MCBound: An Online Framework to Characterize and Classify Memory/Compute-bound HPC Jobs

Francesco Antici
University of Bologna
Bologna, Italy
francesco.antici@unibo.it

Andrea Bartolini
University of Bologna
Bologna, Italy
a.bartolini@unibo.it

Zeynep Kiziltan
University of Bologna
Bologna, Italy
zeynep.kiziltan@unibo.it

Ozalp Babaoglu
University of Bologna
Bologna, Italy
ozalp.babaoglu@unibo.it

Yuetsu Kodama
Riken Center for Computational Science
Kobe, Japan
yuetsu.kodama@riken.jp

***Abstract*—Modern High-Performance Computing (HPC) systems play a fundamental role in driving scientific research, as they execute computationally intensive jobs originating from diverse domains. However, HPC jobs are characterized by conflicting computational requirements, which may cause inefficiencies in resource usage, system throughput and energy consumption. One approach to tackling this problem is to distinguish between *memory-bound* and *compute-bound* jobs at their submission time, with the goal of making informed decisions about their execution. In this paper, we present *MCBound*, the first online data-driven framework to classify HPC jobs as *memory/compute-bound* before job execution, without user intervention. We propose a systematic characterization technique to generate a reference dataset from historical data for initial classification model training. Using the proposed characterization technique, we analyze the data of 2.2 million job runs on the Supercomputer *Fugaku*[1], a production HPC system installed at the RIKEN Center for Computational Science, in Japan. We implement *MCBound* for Fugaku and classify the jobs executed during February 2024. Our approach is proven effective, as it obtains an F1-macro average score of at least 0.89 as prediction quality, while incurring a negligible overhead on the system's operations. Our Python-based implementation of *MCBound* can be seamlessly configured and deployed in other HPC systems.**

I. Introduction

Modern High-Performance Computing (HPC) systems play a fundamental role in driving scientific research, as they execute computationally intensive jobs originating from diverse domains, ranging from genomics and computational chemistry, to weather forecasting and artificial intelligence. HPC jobs , like any application, can be classified based on the intensity of their system resource usage as *memory-bound* and *compute-bound* [8], [9]. The first category refers to the jobs whose attainable performance are bound by their memory access rate, often measured as the utilization level of the available system memory bandwidth. In contrast, the *compute-bound* refers to the ones whose performance is bound by the system's arithmetical performance, often measured as the rate of double-precision floating-point operations computed per second. As jobs are often not engineered to simultaneously saturate the different system resource types, failure in identifying their category prior to their execution is likely to cause inefficiency in resource usage, system throughput and energy consumption [9], [27], [38].

Conversely, knowing if a job is *compute-bound* or *memory-bound* upon submission allows making informed decisions about its scheduling and execution. For instance, it can be used to design specific hardware-software co-design techniques [5], [19], [20], [24], [34], or a job co-scheduling strategy that allocates the same node to jobs with different characteristics [8], [9]. Both techniques have been proven effective in enhancing system throughput, while significantly reducing system energy consumption. Therefore, classifying jobs as *memory-bound* and *compute-bound*, prior to their execution, has the potential to improve the system energy efficiency and throughput, without the need of any intervention by the user, as shown in [5], [8], [9], [19], [20], [24], [34].

To develop reliable classification models to predict the *memory/compute-bound* nature of a job before its execution, a large amount of labelled job data is needed. However, to the best of our knowledge, such a public dataset for a production system does not exist. Without prior knowledge on jobs' computational operations and memory usage, they can only be characterized by analyzing the performance metrics collected during the execution. This requires a systematic characterization technique leveraging the data collected during job execution. Job data analysis based on this characterization could also provide insights into the system usage; for instance, whether the users submit jobs optimized to fully saturate the different system resources and if specific actions can be enacted to improve system throughput.

Despite recognizing its importance, no past work has proposed a solution to systematically and seamlessly characterize and classify *memory/compute-bound* jobs in an HPC system before job execution, nor has proven the feasibility of such an approach. The contributions of this paper are as follows:

- We introduce *MCBound*, the first online data-driven

[1]https://www.fujitsu.com/global/about/innovation/fugaku/

SC24, November 17-22, 2024, Atlanta, Georgia, USA
979-8-3503-5291-7/24/31.00©2024IEEE

framework to classify HPC jobs before job execution as *memory-bound* and *compute-bound*, without user intervention.
- We propose a systematic characterization technique to generate a reference dataset from historical data for our initial classification model training. Using the proposed characterization technique, we analyze the data from 2.2 million job runs on the Supercomputer Fugaku to obtain insights into their *memory/compute-bound* characteristics.
- We employ *MCBound* to classify the jobs executed on Fugaku during February 2024, obtaining an F1-macro average score [32] of at least 0.89 as prediction quality.

The *MCBound* framework is online in the sense that it works in real-time on live streaming data and periodically updates the classification model in the background. The job characterization is performed systematically, leveraging job performance metrics, system's specifics, and the *Roofline* model [36] technique. The classification is achieved through a prediction algorithm relying on Natural Language Processing (NLP) and supervised Machine Learning (ML) models, which is trained on historical and properly characterized job data, and is able to classify unseen jobs upon submission prior to their execution. The algorithm is periodically retrained over time on recent data. Our framework can be configured ad-hoc to meet the needs and characteristics of the system on which it is deployed.

Towards implementing *MCBound* in a production HPC system, we extract job data from the Supercomputer Fugaku. The dataset contains around 2.2 million jobs executed between December 2023 and March 2024. Our job analysis using our characterization technique reveals that the great majority of Fugaku jobs are *memory-bound* and users execute large numbers of *compute-bound* jobs with the system's default execution mode (i.e. 2.0 GHz), instead of the boost mode (i.e. 2.2 GHz), which may result in longer execution time, node-hours wastage and increased energy consumption.

We implement *MCBound* for Fugaku and evaluate the online prediction algorithm with over 700,000 jobs executed during February 2024. We study the impact of choice of recent data for periodic retraining and retraining frequency on prediction accuracy and runtime overhead of training and inference. We show that our approach is effective for the classification task with an F1-macro average score of at least 0.89 as prediction quality and it incurs low runtime overhead on the system.

To the best of our knowledge, this is the first work that systematically and seamlessly characterizes and classifies *memory/compute-bound* jobs in HPC systems before job execution, without requiring any intervention by the user. In the rest of the paper, after we discuss related work in Section II, we present our three contributions in Sections III to V, and conclude in Section VI.

II. Related Work

The use of the *Roofline* model to evaluate computational bottlenecks and characterize *memory-bound* and *compute-bound* applications is a standard in the field, and it has been done in several past work [12], [16], [20], [21]. In [12], [16], the authors rely on a technique based on the *Roofline* model for an in-depth analysis of application bottleneck in the cache memory. Whereas, in [20], [21] it is used to characterize *memory-bound* and *compute-bound* applications and evaluate the impact of optimization techniques on their execution. All the cited work characterized a few well-known kernels or benchmarks via visual analysis of the resulting *Roofline* model of the computations, while we here do it systematically on millions of real jobs, for which we have no prior knowledge on the operations performed.

In recent years, several related work have used ML-based predictive tools in conjunction with operational data. Two main families of approaches exist: one focuses on fault prediction and anomaly detection of system components, the other focuses on job characteristics prediction. We restrict the related work analysis to the approaches that are compatible with live and streaming data in time, i.e. they only require job submission/execution features, or system operational data. [1], [23] belong to the first family. Differently, [3], [4], [6], [22], [29], [31], [33], [35], [37] belong to the second family. [35], [37] predict during job execution the job finish time and job power consumption while [3], [4], [6], [22], [29], [31], [33] focus on predicting job failure, duration or power consumption before job execution. In *MCBound*, we target predicting a new job characteristic prior to job execution, as runtime prediction may incur overhead on the system operations and necessitate modification to the regular workload submission workflow, as often thousands of jobs are submitted every second.

Among the last group, [3], [4], [22], [33] are the only past work that advocate job characteristic prediction using an online approach like ours, periodically retraining the model on recent data. When it comes to ML models, also [3], [4] propose to augment classical ML-models (like Random Forest and k-Nearest Neighbors clustering) with NLP tools to improve prediction performance. Differently from them, *MCBound* targets the prediction of the *memory/compute-bound* job characteristic. Our work proves that NLP-augmented Random Forest and k-Nearest Neighbor clustering models are effective also in predicting the *memory/compute-bound* job class. Moreover, we study the impact of choice of recent data for periodic retraining and retraining frequency on prediction accuracy and runtime overhead of training and inference. Finally, we integrate the algorithm as a component in a deployable framework.

III. *MCBound* Framework

In this section, we first describe the *MCBound* framework at a high-level, then detail its main components, and finally explain how it is deployed on the target system.

The framework is designed to be deployed in a real system where jobs are submitted and executed continuously, and various information regarding job submission, execution and completion (referred to as job data) is streaming in time. In this context, classification of a job before its execution can be done by leveraging only the job submission data, and the historical data of the jobs that are already completed by that

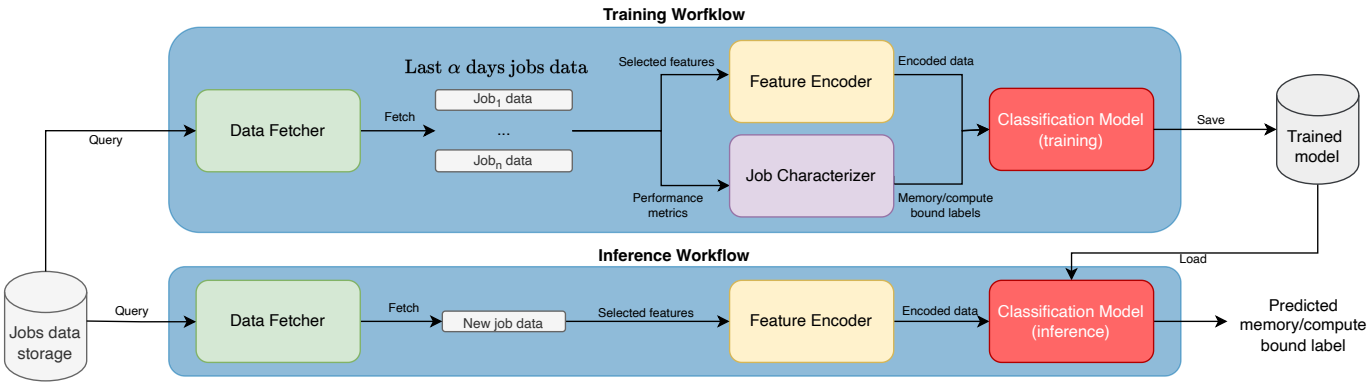

Fig. 1. High-level scheme of the components and the workflows of *MCBound*.

time. The workload of an HPC system can be very similar in a short period, while may vary a lot in the long term [3], [4]. To adapt to changes in the workload and guarantee accurate classification, we periodically update the model over time by using the recently executed job data for training. The framework thus executes in two different modes: (i) periodic model retraining on recent job data and (ii) model inference on a newly submitted job. For this, the framework requires an operational data analytics framework that collects job data and stores them in a jobs data storage.

The framework is depicted in Figure 1, where the rectangular blocks represent the main components and the blue containers show how they are employed in two Continuous Integration/Continuous Delivery (CI/CD) workflows. The components are:

- The *Data Fetcher*, which retrieves the job data by querying the jobs data storage.
- The *Feature Encoder*, which takes as input a series of raw job data, and returns the encoded data to be fed into the *Classification Model*.
- The *Job Characterizer*, which takes the raw job data as input and augments the job data with *memory/compute-bound* labels.
- The *Classification Model*, which uses the encoded data together with the *memory/compute-bound* labels for model training, and just the encoded data for model inference to classify each submitted job as *memory* or *compute-bound* prior to its execution.

When triggered:

- the *Training Workflow* fetches the data of the jobs executed in the last α days to generate an instance of a trained *Classification Model* once every β days;
- the *Inference Workflow* fetches the data of a new (unseen and not yet executed) job and generates a *memory/compute-bound* label for it.

In this paper, we target the Fugaku supercomputer, however, the framework is designed to work universally for any system. It requires only that the jobs data storage is integrated in the system, containing job features referring to submission (such as *requested resources*, *user information*, *job name*), execution and completion (such as *duration* and *#nodes allocated*), and performance metrics (such as $\#flops$ and $\#moved\_memory\_bytes$). The architecture is designed to be modular and easy to customize for different systems, for instance by implementing different *Data Fetcher*, *Feature Encoder*, or *Classification Model*. All the framework components are software components implemented as Python classes, with a method for each functionality they provide.

A. Data Fetcher

The *Data Fetcher* is an interface to retrieve data from the jobs data storage, which contains information of executed and newly submitted jobs. At initialization time, the class allows configuring the *Data Fetcher* object to interact with the specific data storage technology deployed in the target system (e.g., relational database, non-relational database, distributed file system). In this paper, we implement the class to interact with the relational database of Fugaku. The class provides the *fetch* method, which takes as input either a *job_id*, or a *start_time* and *end_time*. With the first parameter the method fetches the data of a single job corresponding to the *job_id*, with the second instead the data of all the jobs executed between *start_time* and *end_time* are fetched. These parameters are used to generate an SQL query to the job's data storage. The query results in a list of job features and their values, which is then returned as the output of the *fetch* method.

B. Feature Encoder

This component represents job features in a format suitable to be fed into the *Classification Model*, i.e. an array of floating point values. The class is endowed with an *encode* method which takes as input the raw job data and outputs the encoded job data. Internally, the method filters out a subset of job features, selected empirically during an initial experimentation phase and accordingly to the chosen classification model. The corresponding feature values are then concatenated into a comma-separated string and encoded with Sentence-BERT (SBERT) [26].

SBERT is a state-of-the-art sentence embedding model, obtained by fine-tuning pre-trained BERT (Bidirectional Encoder

Representations from Transformers) [11]. BERT is trained on millions of textual documents in order to understand patterns of one or more languages. The model is able to generate word-level embeddings, namely a semantically meaningful floating-points array representation. However, when working with pieces of text or strings in regression tasks, this representation is not suited. BERT would encode each sentence as a high dimensional matrix (# of token in the string x 768), which would require additional processing and memory. Thus, SBERT is created by fine-tuning BERT models in sentence similarity tasks, aiming to create a model which is ultimately able to generate meaningful sentence-level embeddings. The resulting representation of a text string from SBERT is a fixed-size 384-dimensional floating-point array, which constitutes the output of the *encode* method.

This method can be modified to select any subset of job features and to leverage any encoding technique (such as classical categorical mapping of feature values to integers, transformers or neural encoder/decoder models) able to map job features to a suitable format for the *Classification Model*.

C. Job Characterizer

The *Job Characterizer* component exploits the *Roofline* model [36] which represents the compute-memory ratio of a computation, and allows identifying if it is *compute-bound* or *memory-bound*. By using system specifics (i.e. peak performance and peak memory bandwidth), it computes the operational intensity op (mean operations per byte of memory traffic) of the ridge point op_r for a machine m, namely the minimum op needed to obtain the peak performance of m. This value is then used to distinguish between *memory-bound* and *compute-bound* computations. In our case, computations are jobs, and machine m is a single node n of an HPC system.

The *Job Characterizer* class is initialized with the peak performance and the peak memory bandwidth of a single node n of an HPC system, and computes the operational intensity of the ridge point op_r. The *generate_labels* method of the class returns the *memory-bound* or *compute-bound* label of a job j given as input its feature values. These are the number of floating point operations ($\#flops_j$), the duration ($duration_j$), the number of nodes allocated (*#nodes_alloc*$_j$), and the amount of moved memory bytes (*#moved_memory_bytes*$_j$). These features can be obtained by filtering the job execution statistics and performance metrics. Internally, the method computes the performance (p_j), the memory bandwidth (mb_j), and operational intensity (op_j) for j. As p_j is a measure of Flops per second, we divide the $flop_j$ by *duration*$_j$. Furthermore, since the *Roofline* model refers to a single node of the machine, the performance of j needs to be normalized on *#nodes_alloc*$_j$, obtaining for each job the per node average p_j, mb_j and op_j. Then, p_j, mb_j and op_j are computed as shown in Equations 1, 2 and 3. The *generate_labels* method returns *compute-bound* if op_j is greater than op_r computed at class initialization time, *memory-bound* otherwise.

$$p_j = \frac{\#flops_j}{duration_j * \#nodes\_alloc_j} \tag{1}$$

$$mb_j = \frac{\#moved\_memory\_bytes_j}{duration_j * \#nodes\_alloc_j} \tag{2}$$

$$op_j = \frac{p_j}{mb_j} \tag{3}$$

In this first version of *MCBound*, we focus only on the classes defined in the original *Roofline* paper [36]. However, by adding to the *Roofline* model the bandwidth of other hardware components (e.g. cache, interconnect and GPUs) it is possible to expand the *Job Characterizer* to create other labels for the job data, such as *interconnect-bound* and *GPU-bound*.

D. Classification Model

This component provides methods for the classification task via data-driven prediction algorithms. When an object of the class is created, the initialization method takes as input the name of the algorithm to employ. In our use case, we implement two instantiations, using supervised ML algorithms which are first trained on historical and properly characterized job data, before performing inference on unseen jobs.

- KNN employs the k-Nearest Neighbors clustering [14] algorithm, which does not rely on internal parameters. Training is performed by only storing the values of a fixed amount of data points in a given feature space. Then, inference is done by a majority voting among the k most similar data. The similarity is computed as the Minkowski distance metric between the data input features.
- RF employs the Random Forest [7] algorithm, which is an ensemble technique relying on a set of Decision Trees (DTs). The DT [28] is an algorithm which learns decision rules from the correlations observed between features and target values of historical data used for training. Differently from the KNN, the training consists of tuning the parameters of the different DTs on a random subset of the training data, and a random subset of the input features. At inference time, a majority voting among the trained DTs is carried out. This is done to make up for the tendency of individual DTs to overfit on the training data and thus obtain less error-prone prediction performance, as explained in [7].

The chosen algorithms spend complementary effort in their training and inference parts. While RF needs to dedicate a significant amount of time to training, the KNN does that for inference. Availability of algorithms with different learning nature allows choosing the best trade off between the quality of prediction and the runtime effort spent on it. We note that it is possible to implement any data-driven prediction algorithm for *Classification Model*, such as neural networks, other ML-based or even heuristic algorithms.

Once initialized, the *Classification Model* instance provides the *training* and *inference* methods. The *training* method takes as input two arrays containing respectively encoded job data and the corresponding *memory/compute-bound* labels. The input data is then used to train the *Classification Model* instance. The *inference* method can be called only after the

TABLE I
FUGAKU SYSTEM ARCHITECTURE.

System characteristic	Description
Architecture	Armv8.2-A SVE 512 bit
OS	Red Hat Enterprise Linux 8
#Nodes	158,976
#Cores (per node)	48 + 4 assistant cores
Memory (per node)	HBM2, 32 GiB, 1024 GBytes/s
Peak Performance	$\approx$ 537 PFlops/s (FP64), $\approx$ 3.3 TFlops/s per node
Internal Network	Tofu D Interconnect (28 Gbps)

Classification Model instance is trained. The method takes as input an array of encoded job data and generates a list of predicted *memory/compute-bound* labels for all the jobs.

E. MCBound Deployment

We implement *MCBound* as a *flask*[2] backend, providing APIs to perform the operations of the framework. Flask is endowed with a built-in development server, but it can be also easily deployed to any HTTP server. We also provide a Docker [25] configuration, to distribute the framework as a container and make it scalable through container orchestration techniques, such as Kubernetes [10].

The workflows shown in Figure 1 are implemented as Python scripts leveraging the framework APIs to perform the necessary steps. The trained model instances are saved to the machine file system by using the skops.io library,[3] in order to handle and maintain different versions of the models.

We provide a `deploy` script for the first deployment of *MCBound*. The script first executes the *Training Workflow* script to generate the trained instance of the *Classification Model*, and then the *flask* application of *MCBound* is started. Finally, a cronjob [17] is scheduled to re-execute the *Training Workflow* script every β days.

The online prediction starts when a trained instance of the *Classification Model* is generated. Then, the trained instance is used by the *Inference Workflow* script to generate predictions for the all jobs submitted in the following β days. Within this period, the inference on a job can be triggered in two different ways depending on how and when the prediction is needed: at each new job submission, or by periodically querying the jobs data storage to retrieve the accumulated new job data. After β days, the cronjob for the *Training Workflow* script is re-triggered, an instance of the *Classification Model* is trained, and the framework is ultimately reloaded.

IV. MEMORY/COMPUTE-BOUND CHARACTERIZATION AND ANALYSIS OF FUGAKU JOBS

In this section, we apply our characterization approach to the job data obtained from the Fugaku system and analyze the outcome. The characterization is necessary to acquire the ground truth for the prediction algorithm evaluation in Section V, while the analysis allows to obtain insights into the system usage; for instance, whether the users submit jobs optimized to fully saturate the different system resources and if specific actions can be enacted to improve system throughput.

[2]https://flask.palletsprojects.com/

[3]https://skops.readthedocs.io/en/stable/index.html

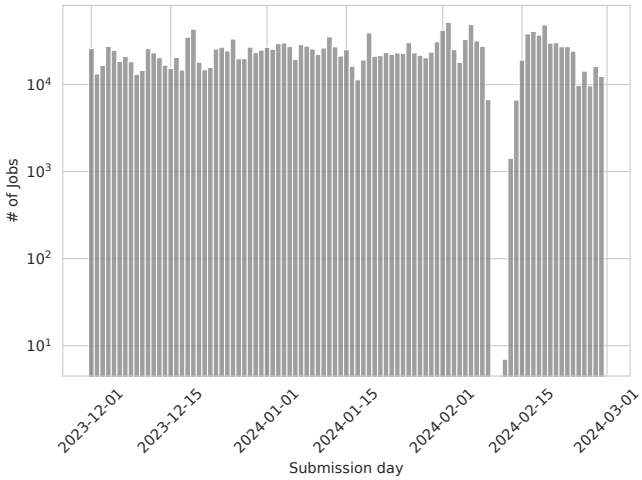

Fig. 2. Job submission distribution over time.

A. Fugaku Job Traces

We extract job data from the operational logs of Fugaku, which was deployed in 2020 and at the time of writing ranked 4^{th} among the most powerful supercomputers in the world. [4] A summary of the system characteristics is shown in Table I.

Fugaku relies on a proprietary operations management software[5], built as an extension of PBS [13] which features workload management operations like job manager, job scheduler, and functions that enable the recording and storage of job data. This data includes information about job submission (such as *submission time*, *requested resources*, *user information*, and *system state*), as well as job execution and completion (such as *duration* and *performance counters*). The *performance counters* can be leveraged to compute performance metrics on the job execution, such as $\#flops$ and $\#moved\_memory\_bytes$.

Fugaku is used by hundreds of users, submitting thousands of jobs to the system every day. We extract the data of 2.2 million jobs submitted and executed between December 2023 and March 2024 (the data is publicly available in Zenodo [2]). Figure 2 shows the distribution of the jobs over the entire period. We observe that the job submission rate is uniform except for a few days in early February, when a scheduled maintenance caused the shutdown of the system.

B. Job Characterization Setup

Following the methodology presented in Section III-C, we extract the peak performance and peak memory bandwidth of a Fugaku node from system's specifications.[6], which are 3380 GFlops/s in FP64 and 1024 GByte/s, respectively. The peak

[4]https://www.top500.org

[5]https://www.fujitsu.com/global/about/resources/publications/technicalreview/2020-03/article10.html#cap-03

[6]https://www.fujitsu.com/global/about/innovation/fugaku/specifications/

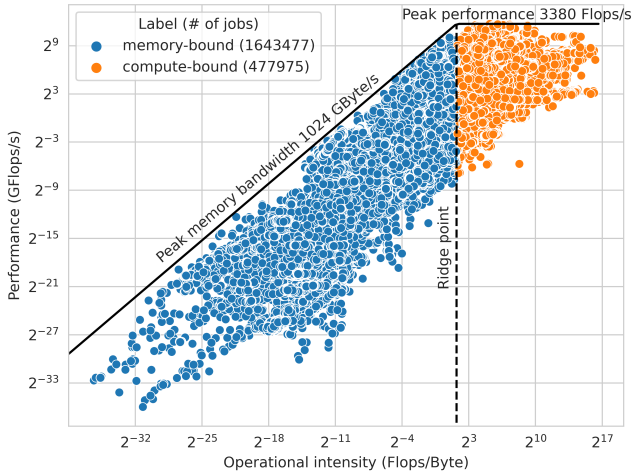

Fig. 3. *Roofline* model of the job data.

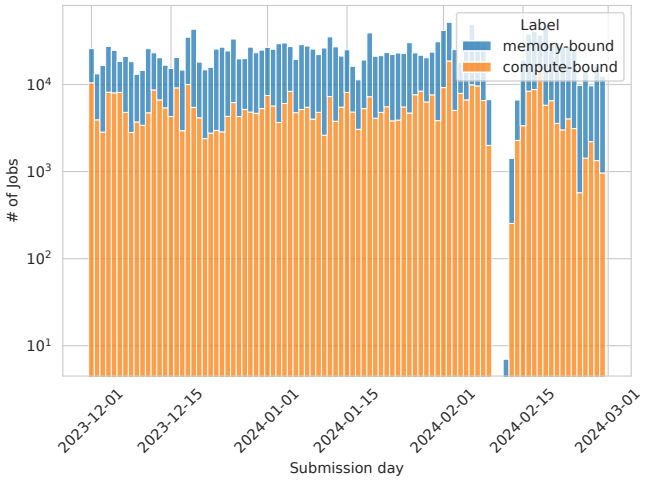

Fig. 4. Distribution of job types over time.

performance reported refers to FX1000 boost-mode configuration (i.e. frequency = 2.2 GHz for the A64FX CPUs), as we need to consider the best performance attainable by the machine. Based on these characteristics, the ridge point is at an op_r of ≈ 3.3 Flops/Byte, which is used for the job labelling.

As described in Section III-C, for job j we compute p_j and mb_j (and consequently op_j), via $\#flops_j$ and $\#moved\_memory\_bytes_j$. These two values in the target system are computed starting from the performance counters ($perf2$, $perf3$, $perf4$, $perf5$). In the Fugaku system, $perf2$ and $perf3$ correspond to the FP_FIXED_OPS_SPEC and FP_SCALE_OPS_SPEC A64FX_PMU_Events, while $perf4$ and $perf5$ correspond to BUS_READ_TOTAL_MEM and BUS_WRITE_TOTAL_MEM [15]. In Equation 4, $perf2$ is the fixed amount of operations, while $perf3$ is the number of operations per 128-bit SVE, and it is multiplied by 4 since the A64FX of Fugaku is 512-bit SVE. In Equation 5, $perf4$ and $perf5$ are summed in order to obtain the total number of requests to the memory, as they represent the amount of memory read and write requests, respectively. Then, they are multiplied by the size of the memory requests, (256 bytes of cache line size), to obtain the total $\#moved\_memory\_bytes_j$. Moreover, the cores of Fugaku nodes are grouped by 12 in Core Memory Groups (CMGs). The $perf4$ and $perf5$ values are generated by summing all the values collected by each core for the whole CMG. Therefore, these values need to be divided by 12 to eliminate redundant information.

$$\#flops_j = perf2_j + (perf3_j * 4) \tag{4}$$

$$\#moved\_memory\_bytes_j = \frac{(perf4_j + perf5_j) * 256}{12} \tag{5}$$

Once we compute p_j and mb_j (and op_j), we label the job j based on the comparison of op_r of the ridge point and op_j.

C. Fugaku Job Analysis

Figure 3 shows the collective *Roofline* model, which reports in x-axis op measured as Flops/Byte, and in y-axis p in GFlops/s. We observe that the distribution of the operational intensity of the jobs submitted to the Fugaku system is significantly skewed toward values lower than the ridge point. Moreover, as reported in Table II, the number of *memory-bound* jobs is around 3.5 times as the number of *compute-bound* jobs. Figure 4 reports the distribution of each job type over the entire period. We notice that the proportion between the *memory-bound* and *compute-bound* jobs is constant in time, suggesting that this difference is a characteristic of the studied Fugaku workload. This is interesting considering that the A64FX of the Fugaku system has been co-designed for *memory-bound* jobs [30], and thus, a more balanced job distribution would be expected.

We can also see that many jobs are far from the *Roofline*. This is particularly notable in the *memory-bound* area, where only a few clusters of jobs are close to the peak memory bandwidth line. The same can be observed in the *compute-bound* area, where only some jobs with operational intensity around the ridge point touch the peak performance line. This means that while there are some well-engineered jobs saturating fully the resources, it is not the case for the majority of the jobs. Therefore, leveraging *MCBound* to classify *memory/compute-bound* jobs has the potential to guide job scheduling, for instance by enacting co-scheduling of *memory-bound* and *compute-bound* jobs on the same node, or by adjusting the amount of resource allocated to the job, and thus to reduce system resource wastage.

Figure 5 shows the distribution of jobs in the *Roofline* plane by highlighting the node frequency selected by the user at job submission time. In Table II, we see that around 54% of the *memory-bound* jobs are executed in normal mode (frequency=2.0 GHz), while only around 30% of *compute-bound* jobs in boost mode (frequency=2.2 GHz). Moreover,

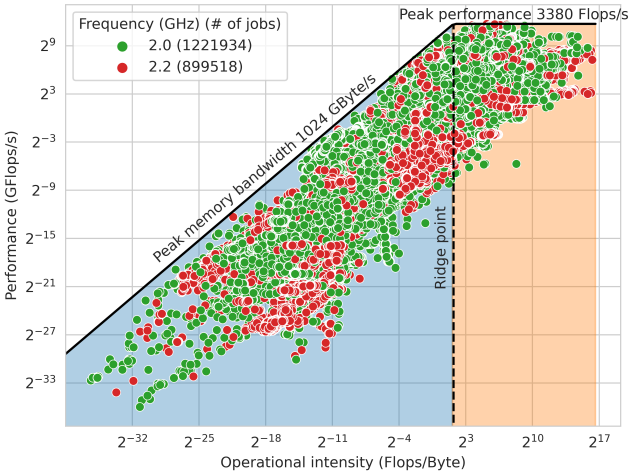

Fig. 5. *Roofline* model of the job data, divided by frequency.

TABLE II
DISTRIBUTION OF JOB TYPES.

Frequency	*memory-bound*	*compute-bound*	Total
2.0 GHz (normal mode)	891,056	330,878	1,221,934
2.2 GHz (boost mode)	752,421	147,097	899,518
Total	1,643,477	477,975	2,121,452

Figure 5 shows that there is no observable correlation between the user-selected frequency at submission time and the position of the given job in the *Roofline*. These observations suggest that users do not necessarily choose appropriate frequencies for their jobs. Indeed, *memory-bound* jobs do not benefit from running at higher frequencies, as their performance bottleneck is the memory bandwidth, while *compute-bound* jobs are likely to increase their performance at higher frequencies, possibly resulting in shorter execution time and energy savings. Therefore, another advantage of leveraging *MCBound* is the possibility to guide frequency selection, and thus the improvement of system energy efficiency and throughput.

V. EXPERIMENTAL STUDY

In this section, we present the implementation of the *Classification Model* of *MCBound* for Fugaku, experimentally evaluate the online prediction algorithm, and discuss the results.

A. Classification Model Implementation for Fugaku

We rely on the *scikit-learn*[7] library for the ML models and use their default implementation. The SBERT model is provided by the *sentence transformers* library[8], while the weights are pulled from Huggingface [9]. We use the pre-trained model *all-MiniLM-L6-v2*[10], since it has the best trade-off between prediction quality and speed [26]. The code we used for the implementation will be released in a public repository.

[7]https://scikit-learn.org/stable/
[8]https://www.sbert.net
[9]https://huggingface.co
[10]https://huggingface.co/sentence-transformers/all-MiniLM-L12-v2

We conduct an initial empirical evaluation of the dataset to find the best set of features to represent the Fugaku jobs. The set should be representative enough for jobs to maximize the prediction accuracy while concise enough to minimize the runtime overhead in processing them. Past work on job power consumption of Fugaku jobs [4] found that the best set is composed of *user name, job name, #cores requested, #nodes requested* and *environment*. Our experiments confirm their value in our prediction task and that including the additional feature *frequency requested* improves the prediction performance. We therefore use *frequency requested* and those of [4] as augmented feature set.

As mentioned in Section III, the *Inference Workflow* can be triggered periodically. For Fugaku, we do it once every β days, using the job data accumulated since the last trigger. We save the job characterizations and encodings of every trigger of the *Training Workflow* and *Inference Workflow*, in order to reuse them and avoid redundant computations during the future triggers of the *Training Workflow*.

The *Training/Inference Workflow*s are performed on a machine detached from Fugaku, accessible via HTTP calls. Thus, no overhead to the HPC computing resources is incurred. As the data were already collected for logging purposes, the only additional storage required is for the saved trained model, which is negligible in today's HDD (around 1GiB)

B. Online Prediction Algorithm Evaluation

To evaluate the online prediction algorithm, we implement an `evaluate` Python script, which is executed once at the end of the testing period. This evaluation targets the assessment of the prediction quality as well as the incurred runtime overhead.

a) Evaluation setup: We employ the two ML models, KNN and RF, described in Section III-D. The models are trained on portions of the data of the jobs executed between December 1^{st}, 2023 and January 31^{st}, 2024 and tested on a subsequent time window composed of over 700,000 jobs executed between February 1^{st} and 29^{th}, 2024.

Prediction quality is measured using the F1-macro average score [32] - a widely used metric for classification problems - computed as the mean of the F1-score obtained on specific *memory-bound* and *compute-bound* classes. The F1-score on a single class is computed as the harmonic mean between the precision and recall on the target values. Hereafter, we will refer to the F1-macro average as F1. The F1 for a model is computed at the end of the testing period by our `evaluate` script, on all the predictions generated by all the *Inference Workflow* executions. The ground truth labels necessary for F1 have been acquired via Fugaku job data characterization, as described in Section IV.

The runtime overhead of the algorithm refers to the time spent in training and inference, which are computed as the average of all the *Training Workflow* and *Inference Workflow* runtimes. While job characterization time is negligible ($1 * 10^{-6}$ seconds per job), the encoding incurs a higher overhead ($2 * 10^{-3}$ seconds per job) which still is negligible. We

note, however, that job encoding takes place during *Training Workflow* only once at the first deployment of *MCBound*. This is because there are no encodings of the historical job data at the beginning. As models are retrained during future triggers of the *Training Workflow*, the job encodings can be retrieved from the previous *Inference Workflow* computations. Thus, we do not include the job characterization and encoding time in the training time, while we include the encoding time in the inference time.

b) Experimental setup: We conduct three experiments. As described in Section III, the online prediction algorithm retrains a model using the recently executed job data (the last α days' data), and continues to do so periodically (at every β days). In the first experiment, we use different combinations of α and β values, to find the best time window of the recent data for periodic retraining and retraining frequency. We iterate $\alpha \in \{15, 30, 45, 60\}$ and $\beta \in \{1, 2, 5, 10\}$. We avoid $\beta = 0$, i.e. retraining upon each new job submission, as it incurs excessive overhead, as well as exclude larger values of β so as not to delay model update for long.

We are not interested in using more than $\alpha = 60$ days of training data either, as otherwise the model would have to deal with a large amount of data during RF training and KNN inference, possibly increasing the runtime overhead. Moreover, the workload of an HPC system is variable and training based on "older" data is not beneficial for prediction. We demonstrate this in our second experiment, where the initial model training is done using the best α found in the first experiment, and then successively the model is retrained using the data of all the past days, without forgetting the data older than α days. We refer to this setting as α^+ time window.

To observe whether the amount of data used within a given α time window influences the prediction quality, in our third experiment we retrain the models using a θ subset of the last α days of data. To this end, we iterate $\theta \; in \; \{10^2, 10^3, 10^4, 10^5\}$ after having analyzed the average training data size.

The experiments are run on the machine where the framework is currently deployed for testing purposes, which is equipped with two AMD EPYC 7302 CPUs, 64 cores and 512 GB RAM, running Python 3.11.5 on Linux Fedora 37. The code of the experiments will be released in a public repository.

C. Experimental Results

a) Experiments with α and β values: Figure 6 shows F1 values over different combinations of α and β values. In both models, as β increases, F1 decreases, due to less frequent model training and knowledge update. We therefore consider the best retraining frequency as $\beta = 1$ (once a day). As α increases, the models behave differently. The parametric model RF tends to benefit from training using data spanning to a larger time window, probably because parameter tuning becomes more precise, but we observe no gains with $\alpha > 15$ when $\beta = 1$. Whereas, the non-parametric nature of KNN does not benefit from "older" data. In the specific case of $\beta = 1$, the best performance is attained with $\alpha = 30$ and then declines with greater values. We theorize that the workload has more

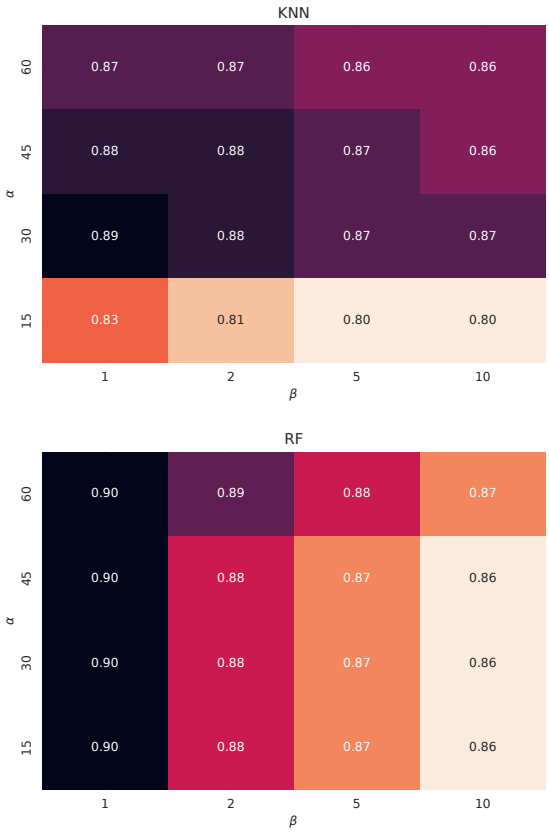

Fig. 6. F1 of KNN (above) and RF (below) with different α and β values.

similarities within 30 days. Given that KNN inference works by finding similar data, training on data older than 30 days infers past job behavior.

Figure 7 shows the average daily training time across various values of α. KNN training time is almost negligible, with a maximum duration of 0.32 seconds with $\alpha = 60$. In fact, KNN training consists of just building a model instance, which stores the training data for future inference, and no parameter is tuned. Conversely, RF requires an actual training phase with parameter tuning, and as α grows, so does the training time with the amount of data growing, up to almost 3 minutes. However, RF reaches the best prediction already with $\alpha = 15$, when the training time is lowest (around 26 seconds).

In Figure 8, we show the daily average inference time per job (including job encoding time) across various values of α. RF has a constant inference time, as inference is done through tuned parameters independently of α. Though this value is around $2 * 10^{-6}$, it is dominated by the average job encoding time ($2 * 10^{-3}$). KNN inference is about finding similarities among the entire training data, thus inference time would grow with larger values of α. However, the inference time is again dominated by the encoding time and is around $2.3 * 10^{-3}$, not changing much across different values of α. Still, the inference time per job of $\alpha = 30$ is negligible (milliseconds) w.r.t. job average waiting time for scheduling (time spent

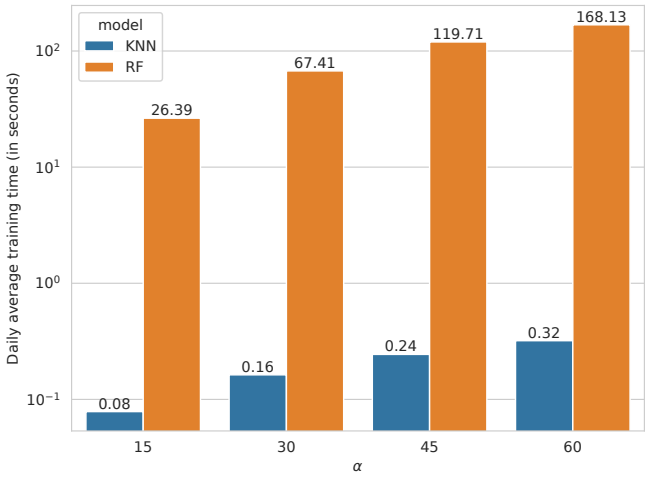

Fig. 7. Average model training time variation when $\beta = 1$.

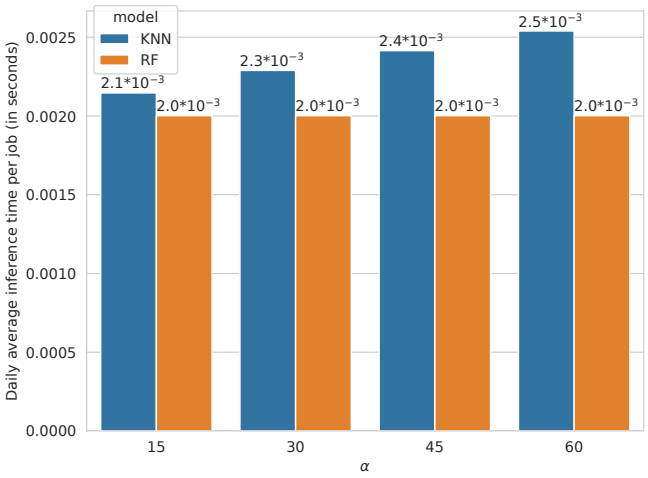

Fig. 8. Average job inference time variation when $\beta = 1$.

until the scheduling decision of a job, after its submission), which is around 3 minutes in the observed period. This means that neither of the models would incur overhead on the job submission workflow of the system. From Figures 6, 7 and 8, we can conclude that the best algorithm settings are $\alpha = 15$ (RF) and $\alpha = 30$ (KNN), coupled with $\beta = 1$.

We further compare the RF and KNN to a simple baseline that maps a tuple of (*job name, # of cores requested*) to a *memory/compute-bound* label (which can be seen as a KNN with k = 1 on the features *job name, # of cores requested*). The baseline is updated over time using the same online algorithm, with $\alpha = 30$ and $\beta = 1$ (as the best KNN settings). While this solution is simpler, it is also less accurate than ours (F1-score: 0.83 vs 0.90), justifying the need for our approach.

b) Experiments with α^+: Starting with the best α and β value combinations as described in the previous experiment, we observe no improvement in prediction considering the α^+ time window during training. F1 of RF with α^+ is 0.90, which is the same as α = 15. This is not surprising, as we saw in the previous experiments that increasing α does not change F1 when $\beta = 1$. F1 of KNN instead decreases to 0.86 with α^+ from 0.89 with α = 30. This supports our hypothesis that jobs are most similar within 30-days.

Moreover, the growing time window jeopardizes the training time of RF and the inference time of KNN, as they are both dependent on the training data size. The average training time of RF increases from 26.39 seconds with α = 15 to more than 200 seconds with α^+. Differently, the training time increase in KNN is marginal, going from 0.16 seconds with α = 30 to 0.39 seconds with α^+. Conversely, while the average inference time per job of RF remains the same, the KNN time increases but slightly, going from $2.3 * 10^{-3}$ seconds per job with α = 30 to around $2.5 * 10^{-3}$ with α^+.

This experiment confirms that a sliding time window, which filters the recent job data for retraining, is beneficial for the proposed online algorithm both for prediction accuracy and overhead on the system's operations. Therefore, in the following experiment, we fix α to its best values of 15 (RF) and 30 (KNN).

c) Experiments with θ: For a given α retraining time window, we select a subset θ of data points either randomly, or by considering the jobs with the most recent ending time. When sampling data randomly, we repeat model training with 5 different random seeds[11] and average the results of the 5 different trained models.

Figures 9 and 10 show F1 values of KNN and RF using latest and random data over different values of θ. We observe that having more data within a fixed time window improves the prediction accuracy for both sampling approaches, where the best result is obtained by using all the available data. Interestingly, random sampling is more effective consistently across all θ values. This can be attributed to the fact that Fugaku jobs are usually submitted in batches of identical jobs, and job data very near in time might lead to replicated data during training. A higher percentage of such replicated training data would result in a less general model, while sampling the data randomly smoothes this effect. Our hypothesis is supported by the fact that with smaller values of θ, i.e. from 10^2 to 10^4, the F1 difference between the two sampling approaches is significant (up to 0.26), while the gap reduces drastically (down to 0.02) with $\theta = 10^5$. In fact, the more jobs take part in training, the less batches of identical jobs would impact the model, as the percentage of replicated data would drop.

d) Discussion of results: The best settings of the algorithm, in terms of retraining data time window α and retraining frequency β, are $\alpha = 15, \beta = 1$ days for RF and $\alpha = 30, \beta = 1$ days for KNN, using all the available training data. With these settings, we obtain accurate predictions (F1=0.90 for RF and F1=0.89 for KNN), at the expense of 26 seconds for RF and 0.16 for KNN daily average training time, and average

[11]The random seeds used for the experiments are 520, 90, 1905, 7, 22

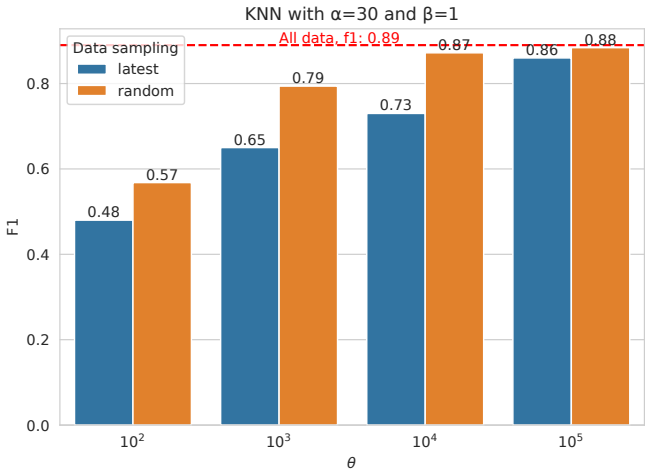

Fig. 9. F1 of KNN with different θ values.

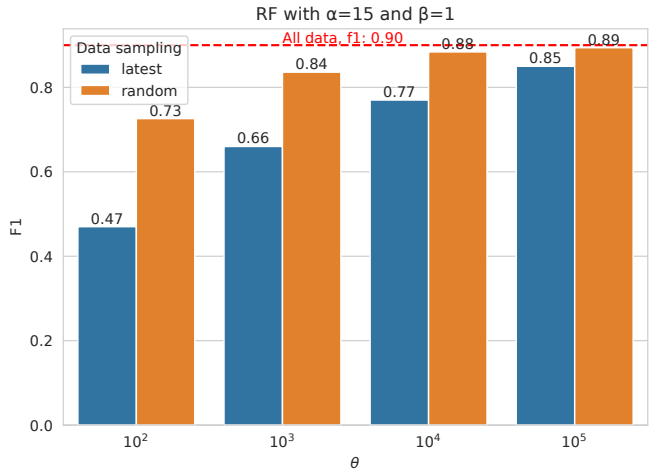

Fig. 10. F1 of RF with different θ values.

inference time per job of $2.0 * 10^{-3}$ seconds with RF and $2.3 * 10^{-3}$ with KNN. As the average number of submitted jobs per day in the observed period is 25K, the daily overhead of model inference can be estimated as around 50 seconds for RF and 60 for KNN. The overall daily training and inference overhead is negligible w.r.t. to job average waiting time for scheduling, which is around 3 mins during our observation.

We conclude that regardless of the model used, the online prediction algorithm is suitable to be deployed in a production system, as it provides accurate predictions with negligible overhead on the job submission workflow of the system. We highlight that our approach can have a significant impact on the system power, energy and performance. We estimate the impact based on previous work on job and frequency mode characterization [18]. The authors showed that using the boost mode on Fugaku for *compute-bound* jobs can reduce the job execution duration by 10% (with respect to normal mode), while using the normal mode for *memory-bound* jobs can reduce the job power consumption by 15% (with respect to boost mode). Our algorithm classifies 90% of the jobs correctly, hence we could perform semi-automatic frequency selection and obtain the following improvements. There are 750k *memory-bound* jobs executed in boost mode, with an average power consumption of 5000 W, and an average duration of 6000 seconds. By executing them in normal mode, we could have reduced the power consumption by around 680W per job, saving 450MW of power, and 14 GJoules of energy, at the system level. Moreover, there are 330k *compute-bound* jobs executed in normal mode, with an average duration of 13,500 seconds. By executing them in boost mode, we could have saved around 20 minutes of computation per job, and more than 1,700 hours of overall system computation.

We note that these kinds of improvements can be potentially obtained in any system where the nodes' frequency can be decided by the user, and thus our results are not limited to the Fugaku system.

VI. Conclusions

We presented *MCBound*, the first online data-driven framework to classify HPC jobs before job execution as *memory-bound* and *compute-bound*. Our framework is designed to be deployed in a real system where jobs are submitted and executed continuously. The framework requires only that the jobs data storage is integrated in the system, containing job features referring to job submission, execution and completion, and performance metrics. The classification is based on an ML-based predictive model which is periodically updated with the recent job data streaming in time. In addition to classification, *MCBound* can be used for job characterization as a stand-alone approach to analyze the *memory/compute-bound* nature of the jobs of a system.

Using the proposed characterization technique, we analyzed the data of 2.2 million jobs executed on the Fugaku supercomputer between December 2023 and March 2024, finding out that the submitted jobs are not optimized to fully saturate the system resources, and thus the system efficiency can be improved. We then implemented *MCBound* for Fugaku and demonstrated that *MCBound* is effective in classifying *memory/compute-bound* jobs before their execution, as it obtains an F1-macro average score of at least 0.89 on the data of more than 700,000 jobs executed on Fugaku during February 2024, with negligible runtime overhead. These results are obtained by studying the best setting for the online prediction algorithm in terms of the choice of recent data for periodic retraining and retraining frequency. We also argued how our approach can have a significant impact on the system power, energy and performance. We conclude that *MCBound* is suitable to be deployed in a production HPC system to guide the job scheduling and allocation (dispatching), so as to improve system energy efficiency and throughput. Our Python-based implementation of *MCBound* can be seamlessly configured and deployed in other HPC systems.

As future work, we plan to deploy the framework in other

systems where similar data can be collected, and expand it for two purposes. First, to predict other job features (such as *duration*, *power consumption* or *failure*) with the KNN predictive model. The KNN finds the most similar jobs regardless of the target feature, hence we can easily adapt the framework for the prediction of multiple features without having to rely on different predictive models. Second, to predict other classes of jobs (e.g. *interconnect-bound* and *GPU-bound*). We are currently developing job dispatching strategies that can benefit from the predictions of *MCBound*, aiming to optimize system throughput and energy efficiency. We will investigate how to integrate *MCBound* and the new dispatchers into the workload management system of HPC systems.

ACKNOWLEDGEMENTS

This research was partly supported by the HE EU Graph-Massivizer project (g.a. 101093202) and the EU DECICE project (g.a. 101092582). We would like to thank Jens Domke [12] from Riken Center for Computational Science for his precious contribution to this work.

REFERENCES

[1] B. Aksar, E. Sencan, B. Schwaller, O. Aaziz, V. J. Leung, J. Brandt, B. Kulis, M. Egele, and A. K. Coskun, "Prodigy: Towards unsupervised anomaly detection in production hpc systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2023, pp. 1–14.

[2] F. Antici, A. Bartolini, J. Domke, Z. Kiziltan, and K. Yamamoto, "F-DATA: A Fugaku Workload Dataset for Job-centric Predictive Modelling in HPC Systems," Jun. 2024. [Online]. Available: https://doi.org/10.5281/zenodo.11467483

[3] F. Antici, A. Borghesi, and Z. Kiziltan, "Online job failure prediction in an hpc system," in *Euro-Par 2023: Parallel Processing Workshops: Euro-Par 2023 International Workshops, Limassol, Cyprus, August 28–September 1, 2023, Revised Selected Papers*. Springer Nature, 2023.

[4] F. Antici, K. Yamamoto, J. Domke, and Z. Kiziltan, "Augmenting ml-based predictive modelling with nlp to forecast a job's power consumption," in *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, 2023, pp. 1820–1830.

[5] K. Asifuzzaman, M. A. H. Monil, F. Liu, and J. S. Vetter, "Evaluating hpc kernels for processing in memory," in *Proceedings of the 2022 International Symposium on Memory Systems*, 2022, pp. 1–6.

[6] A. Borghesi, A. Bartolini, M. Lombardi, M. Milano, and L. Benini, "Predictive modeling for job power consumption in hpc systems," in *High Performance Computing: 31st International Conference, ISC High Performance 2016, Frankfurt, Germany, June 19-23, 2016, Proceedings*. Springer, 2016, pp. 181–199.

[7] L. Breiman, "Random forests," *Machine learning*, vol. 45, pp. 5–32, 2001.

[8] J. Breitbart, S. Pickartz, S. Lankes, J. Weidendorfer, and A. Monti, "Dynamic co-scheduling driven by main memory bandwidth utilization," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, 2017, pp. 400–409.

[9] J. Breitbart, J. Weidendorfer, and C. Trinitis, "Case study on co-scheduling for hpc applications," in *2015 44th International Conference on Parallel Processing Workshops*, 2015, pp. 277–285.

[10] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *Communications of the ACM*, vol. 59, no. 5, pp. 50–57, 2016.

[11] J. Devlin, M.-W. Chang, K. Lee, and et al., "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 NAACL: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186.

[12]https://orcid.org/0000-0002-5343-414X

[12] N. Ding and S. Williams, *An instruction roofline model for gpus*. IEEE, 2019.

[13] H. Feng, V. Misra, and D. Rubenstein, "Pbs: a unified priority-based scheduler," in *Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, 2007, pp. 203–214.

[14] E. Fix and J. L. Hodges, "Discriminatory analysis. nonparametric discrimination: Consistency properties," *International Statistical Review/Revue Internationale de Statistique*, vol. 57, no. 3, pp. 238–247, 1989.

[15] Fujitsu Limited, "A64fx pmu events," 2019. [Online]. Available: https://raw.githubusercontent.com/fujitsu/A64FX/master/doc/A64FX_PMU_Events_v1.2.pdf

[16] A. Ilic, F. Pratas, and L. Sousa, "Cache-aware roofline model: Upgrading the loft," *IEEE Computer Architecture Letters*, vol. 13, no. 1, pp. 21–24, 2013.

[17] M. S. Keller, "Take command: cron: Job scheduler," *Linux Journal*, vol. 1999, no. 65es, pp. 15–es, 1999.

[18] Y. Kodama, T. Odajima, E. Arima, and M. Sato, "Evaluation of power management control on the supercomputer fugaku," in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2020, pp. 484–493.

[19] D. Lee, I. Dinov, B. Dong, B. Gutman, I. Yanovsky, and A. W. Toga, "Cuda optimization strategies for compute-and memory-bound neuroimaging algorithms," *Computer methods and programs in biomedicine*, vol. 106, no. 3, pp. 175–187, 2012.

[20] A. Li, W. Liu, M. R. Kristensen, B. Vinter, H. Wang, K. Hou, A. Marquez, and S. L. Song, "Exploring and analyzing the real impact of modern on-package memory on hpc scientific kernels," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–14.

[21] D. Marques, H. Duarte, A. Ilic, L. Sousa, R. Belenov, P. Thierry, and Z. A. Matveev, "Performance analysis with cache-aware roofline model in intel advisor," in *2017 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 2017, pp. 898–907.

[22] K. Menear, A. Nag, J. Perr-Sauer, M. Lunacek, K. Potter, and D. Duplyakin, "Mastering hpc runtime prediction: From observing patterns to a methodological approach," in *Practice and Experience in Advanced Research Computing*, 2023, pp. 75–85.

[23] A. Netti, Z. Kiziltan, O. Babaoglu, A. Sîrbu, A. Bartolini, and A. Borghesi, "A machine learning approach to online fault classification in hpc systems," *Future Generation Computer Systems*, vol. 110, pp. 1009–1022, 2020.

[24] M. Orenes-Vera, E. Tureci, D. Wentzlaff, and M. Martonosi, "Dalorex: A data-local program execution and architecture for memory-bound applications," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 718–730.

[25] B. B. Rad, H. J. Bhatti, and M. Ahmadi, "An introduction to docker and analysis of its performance," *International Journal of Computer Science and Network Security (IJCSNS)*, vol. 17, no. 3, p. 228, 2017.

[26] N. Reimers and I. Gurevych, "Sentence-bert: Sentence embeddings using siamese bert-networks," *arXiv preprint arXiv:1908.10084*, 2019.

[27] E. R. Rodrigues, R. L. Cunha, M. A. Netto, and M. Spriggs, "Helping hpc users specify job memory requirements via machine learning," in *2016 Third International Workshop on HPC User Support Tools (HUST)*. IEEE, 2016, pp. 6–13.

[28] L. Rokach and O. Maimon, "Decision trees," *Data mining and knowledge discovery handbook*, pp. 165–192, 2005.

[29] T. Saillant, J.-C. Weill, and M. Mougeot, "Predicting job power consumption based on rjms submission data in hpc systems," in *High Performance Computing: 35th International Conference, ISC High Performance 2020, Frankfurt/Main, Germany, June 22–25, 2020, Proceedings 35*. Springer, 2020, pp. 63–82.

[30] M. Sato, Y. Ishikawa, H. Tomita, Y. Kodama, T. Odajima, M. Tsuji, H. Yashiro, M. Aoki, N. Shida, I. Miyoshi *et al.*, "Co-design for a64fx manycore processor and" fugaku"," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–15.

[31] A. Sîrbu and O. Babaoglu, "Power consumption modeling and prediction in a hybrid cpu-gpu-mic supercomputer," in *Euro-Par 2016: Parallel Processing: 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24-26, 2016, Proceedings 22*. Springer, 2016, pp. 117–130.

[32] M. Sokolova, N. Japkowicz, and S. Szpakowicz, "Beyond accuracy, f-score and roc: A family of discriminant measures for performance evaluation," vol. Vol. 4304, 01 2006, pp. 1015–1021.

[33] X. Tian, X. Li, J. Zhang, Z. Zhao, C. Wang, X. Wang, and J. Wang, "An online incremental learning framework for hpc job power consumption prediction," in *Proceedings of the 2023 7th International Conference on High Performance Compilation, Computing and Communications*, 2023, pp. 176–183.

[34] M. Wahib and N. Maruyama, "Scalable kernel fusion for memory-bound gpu applications," in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2014, pp. 191–202.

[35] Q. Wang, H. Zhang, J. Li, Y. Shen, and X. Liu, "Predicting job finish time based on parameter features and running logs in supercomputing system," *The Journal of Supercomputing*, vol. 78, no. 17, pp. 18 551–18 577, 2022.

[36] S. Williams, "Roofline: An insightful visual performance model for floating-point programs and multicore," *ACM Communications*, p. 16, 2009.

[37] K. Yamamoto, Y. Tsujita, and A. Uno, "Classifying jobs and predicting applications in hpc systems," in *High Performance Computing: 33rd International Conference, ISC High Performance 2018, Frankfurt, Germany, June 24-28, 2018, Proceedings 33*. Springer, 2018, pp. 81–99.

[38] F. V. Zacarias, P. Carpenter, and V. Petrucci, "Memory demands in disaggregated hpc: How accurate do we need to be?" in *2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. IEEE, 2021, pp. 1–6.

Appendix: Artifact Description/Artifact Evaluation

Artifact Description (AD)

I. Overview of Contributions and Artifacts

A. Paper's Main Contributions

In the paper, we propose the following contributions:

- C_1 We introduce *MCBound*, the first online data-driven framework to classify HPC jobs before job execution as *memory-bound* and *compute-bound*, without user intervention.
- C_2 We propose a systematic characterization technique to generate a reference dataset from historical data for our initial classification model training. Using the proposed characterization technique, we analyze the data from 2.2 million job runs on the Supercomputer Fugaku to obtain insights into their *memory/compute-bound* characteristics.
- C_3 We employ *MCBound* to classify the jobs executed on Fugaku during February 2024, obtaining an F1-macro average score of at least 0.89 as prediction quality.

B. Computational Artifacts

For the paper, we produce 3 artifacts, listed below along with their DOIs.

- A_1 https://github.com/francescoantici/MCBound-framework/tree/sc_ad.
- A_2 https://github.com/francescoantici/MCBound-framework/tree/sc_ad/characterize_jobs.py
- A_3 https://github.com/francescoantici/MCBound-framework/tree/sc_ad/online_algorithm_evaluation.py

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1	Figure 1
A_2	C_2	Table 2 Figures 2-5
A_3	C_2	Figures 6-10

II. Artifact Identification

A. Computational Artifact A_1

Relation To Contributions

The artifact A_1 is related to the contribution C_1, as this artifact contains the source code allowing to deploy and execute the operations of the *MCBound* framework.

Expected Results

The experiments are needed to show how *MCBound* is deployed to the machine and used. This substantiates the contribution A_1, as it demonstrates that the framework is implemented in a reproducible fashion.

Expected Reproduction Time (in Minutes)

All the times are evaluated considering the execution of the artifact on a machine with two AMD EPYC 7302 CPUs, 64 cores and 512 GB RAM. The time to perform the artifact A_1 setup is the time needed to install a Python distribution and all the required Python libraries to the system, which is around 1 minute. The artifact execution is the time needed to execute the deployment, and it is around $5*10^{-5}$ minutes (0.003 seconds). The analysis time can be estimated by computing the average time needed to perform a request to an endpoint of the backend instance just started, and get a response in return. This time is around $2*10^{-4}$ minutes (0.01 seconds).

Artifact Setup (incl. Inputs)

Hardware: The hardware requirements for the artifact A_1 are the same requirements of Python3.11.5, namely at least 2 CPUs, 4 GB of RAM, and 5 GB of free disk space.

Software: The required software packages are listed below, along with their version and URL. The deployment can be performed either on the machine with Python, or through containerization with Docker. Therefore, the requirements are either:

Docker 4.29.0 [1];

or:

Python 3.11.5[2].
dependency_injector 4.41.0[3].
Flask 3.0.3[4].
Flask_Cors 4.0.0[5].
numpy 1.26.4[6].
pandas 2.2.2[7].
scikit_learn 1.4.2[8].
sentence_transformers 2.6.1[9].
skops 0.9.0[10].

Datasets / Inputs: The artifact does not require any dataset.

Installation and Deployment: The requirement for the deployment and execution of the experiments is the installation of the packages listed in the *Software* paragraph of this section, namely either the Docker distribution, or the Python distribution along with the needed libraries. Then, no other requirement needs to be satisfied to proceed with the deployment.

[1] https://www.docker.com
[2] https://www.python.org/downloads/release/python-3115/
[3] https://python-dependency-injector.ets-labs.org
[4] https://flask.palletsprojects.com/en/3.0.x/
[5] https://flask-cors.readthedocs.io/en/latest/
[6] https://numpy.org
[7] https://pandas.pydata.org
[8] https://scikit-learn.org/stable/
[9] https://www.sbert.net
[10] https://skops.readthedocs.io

Artifact Execution

The experiment workflow is composed of two tasks, T_1 and T_2. T_1 consists of the installation of the dependencies for the deployment of the framework, executed in T_2. To this end, T_2 is strictly dependent on the correct execution of T_1, otherwise it cannot be completed. T_2 takes as input a port on which the backend is deployed. For the sake of this experiment, we pick port 8080, as it is not used by any other processes on the deployment machine. Both T_1 and T_2 are executed only once.

Artifact Analysis (incl. Outputs)

The output of the execution of the A_1 is a running instance of a backend server on the selected port of a machine (8080 for this experiment). Moreover, if the deployment is performed with Python, the Python environment would contain the packages described in the *Software* paragraph of this section. Conversely, if performed with Docker, there would be a running instance of a docker container, built on a docker image of around 2 GB in size.

B. Computational Artifact A_2

Relation To Contributions

The artifact A_2 is related to the contribution C_2, as this artifact contains the source code to perform the systematic *memory/compute-bound* characterization on the data of 2.2 million job runs on the Supercomputer Fugaku.

Expected Results

The outcome of the corresponding experiment is the data to perform analysis of the *memory/compute-bound* nature of the job executed on Supercomputer Fugaku between December 2023 and February 2024. The abstract substantiates the contributions by allowing to perform such analysis and draw conclusions on the *memory/compute-bound* nature of the job runs of Fugaku.

Expected Reproduction Time (in Minutes)

The time to execute the artifact A_2 on a machine with two AMD EPYC 7302 CPUs, 64 cores and 512 GB RAM, is around 2.2 minutes, where $3*10^{-2}$ minutes (2 seconds) are spent for the job characterization, and the remaining is needed to generate the plots.

Artifact Setup (incl. Inputs)

Hardware: The hardware requirements for the artifact A_2 are the same as A_1.

Software: The required software packages are listed below, along with their version and URL.

A_1.
Python 3.11.5[2].
seaborn 0.13.2[11].
matplotlib 3.8.4[12].
pandas 2.2.2[7].

[11]https://seaborn.pydata.org
[12]https://matplotlib.org

Datasets / Inputs: The artifact A_2 requires the dataset of the 2.2 million job runs on Supercomputer Fugaku. This dataset can be generated by using the A_1 to fetch the data concerning the job runs on the system between December 2023 and February 2024.

Installation and Deployment: The requirements for the execution of the experiments are the presence of a Python distribution, the installation of the Python packages listed in the *Software* paragraph of this section, and the execution of A_1, namely a running instance of the *MCBound* framework.

Artifact Execution

The experiment workflow is composed of three tasks, T_1, T_2 and T_3, where the correct execution of each of the tasks is mandatory for the following ones. T_1 consists of fetching the data of the 2.2 job runs on Fugaku through the running instance of *MCBound*. Then, such data are characterized in T_2 to create the *memory/compute-bound* labels. Finally, in T_3 the labels are used to generate several plots on their distribution. All the three tasks are executed only once.

Artifact Analysis (incl. Outputs)

The output of the execution of A_2 is a dataset containing the 2.2 million *memory/compute-bound* labels for all the job data, and a series of plots showing their distributions.

C. Computational Artifact A_3

Relation To Contributions

The artifact A_3 is related to the contribution C_3, as this artifact contains the code to train the classification models, classify the jobs executed on Fugaku during February 2024 with *MCBound*, and evaluate its prediction performance.

Expected Results

The outcome of the experiment should be the best performing classification model setting, and the fact that *MCBound* is able to accurately classify Fugaku jobs as *memory-bound* or *compute-bound*, before their execution.

Expected Reproduction Time (in Minutes)

The time to perform the artifact A_3 setup is the time needed to perform the training and inference of all the models, plus the final evaluation of the prediction accuracy. In a machine with two AMD EPYC 7302 CPUs, 64 cores and 512 GB RAM, these operations are performed in around 500 minutes.

Artifact Setup (incl. Inputs)

Hardware: The hardware requirements for the artifact A_3 are the same as A_1 and A_2.

Software: The required software packages are listed below, along with their version and URL.

A_1.
A_2.
Python 3.11.5[2].
seaborn 0.13.2[11].
matplotlib 3.8.4[12].

pandas 2.2.2[7].

Datasets / Inputs: The artifact A_3 requires two datasets. The first is the dataset of the job executed on Supercomputer Fugaku during February 2024, which is generated by using the A_1 to fetch the corresponding job submission data. Moreover, it also needs the dataset containing the *memory/compute-bound* labels of all the jobs executed from December 2023 though February 2024. This dataset is obtained from the output of A_2, and it is necessary to train the classification models and evaluate their prediction accuracy.

Installation and Deployment: The requirements for the execution of the experiments are the presence of a Python distribution, the installation of the Python packages listed in the *Software* paragraph of this section, and the execution of A_1 and A_2.

Artifact Execution

The experiment workflow is composed of three tasks, T_1, T_2 and T_3. T_1 performs the training of a set of classification models on the past job data. The trained models are used in T_2 to perform inference on the future job data and generate the *memory/compute-bound* predictions. T_2 depends on T_1 and always comes after it. These two tasks are repeated 29 times, once per day, for all the day between February 1^{st} and 29^{th}. Finally, T_3 is executed, which compares the predicted labels to the actual ones stored in the dataset generated in A_2, and outputs the prediction performance of the classification models.

Artifact Analysis (incl. Outputs)

The output of the execution of the A_3 is a series of trained instances of the classification models, and the prediction accuracy of each model.