Are We There Yet?
Predicting if Executing Applications are Near Completion

Mohammad Sonji
mms158@mail.aub.edu
American University of Beirut
Beirut, Lebanon

Mohammed Baydoun
mohbay@gmail.com
American University of Beirut
Beirut, Lebanon

Safaa Diab
syd04@mail.aub.edu
American University of Beirut
Beirut, Lebanon

Amir Nassereldine
aan34@mail.aub.edu
American University of Beirut
Beirut, Lebanon

Pedro Bruel
bruel@hpe.com
Hewlett Packard Enterprise Labs
Milpitas, CA, USA

Aditya Dhakal
aditya.dhakal@hpe.com
Hewlett Packard Enterprise Labs
Milpitas, CA, USA

Rolando Pablo Hong Enriquez
rhong@hpe.com
Hewlett Packard Enterprise Labs
Milpitas, CA, USA

Gourav Rattihalli
gourav.rattihalli@hpe.com
Hewlett Packard Enterprise Labs
Milpitas, CA, USA

Diman Zad Tootaghaj
diman.zad-tootaghaj@hpe.com
Hewlett Packard Enterprise Labs
Milpitas, CA, USA

Gallig Renaud
gallig.renaud@hpe.com
Hewlett Packard Enterprise Labs
Milpitas, CA, USA

Barbara Chapman
barbara.chapman@hpe.com
Hewlett Packard Enterprise Labs
Milpitas, CA, USA

Fatima K. Abu Salem
fa21@aub.edu.lb
American University of Beirut
Beirut, Lebanon

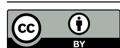
Eitan Frachtenberg
eitan.frachtenberg@hpe.com
Hewlett Packard Enterprise Labs
Milpitas, CA, USA

Dejan Milojicic
dejan.milojicic@hpe.com
Hewlett Packard Enterprise Labs
Milpitas, CA, USA

Izzat El Hajj
izzat.elhajj@aub.edu.lb
American University of Beirut
Beirut, Lebanon

Abstract

Predicting the running time or remaining time of batch-style applications is useful to schedulers and resource managers. However, it is fundamentally challenging to make such predictions accurately for applications that have not been seen before or that run on datasets with varying sizes. For this reason, we aim to answer a simpler, but nevertheless instrumental, question: is an executing application about to finish executing? To this end, we present AWTY, a workflow for predicting whether or not a running batch-style application is near completion. AWTY analyzes application profiles to identify what applications' last phases look like while treating applications as black boxes. It then uses this data to train classifiers that can identify whether or not an executing application is in its last phase. AWTY employs both single-application classifiers that work on applications that have been seen before and general classifiers that work on applications that have not been seen before. Our evaluation shows that AWTY can predict if an application is near completion reasonably well. AWTY can inform schedulers and resource managers in making decisions about whether to kill applications that have overstayed their time or to let them finish.



This work is licensed under a Creative Commons Attribution 4.0 International License.

ICPE '26, Florence, Italy

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2325-4/2026/05
https://doi.org/10.1145/3777884.3796991

CCS Concepts

• **General and reference** → **Metrics**; **Measurement**.

ACM Reference Format:
Mohammad Sonji, Mohammed Baydoun, Safaa Diab, Amir Nassereldine, Pedro Bruel, Aditya Dhakal, Rolando Pablo Hong Enriquez, Gourav Rattihalli, Diman Zad Tootaghaj, Gallig Renaud, Barbara Chapman, Fatima K. Abu Salem, Eitan Frachtenberg, Dejan Milojicic, and Izzat El Hajj. 2026. Are We There Yet? Predicting if Executing Applications are Near Completion. In *Proceedings of the 17th ACM/SPEC International Conference on Performance Engineering (ICPE '26), May 04–08, 2026, Florence, Italy*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3777884.3796991

1 Introduction

Knowing how long a batch-style application will take to execute is useful for schedulers and resource managers. For this reason, many prior works attempt to predict the running time of such an application before it executes [8, 29, 32, 36, 56, 65–67, 71, 75, 90–92, 99, 101, 115], or the remaining time of an application that is already executing [18, 104, 105]. However, these works tend to rely on running times being repetitive and do not perform well for new applications or applications with significantly varying datasets across runs. While some works adapt their running time predictions to different datasets [8, 29, 67], they use application-specific input features that describe the dataset and require an application to have been executed before on different datasets. The use of customized features makes it difficult to deploy such works in general purpose

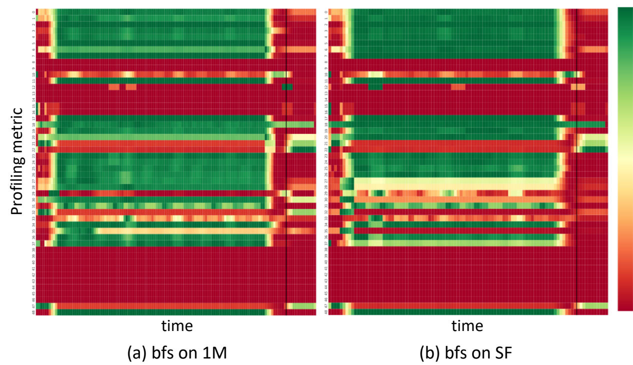


Figure 1: Time series of profiling metrics for breadth-first search [97] executing on two different graph datasets (the black vertical line indicates the beginning of the last phase)

environments that run many different applications [52]. It remains fundamentally challenging to make accurate running time or remaining time predictions using application-agnostic features while adapting to new datasets, let alone new applications.

In light of the aforementioned challenges, our work aims to answer a simpler, but nevertheless instrumental, question. Rather than asking how much time an application needs to finish executing, we ask whether or not an executing application is near completion. In particular, we target batch-style applications typical in scientific computing and offline data analytics, and we do not aim to detect completion for always-on services without a natural endpoint. We expect that this question can more easily be answered using application-agnostic features while adapting to new datasets and applications. Our rationale behind this expectation is threefold. First, we expect applications to behave differently in their last phase of execution than they do during their main phase of execution. For example, Figure 1(a) shows the time series of application-agnostic profiling metrics for breadth-first search (see Section 4 for the details of the experimental setup and visualization). It is clear that the values of these metrics change at the end of the time series. Second, we expect an application to behave in the same way in its last phase of execution regardless of the dataset it has executed on. For example, Figure 1(b) shows the time series of profiling metrics for breadth-first search executed on a different dataset from that in Figure 1(a). It is clear that the last phases in the two time series have a lot in common, notably an increase in L1-dcache-stores (metric 21), L1-icache-load-misses (metric 22), dTLB-stores (metric 32), and mem-stores (metric 47). Third, we expect that different applications, despite having different running times, may have similar behavior in their last phases of execution. We show an example of this similarity in Section 5.2.

Based on these observations, we present *Are We There Yet* (AWTY), a workflow for predicting whether or not a running application is near completion. AWTY analyses a large number of application profiles and labels their last phases. It then uses this information to train single-application and general classifiers that predict whether or not a running application is near completion. At run time, a running application in question is profiled for a few sampling periods. These samples are passed to the pre-trained classifiers to classify if the application is in its last phase or not. If the application has been seen before, the single-application classifier for that application is used, whereas if the application has not been seen before, the general classifier is used.

We evaluate AWTY using 44 data analytics and scientific computing benchmarks from six benchmark suites and libraries. Our evaluation shows that AWTY achieves mean F2 scores of 0.94, 0.77, and 0.77 for the single-application classifiers when the dataset has been seen before, the single-application classifiers when the dataset has not been seen before, and the general classifier where the application has not been seen before, respectively. Moreover, since the F2 scores vary significantly across applications, AWTY can distinguish between applications for which it can make accurate predictions and applications for which it can not. Our evaluation demonstrates AWTY on a production multi-node application as well. It also shows that AWTY's performance replicates across different systems.

AWTY can be used in schedulers and resource managers for a variety of purposes. For example, in a shared system with users having different priorities, when a high-priority job arrives, AWTY's predictions can be used to decide whether to kill a low-priority job (e.g., a spot instance in the cloud) or to let it finish. As another example, in a scheduler that uses predicted running times for job scheduling, if an application's running time has been underpredicted and the application has exceeded its time budget, AWTY's predictions can help the scheduler decide whether to give the application a little more time to finish or to kill it. Integrating AWTY into such systems is the subject of future work. Our main objective in this paper is to validate the hypothesis that application last-phases can indeed be learned and predicted, and to develop a mechanism for doing so.

The rest of this paper is organized as follows. Section 2 provides background to the problem and reviews related works. Section 3 describes AWTY and its components. Section 4 describes our methodology and Section 5 evaluates the effectiveness of our approach. Finally, Section 6 concludes.

2 Related Work

Many prior works aim to predict different aspects of the performance of applications, such as running time, remaining time, or relative performance across different systems and configurations. These predictions are used for different purposes such as resource management or best configuration selection. This section presents an overview of these prior works and contrasts them with our work, which aims to predict whether an application has neared completion. We also review prior works on phase detection, which is a component of our work.

Running time prediction. The expected running time of an application is an important input parameter in many schedulers and resource managers [3, 28, 61, 62, 73, 89, 95, 110, 113], including for the backfilling [33, 57] process. Since user-specified times tend to be inaccurate [7, 51, 69, 100], predicting the running time of an application is an important problem [29, 32, 36, 56, 65, 66, 71, 75, 90–92, 101, 115]. Tsafrir et al. [101] predict running time using the average running time of the last two jobs from the same user.

Gaussier et al. [36] use linear regression models that take the features from the job's description, historical information, and the current state of the system. PREP [115] also considers the job's running path as part of the input features. TRIP [32] uses a Tobit model to achieve good prediction accuracy while also keeping the underprediction rate low. Park and Kim [75] predict running time using a support vector regression model. OKCM [56] uses k-nearest neighbors and falls back to the user-specified time in the case of underprediction. Smith et al. [91, 92] use genetic algorithms to identify application categories and predict running times based on an application's category. Naghshnejad and Singhal [71] use generative state space models to predict running time while tolerating the high variability in the cloud. Smith [90] uses instance-based learning to predict running time as well as file transfer time and queue wait time. Duan et al. [29] use a Bayesian neural network to predict the running time of scientific workflow activities in a grid. Matsunaga et al. [65] and McKenna et al. [66] compare the effectiveness of multiple different machine learning algorithms at predicting running time. While these works achieve reasonable accuracy with varying levels of success, they tend to rely on running times being repetitive and do not perform as well for new applications or applications with significantly varying datasets across runs. Moreover, performance variability can cause significant slowdown making user provided runtime estimates unreliable [9, 30, 64]. Our work supplements these works by providing a second level of defense. If an application's running time has been underpredicted and a scheduler has used that prediction, when the application exceeds its time budget, our work can assist that scheduler in deciding whether to give the application a little more time to finish or to kill it.

Application-specific running time prediction. Multiple prior works predict the running times of specific applications and how these running times vary across different datasets and system configurations. Mendes and Reed [67] make predictions for a PDE application for different datasets. Duan et al. [29] make predictions for four different scientific workflows for different datasets. Baughman et al. [8] make predictions for six different bioinformatics workflows. To predict running times across different datasets, these works use application-specific input features that describe the dataset and require an application to have been executed before on different datasets. In contrast, our work makes predictions for arbitrary applications that may have not been seen before. It treats applications as black boxes and uses application-agnostic features instead of using manually selected application-specific input parameters such as properties of the dataset.

Remaining time prediction. A number of works predict the remaining time to completion of an already running application to overcome the possible inaccuracy of running time prediction. RLSchert [104, 105] predicts the remaining time of a job using intermediate logs passed to a recurrent neural network as well as the job's input parameters. However, it is specific to VASP jobs and uses VASP-specific input parameters to make predictions. In contrast, our work focuses on making predictions for arbitrary applications and does not use any application-specific input parameters. Chen et al. [18] predict an arbitrary job's remaining time using system log information passed to Hidden Markov Models. In our work, we use hardware and software counters collected by profiling to make predictions, however, combining profiling information with system log information to make predictions is interesting for future work. Furthermore, our work does not attempt to predict the remaining time, but instead, simplifies the problem to only predicting whether a job is near completion or not.

Performance-cost trade-off prediction. Many prior works predict the performance of applications across different systems and configurations and infer the corresponding cost [1, 8, 19, 25, 26, 58, 63, 72, 103, 108, 109]. These works are not concerned with predicting the absolute running time of an application for a specific dataset. Instead, they may use a proxy application, a representative dataset, or a partial run to characterize how the application's performance and cost vary across systems and configurations because their goal is to assist users in selecting the best system and configuration for their application in general. The goal of our work is different. We aim to predict whether a specific application that is already running with a specific dataset has neared completion.

Near optimal system/configuration prediction. Many prior works predict the near-optimal system and configuration for running a specific application [4, 10, 11, 14, 44–46, 50, 59, 60, 106, 107]. Similar to the previous category, the goal of these works is to assist users in selecting the best system and configuration to run on. Again, the goal of our work is different. We aim to predict whether a specific application that is already running with a specific dataset has neared completion.

Phase detection. Detecting phases in applications has been the subject of much research [23]. Phase detection has been used for different objectives such as adapting hardware resources to application phases [16, 27, 37, 47, 112] or reducing hardware simulation time by simulating each distinct phase only once [40, 53, 54, 76, 79, 86]. In our work, we specifically aim to detect the last phase of an application to be able to predict whether an application has neared completion. Some phase detectors detect phases offline [12, 13, 34, 39, 43, 48, 78, 80, 84, 85, 93, 96, 114] for the purpose of application characterization and optimization, while others detect phases online [20, 21, 35, 38, 49, 81–83, 94, 98] for the purpose of real-time adaptation. In our work, we detect phases offline to identify the last phase of previously executed applications to train our prediction models. The online aspect of our tool is concerned with detecting if an application is currently in its last phase, not detecting whenever a phase change occurs. Prior works on phase detection are distinguished by many aspects such as the type of sampling interval used (per instruction [48, 49, 78, 81, 83, 85, 94, 114] or per unit time [35, 84, 93]), the choice of classification metrics (microarchitecture-dependent [21, 31, 43, 80, 83, 84, 87, 88, 114] or microarchitecture-independent [12, 38, 39, 48, 84, 88, 94]), and the phase analysis method (e.g., clustering [82], Manhattan distance [12, 20, 35, 38, 78, 81, 83, 94, 114], digital signal processing [13, 21, 48, 49, 85, 93], etc.). In our work, we sample applications per unit time, use microarchitecture-dependent metrics, and perform phase analysis based on the Manhattan distance [20]. However, other phase detection strategies proposed in prior works can also be used and our contribution is orthogonal to the choice of phase detection strategy. For example, we also tried performing phase analysis based on clustering and found similar phases, but we opted for performing phase analysis based on the Manhattan distance because it was faster.

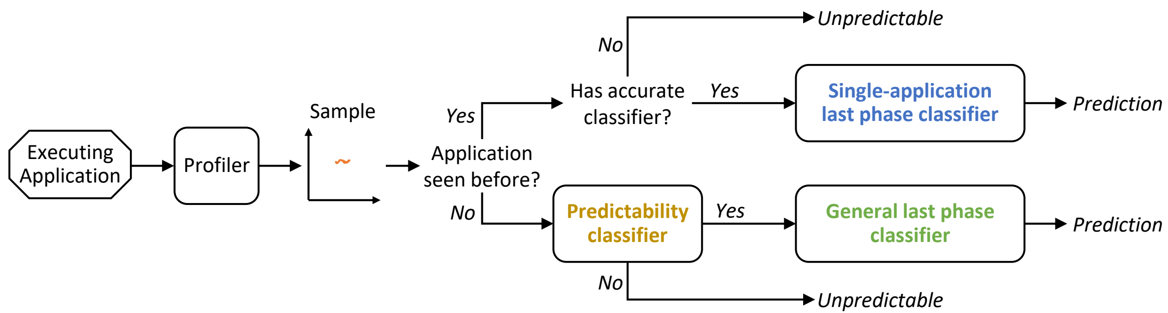


Figure 2: AWTY Run-time Workflow

3 AWTY Design and Implementation

This section describes the overall design and implementation of AWTY. Section 3.1 describes the run-time workflow for predicting is an executing application is near completion. Section 3.2 describes the steps taken to train the prediction models used in that workflow. We then zoom in on specific stages of the workflow including the last-phase detection stage (Section 3.3), the different classifiers (Sections 3.4 and 3.5), and the handling of multi-node applications (Section 3.6).

3.1 Run-time Workflow

The run-time workflow of AWTY is shown in Figure 2. We first attach a profiler to the executing application of interest and extract a sequence of a few profiling samples. We then check if the application has been seen before, i.e., if we have included runs from the application in the training of our classifiers.

If the application has been seen before, we check if the application's single-application last phase classifier is sufficiently accurate. The *single-application last phase classifier* classifies whether or not a given sequence of profiling samples is in the last phase of a specific application's execution, i.e., it predicts if the executing application has neared completion (details in Section 3.4). If the application's classifier is not sufficiently accurate, we declare that we cannot make a good prediction. If the application's classifier is sufficiently accurate, we pass the extracted sequence of profiling samples to the application's classifier and make a prediction.

If the application has *not* been seen before, we pass the extracted sequence of profiling samples to the predictability classifier. The *predictability classifier* classifies whether or not a given sequence of profiling samples came from an application that is easy or difficult to make predictions for, i.e., the application's *predictability* (details in Section 3.4). If the classifier classifies the samples as coming from an application with low predictability, we declare that we cannot make an accurate prediction. On the other hand, if the classifier classifies the samples as coming from an application with high predictability, we pass the extracted sequence of profiling samples to the general last phase classifier. The *general last phase classifier* classifies whether or not a given sequence of profiling samples is in the last phase of the execution of whatever application it came from, i.e., it predicts if an application that has not been seen before has neared completion (details in Section 3.4).

In an alternative design, the predictability classifier and general classifier can be combined into one classifier with three output classes: not in the last phase, in the last phase, and unpredictable. We tried this design and found that separating the two classifiers yields better results. In particular, separating the two classifiers allows them to be trained with different datasets for increased specialization of the general classifier. As we see in Section 3.2, the predictability classifier is trained using data from all applications, whereas the general classifier is trained with data from applications with high predictability only.

To know if an application has been seen before, we rely on the name of the application. This approach is sufficient for the common case where users run the same application repetitively without changing its name. More robust approaches can be used for detecting if an application has been seen before, such as taking a hash of the binary or combining the application name with the username. However, this aspect of the work is orthogonal to our main contribution.

3.2 Training Workflow

The training workflow of AWTY is shown in Figure 3. It is executed offline when AWTY is first deployed on the system of interest to train the classifiers used in the run-time workflow. The classifiers are not retrained online when new applications are observed. The training workflow can be re-executed periodically to benefit from newly collected application data.

The training workflow takes many application profiles as input. An application profile is a time series of profiling metric values collected during the entirety of an application's execution. The application profiles are first analyzed to detect the last phase in each time series (details in Section 3.3). The result of this analysis is a last-phase label for each point in each time series that indicates whether that point belongs to the last phase of the time series or not.

The labeled points are used to train a single-application last-phase classifier for each application. The trained classifiers are then evaluated, and each application receives a predictability label indicating whether or not it has high predictability, based on whether or not its classifier's quality score is above a certain threshold (details in Section 3.4). These predictability labels are used by the run-time workflow to decide whether an application that has been seen before proceeds to its single-application classifier or is declared as

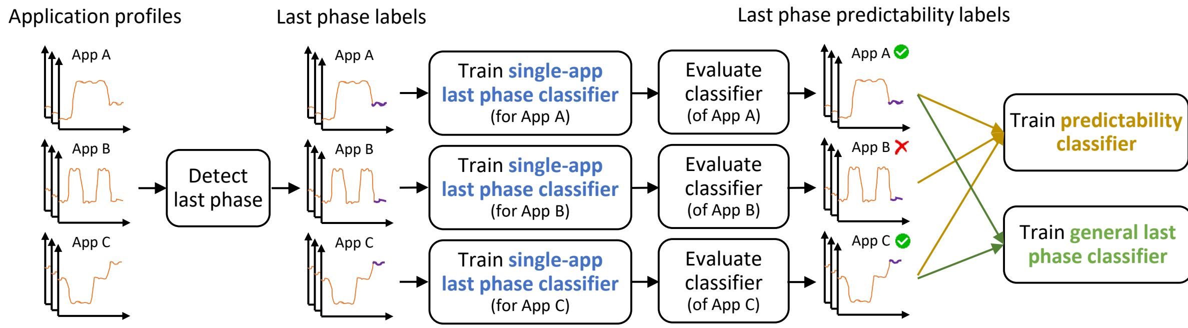


Figure 3: AWTY Training Workflow

unpredictable. The predictability labels are also used to train the predictability classifier that predicts if new unseen applications have high predictability. Moreover, the profiles of the applications with accurate single-application classifiers are jointly used to train the general last-phase classifier to predict if new unseen applications are in their last phase.

3.3 Last Phase Detection

The last phase detection stage takes application profiles as input. An application profile is a time series of profiling metric values collected during an application's execution. That is, every point in the time series represents a vector of profiling metric values collected over a sampling period. These time series may be collected from different runs of different applications, the same application with different datasets, or the same application with the same dataset. The specific applications, datasets, and profiling metrics used in our evaluation are reported in Section 4.

To identify the last phase in an application's profile, we first identify all the points at which a phase change occurs. We detect phase changes using the same method from Chetsa et al. [20] based on the Manhattan distance. Briefly, the technique detects a phase change whenever the Manhattan distance between two consecutive points in the profile exceeds a certain threshold. The threshold is determined relative to all the distances encountered in the application's profile since different applications may have different degrees of variability across time points. To be robust against noisy perturbations in the profile, we modify the technique in Chetsa et al. [20] to take the distance between the mean of a window of past points and the mean of a window of future points, rather than the distance between individual points. We use five and three as the sizes of the past and future windows, respectively, but these values can be tuned depending on their desired sensitivity to perturbations.

After identifying the points at which phase changes occur, we pick the last such point as the beginning of the last phase provided that it falls within the last 10% of the time series but before the last 1%. Different applications can have different last phase lengths. The motivation for requiring the last phase to start within the last 10% of execution is that we ultimately would like to predict that an application is near completion. Hence, if the last phase is too long, the prediction will not be relevant. The motivation for requiring the last phase to start before the last 1% of execution is to avoid minor perturbations we have seen to occur in the last one or two points in the time series. If no phase change is detected between the last 10% and 1% of execution, we pick the starting point of the last 10% of execution as the beginning of the last phase. Although we use the last 10% and last 1% markers to dilineate the start of the last phase, these values are configurable and can be tuned depending on the context. Once we have picked a point as the beginning of the last phase, we label it and all subsequent points as being part of the last phase, and we label all prior points as being not part of the last phase. This last phase serves as the basis for our "near completion" classification.

3.4 Last Phase Classifiers

The last phase classifiers take as input a sequence of profiling samples and classify whether or not that sequence of profiling samples came from the last phase of an application's execution. We train these classifiers using the labeled time series generated in the last phase detection stage.

For all last phase classifiers (single-application and general), we use a Recurrent Neural Network (RNN) with the following architecture. The input layer takes five consecutive points from the time series each consisting of that point's profiling metric values. The input layer is followed by two bidirectional LSTM layers, the first having 64 units and the second having 32 units. The LSTM layers are followed by a batch normalization layer then four fully connected layers with 32, 16, 8, and 4 units before the final output layer. We experimented with having longer sequences at the input, larger layers, and more layers in the network, but we found that it did not improve performance and led to overfitting.

Using the same model architecture and input parameters for all the single-application last-phase classifiers of the different applications is an important design choice. This design choice allows us to seamlessly support any application that we have previously profiled by treating applications as black boxes. In contrast, using application-specific input features, such as properties of the dataset or application output logs, would require manual feature selection for every new application which would limit the scalability of our framework to a large number of applications.

The key difference between the classifiers is not their architecture or input parameters, but the data used to train them. We have a single-application classifier for each application that is trained with data collected from just that application. On the other hand, we only have one general classifier that is trained with data collected

from all the applications whose single-application classifier quality score exceeds a certain threshold.

We use the weighted harmonic mean F-beta score (beta=2) from Scikit-Learn [77] as the quality score, and we use a threshold of 0.7 to consider an application as having high predictability. However, other quality scores and thresholds may be used by the deployer depending on the deployer's priorities. Our motivation for using the F2 score is to give more weight to recall, because our positive class (points in the last phase) is a minority class and because reducing false negatives is more important than reducing false positives in our context. For example, assume AWTY was used by a scheduler to kill applications that exceed their time budget and are not close to completion. A false negative would result in killing an application that is in its last phase, which wastes all the resources that would be needed to rerun that application. On the other hand, a false positive would result in giving more slack time to an application that is not in its last phase and then killing it, which only wastes the resources used for that additional slack time.

One characteristic of our training data is its high imbalance. First, the number of time series points that are in the last phase is much smaller than the number of points that are not in the last phase. Second, the number of time series points collected for an application may significantly vary across datasets. Third, for the general predictor, the number of time series points collected may significantly vary across applications since different applications have different running times. To avoid biasing the models towards non-last phase points, larger datasets, and longer-running applications, we rebalance the data before training. We use the Adaptive Synthetic (ADASYN) sampling technique [42] from Imbalanced-Learn [55] for rebalancing.

The rationale behind the single-application last-phase classifiers is that we expect the last phase of an application to look similar across executions regardless of what dataset the application executes on. The reason is that we expect an application to perform the same kind of wrap-up activities at the end of its execution for any dataset. The rationale behind the general last phase classifier is that we expect the last phases of different applications to look similar because different applications may perform similar wrap-up activities when they are close to finishing. We evaluate the success of the single-application and general last phase classifiers in Sections 5.1 and 5.2, respectively.

3.5 Predictability Classifier

The predictability classifier takes as input a sequence of profiling samples and classifies whether that sequence came from an application that has high or low predictability. We use an RNN with the same architecture as that used for the last phase classifiers which has been described in Section 3.4. To label an application as having high predictability, its single-application classifier quality score must exceed a certain threshold, and we use the F2 score and a threshold of 0.7 as stated in Section 3.4. However, other quality scores and thresholds may be used by the deployer depending on the deployer's priorities.

The rationale behind the predictability classifier is that we expect applications with high predictability to have common characteristics, and likewise for applications with low predictability. We evaluate the success of the predictability classifier in Section 5.2.

3.6 Multi-node Profiling and Prediction

For applications running on multiple nodes, profilers typically provide multiple time series for the application, one for each core. In this case, one challenge is how to consolidate the different time series to make a prediction. We explore three different approaches for handling multi-node applications:

1. **Select then predict**: We make a prediction based on the profile of a single selected core.
2. **Predict then aggregate**: We make a prediction for each time series separately, then aggregate the prediction by checking if at least one-third of the time series are in their last phase.
3. **Aggregate then predict**: We aggregate multiple time series into one (by taking the mean across all the time series for each time point) and make a prediction based on the aggregated time series.

The advantage of the first approach is that it avoids having to gather profiling information from multiple nodes. The advantage of the latter two approaches is that they leverage more information from multiple cores/nodes, which is particularly important for asymmetrical applications where activities on one core may not be representative of the entire application. We evaluate the quality of each approach in Section 5.3.

4 Methodology

Our training and inference workflows are implemented in Python. We use NumPy [41] and pandas [74] for performing the data preparation and pre-processing tasks. We use TensorFlow [2] for implementing the RNN models, and we also compare the RNN's performance to a dummy classifier [77] and XGBoost [17]. We use the ADASYN [42] technique from Imbalanced-Learn [55] for balancing the training data. We use a standard scaler [77] to normalize the features before providing them to the models. We use the F2 score [77] as the main classification quality score for the reasons discussed in Section 3.4.

We use leave-one-group-out cross-validation from Scikit-Learn [77] to evaluate the performance of all the classifiers. For the predictability and general classifiers, all runs of an application on all datasets serve as a group, such that the training set only contains runs of different applications. For the single-application classifiers, we evaluate two cases: (1) when the dataset has been during training, and (2) when the dataset has not been seen during training. In the first case, a single run on a single dataset serves as a group, such that the training set contains other runs of the application on both the same and different datasets. In the second case, all runs on the same dataset serve as a group, such that the training set only contains other runs of the application on different datasets. Note that our run-time workflow only contains one single-application classifier per application. It does not differentiate whether or not a dataset has been seen before. The objective of evaluating the two aforementioned cases is to show how the classifier would perform if a

Table 1: Benchmarks used in the evaluation

Suite	Benchmarks	Datasets
NPB [6]	bt, cg, ep, ft, lu, mg, sp, ua	B, C, D
PARSEC3.0 [111]	canneal, dedup, facesim, fluidanimate, netdedup, netstreamcluster, streamcluster, vips, x264	native, simlarge, simmedium
SPEC OMP 2012 [70]	350.md, 352.nab, 359.botsspar, 360.ilbdc, 367.imagick, 371.applu, 372.smithwa	ref, train, test
Parboil [97]	bfs, histo, tpacf	small, large, default
Rodinia [15]	hotspot, lavaMD, leukocyte, particle_filter, pathfinder	large, medium, small
MLlib [68]	correlation, dtclassifier, fmclassifier, gbtclassifier, gmm, kmeans, logisticregression, lsvc, mlp, pca, randomforestclassifier, summarizer	Three synthetically generated datasets [77]

Table 2: Profiling metrics collected

ID	Metric	ID	Metric
0	branch-instructions	25	LLC-store-misses
1	branch-misses	26	LLC-stores
2	bus-cycles	27	branch-load-misses
3	cache-misses	28	branch-loads
4	cache-references	29	dTLB-load-misses
5	cpu-cycles	30	dTLB-loads
6	instructions	31	dTLB-store-misses
7	ref-cycles	32	dTLB-stores
8	alignment-faults	33	iTLB-load-misses
9	bpf-output	34	node-load-misses
10	context-switches	35	node-loads
11	cpu-clock	36	node-store-misses
12	cpu-migrations	37	node-stores
13	emulation-faults	38	cycles-ct
14	major-faults	39	cycles-t
15	minor-faults	40	el-abort
16	page-faults	41	el-capacity-read
17	task-clock	42	el-capacity-write
18	duration_time	43	el-commit
19	L1-dcache-load-misses	44	el-conflict
20	L1-dcache-loads	45	el-start
21	L1-dcache-stores	46	mem-loads
22	L1-icache-load-misses	47	mem-stores
23	LLC-load-misses	48	slots
24	LLC-loads		

different run with the same dataset has incidentally been seen in the data used for training.

To train and test our prediction models, we use 44 data analytics and scientific computing benchmarks from six benchmark suites and libraries, with multiple datasets each. The benchmarks and datasets used are listed in Table 1. We use all the benchmarks in the single-node evaluation, and the NAS benchmarks in the multi-node evaluation because they have MPI versions. We collect three runs per benchmark per dataset. The running times of the benchmarks range from 1.85 s to 22.6 min, with a mean running time of 94.01 s. We also include a case study on a real application, GROMACS [102], in Section 5.4.

For the single-node evaluation, we evaluate our tool using a dual-processor Intel Xeon Platinum 8358 CPU system with 64 cores (64 threads, hyperthreading disabled) and 512GB of main memory. For the mult-node evaluation, we use 256 MPI ranks distributed across 256 cores of a two-node cluster, each node is a dual socket AMD EPYC 7713 CPU system with 128 cores (128 threads) and 256GB of main memory. Although the scale is not very large, the main purpose of our evaluation is to compare approaches for consolidating predictions across many distributed ranks, as opposed to testing scalability to massive scales.

We use Linux `perf` [24] to profile applications in the single-node setup. We collect 48 profiling metrics listed in Table 2. Despite the large number of metrics, AWTY introduces very little overhead to the application in question at run time because it only extracts a small number of samples and does not run for the entire application duration. We use a sampling period of 200 ms. We use Performance Co-Pilot [22] to profile applications in the multi-node setup.

We use heatmaps to visualize the time series of applications in the motivating example in Section 1 and throughout the evaluation in Section 5. The values in the heatmap are normalized using a min-max scaler [77] before being plotted to enhance visual clarity. The x-axes in the heatmaps represent time, and although the heatmaps have the same width, their x-axes have different scales since the benchmarks have substantially different execution times. To assess which features distinguish the last phase of an application in our discussions, we use visual inspection of the heatmaps aided with the feature importance scores provided by the permutation importance method [5].

5 Evaluation

5.1 Single-application Classifiers

Table 3 shows the F2 score for each benchmark and classifier and the mean across all benchmarks. The benchmarks are sorted in decreasing order of their F2 score for the single-application predictor when the dataset is not previously seen.

It is clear that the single-application classifier exhibits high prediction quality across most benchmarks when it has seen the dataset before. Moreover, even when the dataset has not been seen before, the prediction quality remains high for many benchmarks. This result indicates that our single-application classifiers are effective in many cases.

To further understand what makes the single-application classifiers perform well or not, we inspect the profiles of the individual benchmarks. Figure 4 shows the profiles of the five benchmarks with the highest F2 scores for the single-application predictor (the benchmark with the sixth highest score, bfs, was used as a motivating example in Section 1). It is visually clear that all these applications have distinct last phases. For example, lsvc and mlp witness a notable increase in cpu-migrations (12) and dTLB-load-misses (29). x264 witnesses a notable drop in context-switches (10) and iTLB-load-misses (33), and an increase in cpu-migrations (12). gmm witnesses a gradual drop in cycles and an increase in load misses, particularly bus-cycles (2), cpu-cycles (5), ref-cycles (7), LLC-load-misses (23), and node-load-misses (34). vips witnesses a notable increase in a large number of profiling metrics. The fact that these benchmarks have distinct last phases explains why their classifiers can distinguish their last phases so well.

On the other hand, Figure 5 shows the profiles of the five applications with the lowest F2 scores for the single-application predictor.

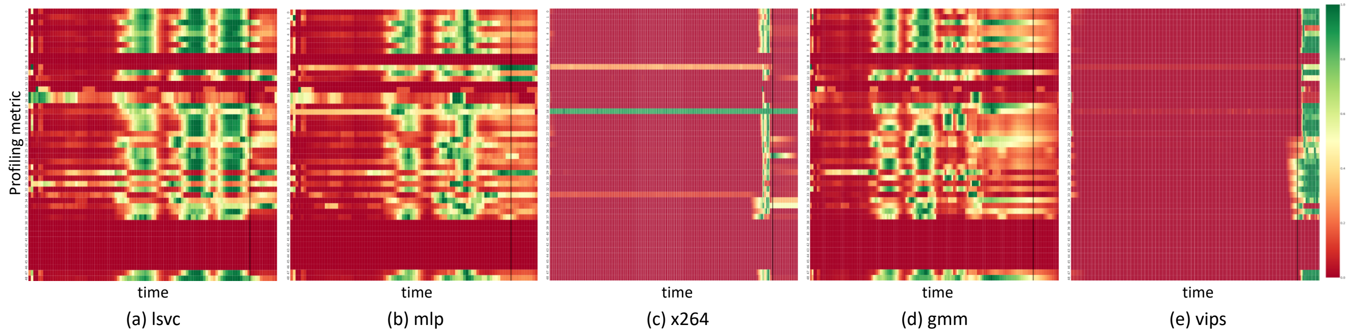


Figure 4: Time series for the five applications with the highest F2 scores for the single-application predictor. The profiling metric values are normalized. The vertical black line indicates the beginning of the last phase detected by AWTY.

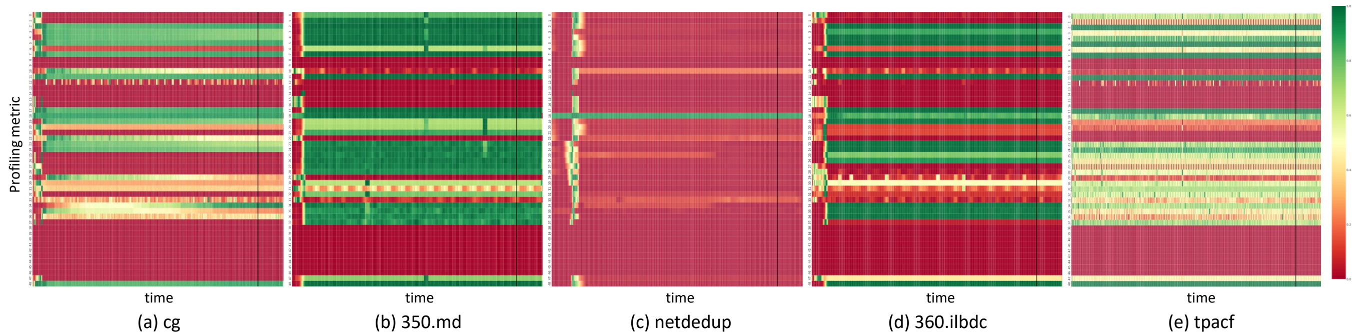


Figure 5: Time series for the five applications with the lowest F2 scores for the single-application predictor. The profiling metric values are normalized. The vertical black line indicates the beginning of the last phase detected by AWTY.

It is clear that all five applications do not have distinct last phases, so it is not surprising that the classifiers could not distinguish the last phases well. However, recall that our workflow screens these cases out and simply reports that it cannot make accurate predictions for that application.

5.2 Predictability and General Classifiers

The results in Table 3 also show that the predictability classifier is quite effective at determining if an application has high predictability. The general classifier also performs well for many benchmarks. This result indicates that our tool is also effective at predicting if many applications that have not been seen before are near completion.

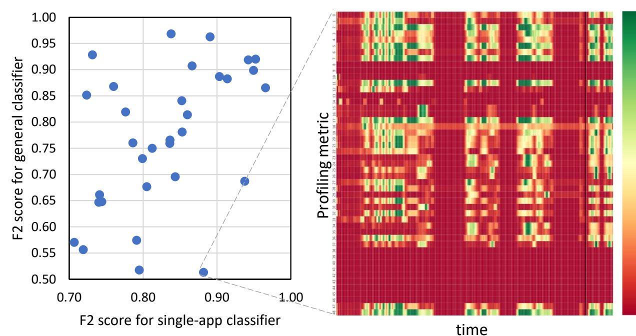
To further understand when our tool is effective at making predictions for applications that have not been seen before, we inspect the profiles of the individual applications with high and low F2 scores for the general classifier. The applications with high F2 scores for the general classifier tend to have other applications with partially similar last phases in the training set. For example, 359.botsspar (F2=0.93), mlp (F2=0.92), 372.smithwa (F2=0.91), and x264 (F2=0.90) all witness an increase in cpu-migrations (12) in their last phase. Hence, each of these benchmarks benefits from having the other benchmarks in the training set, which explains why they all perform well with the general classifier. Note that CPU migrations at the end of an application are common to occur because when some threads begin to finish their execution, the OS might start migrating other threads to rebalance the load across cores.

On the other hand, the applications with low F2 scores for the general classifier tend to either be more difficult to classify with the single-application classifier, or to not have applications with similar last phases in the training set. Figure 6(a) compares the F2 scores of the single-application and general classifiers (each point represents one benchmark). There is a clear association between the two scores. Hence, applications that are more difficult to classify with the single-application classifier are more difficult to classify with the general classifier. However, some applications perform well with the single-application classifier but not the general classifier. For example, the application with the biggest difference in F2 scores between the two classifiers is correlation. Figure 6(b) shows the time series for correlation. The most distinguishing aspect of correlation's last phase is a moderate value for node-loads (35), but few other benchmarks have such a last phase. On the contrary, 359.botsspar, where node-loads are also important, actually witnesses very low values for node-loads in its last phase. Hence, the absence of similar last phases in the training data prevents the general classifier from recognizing correlation's last phase well.

These observations indicate that our rationale that different applications may have similar behavior at the end of their execution is valid. However, there is still room for improving the general classifier by expanding the pool of applications used for training, which is the subject of future work. Fortunately, the general classifier

Table 3: F2 scores for each benchmark and classifier

Benchmark	Single-app Classifier		Predictability Classifier	General Classifier
	Dataset seen	Dataset unseen		
lsvc	0.99	0.97	0.99	0.87
mlp	0.96	0.95	0.99	0.92
x264	1.00	0.95	0.99	0.90
gmm	0.95	0.94	0.99	0.92
vips	0.99	0.94	0.97	0.69
bfs	0.97	0.91	0.29	0.88
kmeans	1.00	0.90	1.00	0.89
pca	0.98	0.89	0.99	0.96
correlation	0.97	0.88	0.90	0.51
372.smithwa	0.93	0.87	0.97	0.91
facesim	0.96	0.86	0.99	0.81
particlefilter	0.95	0.85	0.83	0.78
randomforestclassifier	0.96	0.85	1.00	0.84
logisticregression	0.96	0.84	0.99	0.70
dtclassifier	0.97	0.84	1.00	0.97
summarizer	0.85	0.84	0.94	0.77
pathfinder	0.98	0.84	0.93	0.76
netstreamcluster	0.95	0.81	1.00	0.75
streamcluster	0.96	0.80	1.00	0.68
fluidanimate	1.00	0.80	0.98	0.73
352.nab	0.98	0.79	0.91	0.52
mg	0.99	0.79	0.89	0.57
gbtclassifier	0.95	0.79	1.00	0.76
371.applu	1.00	0.78	0.78	0.82
lavaMD	0.89	0.76	0.44	0.87
fmclassifier	0.93	0.74	0.72	0.65
leukocyte	0.80	0.74	0.99	0.66
ep	0.97	0.74	0.99	0.65
359.botsspar	0.99	0.73	0.69	0.93
canneal	0.96	0.72	0.97	0.85
ua	0.93	0.72	0.16	0.56
367.imagick	0.88	0.71	0.22	0.57
lu	0.94	0.69	0.99	-
bt	0.88	0.66	0.87	-
sp	0.93	0.65	0.06	-
dedup	0.94	0.65	0.98	-
ft	0.84	0.63	0.27	-
histo	0.80	0.63	1.00	-
hotspot	0.95	0.62	0.07	-
cg	0.92	0.60	1.00	-
350.md	0.79	0.59	0.97	-
netdedup	0.99	0.54	0.99	-
360.ilbdc	0.98	0.53	0.51	-
tpacf	0.66	0.53	1.00	-
Mean	**0.94**	**0.77**	**0.82**	**0.77**

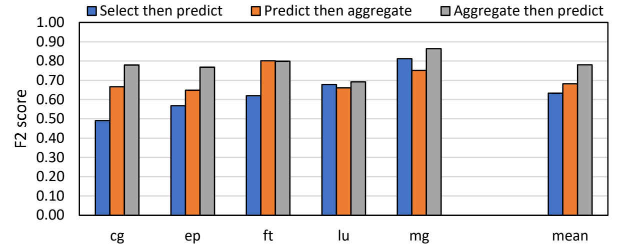


(a) Single-app and general classifier F2 scores

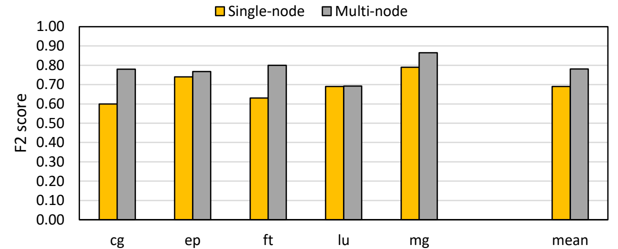
(b) Time series for correlation

Figure 6: Single-application vs. general classifiers

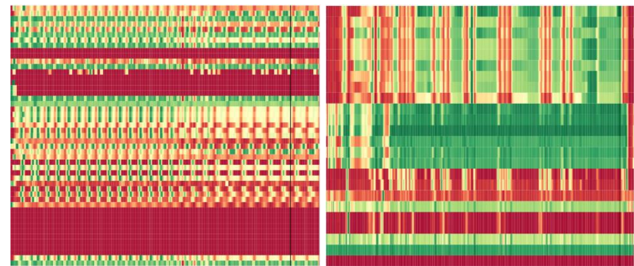
is expected to be less frequently used than the single-application classifier in practice because most systems host users that run the same applications repetitively, albeit on different datasets.



(a) Comparing different approaches for handling multi-node applications



(b) Comparing prediction quality for single- and multi-node versions of the same application



(c) Comparing the time series for single-node (left) and multi-node (right) versions of ft

Figure 7: Evaluating the single-application classifier when the dataset has not been seen for multi-node applications

5.3 Multi-node Applications

Figure 7(a) compares the three different approaches for handling multi-node applications described in Section 3.6 for the single-application classifier when the dataset has not been seen. We observe that *select then predict* has the lowest classification quality, indicating that one rank cannot be used to represent the entire application. We also observe that *aggregate then predict* has better classification quality than *predict then aggregate*, indicating that the aggregation of the time series has useful information that is missed when making predictions for each rank separately.

Figure 7(b) compares the classification quality of the single-application classifier when the dataset has not been seen for the single-node and multi-node versions of the same benchmark. It is clear that multi-node applications are consistently easier to make predictions for. To further understand this observation, Figure 7(c) compares the time series for the single-node and multi-node versions of the *ft* benchmark, which has a large difference in quality between the two versions, for the same dataset. It is clear that the single-node version does not have a distinct last phase, whereas the

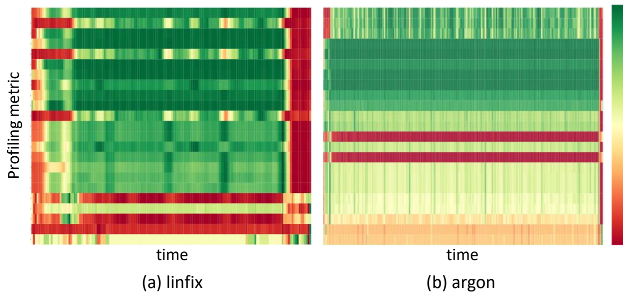


Figure 8: GROMACS time series with two different datasets

Table 4: Comparison of mean F2 scores of each classifier across different models

Classifier	Dummy [77]	XGBoost [17]	RNN
Single-app (dataset seen)	0.33	0.88	0.94
Single-app (dataset unseen)	0.34	0.72	0.77
Predictability	NA	0.71	0.82
General	NA	0.67	0.77

multi-node version does, possibly due to the MPI wrap-up activities done by the latter version.

5.4 Case Study: GROMACS

To demonstrate the effectiveness of our tool on real applications, not just benchmarks, we evaluate it on GROMACS [102], a widely used molecular dynamics package for simulating proteins, lipids, and nucleic acids. We use nine datasets in our evaluation: argon, cbt, linacc, linfix, position-restraints, radacc, radcon, radfix, and rzero. We perform the evaluation on our multi-node setup and use the *aggregate then predict* technique. Our evaluation shows that the single-application classifier for GROMACS when the dataset has not been seen achieves a mean F2 score of 0.82. Figure 8 shows the time series for GROMACS with two different datasets. The *argon* dataset runs longer than the *linfix* dataset, which explains why the last phase detected is relatively shorter. Nevertheless, the last phases are clear in both cases with many profiling metrics significantly dropping in value. The last phases are also similar across the two datasets (and all the other datasets we evaluated), which explains why the mean F2 score is good. These results indicate that our premises and techniques do not only apply to benchmarks but also extend to real applications.

5.5 Comparison to Other Models

We use an RNN as our classification model because RNNs are effective at handling time series data. To demonstrate the effectiveness of our choice of an RNN, we compare its performance to a dummy classifier [77] as a reference, and to XGBoost [17] which is commonly used for classification and known for its ability to deal with many diverse features. Table 4 compares the mean F2 scores for each classifier across different models. It is clear that our RNN model substantially outperforms the dummy classifier which indicates that it is indeed learning something useful. Moreover, it is clear that our RNN model outperforms XGBoost for all classifiers, making it a suitable choice for our classification problem.

Table 5: Profiling metrics collected on System 2

ID	Metric	ID	Metric
0	branch-instructions	27	ic_fetch_stall.ic_stall_back_pressure
1	branch-misses	28	ic_fetch_stall.ic_stall_dq_empty
2	cache-misses	29	l2_cache_req_stat.ls_rd_blk_c
3	cache-references	30	l2_cache_req_stat.ls_rd_blk_cs
4	cpu-cycles	31	l2_latency.l2_cycles_waiting_on_fills
5	instructions	32	l2_request_g1.cacheable_ic_read
6	stalled-cycles-backend	33	l2_request_g1.ls_rd_blk_c_s
7	stalled-cycles-frontend	34	l2_request_g1.rd_blk_l
8	alignment-faults	35	l2_request_g1.rd_blk_x
9	context-switches	36	l2_request_g2.bus_locks_originator
10	cpu-clock	37	ex_ret_instr
11	cpu-migrations	38	ex_ret_mmx_fp_instr.mmx_instr
12	major-faults	39	ex_ret_mmx_fp_instr.sse_instr
13	minor-faults	40	fp_ret_sse_avx_ops.all
14	page-faults	41	ls_l1_d_tlb_miss.tlb_reload_1g_l2_hit
15	task-clock	42	all_tlbs_flushed
16	L1-dcache-load-misses	43	l1_dtlb_misses
17	L1-dcache-loads	44	l2_cache_accesses_from_dc_misses
18	L1-dcache-prefetches	45	l2_cache_accesses_from_ic_misses
19	L1-icache-load-misses	46	l2_cache_hits_from_dc_misses
20	L1-icache-loads	47	l2_cache_hits_from_ic_misses
21	dTLB-load-misses	48	l2_cache_hits_from_l2_hwpf
22	dTLB-loads	49	l2_cache_misses_from_dc_misses
23	iTLB-load-misses	50	l2_cache_misses_from_ic_miss
24	iTLB-loads	51	l2_dtlb_misses
25	bp_l1_tlb_miss_l2_tlb_miss	52	l2_itlb_misses
26	ic_fetch_stall.ic_stall_any	53	sse_avx_stalls

Table 6: Comparison of mean F2 scores of each classifier across systems

Classifier	System 1	System 2
Single-app (dataset seen)	0.94	0.93
Single-app (dataset unseen)	0.77	0.75
Predictability	0.82	0.84
General	0.77	0.74

5.6 Replication Study

To assess the generality of our results, we also evaluate our workflow on a dual-socket AMD EPYC 7532 CPU system with 64 cores (64 threads) and 512GB of main memory. Throughout this section, we refer to the original system we used as *System 1* and the system we used for the replication study as *System 2*. We collect a total of 54 profiling metrics on System 2 which are shown in Table 5. Note that some of these metrics are the same as those from System 1 and some are different because the CPUs on the two systems do not have the same set of hardware counters.

Table 6 compares the mean F2 score for each classifier across both systems. It is clear that our workflow achieves comparable performance on the two systems on average. Figure 9 shows the F2 scores of each benchmark across the two systems for each classifier. It is clear that the F2 scores across the two systems are closely associated. These observations indicate that our results replicate across systems and that the extent to which we can predict if an application is in its last phase is more dependent on the application than the system it executes on.

Although the F2 scores across the two systems are closely associated, there are instances where one benchmark may have a

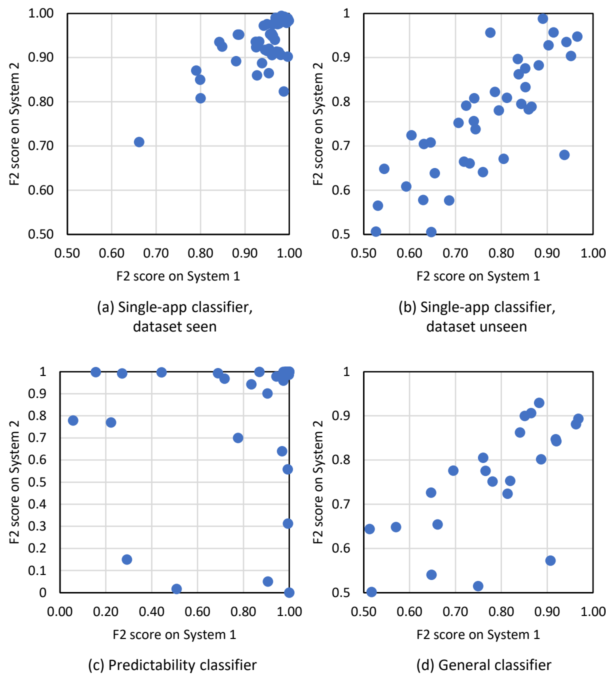


Figure 9: Comparing F2 scores of each classifier across systems (each point represents a single benchmark)

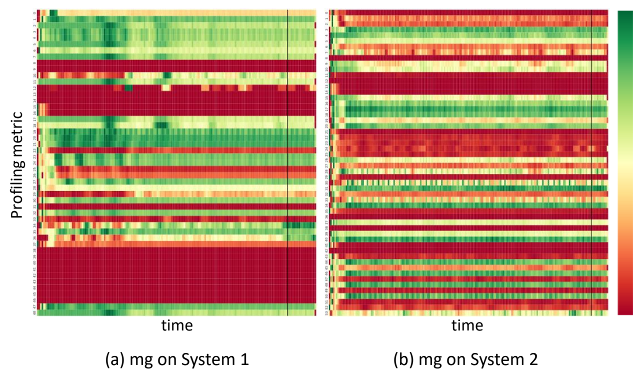


Figure 10: Comparing mg's time series across systems

high F2 score on one system but not the other. For example, the benchmark with the highest difference across systems in the F2 scores of the single-application classifier is mg. Figure 10 shows the time series of mg for the two systems. For System 1, the last phase is distinguished by the increase in node-load-misses (34) and node-store-misses (36), while all other metrics remain the same. On the other hand, System 2 does not have these two metrics, and all the other metrics collected from System 2 do not change at the end of the execution. This observation explains why the classifier for System 2 was unable to distinguish the last phase as well as that for System 1 for this particular benchmark.

6 Conclusion

We present a workflow for predicting if an executing batch-style application is near completion. Our workflow analyses a large number of application profiles to find their last phases, then uses this information to train classifiers that predict whether or not an executing application is in its last phase. The workflow includes single-application classifiers for applications that have been seen before and a general classifier for applications that have not been seen before. At run time, an executing application of interest is profiled briefly, and its profiling samples are passed to the classifiers to classify if the application is in its last phase or not. Our evaluation shows that our workflow can predict if an executing application is in its last phase reasonably well, with results replicating across different systems. Our future work includes expanding the set of applications used for training our predictors, and combining our tool with a scheduler or resource manager to inform the latter when deciding whether or not to kill a job.

Acknowledgments

This work was supported by Hewlett Packard Enterprise.

References

[1] Omar Aaziz, Jonathan Cook, and Mohammed Tanash. 2018. Modeling expected application runtime for characterizing and assessing job performance. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 543–551.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[3] Dong H Ahn, Jim Garlick, Mark Grondona, Don Lipari, Becky Springmeyer, and Martin Schulz. 2014. Flux: A next-generation resource management framework for large HPC centers. In *2014 43rd International Conference on Parallel Processing Workshops*. IEEE, 9–17.

[4] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. 2017. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 469–482.

[5] André Altmann, Laura Toloşi, Oliver Sander, and Thomas Lengauer. 2010. Permutation importance: a corrected feature importance measure. *Bioinformatics* 26, 10 (2010), 1340–1347.

[6] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. 1991. The NAS parallel benchmarks summary and preliminary results. In *Supercomputing'91: Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*. IEEE, 158–165.

[7] Cynthia Bailey Lee, Yael Schwartzman, Jennifer Hardy, and Allan Snavely. 2005. Are user runtime estimates inherently inaccurate?. In *Job Scheduling Strategies for Parallel Processing: 10th International Workshop, JSSPP 2004, New York, NY, USA, June 13, 2004. Revised Selected Papers 10*. Springer, 253–263.

[8] Matt Baughman, Ryan Chard, Logan Ward, Jason Pitt, Kyle Chard, and Ian Foster. 2018. Profiling and predicting application performance on the cloud. In *11th IEEE/ACM International Conference on Utility and Cloud Computing (UCC)*.

[9] Mohammed Baydoun, Mohammad Sonji, Pedro Bruel, Dejan Milojicic, Eitan Frachtenberg, and Izzat El Hajj. 2025. Predicting Performance Variability. In *2025 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 225–234. doi:10.1109/IPDPSW66978.2025.00044

[10] Muhammad Bilal, Marco Canini, and Rodrigo Rodrigues. 2020. Finding the right cloud configuration for analytics clusters. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 208–222.

[11] Muhammad Bilal, Marco Serafini, Marco Canini, and Rodrigo Rodrigues. 2020. Do the best cloud configurations grow on trees? An experimental evaluation of black box algorithms for optimizing cloud workloads. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2563–2575.

[12] Aleksandar Branković, Kyriakos Stavrou, Enric Gibert, and Antonio González. 2014. Accurate off-line phase classification for HW/SW co-designed processors. In *Proceedings of the 11th ACM Conference on Computing Frontiers*. 1–10.

[13] Marc Casas, Rosa M Badia, and Jesús Labarta. 2010. Automatic phase detection and structure extraction of mpi applications. *The International Journal of High Performance Computing Applications* 24, 3 (2010), 335–360.

[14] Maria Casimiro, Diego Didona, Paolo Romano, Luis Rodrigues, Willy Zwaenepoel, and David Garlan. 2020. Lynceus: Cost-efficient tuning and provisioning of data analytic jobs. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 56–66.

[15] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 44–54.

[16] Liming Chen, Xuecheng Zou, Jianming Lei, and Zhenglin Liu. 2007. Dynamically reconfigurable cache for low-power embedded system. In *Third International Conference on Natural Computation (ICNC 2007)*, Vol. 5. Ieee, 180–184.

[17] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. 785–794.

[18] Xin Chen, Charng-Da Lu, and Karthik Pattabiraman. 2013. Predicting job completion times using system logs in supercomputing clusters. In *2013 43rd Annual IEEE/IFIP Conference on Dependable Systems and Networks Workshop (DSN-W)*. IEEE, 1–8.

[19] Yanjiao Chen, Long Lin, Baochun Li, Qian Wang, and Qian Zhang. 2021. Silhouette: Efficient cloud configuration exploration for large-scale analytics. *IEEE Transactions on Parallel and Distributed Systems* 32, 8 (2021), 2049–2061.

[20] Ghislain Landry Tsafack Chetsa, Laurent Lefevre, Jean-Marc Pierson, Patricia Stolf, and Georges Da Costa. 2013. A user friendly phase detection methodology for hpc systems' analysis. In *2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing*. IEEE, 118–125.

[21] Meng-Chieh Chiu, Benjamin Marlin, and Eliot Moss. 2016. Real-time program-specific phase change detection for java programs. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. 1–11.

[22] Performance Co-Pilot. [n. d.]. *https://pcp.io*. https://pcp.io

[23] Keeley Criswell and Tosiron Adegbija. 2019. A survey of phase classification techniques for characterizing variable application behavior. *IEEE Transactions on Parallel and Distributed Systems* 31, 1 (2019), 224–236.

[24] Arnaldo Carvalho De Melo. 2010. The new linux'perf'tools. In *Slides from Linux Kongress*, Vol. 18. 1–42.

[25] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, USA) *(ASPLOS '13)*. Association for Computing Machinery, New York, NY, USA, 77–88.

[26] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (Salt Lake City, Utah, USA) *(ASPLOS '14)*. Association for Computing Machinery, New York, NY, USA, 127–144.

[27] Ashutosh S. Dhodapkar and James E. Smith. 2002. Managing Multi-Configuration Hardware via Dynamic Working Set Analysis. In *Proceedings of the 29th Annual International Symposium on Computer Architecture* (Anchorage, Alaska) *(ISCA '02)*. IEEE Computer Society, USA, 233–244.

[28] Giacomo Domeniconi, Eun Kyung Lee, Vanamala Venkataswamy, and Swaroopa Dola. 2019. Cush: Cognitive scheduler for heterogeneous high performance computing system. *Proceedings of DRL4KDD* 19 (2019).

[29] Rubing Duan, Farrukh Nadeem, Jie Wang, Yun Zhang, Radu Prodan, and Thomas Fahringer. 2009. A hybrid intelligent method for performance modeling and prediction of workflow activities in grids. In *2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*. IEEE, 339–347.

[30] Dmitry Duplyakin, Alexandru Uta, Aleksander Maricq, and Robert Ricci. 2019. On Studying CPU Performance of CloudLab Hardware. In *2019 IEEE 27th International Conference on Network Protocols (ICNP)*. 1–2. doi:10.1109/ICNP.2019.8888128

[31] Lieven Eeckhout, John Sampson, and Brad Calder. 2005. Exploiting program microarchitecture independent characteristics and phase behavior for reduced benchmark suite simulation. In *IEEE International. 2005 Proceedings of the IEEE Workload Characterization Symposium, 2005*. IEEE, 2–12.

[32] Yuping Fan, Paul Rich, William E Allcock, Michael E Papka, and Zhiling Lan. 2017. Trade-off between prediction accuracy and underestimation rate in job runtime estimates. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 530–540.

[33] Dror G Feitelson and Ahuva Mu'alem Weil. 1998. Utilization and predictability in scheduling the IBM SP2 with backfilling. In *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*. IEEE, 542–546.

[34] Karl Fürlinger and Shirley Moore. 2008. Detection and analysis of iterative behavior in parallel applications. In *Computational Science–ICCS 2008: 8th International Conference, Kraków, Poland, June 23-25, 2008, Proceedings, Part III 8*. Springer, 261–267.

[35] Kunal Ganeshpure and Sandip Kundu. 2013. On runtime task graph extraction in MPSoC. In *2013 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 171–176.

[36] Eric Gaussier, David Glesser, Valentin Reis, and Denis Trystram. 2015. Improving backfilling by using machine learning to predict running times. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–10.

[37] Ann Gordon-Ross, Jeremy Lau, and Brad Calder. 2008. Phase-based cache reconfiguration for a highly-configurable two-level cache hierarchy. In *Proceedings of the 18th ACM Great Lakes symposium on VLSI*. 379–382.

[38] Dayong Gu and Clark Verbrugge. 2007. Using hardware data to detect repetitive program behavior. *Sable Research Group, School Comput. Sci., McGill Univ., Montréal, Québec, Tech. Rep. SABLE-TR-2007–2* (2007).

[39] Dayong Gu and Clark Verbrugge. 2008. Phase-based adaptive recompilation in a JVM. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*. 24–34.

[40] Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. 2005. Simpoint 3.0: Faster and more flexible program phase analysis. *Journal of Instruction Level Parallelism* 7, 4 (2005), 1–28.

[41] Charles R. Harris, K. Jarrod Millman, St'efan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fern'andez del R'ıo, Mark Wiebe, Pearu Peterson, Pierre G'erard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. doi:10.1038/s41586-020-2649-2

[42] Haibo He, Yang Bai, Edwardo A Garcia, and Shutao Li. 2008. ADASYN: Adaptive synthetic sampling approach for imbalanced learning. In *2008 IEEE international joint conference on neural networks (IEEE world congress on computational intelligence)*. Ieee, 1322–1328.

[43] Kenneth Hoste and Lieven Eeckhout. 2007. Microarchitecture-independent workload characterization. *IEEE micro* 27, 3 (2007), 63–72.

[44] Chin-Jung Hsu, Vivek Nair, Vincent W Freeh, and Tim Menzies. 2018. Arrow: Low-level augmented bayesian optimization for finding the best cloud VM. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 660–670.

[45] Chin-Jung Hsu, Vivek Nair, Tim Menzies, and Vincent Freeh. 2018. Micky: A cheaper alternative for selecting cloud instances. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 409–416.

[46] Chin-Jung Hsu, Vivek Nair, Tim Menzies, and Vincent W Freeh. 2018. Scout: An experienced guide to find the best cloud configuration. *arXiv preprint arXiv:1803.01296* (2018).

[47] MC Huang, J Renau, and J Torrellas. 2003. Positional adaptation of processors: application to energy reduction. In *30th Annual International Symposium on Computer Architecture, 2003. Proceedings*. IEEE, 157–168.

[48] Ted Huffmire and Tim Sherwood. 2006. Wavelet-based phase classification. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*. 95–104.

[49] Omer Khan and Sandip Kundu. 2011. Microvisor: A runtime architecture for thermal management in chip multiprocessors. In *Transactions on high-performance embedded architectures and Compilers IV*. Springer, 84–110.

[50] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. 2018. Selecta: Heterogeneous cloud storage configuration for data analytics. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 759–773.

[51] Dalibor Klusáček and Václav Chlumský. 2019. Evaluating the impact of soft walltimes on job scheduling performance. In *Job Scheduling Strategies for Parallel Processing: 22nd International Workshop, JSSPP 2018, Vancouver, BC, Canada, May 25, 2018, Revised Selected Papers 22*. Springer, 15–38.

[52] Kenneth Lamar, Alexander Goponenko, Omar Aaziz, Benjamin A Allan, James M Brandt, and Damian Dechev. 2023. Evaluating HPC job run time predictions using application input parameters. In *Proceedings of the 17th ACM International Conference on Distributed and Event-based Systems*. 127–138.

[53] Jeremy Lau, Jack Sampson, Erez Perelman, Greg Hamerly, and Brad Calder. 2005. The strong correlation between code signatures and performance. In *IEEE International Symposium on Performance Analysis of Systems and Software, 2005. ISPASS 2005*. IEEE, 236–247.

[54] Jeremy Lau, Stefan Schoemackers, and Brad Calder. 2004. Structures for phase classification. In *IEEE International Symposium on-ISPASS Performance Analysis of Systems and Software, 2004*. IEEE, 57–67.

[55] Guillaume Lemaître, Fernando Nogueira, and Christos K. Aridas. 2017. Imbalanced-learn: A Python Toolbox to Tackle the Curse of Imbalanced Datasets in Machine Learning. *Journal of Machine Learning Research* 18, 17 (2017), 1–5.

http://jmlr.org/papers/v18/16-365.html
[56] Jingbo Li, Xingjun Zhang, Li Han, Zeyu Ji, Xiaoshe Dong, and Chenglong Hu. 2021. OKCM: improving parallel task scheduling in high-performance computing systems using online learning. *The Journal of Supercomputing* 77 (2021), 5960–5983.
[57] David A Lifka. 1995. The ANL/IBM SP scheduling system. In *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 295–303.
[58] Yuhui Lin, Adam Barker, and John Thomson. 2020. Modelling VM Latent Characteristics and Predicting Application Performance using Semi-supervised Non-negative Matrix Factorization. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*. IEEE, 470–474.
[59] Yuhui Lin, Jack Briggs, and Adam Barker. 2020. FIFE: an Infrastructure-as-Code Based Framework for Evaluating VM Instances from Multiple Clouds. In *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*. IEEE, 91–100.
[60] Yang Liu, Huanle Xu, and Wing Cheong Lau. 2019. Accordia: Adaptive cloud configuration optimization for recurring data-intensive applications. In *Proceedings of the ACM Symposium on Cloud Computing*. 479–479.
[61] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. 2016. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM workshop on hot topics in networks*. 50–56.
[62] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM special interest group on data communication*. 270–288.
[63] Giovanni Mariani, Andreea Anghel, Rik Jongerius, and Gero Dittmann. 2017. Predicting cloud performance for HPC applications: A user-oriented approach. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 524–533.
[64] Aleksander Maricq, Dmitry Duplyakin, Ivo Jimenez, Carlos Maltzahn, Ryan Stutsman, and Robert Ricci. 2018. Taming performance variability. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 409–425.
[65] Andréa Matsunaga and José AB Fortes. 2010. On the use of machine learning to predict the time and resources consumed by applications. In *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*. IEEE, 495–504.
[66] Ryan McKenna, Stephen Herbein, Adam Moody, Todd Gamblin, and Michela Taufer. 2016. Machine learning predictions of runtime and IO traffic on high-end clusters. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 255–258.
[67] Celso L Mendes and Daniel A Reed. 1998. Integrated compilation and scalability analysis for parallel systems. In *Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. 98EX192)*. IEEE, 385–392.
[68] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. 2016. MLlib: Machine learning in Apache Spark. *The Journal of Machine Learning Research* 17, 1 (2016), 1235–1241.
[69] Ahuva W. Mu'alem and Dror G. Feitelson. 2001. Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. *IEEE transactions on parallel and distributed systems* 12, 6 (2001), 529–543.
[70] Matthias S Müller, John Baron, William C Brantley, Huiyu Feng, Daniel Hackenberg, Robert Henschel, Gabriele Jost, Daniel Molka, Chris Parrott, Joe Robichaux, et al. 2012. SPEC OMP2012 — an application benchmark suite for parallel systems using OpenMP. In *International Workshop on OpenMP*. Springer, 223–236.
[71] Mina Naghshnejad and Mukesh Singhal. 2018. Adaptive online runtime prediction to improve HPC applications latency in cloud. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 762–769.
[72] Amir Nassereldine, Safaa Diab, Mohammed Baydoun, Kenneth Leach, Maxim Alt, Dejan Milojicic, and Izzat El Hajj. 2023. Predicting the Performance-Cost Trade-off of Applications Across Multiple Systems. In *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 216–228.
[73] Bill Nitzberg, Jennifer M Schopf, and James Patton Jones. 2004. PBS Pro: Grid computing and scheduling attributes. In *Grid resource management: state of the art and future trends*. Springer, 183–190.
[74] The pandas development team. 2020. pandas-dev/pandas: Pandas. doi:10.5281/zenodo.3509134
[75] Ju-Won Park and Eunhye Kim. 2017. Runtime prediction of parallel applications with workload-aware clustering. *The Journal of Supercomputing* 73, 11 (2017), 4635–4651.
[76] Harish Patil, Robert Cohn, Mark Charney, Rajiv Kapoor, Andrew Sun, and Anand Karunanidhi. 2004. Pinpointing representative portions of large intel® itanium® programs with dynamic instrumentation. In *37th International Symposium on Microarchitecture (MICRO-37'04)*. IEEE, 81–92.
[77] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, and Others. 2011. Scikit-learn: Machine learning in Python. *the Journal of machine Learning research* 12 (2011), 2825–2830.
[78] Nitzan Peleg and Bilha Mendelson. 2007. Detecting change in program behavior for adaptive optimization. In *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*. IEEE, 150–162.
[79] Erez Perelman, Marzia Polito, J-Y Bouguet, Jack Sampson, Brad Calder, and Carole Dulong. 2006. Detecting phases in parallel applications on shared memory architectures. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*. IEEE, 10–pp.
[80] Paruj Ratanaworabhan and Martin Burtscher. 2008. Program phase detection based on critical basic block transitions. In *ISPASS 2008-IEEE International Symposium on Performance Analysis of Systems and software*. IEEE, 11–21.
[81] Rance Rodrigues, Arunachalam Annamalai, Israel Koren, and Sandip Kundu. 2013. Improving performance per watt of asymmetric multi-core processors via online program phase classification and adaptive core morphing. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 18, 1 (2013), 1–23.
[82] Andreas Sembrant, David Black-Schaffer, and Erik Hagersten. 2012. Phase behavior in serial and parallel applications. In *2012 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 47–58.
[83] Andreas Sembrant, David Eklov, and Erik Hagersten. 2011. Efficient software-based online phase classification. In *2011 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 104–115.
[84] Xipeng Shen, Michael L Scott, Chengliang Zhang, Sandhya Dwarkadas, Chen Ding, and Mitsunori Ogihara. 2007. Analysis of input-dependent program behavior using active profiling. In *Proceedings of the 2007 workshop on experimental computer science*. 5–es.
[85] Xipeng Shen, Yutao Zhong, and Chen Ding. 2007. Predicting locality phases for dynamic memory optimization. *J. Parallel and Distrib. Comput.* 67, 7 (2007), 783–796.
[86] Timothy Sherwood, Erez Perelman, and Brad Calder. 2001. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings 2001 International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 3–14.
[87] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California) *(ASPLOS X)*. Association for Computing Machinery, New York, NY, USA, 45–57. doi:10.1145/605397.605403
[88] Timothy Sherwood, Suleyman Sair, and Brad Calder. 2003. Phase tracking and prediction. *ACM SIGARCH Computer Architecture News* 31, 2 (2003), 336–349.
[89] Joseph Skovira, Waiman Chan, Honbo Zhou, and David Lifka. 1996. The easy—loadleveler api project. In *Job Scheduling Strategies for Parallel Processing: IPPS'96 Workshop Honolulu, Hawaii, April 16, 1996 Proceedings 2*. Springer, 41–47.
[90] Warren Smith. 2007. Prediction services for distributed computing. In *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 1–10.
[91] Warren Smith, Ian Foster, and Valerie Taylor. 1998. Predicting application run times using historical information. In *Job Scheduling Strategies for Parallel Processing: IPPS/SPDP'98 Workshop Orlando, Florida, USA, March 30, 1998 Proceedings 4*. Springer, 122–142.
[92] Warren Smith, Ian Foster, and Valerie Taylor. 2004. Predicting application run times with historical information. *J. Parallel and Distrib. Comput.* 64, 9 (2004), 1007–1016.
[93] Sudarshan Srinivasan, Kunal P Ganeshpure, and Sandip Kundu. 2011. Maximizing hotspot temperature: Wavelet based modelling of heating and cooling profile of functional workloads. In *2011 12th International Symposium on Quality Electronic Design*. IEEE, 1–7.
[94] Sudarshan Srinivasan, Israel Koren, and Sandip Kundu. 2016. Improving performance per watt of non-monotonic multicore processors via bottleneck-based online program phase classification. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*. IEEE, 528–535.
[95] Garrick Staples. 2006. Torque resource manager. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. 8–es.
[96] Mathieu Stoffel, François Broquedis, Frédéric Desprez, and Abdelhafid Mazouz. 2021. Phase-TA: Periodicity Detection and Characterization for HPC Applications. In *HPCS 2020-18th IEEE International Conference on High Performance Computing and Simulation*. IEEE, 1–12.
[97] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing* 127 (2012), 27.
[98] Karl Taht, James Greensky, and Rajeev Balasubramonian. 2019. The pop detector: A lightweight online program phase detection framework. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 48–57.
[99] Mohammed Tanash, Brandon Dunn, Daniel Andresen, William Hsu, Huichen Yang, and Adedolapo Okanlawon. 2019. Improving HPC system performance by predicting job resources via supervised machine learning. In *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the*

Machines (learning). 1–8.

[100] Dan Tsafrir, Yoav Etsion, and Dror G Feitelson. 2005. Modeling user runtime estimates. In *Job Scheduling Strategies for Parallel Processing: 11th International Workshop, JSSPP 2005, Cambridge, MA, USA, June 19, 2005, Revised Selected Papers 11*. Springer, 1–35.

[101] Dan Tsafrir, Yoav Etsion, and Dror G Feitelson. 2007. Backfilling using system-generated predictions rather than user runtime estimates. *IEEE Transactions on Parallel and Distributed Systems* 18, 6 (2007), 789–803.

[102] David Van Der Spoel, Erik Lindahl, Berk Hess, Gerrit Groenhof, Alan E Mark, and Herman JC Berendsen. 2005. GROMACS: fast, flexible, and free. *Journal of computational chemistry* 26, 16 (2005), 1701–1718.

[103] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. 2016. Ernest: Efficient Performance Prediction for {Large-Scale} Advanced Analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. 363–378.

[104] Qiqi Wang, Hongjie Zhang, Jing Li, Yu Shen, and Xiaohui Liu. 2022. Predicting job finish time based on parameter features and running logs in supercomputing system. *The Journal of Supercomputing* 78, 17 (2022), 18551–18577.

[105] Qiqi Wang, Hongjie Zhang, Cheng Qu, Yu Shen, Xiaohui Liu, and Jing Li. 2021. RLSchert: an hpc job scheduler using deep reinforcement learning and remaining time prediction. *Applied Sciences* 11, 20 (2021), 9448.

[106] Yue-Wen Wu, Yuan-Jia Xu, Heng Wu, Lin-Gang Su, Wen-Bo Zhang, and Hua Zhong. 2021. Apollo: Rapidly Picking the Optimal Cloud Configurations for Big Data Analytics Using a Data-Driven Approach. *Journal of Computer Science and Technology* 36, 5 (2021), 1184–1199.

[107] Ai Xiao, ZhiHui Lu, Junnan Li, Jie Wu, and Patrick CK Hung. 2019. SARA: Stably and quickly find optimal cloud configurations for heterogeneous big data workloads. *Applied Soft Computing* 85 (2019), 105759.

[108] Yajing Xu, Junnan Li, Zhihui Lu, Jie Wu, Patrick CK Hung, and Abdulhameed Alelaiwi. 2020. ARVMEC: Adaptive Recommendation of Virtual Machines for IoT in Edge–Cloud Environment. *J. Parallel and Distrib. Comput.* 141 (2020), 23–34.

[109] Neeraja J Yadwadkar, Bharath Hariharan, Joseph E Gonzalez, Burton Smith, and Randy H Katz. 2017. Selecting the best VM across multiple public clouds: A data-driven performance modeling approach. In *Proceedings of the 2017 Symposium on Cloud Computing*. 452–465.

[110] Andy B Yoo, Morris A Jette, and Mark Grondona. 2003. Slurm: Simple linux utility for resource management. In *Workshop on job scheduling strategies for parallel processing*. Springer, 44–60.

[111] Xusheng Zhan, Yungang Bao, Christian Bienia, and Kai Li. 2017. PARSEC3.0: A multicore benchmark suite with network stacks and SPLASH-2X. *ACM SIGARCH Computer Architecture News* 44, 5 (2017), 1–16.

[112] Chuanjun Zhang, Frank Vahid, and Walid Najjar. 2003. A highly configurable cache architecture for embedded systems. In *Proceedings of the 30th annual international symposium on Computer architecture*. 136–146.

[113] Di Zhang, Dong Dai, Youbiao He, Forrest Sheng Bao, and Bing Xie. 2020. RLScheduler: an automated HPC batch job scheduler using reinforcement learning. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[114] Weihua Zhang, Jiaxin Li, Yi Li, and Haibo Chen. 2015. Multilevel phase analysis. *ACM Transactions on Embedded Computing Systems (TECS)* 14, 2 (2015), 1–29.

[115] Longfang Zhou, Xiaorong Zhang, Wenxiang Yang, Yongguo Han, Fang Wang, Yadong Wu, and Jie Yu. 2021. Prep: Predicting job runtime with job running path on supercomputers. In *Proceedings of the 50th International Conference on Parallel Processing*. 1–10.