Intelligent Colocation of Workloads for Enhanced Server Efficiency

Conference Paper · August 2019

DOI: 10.1109/SBAC-PAD.2019.00030

6 authors, including:

Felippe Vieira Zacarias
Barcelona Supercomputing Center
10 PUBLICATIONS **78** CITATIONS
SEE PROFILE

Vinicius Petrucci
Micron Technology, Inc
50 PUBLICATIONS **788** CITATIONS
SEE PROFILE

Rajiv Nishtala
Norwegian University of Science and Technology
18 PUBLICATIONS **117** CITATIONS
SEE PROFILE

Paul M. Carpenter
Barcelona Supercomputing Center
69 PUBLICATIONS **615** CITATIONS
SEE PROFILE

Intelligent Colocation of Workloads for Enhanced Server Efficiency

Felippe Vieira Zacarias*†, Vinicius Petrucci*§, Rajiv Nishtala‡, Paul Carpenter†, Daniel Mossé§
*Universidade Federal da Bahia
†Barcelona Supercomputing Center
‡Norwegian University of Science and Technology
§University of Pittsburgh

***Abstract*—Many server applications achieve only a fraction of their theoretical peak performance due to bottlenecks in the shared caches, instruction execution units, I/O or memory bandwidth, even though the remaining resources may be under-utilized. It is very hard for developers and runtime systems to ensure that all these critical resources are fully exploited by a single application. An attractive technique for increasing server system utilization is to colocate multiple applications on the same server. When applications share critical resources, however, these applications may adversely affect each other, due to contention on the shared resources.**

In this paper, we show that server efficiency can be improved by modeling the expected performance degradation of colocated applications from measured hardware performance counters, and exploiting such a model to determine an optimized mix of colocated applications. This paper presents a novel resource management approach and makes the following contributions: (1) a new machine learning model to predict the performance degradation of colocated applications from hardware counters and (2) an intelligent scheduling scheme deployed on an existing resource manager to enable application co-scheduling with minimum performance degradation. Our results show that our approach achieves performance improvements of $15\,\%$ (avg) and $26\,\%$ (max) compared to the standard policy commonly used by existing job managers.

***Index Terms*—Resource manager, Machine learning, Colocation, Performance degradation, Performance counters**

I. Introduction

Data center providers need to maximize server utilization, in order to obtain the greatest possible benefit from their large capital investments [1], [2]. Many server applications, however, only achieve a fraction of the theoretical peak performance, even when they have been carefully optimized [3]. This can lead to a substantial waste of resources across the whole data center.

A promising way to increase overall system utilization and efficiency is to run multiple applications concurrently on a node, an approach that is known as workload colocation [1], [2], [5]–[7]. The biggest disadvantage of workload colocation is the potential degradation in application performance due to sharing of resources such as caches, memory controllers, data prefetchers and I/O devices. Such degradation is hard to predict in real systems, and it is impractical to measure the degradation of all pairs of applications ahead of time. For this reason server systems usually disallow applications from sharing resources in the same computing node [1], [8]. Nevertheless, workload colocation does have a substantial potential to improve system throughput, especially when the colocated applications are bottlenecked on different resources [3]. Note that this improvement in system utilization is made without any need to modify the application's source code.

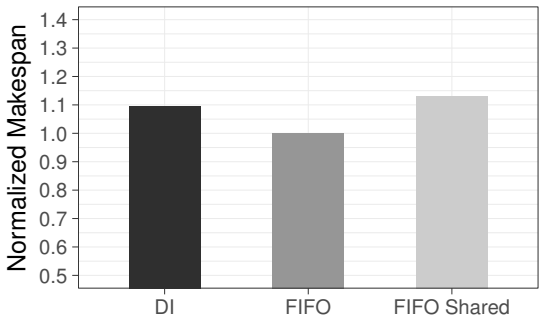

Fig. 1: Normalized makespan for Distributed Intensity [4] (prior work) vs FIFO sequential (no-sharing) and FIFO shared policies, while executing 20 randomized application queues (see Section III for details on our experimental setup).

Prior research has addressed the workload colocation problem, including [4], [9], and [10] (see Section IV), which favor co-scheduling applications with wide variance in the shared resource (e.g., the Last-Level Cache, or LLC). Distributed Intensity (DI) [4] is an example of such a heuristic, which collects the LLC miss rate via hardware counters and avoids co-scheduling two applications with high LLC miss rates in the same memory hierarchy.

Figure 1 shows the result of an experiment that DI actually increased the makespan[1] in 10% compared to the default FIFO policy used by a typical job manager system (see Section III details), but could improve by 5% over FIFO shared (allowing applications to run on the same node). This behavior is due to bad pairs co-scheduled by DI, observed in most executed queues. In conclusion, relying on simple heuristics

* Work was mainly carried out when the authors were affiliated with UFBA.

[1]The makespan is a performance metric given by the time between the start of the first application and the end of the last application in the job queue.

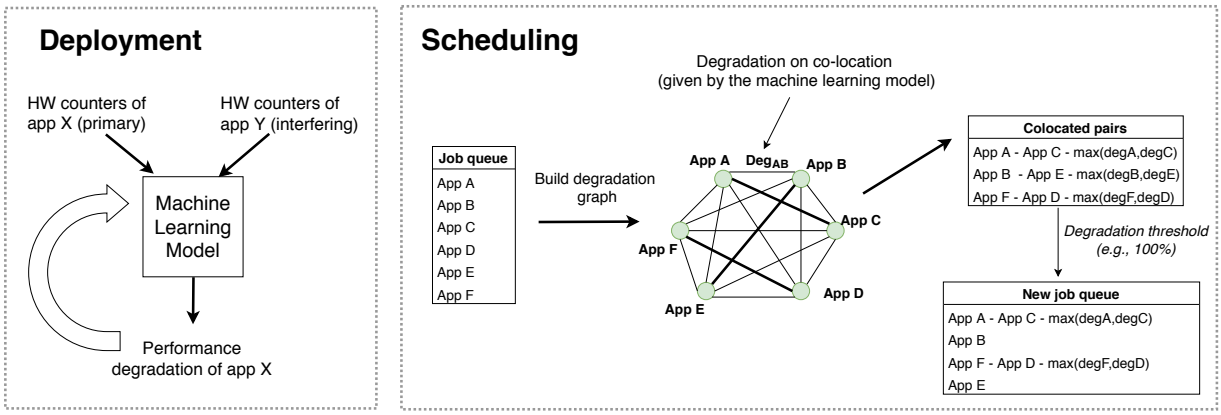

Fig. 2: Overview of our approach consisting of two phases: (1) deployment of the machine learning model for performance degradation prediction due colocation and (2) scheduling of the submitted job queue for improved execution efficiency given a degradation graph.

(for example, using cache misses as a sole indicator) was not sufficient to minimize overall resource contention and improve server efficiency execution.

In contrast to prior heuristics, the key idea of our approach is to build a model that abstracts the application, by predicting its degradation introduced by colocating a pair of applications through automatic analysis of hardware performance monitoring counters. Hardware performance counters can track the occurrence of events with negligible overhead, but they do not directly reveal how much degradation the application will suffer when it is colocated with another application. In addition, there are many hardware counters that can be used for monitoring, and only some of them may reveal the degradation due to colocation. It is hard to identify the right performance counters and manually build the right non-linear model that maps the performance counters to the predicted degradation. For these reasons, we build a machine learning model.

This paper presents an approach that encompasses a machine learning model and its integration into an intelligent application scheduler. The key idea is to fully utilize the server hardware, consolidate workloads onto a server, and improve server efficiency. We make the following contributions:

1) We design and build prediction models to estimate performance degradation due to workload colocation. Our best model (Random Forest) is capable of providing good accuracy (78%) in predicting performance degradation, even for new unseen applications.
2) We present an intelligent workload management solution, that leverages hardware performance counters given predicted degradation between colocated applications in the server system. The goal is to improve the workload makespan without making the turnaround time of the jobs much longer.
3) We evaluate our solution through an actual implementation deployed on an existing resource manager executing multithreaded applications from PARSEC, NPB, Splash and Rodinia and show improvements over a default scheduling policy by 15% (average) and 26% (maximum) in system performance (makespan).

The key ingredient of our technique is that it uses a machine learning model that does not require application modifications or annotations; rather it is based on hardware performance counter data for deciding which applications can run together with minimum degradation in a server system.

II. Intelligent Workload Management

This section describes our solution to perform intelligent colocation of workloads in server systems.

A. Overview

A typical cluster scheduler uses a flavor of bin packing algorithm to assign ready jobs to physical nodes based on user-defined resource demands and constraints. In our solution we keep the same distribution methodology performed by an existing workload manager and add to it non-exclusive access to physical nodes in a degradation-aware fashion.

Figure 2 shows a high-level overview of our solution, organized in two phases: Deployment and Scheduling. The deployment stage is responsible for training a machine learning model to estimate performance degradation due to workload colocation. The scheduling stage takes as input a job queue to execute in the server and decides on colocations (with minimal degradation) and a new ordering for those jobs.

The machine learning model built in the deployment stage is used to predict the degradation for any pair of applications submitted to execute in the queue. We focus on finding a good colocation for pairs of applications as in [1], [2]. We use the maximum performance loss within each pair to characterize degradation. We assume that the applications have already executed in the system and its profile (that is, the hardware counters measured for the application execution alone on the target system) is accessible to the resource manager in place.

In the scheduling phase, our technique receives as input both the list of jobs in the ready queue as well as the profile

data for such jobs. Next, it uses the trained model to predict the degradation when two particular jobs are expected to run colocated with each other. Finally, it generates an optimized schedule queue that minimizes the total degradation between the pairs of applications. In fact, a new job queue is not created, however it reorders the job's priority in the ready queue in order to execute the pair in the right order.

B. Predicting performance degradation

We collect performance counters and execution time from the applications to build a training set, which is usual in many data centers [11]. An application profile (input features) consists of its execution time on the target system and its collected hardware counters.

We develop the input dataset used for training the models, as follows. First, we execute the applications alone in the system (without any other user applications). This was necessary to obtain a baseline for the application execution time and hardware counters considering the ideal condition: the application has all resources available without competing with other applications. We allow each application to use all available cores. All applications were executed multiple times and we collect data for the counters shown in Table I; we use the mean, minimum, maximum value, and the standard deviation of those counters in our models. The baseline execution is also needed to calculate the degradation experienced by the application during colocation (see Equation 1), which is calculated as the percentage increase in the application execution time, where T_{coloc} is the execution time when the application executes on colocation, and T_{alone} is its execution time under the ideal condition.

$$Deg_{ij} = 100 \times ((T_{\mathrm{coloc}_{ij}} - T_{\mathrm{alone}_i})/T_{\mathrm{alone}_i}) \qquad (1)$$

After collecting the application performance for the alone execution, we execute every (primary) application concurrently with another (interfering) application. We execute all possible pairs on our application set by starting every two applications at the same time. If the interfering application finishes before the primary application, we restart it to keep the primary application suffering contention during its entire execution.

Although the complexity of $O(n^2)$ for data collecting from job colocation can be high if the number of applications (n) is large, in the long term, this cost can be reduced by executing a subset of the pairs of common applications, and estimating the rest (e.g. via matrix factorization), or by using fast $O(n)$ microbenchmarking as in Bubble-Up [1].

In the end, the training dataset contained three parts: the first contains the performance counters data for the primary application when running alone on the system; the second contains the performance counter data for the interfering application; and the third contains the performance degradation suffered by the primary application for this pair. The most important counters are automatically inferred by the machine learning models after the training phase. We use the model at runtime to predict the degradation of every pair of application given their profiles, as described in the next section.

TABLE I: Hardware counters and derived metrics.

Events	Counters
Hardware	cycles, instructions, resource_stalls.any, branch_instructions, stalled_cycles_frontend, stalled_cycles_backend,branch_misses,
Software	page_faults, context_switches, cpu_migrations
Cache	cache_references, cache_misses, LLC_prefetch_misses, LLC_prefetches, l2_rqsts.demand_data_rd_hit l2_rqsts.pf_hit, l2_l1d_wb_rqsts.miss l2_lines_out.pf_clean, l2_lines_out.pf_dirty l2_rqsts.all_pf, l2_lines_in.all, l1d.allocated_in_m l2_lines_out.demand_clean, l1d.eviction l2_rqsts.all_demand_data_rd, L1_dcache_store_misses, L1_dcache_load_misses, L1_dcache_loads L1_dcache_prefetch_misses, l1d.replacement
Memory	mem_uops_retired.all_stores mem_uops_retired.all_loads mem_load_uops_retired.llc_miss mem_load_uops_retired.llc_hit
Calculated	*IPC, cache_ref_per_instructions, CPU_usage cache_misses_per_instructions, miss_ratio*

C. Scheduling colocated applications

We adopted a scheme to schedule the applications in such a way that the overall degradation on the target system could be minimized, similar to [12]. Given a set of *n* independent applications $A_1, A_2, ..., A_n$, the goal is find a schedule that chooses a pair of applications (i and j) onto a server node with minimum colocation degradation, expressed by the term $\max(Deg_{i,j}, Deg_{j,i})$. The total cost of the schedule to be minimized is the sum of all these terms for the chosen pairs.

For two applications sharing the same node, given the degradation knowledge as input, the optimal co-scheduling can be found in polynomial time [12]. In this case, the problem can be modeled as a fully connected graph, called *degradation graph*. As shown in Figure 2 in the scheduling stage, each vertex of the degradation graph is an application (from the job queue) and each edge is the highest predicted degradation of the pair when they run on the same node.

The optimal co-scheduling problem can be viewed as a *minimum-weight perfect matching* problem given the degradation graph [12]. Finding the solution for this matching problem is equivalent to finding an optimal colocation scheduling for pairs of applications because a valid scheduling means all selected pairs should cover all vertices without sharing the same vertex, which is a condition for a perfect matching. The minimum-weight ensures the objective function of the colocation problem is satisfied, which stands for minimizing the sum of the degradations.

We use the *blossom* algorithm [13] to optimally solve the matching problem for application pairs in polynomial-time: $\mathcal{O}(n^2m)$, where n and m are the number of nodes and edges in the degradation graph, respectively. We show in Section III-E4 an analysis on the computational time required to solve this matching problem and produce a scheduling solution, optimally based on the degradation graph and approximately using a greedy heuristic, which simply selects the pairs with the lowest degradation to execute.

Although the output given by the solution of the *minimum-weight perfect matching* results in a set with minimum overall degradation, the solution may produce pairs with very high degradation when compared to its solo/serial execution. We schedule those pairs with excessive degradation in a serial fashion (exclusive resources). A default acceptable degradation threshold is set to 100% for the chosen pairs, unless otherwise stated. This is because at this point application pair colocation causes system performance to actually go down (reduction of performance is an increase of makespan in this case).

III. Experimental Evaluation

We evaluate our approach in a multi-core server system running several compute-intensive applications. We deploy the models in the Slurm manager. We have adopted Slurm [14] because it is a popular, scalable, widely deployed, open source, fault-tolerant job scheduling system for Linux clusters.

We perform several experiments with varying job queues to compare our approach with the existing default policies in Slurm's workload manager: (a) the first executes the applications sequentially on a single node, and (b) the second allows for node sharing without degradation knowledge. We also compare (c) a greedy solution vs optimal (perfect matching) solution given the degradation graph. The goal is to contrast the quality of output (metric of interest is makespan) and time required to produce a scheduling solution.

A. Machine Learning Models

We used the training dataset with different machine learning models to evaluate their prediction accuracy outcomes. We investigated four types of machine learning techniques: *Elastic Net* [15], a regularization method that does variable selection and shrinkage of regression coefficients applying penalty, and it groups strongly correlated variables; *Support Vector Machine (SVM) Regressor* [16], a generalization of SVM to solve regression problems; *Random Forest Regressor* [17], a supervised learning algorithm that assembles multiple decision trees to predict a decision for a given mapping function; and *Multilayer Perceptron (MLP) Regressor* that learns a non-linear function allowing one or more hidden layers between the input and output layers. These layers work by specifying a set of "neurons" that can propagate the inputs through a network using a series of functions located at each node.

We used the *Scikit-learn* library [18] to implement the machine learning techniques. We show the accuracy by using the coefficient of determination, R-squared (R^2). For a given prediction function $y = f(x)$, R^2 determines how much the total variation of Y (dependent variable) is due to X (independent variable).

During the training phase we tried different parameters in the models in order to achieve the highest scores. These parameters are called hyper-parameters in machine learning and are not directly learned within the training. For instance, the parameters *alpha*, *gamma* and *kernels* are passed as arguments to the desired model to be trained. We ran an experiment to find the most suitable combination of those parameters that could give us the highest accuracy for our problem.

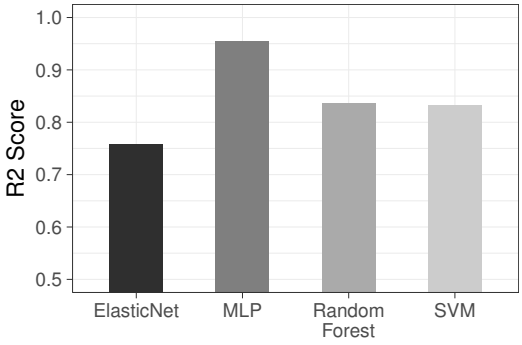

Fig. 3: Highest accuracy achieved during training phase.

B. Experimental Setup

We carried out the experiments on a dual-socket quad-core Intel Xeon E5-2407 processor. To minimize Linux interference on the makespan, the CPU *intel_pstate* governor was set to "performance" mode with the clock speed fixed at highest frequency, 2.2 GHz. Each socket has 10 MB L3 cache (Last Level Cache), shared among all cores. Each core has a private 256 kB L2 and 32 kB L1 cache. The machine has 64 GB of DDR3-1333 MHz RAM. The operational system was CentOS 7 with kernel version 3.10.0-693.21.1 x86_64. Hyperthreading was disabled as in most data center systems.

We used a total of 32 applications from different benchmarks to perform our experiments, including 27 applications from Parsec [19], Rodinia [20], NPB [21] for training and validation sets and 5 applications from Splash [22] for the test set. The Splash dataset was set apart (holdout method) to provide an unbiased evaluation of the final model.

We selected applications to cover a variety of computational patterns found in multithreaded and high performance codes. All applications were compiled using GNU/Linux GCC 4.8.5 with optimization flags and multithreading enabled, so each application could use the number of threads equals the number of cores on the server node. We profiled the application using the Linux tool `Perf` at a per-thread level and the working set for the applications was tuned to exceed the size of private caches, as it is common for native input size on real machines.

C. Hardware Counter Metrics

To build the training dataset, we record the performance counter measurements (specific counters are described in Table I) to help understand the application behavior. Each application was executed multiple times, yielding standard deviation of 0.2% to 4.5%. In the model we characterize applications by extracting (i.e., considering the following features:) the mean, minimum, maximum value, and standard deviation of the collected values.

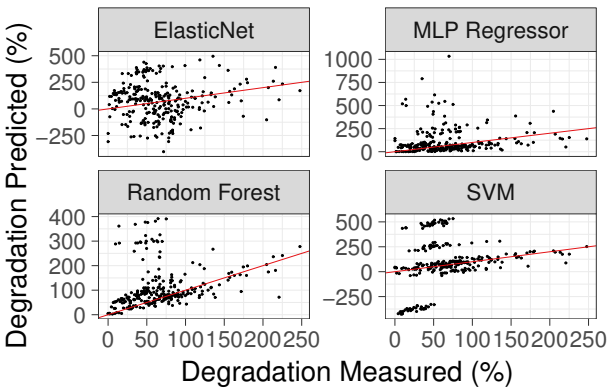

Fig. 4: Prediction behavior for each model during validation with unseen data.

D. Model Evaluation

To build the prediction models we iterated over the four models described in Section III-A applying 5-fold cross-validation to achieve higher scores during the training phase. The method splits the input data (27 applications and 729 pairs) into k number of subsets (in our case $k = 5$), then performs the training on *k-1* subsets, leaving one subset for the evaluation of the trained model and iterating k times with a different subset for testing each time.

We fed the Splash dataset (consisting of 5 unseen applications) to each model as a "blind test", to assess its performance for new incoming workload data. We observed that the machine learning models were capable of predicting application interference with high accuracy despite using applications from PARSEC, Rodinia and NPB benchmark suites for training. For that reason, using Splash as a blind test the goal was to estimate how general and useful the models are in practice given that new applications is what we would observe in a real environment.

When predicting performance degradation among all pairs of colocated applications, Figure 3 presents the highest accuracy (*r2_score*) obtained for each machine learning model. We can observe that all machine learning techniques achieved accuracy above 70%. Random Forest regressor and SVM Regressor showed similar results, with accuracy of over 80%, both outperforming ElasticNet. MLP Regressor attained over 90% of accuracy, the best in our training phase.

For the validation step, as usual, we used a dataset with unseen application profiles, consisting of benchmarks not used in the training phase. While in Figure 3 we observe that all models achieved over 70% of accuracy during training phase, only Random Forest achieved more than 50% of accuracy in the validation data, at 78%.

Figure 4 presents how each model predicted the degradation on the unseen data. We can see that Random Forest predictions are more concentrated around the straight line. Points close to the line means that the predicted degradation is similar to the measured true value, thus showing high accuracy. MLP has some points far from the true value, but the majority is around the true value. However, both ElasticNet and SVM have more spread out predictions and many predictions below zero.

Predictions that deviate from the true value does not contribute to maximize server utilization, the scheduler will rely on the mispredicted values and make decisions that will not lead to the most efficient solution. We consider predicting negative degradations (below the true value) as critical because the scheduler can use this value to pair applications that should not be together, thus making them suffer high levels of degradation. Predicting positive degradation (above the true value) is also not desirable, but it is less critical because the scheduler can assume a conservative approach and schedule the applications in a First in First out (FIFO) fashion.

In our tests (Figure 4) ElasticNet and SVM predicted negative values when confronted with the validation data and thus were deemed not suitable to be used in the resource manager as they could lead to bad application pairs. On the other hand, Random Forest and MLP Regressor did not present negative predictions, therefore showing that they can at least perform as well as an approach without colocation. Our next experiments will show how such accuracy influences scheduler decisions.

E. Application Mapping Evaluation

Given the results from model training and validation phases, we integrate both the MLP Regressor (best in the training phase) and Random Forest (best in the validation phase) in our scheduling plugin for Slurm. As a baseline, we disable the shared resource and let the resource manager apply its FIFO execution, which we called *FIFO*. Slurm allows for setting priority and allowing the applications to share the whole allocated node without restrictions. This approach was called *FIFO Shared*. It is agnostic to degradation between applications and executes the applications in order of arrival.

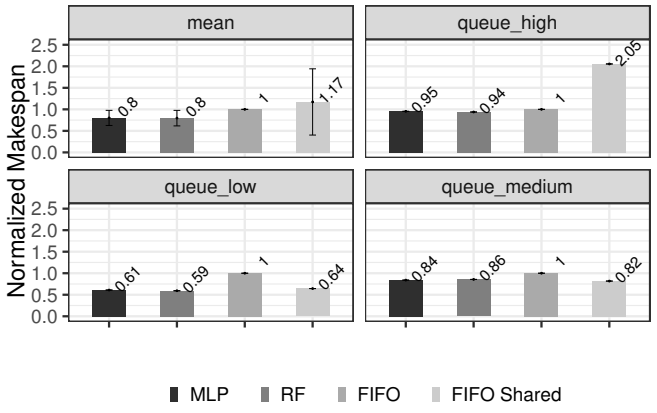

Fig. 5: Normalized makespan for each approach on the described scenarios.

1) Predefined Degradation Levels: To investigate the effect of degradation levels on performance, we evaluate the

potential of sharing resources on a set of defined group of applications with distinct degradation levels then we analyze the behavior of the scheduling strategies. We separated the arriving applications into 3 different queues with 50 applications on each, according to their degradation amounts: *queue_low* contained applications whose degradation levels were 0%; *queue_medium*, contained applications whose execution order would generate degradation of more than 0% and less than 100%; and *queue_high* containing applications whose degradation would be higher than 100%. This analysis aims to highlight the potential worst and best scenarios given the application set for the scheduling strategies.

Figure 5 presents the makespan, normalized to FIFO (serial execution). As expected, on *queue_high*, FIFO shared had the worst performance (2x makespan) compared with FIFO. Because FIFO shared is not degradation aware, it has no flexibility to rearrange sequential bad pairs: it executes them in order of arrival, causing excessive amount of degradation. Although the queue are made of applications with high degradation between them, by using both machine learning models we were able to outperform the FIFO execution, mainly by its capacity of rearranging the queue to find which applications can and can not execute together. It also outperformed FIFO shared as sequential bad pairs will not share resources.

In *queue_medium*, FIFO shared outperformed FIFO with an improvement of about 20% on makespan. Even better results were obtained on *queue_low*, when the improvement on makespan reached around 40%. It happens because the serial execution misses opportunities when the applications can safely share resources as they do not fully utilize the resources. In these scenarios, FIFO shared can take advantage of the combination on applications' dispatch. As long as the order of arrival is beneficial to colocation, FIFO shared can improve the makespan over FIFO.

The top-left part of Figure 5 shows the *mean*, representing the overall performance of each approach. FIFO shared had an increase of almost 20% on the makespan necessary to complete all three queues, because this strategy is unable to avoid scheduling bad combinations that arrive in the queue. However, we outperformed FIFO by nearly 20%, because our approach not only avoided bad combinations but also found other possibilities through reordering.

This result demonstrates the effectiveness of our predictive approach that could correctly identify the scenarios where it is worth performing workload colocation vs. non-colocation.

2) Random job queues: To evaluate a more realistic scenario, in which the degradation level from the arrival of applications is unknown, we ran an experiment in which the order of dispatch was randomly chosen (uniform distribution). Differently from the previous experiment, the applications on each queue were randomly chosen.

Figure 6 presents the average makespan normalized to FIFO for each scheme when executing 20 queues of 50 applications each. FIFO shared increased the makespan in almost 13%, while our solution improved the makespan in 15%. It is worth mentioning that our solution had better performance than FIFO for every queue and that, despite MLP achieving 43% in our validation test, it had very similar makespan performance when compared to Random Forest (less than 1% difference). When investigating the results of the individual queues, in comparison to FIFO, our solution had 5% of makespan improvement in the worst scenario and 26% in the best scenario. In addition, our approach also outperformed the DI heuristic presented in Figure 1, which allowed bad pairs to execute together in most executed queues, thus increasing the makespan in 10%. This result highlights the importance of considering additional performance counters when characterizing the degradation instead of relying on chache miss as a sole indicator.

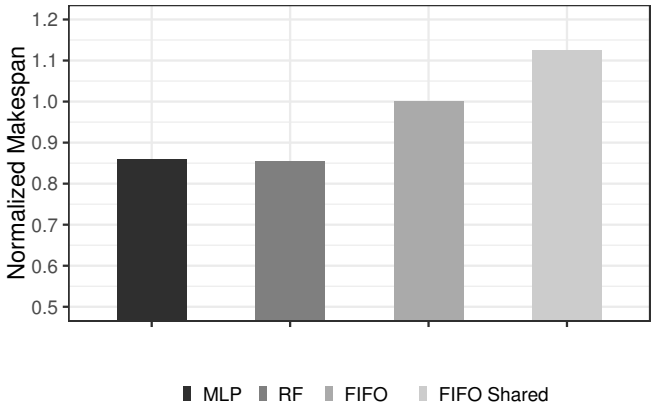

Fig. 6: Normalized makespan for each approach to execute 20 randomly-generated queues.

3) Greedy approach comparison: We also evaluate a greedy strategy to assess if a simpler approach would also yield satisfactory results. The greedy strategy (a) predicted the degradation for all pairs of applications; however, instead of creating a degradation graph it created an ordered list of application pairs, sorted from the lower to highest degradation when colocated; and (b) selected the pair with the lowest degradation to execute.

Figure 7 presents the makespan results normalized to FIFO, for optimal and greedy strategies. Both strategies were able to outperform the FIFO execution and the greedy strategy achieved performance similar to that of the optimal strategy. This happens as we apply the threshold that cut down pairs with excessive degradation (in our case higher than 100%). In this case, the greedy goal of picking always the lower degraded pairs took more advantage than the well balanced optimal strategy, since the applications in pairs with excessive degradation will be executed alone when the threshold is applied. Note that using a simpler strategy already reduces the makespan significantly (about 15%) over FIFO.

4) Overhead analysis: In this experiment, we evaluate the time taken to compute a scheduling solution. It includes the time to predict the degradation of each pair of applications

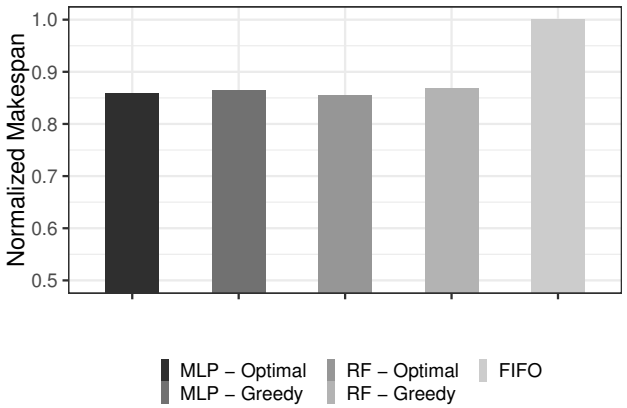

Fig. 7: Normalized makespan for optimal and greedy strategies to execute all queues.

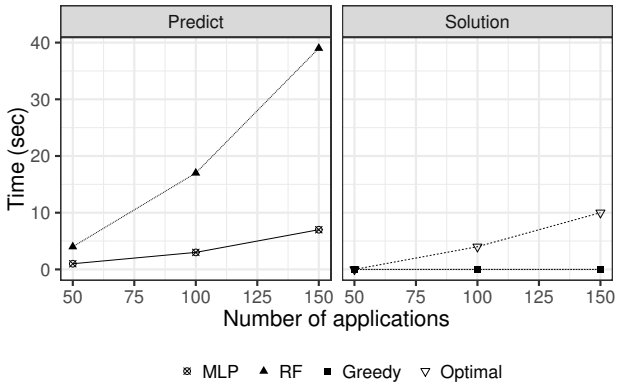

Fig. 8: Measured time spent to predict colocation and produce scheduling decisions using two strategies: Optimal and Greedy.

offline and the time to compute the scheduling solution, solving the graph matching problem.

Figure 8 presents the time measured to predict all degradation pairs (left) and apply (right) for the optimal and greedy strategies. It was measured multiple times, yielding standard deviation of 0.2% to 5.5%. The prediction using the MLP Regressor is much faster than using Random Forest because the latter averages the predictions of several base estimators, thus increasing the time spent on the prediction phase, especially when there are more than 50 applications.

To compute the final scheduling solution, greedy (0.01-0.15 ms) shows much lower processing time than optimal strategy (0-10s). The time spent to compute the scheduling with greedy never exceeded 1 second in our evaluation, while optimal spent up to 10 seconds. For optimal strategy, the time increased 2.5x when we varied the number of queued applications from 100 to 150.

Given the results presented, we can observe that the MLP Regressor model and adopting a greedy strategy could best balance prediction time and effectiveness in reducing the makespan. Moreover, its lower overhead allows for scaling with much higher number of jobs. This is important in real systems, since our scheme could be triggered at shorter intervals without negatively impacting the scheduling manager.

IV. Related Work

Georgiou *et al.* [23] implemented in Slurm three powercap policies for dealing with power limitations in large scale HPC clusters. Ellsworth *et al.* [24] implemented power-aware Slurm plugins to explore and compare power management strategies using existing hardware platforms. Rajagopal *et al.* [25] developed a power-aware mechanism integrated within the Slurm to improve the system resource utilization and to reduce job waiting times by redistributing the system power under strict powercap regime. Sakamoto *et al.* [26] proposes a power-aware resource manager based on the Slurm to maintain a system wide power limit using a set of interfaces in combination with portable APIs for power measurement and control. Simakov *et al.* [27] experiment on a Slurm simulator to study the benefits of node sharing. Dynamic Voltage and Frequency Scaling (DVFS) and Intel Running Average Power Limit (Intel RAPL) techniques have been extensively studied [28]–[31] to improve energy efficiency and system throughput under over-provisioned scenarios.

To avoid negative interference between workloads Bubble-Up [1] uses the application's sensitivity and contentiousness to predict the degradation due to contention sharing the memory subsystem. The profiling complexity for all pairwise colocations with Bubble-Up would be $\mathcal{O}(N^2)$ (N as the number of applications). Alves *et al.* [32] provides a model based on multiple regression analysis and trained using micro-benchmarks to predict the average slowdown of collocated applications, given the LLC, memory and network pressures. The work is extended in [33] to provide a scheduler that colocates Virtual Machines (VMs) using an Integer Linear Programming (ILP) formulation to minimize the number of physical machines. In contrast to these works, our solution is designed to be readily adopted as a plugin in an open-source cluster resource manager.

Dwyer *et al.* [7] investigate the effectiveness of machine learning on predicting performance degradation due to colocation of applications. Their model estimates the degradation based on per-core and system-wide hardware counter values. For multithreaded applications the model could be used to estimate the degradation for each thread. Then, the scheduler can decide whether is necessary to allocate more hardware by averaging the degradation of all threads.

Delimitrou *et al.* [34], [35] propose a cluster manager that maximizes resource utilization while meeting workload performance requirement. It uses classification to determine the impact of the amount of resources, type of resources and the interference on performance for each workload. The classification approach eliminates the need for exhaustive online characterization. Regularization models [10] were previously explored for predicting application interference. These

models combined linear and non-linear approaches to produce accurate predictions. The models were trained with low-level hardware features (instructions retired, cache miss, etc) acquired from off-line measurements.

V. Conclusion

We propose a solution that works by predicting degradation of colocated applications and deciding a combination with minimum degradation. This way we can better allocate the computing capacity of the server systems, and hence improve server efficiency. We experimentally demonstrated that using hardware counters in machine learning is a promising alternative to tackle this problem. We implemented our solution in an open-source cluster resource manager, Slurm, as a plugin in its scheduling decision process. We carried out several experiments to evaluate its effectiveness in a practical setting.

Our experimental results showed that machine learning models, in particular Random Forest, achieves good accuracy (78%) in the evaluation phase, outperforming other models such as regression. Our solution achieved performance improvements of 15% (avg) and 26% (max) compared to the policies used by the Slurm resource manager. The solution based on MLP Regressor model and greedy strategy presented a well balanced approach considering both makespan improvement and overhead to compute the scheduling solutions.

Acknowledgement

This work is part of a project that has received funding from the European Unions Horizon 2020 research and innovation programme under grant agreement No 754337 (EuroEXA); it has been supported by the Spanish Ministry of Science and Technology (project TIN2015-65316-P), Generalitat de Catalunya (contracts 2014-SGR-1051 and 2014-SGR-1272), and the Severo Ochoa Programme (SEV-2015-0493). The work is also supported by FAPESB (Edital JCB 008/2015).

References

[1] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *MICRO*. ACM, 2011, pp. 248–259.

[2] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers," in *ACM SIGARCH*, vol. 41, no. 3. ACM, 2013, pp. 607–618.

[3] J. Breitbart, J. Weidendorfer, and C. Trinitis, "Case study on co-scheduling for hpc applications," in *ICPPW*. IEEE, 2015, pp. 277–285.

[4] S. Blagodurov, S. Zhuravlev, and A. Fedorova, "Contention-aware scheduling on multicore systems," *TOCS*, vol. 28, no. 4, p. 8, 2010.

[5] R. Nishtala, D. Mossé, and V. Petrucci, "Energy-aware thread co-location in heterogeneous multicore processors," in *EMSOFT*, Sept 2013.

[6] L. Tang, J. Mars, W. Wang, T. Dey, and M. L. Soffa, "Reqos: Reactive static/dynamic compilation for qos in warehouse scale computers," in *ACM SIGPLAN Notices*, vol. 48, no. 4. ACM, 2013, pp. 89–100.

[7] T. Dwyer, A. Fedorova, S. Blagodurov, M. Roth, F. Gaud, and J. Pei, "A practical method for estimating performance degradation on multicore processors, and its application to HPC workloads," in *SC*. IEEE, 2012.

[8] A. de Blanche and T. Lundqvist, "Disallowing same-program co-schedules to improve efficiency in quad-core servers." in *COSH/VisorHPC@ HiPEAC*, 2017, pp. 13–20.

[9] A. Merkel, J. Stoess, and F. Bellosa, "Resource-conscious scheduling for energy efficiency on multicore processors," in *EuroSys*. ACM, 2010.

[10] N. Mishra, J. D. Lafferty, and H. Hoffmann, "Esp: A machine learning approach to predicting application interference," 2017.

[11] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *ISCA*. New York, NY, USA: ACM, 2015, pp. 158–169.

[12] Y. Jiang, X. Shen, C. Jie, and R. Tripathi, "Analysis and approximation of optimal co-scheduling on chip multiprocessors," in *PACT*. IEEE, 2008, pp. 220–229.

[13] J. Edmonds, "Maximum matching and a polyhedron with 0, 1-vertices," *Journal of research of the National Bureau of Standards B*, vol. 69, no. 125-130, pp. 55–56, 1965.

[14] A. B. Yoo, M. A. Jette, and M. Grondona, "Slurm: Simple linux utility for resource management," in *JSSPP*. Springer, 2003, pp. 44–60.

[15] H. Zou and T. Hastie, "Regularization and variable selection via the elastic net," *Journal of the Royal Statistical Society, Series B*, vol. 67, pp. 301–320, 2005.

[16] B. E. Boser, I. M. Guyon, and V. N. Vapnik, "A training algorithm for optimal margin classifiers," in *COLT*. ACM, 1992, pp. 144–152.

[17] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, Oct. 2001. [Online]. Available: https://doi.org/10.1023/A:1010933404324

[18] "Scikit-learn: Machine learning in python," http://scikit-learn.org/stable/index.html, 2018, accessed: 2018-03-01.

[19] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *PACT*. ACM, 2008, pp. 72–81.

[20] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC*. Washington, DC, USA: IEEE, 2009, pp. 44–54.

[21] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The nas parallel benchmarks: Summary and preliminary results," in *SC*. New York, NY, USA: ACM, 1991, pp. 158–165.

[22] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros, "Splash-3: A properly synchronized benchmark suite for contemporary research," in *ISPASS*. IEEE, 2016, pp. 101–111.

[23] Y. Georgiou, D. Glesser, and D. Trystram, "Adaptive resource and job management for limited power consumption," in *IPDPSW*. IEEE, 2015.

[24] D. Ellsworth, T. Patki, M. Schulz, B. Rountree, and A. Malony, "A unified platform for exploring power management strategies," in *E2SC*. IEEE, 2016, pp. 24–30.

[25] D. Rajagopal, D. Tafani, Y. Georgiou, D. Glesser, and M. Ott, "A novel approach for job scheduling optimizations under power cap for arm and intel hpc systems," in *HiPC*. IEEE, 2017, pp. 142–151.

[26] R. Sakamoto, T. Cao, M. Kondo, K. Inoue, M. Ueda, T. Patki, D. Ellsworth, B. Rountree, and M. Schulz, "Production hardware overprovisioning: real-world performance optimization using an extensible power-aware resource management framework," in *IPDPS*. IEEE, 2017.

[27] N. A. Simakov, R. L. DeLeon, M. D. Innus, M. D. Jones, J. P. White, S. M. Gallo, A. K. Patra, and T. R. Furlani, "Slurm simulator: Improving slurm scheduler performance on large hpc systems by utilization of multiple controllers and node sharing," in *PEARC*. ACM, 2018, p. 25.

[28] B. Rountree, D. H. Ahn, B. R. de Supinski, D. K. Lowenthal, and M. Schulz, "Beyond dvfs: A first look at performance under a hardware-enforced power bound," in *IPDPS*, May 2012, pp. 947–953.

[29] O. Sarood, A. Langer, A. Gupta, and L. Kale, "Maximizing throughput of overprovisioned hpc data centers under a strict power budget," in *SC*. IEEE Press, 2014, pp. 807–818.

[30] D. A. Ellsworth, A. D. Malony, B. Rountree, and M. Schulz, "Dynamic power sharing for higher job throughput," in *SC*. ACM, 2015, p. 80.

[31] T. Patki, D. K. Lowenthal, A. Sasidharan, M. Maiterth, B. L. Rountree, M. Schulz, and B. R. De Supinski, "Practical resource management in power-constrained, high performance computing," in *ICPADS*. ACM, 2015, pp. 121–132.

[32] M. M. Alves and L. M. de Assumpção Drummond, "A multivariate and quantitative model for predicting cross-application interference in virtual environments," *Journal of Systems and Software*, vol. 128, 2017.

[33] M. M. Alves, L. Teylo, Y. Frota, and L. M. Drummond, "An interference-aware virtual machine placement strategy for high performance computing applications in clouds," in *WSCAD*. IEEE, 2018, pp. 94–100.

[34] C. Delimitrou and C. Kozyrakis, "Quasar: resource-efficient and qos-aware cluster management," in *ACM SIGPLAN Notices*. ACM, 2014.

[35] ——, "Paragon: Qos-aware scheduling for heterogeneous datacenters," in *ACM SIGPLAN Notices*. ACM, 2013.