends to be lowly degraded under power cap;
- MIXED tag: The application exhibits several interlaced behaviours, and hence the impact of a power cap on its performance tends to be moderate on average.

From the point of view of the AAPC mechanism, the tag associated with an HPC application defines a soft relative lower bound on the power constraint to be applied to a node executing the latter.

Relative, since the associated power cap is defined as a percentage of the nominal power consumption of the node. Soft, because the AAPC mechanism tries to find a sharing of the power budget allocated to the partition which makes it possible for each individual power cap enforced on the compute nodes to be compliant with the tags associated with the jobs they execute, but it can set lower power constraints if such a sharing does not exist. On top of that, the application tags are used to build a priority order regarding nodes to be power-capped: nodes executing an application tagged as MEMORY should be power-constrained before nodes executing an application tagged as COMPUTE. More details about this last point in the next subsection.

At this stage of the design of AAPC, HPC users who submit jobs will be trusted to specify the tag of the job. However, more refined techniques (e.g. AI/ML based) to infer the tag of a job are being explored, and could be integrated in this architecture by interfacing with both the RJMS and the AAPC module.

Description of the algorithm

To begin with, this subsection notably relies on schemas representing examples of power budget sharing, so as to illustrate the description of the AAPC mechanism. These schemas are presented by Figure 34, Figure 35, Figure 36, and Figure 37. On the latter, each node is represented by a square containing four pieces of information regarding the node:

- PWR: its average power consumption (in Watts) on a coarse time unit (e.g. 5 minutes);
- CAP: the power cap (in Watts) enforced on the node;
- MOP: its Minimal Operation Power (in Watts), which is the lowest value for a power cap enforced on the node so that it should still operate properly;
- TDP: its Thermal Design Power (in Watts), which is the thermal power the cooling system associated with the node should be able to dissipate so that the computing components (e.g. CPUs, GPUs, ...) of the latter could operate at their nominal performance level safely. TDP is quite commonly regarded as an estimation or an accurate upper bound of the power consumption of computing components while operating at their nominal performance level under load.

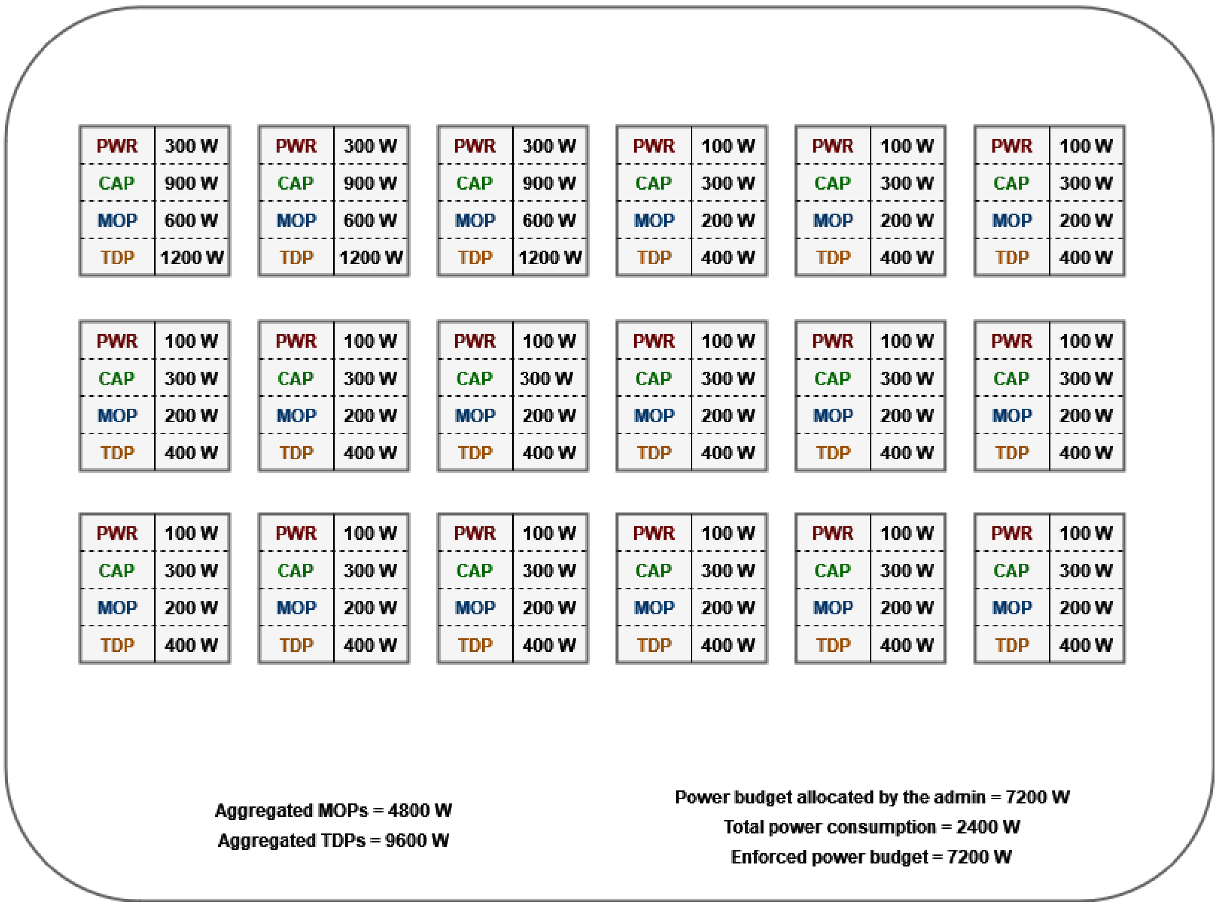


Figure 34: Example of the power budget sharing for a partition of idle compute nodes (in grey), according to the Fair Sharing Power Capping (FSPC) strategy.

Additionally, idle nodes are in grey, active nodes in green if the power cap enforced on them does not limit the power consumption associated with the execution of an application, and in red if it does. In other words, nodes in green are active and the power caps enforced on them does not significantly degrade the performance of the application they execute. On the contrary, the nodes in red are active and the power caps enforced on them limit their power consumptions, which has a significant impact on the performance of the application they execute. Finally, jobs are represented in blue if the associated applications are tagged as memory-bound, in purple if they are tagged as compute-bound, and in orange if they are tagged as exhibiting mixed behaviours.

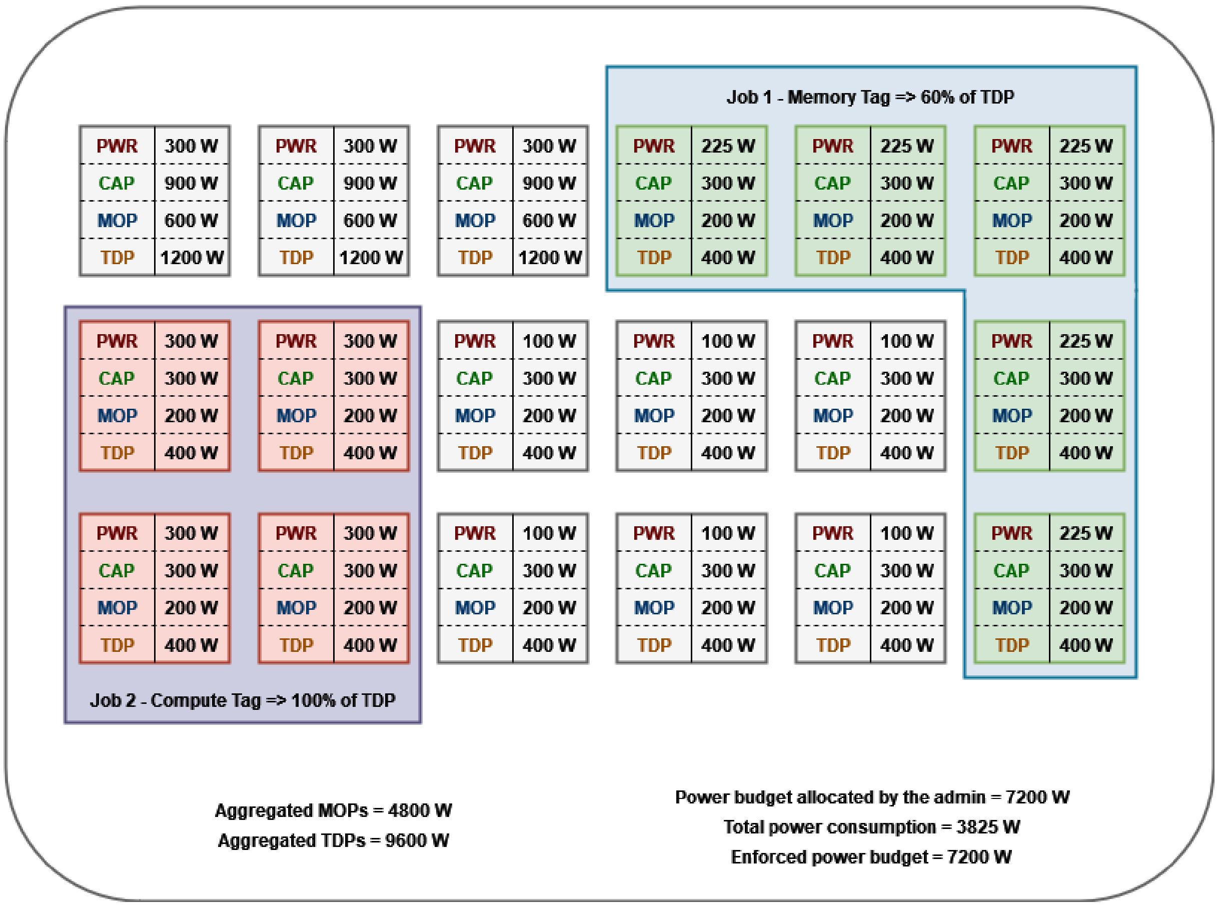

Job 1 - Memory Tag => 60% of TDP
PWR 300 W CAP 900 W MOP 600 W TDP 1200 W
PWR 300 W CAP 900 W MOP 600 W TDP 1200 W
PWR 300 W CAP 900 W MOP 600 W TDP 1200 W
PWR 225 W CAP 300 W MOP 200 W TDP 400 W
PWR 225 W CAP 300 W MOP 200 W TDP 400 W
PWR 225 W CAP 300 W MOP 200 W TDP 400 W
PWR 300 W CAP 300 W MOP 200 W TDP 400 W
PWR 300 W CAP 300 W MOP 200 W TDP 400 W
PWR 100 W CAP 300 W MOP 200 W TDP 400 W
PWR 100 W CAP 300 W MOP 200 W TDP 400 W
PWR 100 W CAP 300 W MOP 200 W TDP 400 W
PWR 225 W CAP 300 W MOP 200 W TDP 400 W
PWR 300 W CAP 300 W MOP 200 W TDP 400 W
PWR 300 W CAP 300 W MOP 200 W TDP 400 W
PWR 100 W CAP 300 W MOP 200 W TDP 400 W
PWR 100 W CAP 300 W MOP 200 W TDP 400 W
PWR 100 W CAP 300 W MOP 200 W TDP 400 W
PWR 225 W CAP 300 W MOP 200 W TDP 400 W
Job 2 - Compute Tag => 100% of TDP
Aggregated MOPs = 4800 W
Aggregated TDPs = 9600 W
Power budget allocated by the admin = 7200 W
Total power consumption = 3825 W
Enforced power budget = 7200 W

Figure 35: Example of power budget sharing for a partition of nodes executing two jobs, according to the Fair Sharing Power Capping (FSPC) strategy.

As stated earlier, the goal of the AAPC mechanism is to dynamically share the power budget allocated to a partition of nodes according to the set of jobs executed on this partition so as to increase the associated job throughput when compared to the Fair Sharing Power Capping (FSPC) strategy. The latter consists in statically enforcing a power cap on each node of the partition so that each node should be constrained to the same percentage of its nominal power consumption. An example of power budget sharing according to the FSPC strategy is represented by Figure 34. The power envelope allocated to the partition being equal to 75% of the aggregated nominal power consumptions of the nodes, each node is capped to 75% of its individual nominal power consumption.

That being said, the algorithm designed to implement the AAPC mechanism and make it a smarter and more efficient power capping strategy when compared to FSPC can now be presented. The first step of the update of the power capping rules by the AAPC mechanism consists in checking that the power budget allocated to the managed partition abides by the prerequisite specified earlier, namely that it is sufficient for all the nodes of the partition to consume their Minimal Operation Powers (MOPs).

Then, the first path of the AAPC mechanism corresponds to the case where the power budget allocated to the managed partition is greater or equal to the sum of the aggregated MOPs of the idle nodes and of the aggregated TDPs of the active nodes. In this case, all the active nodes can operate at their nominal power consumption, without any degradation in the performance of the executed applications.

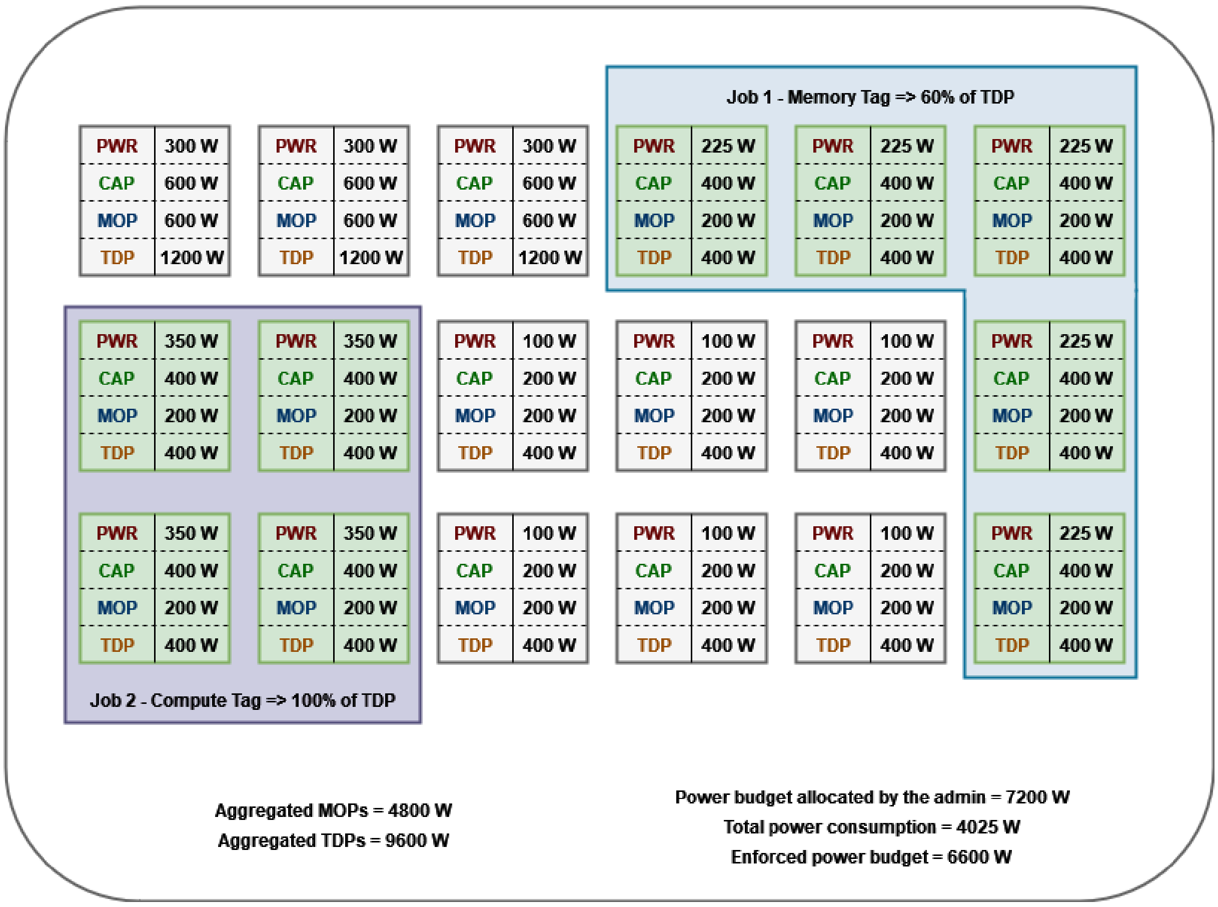

Job 1 - Memory Tag => 60% of TDP
Job 2 - Compute Tag => 100% of TDP
PWR
CAP
MOP
TDP
300 W
600 W
1200 W
225 W
400 W
200 W
350 W
100 W
Aggregated MOPs = 4800 W
Aggregated TDPs = 9600 W
Power budget allocated by the admin = 7200 W
Total power consumption = 4025 W
Enforced power budget = 6600 W

Figure 36: Example of power budget sharing for a partition of nodes executing two jobs, according to the Application-Aware Power Capping (AAPC) mechanism.

As illustrated by Figure 35 and Figure 36, this first path is already a significant advantage of AAPC over FSPC when it comes to power capping strategies. Indeed, by shifting power budget from idle nodes, which do not require it, to active nodes, the AAPC mechanism makes it possible to avoid performance degradations due to power capping for compute-bound jobs.

Regarding the main path of the AAPC mechanism, the first thing to say is that it consists in an iterative refinement process of the individual power caps which takes into account the tags of the executed applications. Since the power caps are iteratively refined, they are initialised to the TDPs of the nodes. Then, each refinement step will potentially update those intermediary power caps. When the refinement process terminates, power capping rules are created by the AAPC module and enforced by BEO.

As explained beforehand, the rationale of the AAPC mechanism is to power-constrain first the nodes executing jobs which are less affected by power capping so as to let the more power-hungry jobs run uncapped. On top of that, since the exhibited level of performance for a given power cap can vary greatly as demonstrated by Pedretti et al.[59], it seems well advised to start by enforcing power constraints on jobs to which less nodes are allocated. Indeed, due to the fact that they are executed on less nodes, the performance disparities between the power capped nodes are less likely to induce additional performance degradations during the synchronisation steps.

Let's now focus on the iterative refinement process implemented by the AAPC mechanism to share the power envelope allocated to the partition it manages between the nodes belonging to it. It can be outlined, sequentially, as follows:

(1) To begin with, each idle node is capped to its MOP, and each active node is temporarily capped to its TDP. Since the refinement process is executed because the partition power budget is not enough for all active nodes to consume their TDPs, it means that the total power balance is less than 0. The goal of the refinement process is to cap the active nodes, one job at a time, to make this power balance equal to 0.

(2) Then, iterate over all the jobs, one at a time and sequentially, to enforce power caps on their allocated nodes. The aforementioned power caps are called “tag-related power caps”. In short, it means that the power constraint enforced on a node allocated to a job is equal to a percentage of its TDP, the percentage depending solely on the tag (e.g. 90% of TDP for MIXED jobs). If, after power capping a job, the power balance is greater or equal to 0, then the refinement process and hence the AAPC algorithm terminates (note that if the power balance is strictly greater than 0, the excess power budget is fairly shared between the nodes allocated to the lastly capped job).

(3) If, at the end of this first refinement iteration when all the jobs are tag-related power capped, the power balance is still lesser than 0, a second and last refinement iteration is performed. The same sorted list of jobs is parsed once more, one at a time and sequentially. This time, the nodes allocated to the considered job are power constrained at their MOPs. Similarly to what is done during the first iteration, the AAPC algorithm terminates when power balance is greater or equal to 0, and the excess power budget is fairly shared between the nodes allocated to the lastly capped job.

Note that since it is enforced that the global power budget makes it possible for all the nodes of the partition to consume their MOPs at the same time, this second iteration of the refinement process terminates, for sure. Once again, the rationale is to first and foremost power constrain MEMORY-tagged jobs, as they tend to be less impacted by power capping than COMPUTE-tagged jobs. This way, the job throughput at the scale of the partition should be better with the AAPC mechanism than with the FSPC strategy.

[59] K. Pedretti et al., "A Comparison of Power Management Mechanisms: P-States vs. Node-Level Power Cap Control," 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2018.

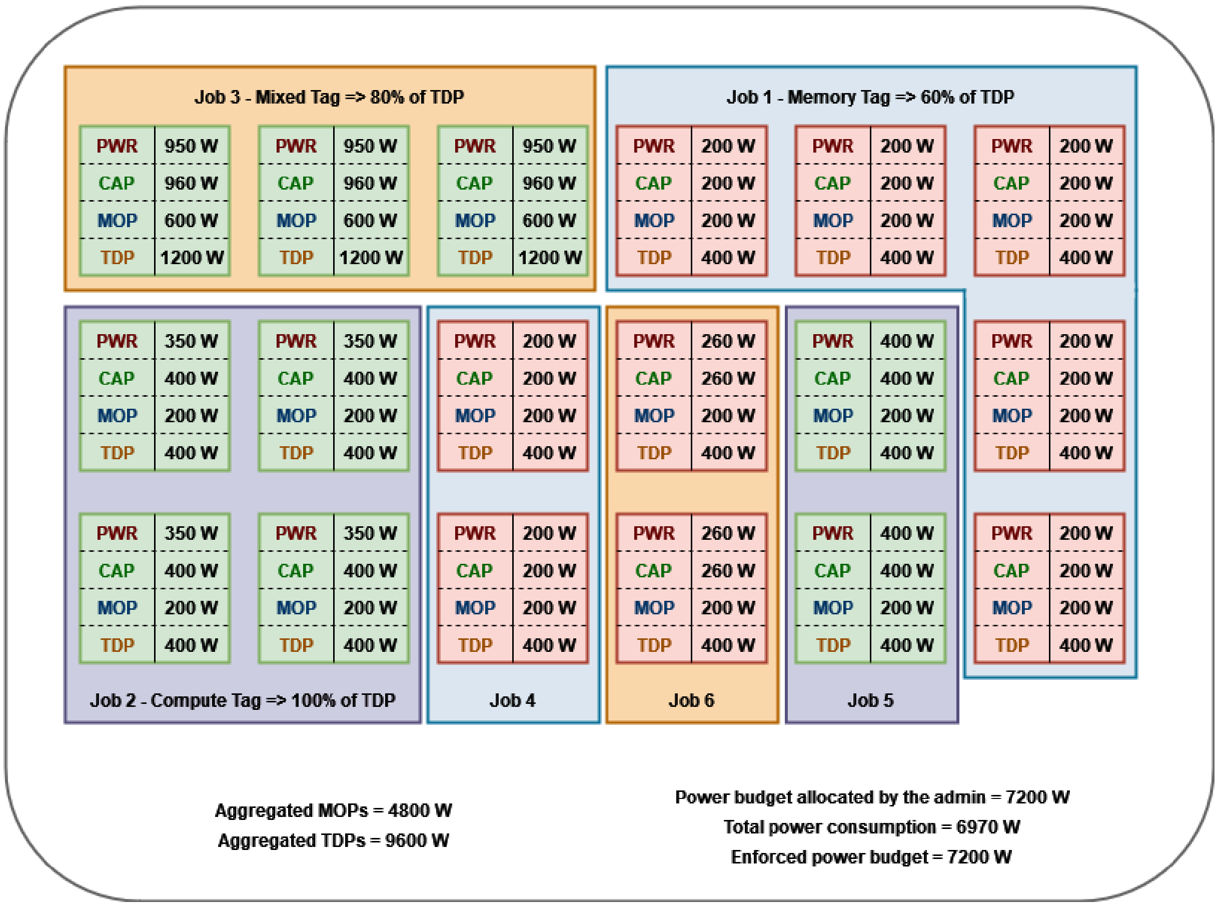


Figure 37: Example of power budget sharing for a partition of nodes completely loaded by the execution of six jobs, according to the Application-Aware Power Capping (AAPC) mechanism..

Figure 37 shows an example where the managed partition is fully loaded and needs to be power-constrained: all its nodes are active and all of them cannot consume their TDPs at once. At the end of the first iteration of the refinement process, pwr_left still was less than 0. Thus, the nodes allocated to the two MEMORY-tagged jobs were power capped to their respective MOPs. On top of that, the MIXED-tagged job N°6 had also to be power-constrained further than the tag-related power cap. However, it was not compulsory to cap its nodes to their MOPs and the excess power budget was fairly shared between the latter. When compared to FSCP, the same number of nodes are impacted by performance degradations due to power capping. However, here, out of the three impacted jobs, 2 are MEMORY-tagged and 1 is MIXED-tagged (the latter being only slightly impacted). In comparison, with FSCP, out of the three impacted jobs, 2 would be COMPUTE-tagged and 1 would be MIXED-tagged, and they would suffer significant performance degradations.

(4) Finally, the last step of the refinement process of the AAPC mechanism consists in enforcing the individual power caps on the nodes belonging to the managed partition.

Conclusion and possible refinements

To conclude this section, a short digression on the temporality of the enforcement of a power cap. Empirically, it was observed that a potentially significant delay could occur between the time a power cap is set on a compute node, and the time it is effectively enforced (i.e. a limitation of the power consumption of the node is observable, for instance thanks to BEO). This delay tends to increase with the load of the node. As a result, two additional features might need to be implemented in the AAPC mechanism:

- Enforcing the power caps which are lesser than the previously enforced ones first, so as to make room for the jobs that will be less constrained before granting them the capability to consume more power;
- A power cap should be considered as enforced by the AAPC mechanism only after an effective change regarding the frequency of the computing cores is observed.

Those two features could make the update of the power caps enforced on the nodes belonging to the managed partition counter-productive if it is performed too frequently. Thus, a minimal delay (e.g. 1 minute) between two updates might be necessary. However, the callbacks hooked on the events associated with the start and termination of a job should be executed normally, so as to maintain the internal representation of the state of the partition of the AAPC mechanism.

5 Combining Sophistications

To facilitate a deeper understanding of how the proposed sophistications detailed in the previous sections and integrated in the REGALE components interact, we will now map them to the overall REGALE architecture, as detailed in Section 3 of Deliverable 1.3 and provided here for reference in Figure 38. This analysis will go from the top of the Figure to the bottom, discussing the specific sophistications made to each module in the system and their interactions, both within the module itself and with the entire software stack.

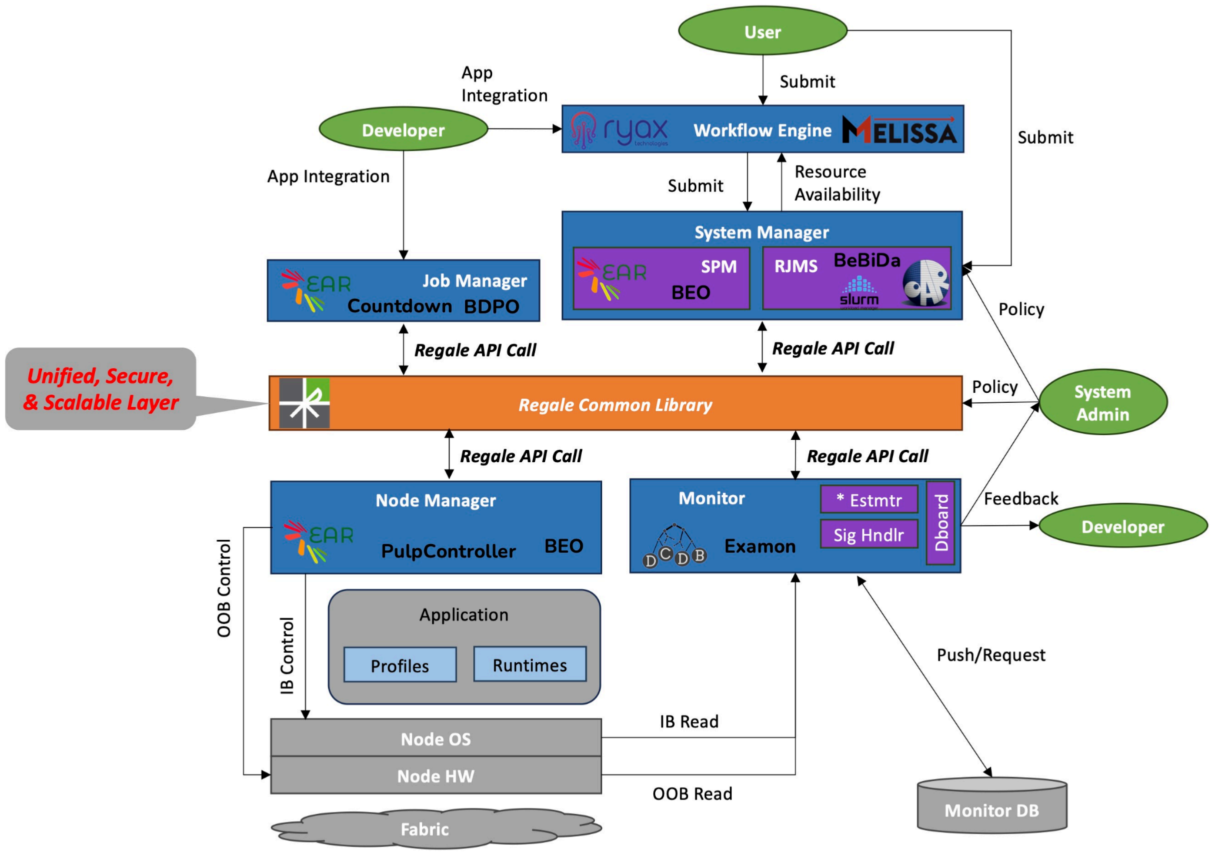


Figure 38: The final REGALE Architecture

Workflow Engine

Within this component, there are two sophistications aimed at improving the throughput, related to the Elastic Resource Management presented in Subsection 2.3 and Data-aware resource allocation presented in Subsection 2.4. While those sophistications were defined and analyzed in two different workflow engines (respectively Ryax and Melissa), the core concepts of the sophistications are applicable in both components. Similarly, concurrent execution of workflows deployed with either workflow engine is also possible without interference. Finally, there are no foreseeable conflicts with the other sophistications in the rest of the software stack.

System Manager

At the System Manager level, there are sophistications for all the sophistications objectives. For performance and throughput, Co-scheduling for throughput and application coupling at the node level was described in Subsection 2.1 and Co-scheduling within multicore processing units was described in Subsection 2.2. While combining these two sophistications on the same applications seem unrealistic due to the complexity required to manage processes at this level of granularity, these can coexist on a cluster provided the resource manager is able to manage both.

They are also compatible with the third sophistication which is Moldability for energy efficiency presented in Subsection 3.1, as this sophistication is designed to choose the best resource allocations for applications which can execute on a range of resources (as opposed to a fixed predetermined number of resources). Since the co-scheduling at the node level and this moldability approach were both integrated in OAR, their integration is straightforward. Another sophistication is brought through BeBiDa elastic resource management presented in subsection 2.3. This elasticity is brought without modifying the typical usage of system managers such as OAR or Slurm hence keeping the compatibility with all the other sophistications brought at this level.

Finally the last two sophistications are the System Level Power-Capping with OAR presented in Subsection 4.1 and the Application level Power-Capping with BEO presented in Subsection 4.2. As these sophistications are aimed at managing the global power consumption of the system by managing jobs power budgets, they are not conflicting with the other sophistications, as long as there is a communication and agreement through the REGALE common library on the power allocation to every job. They are however incompatible with each other, as they are based on a completely different philosophy: the first one is based on a volunteer scheme, where users can willingly allow the system power manager to reduce their power consumption if needed and manages power allocation with simple greedy heuristics, while the second is an automated approach where jobs characteristics are analyzed on the fly during execution to adjust the power knobs depending on which job can be executed at a lower power setting with the smallest penalty.

Job Manager and Node Manager

While these are two different components in our architecture, they are used jointly in both related sophistications: Node level power controls (BDPO) presented in Subsection 3.4 and Thermal and Power control on a node level (ControlPULP) presented in Subsection 3.5. These sophistications are highly specific to the software used on the platform, and thus are not directly compatible. They are also directly setting power consumption at the node level and job level, and thus need to closely report to the system power manager to avoid conflicting directives from the global sophistications and the more local sophistications explored here.

Monitor

At the monitoring level, the sophistications are aimed at a better understanding of the applications and power patterns, to help other components in their decision making. As these two sophistications ML-based User-Labelling of the Job presented in Subsection 3.2 and Integration of ML Models in Production System presented in Subsection 3.3 are defined as inputs to the other sophistications, they are naturally designed as compatible with most sophistications. They are even compatible with each other, as the first one is analyzing high level data and job activity logs, while the second one is directly integrated in the monitoring system.