The handle http://hdl.handle.net/1887/18620 holds various files of this Leiden University dissertation.

**Author:** Tran, Minh Ngoc  
**Title:** Workload modeling and performance evaluation in parallel systems  
**Issue Date:** 2012-03-22
Chapter 8

A History-Based Predictor of Parallel Application Runtimes

Backfilling algorithms [73] have become more popular for scheduling in many space-shared computing resources. For instance, the clusters in the High Performance Computing Center North in Sweden [34] and the National Center for Supercomputing Applications in USA [78] all use Maui/Moab [61] scheduler which enables backfilling. Also the SDSC Blue Horizon supercomputer [88] implements backfilling policies, etc. Recall that backfilling means that short jobs which are expected to terminate in time are allowed to leapfrog and execute before previously queued large jobs provided that they do not delay these large jobs [73]. With backfilling, an estimate of the runtime is necessary for each job to determine whether it is short enough to be backfilled. The estimate can be either provided by the user or predicted automatically by the system.

Most parallel systems that implement backfilling policies ask users for the estimate of the runtime with an assumption that users would provide accurate estimates since their jobs could start faster if the estimates are tight but would be killed in case the estimates are too small. However, the user requested runtime\textsuperscript{1} is found to be extremely inaccurate because users are motivated to overestimate their jobs so that jobs will not be killed much stronger than to provide accurate estimate to enable jobs to start faster, according to the study in [73]. Although studies in [73, 20, 93, 113, 114] show that accurate requested runtimes only have a small impact on the performance of FCFS-backfilling systems, Chiang et al. have indicated that this result is correct on FCFS-backfilling systems, but not on systems using high performance backfilling policies such as L√XF&W-backfill, S√TF&W-backfill and LXF&W-backfill [9, 10]. It means a more accurate estimated runtime can improve system performance significantly [9]. Therefore, user estimated runtime should not be used as a runtime predictor in backfilling systems if possible.

With respect to the system predictor, several studies have shown that using histori-

\textsuperscript{1}The phrases “user requested runtime” and “user estimated runtime” are used interchangeably with the same meaning throughout this chapter.
cal data for runtime prediction can considerably improve the accuracy of the estimates [14, 104, 89, 29, 92]. Nevertheless, these studies were done in a context that does not take care of the problem of underestimation. In theory with backfilling systems, jobs that are underestimated will be killed when they exceed their declared runtime [73]. However, user jobs cannot be killed just because they are underestimated by a system predictor. In this case, the system can only let underestimated jobs continue to run and re-schedule jobs in the waiting queue. Consequently, more jobs may be backfilled and several jobs in the waiting queue will be delayed and finally it leads to an adverse impact on system performance.

Both inaccuracy and underestimation in prediction are not good for backfilling systems. Using user estimated runtime can avoid the problem of underestimation but it is inaccurate. System predictor using recent studies [104, 50, 55, 94] can obtain more accuracy but the problem of underestimation is not solved. Therefore in this chapter, we present a novel predictor which tries to reduce the number of jobs that are underestimated and reduce the prediction error as much as possible.

8.1 Related Work

It is well known that workloads on parallel systems are highly repetitive, because users tend to run the same applications over and over again [15, 21]. This means that the estimate of job runtime can be done based on the information of jobs that finished in the past. In fact, several studies have been proposed to estimate job runtime using historical information. Basically, we can divide these studies into two kinds based on the way they look for similar jobs: categorization and instance based learning.

Predicting studies with categorization consist of [13, 104, 29, 92]. Each of them applies a template of job characteristics to determine similar jobs that already finished in the past. In [13], the system queue is used to categorize jobs into classes and a model is created for each class to predict execution times. Gibbons [29] defines a historical application profiler to classify parallel applications in categories based on static templates including attributes such as user, job name and system queue. Parallel applications are grouped according to these templates and the average execution time of each group is used to predict execution times of future applications. Smith et al. [92] find that using static templates to classify jobs is simple and a more sophisticated technique can be applied. They use a genetic algorithm [65] to dynamically determine which job characteristics produce the best definition of similarity. The most recent study [104] finds that using only the previous two jobs submitted by the same user will also give a good prediction. This predictor is really attractive due to its simplicity.

Studies in [50, 89, 94] use the same instance based learning technique to predict parallel application runtimes. With this technique, information about N most recent finished jobs (called the experience base) is kept. The runtime of a future job is
8.2 Workloads Under Study

estimated by searching $K$ nearest neighbor jobs (jobs that are most similar to the one being predicted) in the experience base. The similarity between job $X$ and job $Y$ is measured by a distance function

$$D(x, y) = \sqrt{\frac{\sum_{a=1}^{m} w_a \times d_a(x_a, y_a)^2}{\sum_{a=1}^{m} w_a}}, \quad (8.1)$$

where $x$ and $y$ are attribute vectors of $X$ and $Y$, respectively. In addition, $w_a$ is the weight and $d_a$ is the corresponding distance function for attribute $a$ with

$$d_a(x_a, y_a) = \begin{cases} 
0 & \text{if } a \text{ is nominal, } x_a = y_a \\
1 & \text{if } a \text{ is nominal, } x_a \neq y_a \\
\frac{|x_a - y_a|}{max_a - min_a} & \text{if } a \text{ is numeric scalar.}
\end{cases} \quad (8.2)$$

Once a set of $K$ nearest neighbors are identified, an induction model is applied to generate predictions. The induction models used in [50, 89, 94] include Weighted Average (WA) and Linear Locally Weighted Regression (LLWR), where WA is shown to be better than LLWR. Model WA estimates the execution time by the following function

$$Estimate(job) = \frac{\sum_{i=1}^{K} e^{-\frac{(D(job, job_i))^2}{\sigma^2}} \times R(job_i)}{\sum_{i=1}^{K} e^{-\frac{(D(job, job_i))^2}{\sigma^2}}}, \quad (8.3)$$

where $R(job_i)$ is the actually runtime of job $i$ in the set of $K$ nearest neighbors and $\sigma$ is the kernel bandwidth, which can be a fixed constant value or can be set to the largest distance in the nearest neighbor set.

The differences between the studies in [50, 89, 94] are the applied parameters such as the experience base size $N$, the neighbor size $K$, the weights $w_a$ in Eq. (8.1) and the kernel bandwidth $\sigma$ in Eq. (8.3). These parameters are assigned with fixed values by Senger et al. in [89]. More flexibly, Smith et al. [94] and Li et al. [50] use a genetic algorithm [30] to determine the best values for these parameters. An implementation of this technique is available under the name “Performance Data Miner” [81].

8.2 Workloads Under Study

Table 8.1 describes details of the traces used in this study. Note that, these traces are different from those described in Section 3.1. SDSC02 and SDSC04 are two separate traces collected from the San Diego Supercomputer Center Intel Paragon machine,
whose scheduler is Catalina [7]. SDSC04 includes 13-month data and we use the whole trace in our experiments. HPC2N is from a 120-node Linux cluster named Seth at the High Performance Computing Center North in Sweden [34]. Seth is scheduled by Maui [61]. This trace contains three and a half years worth of accounting records, starting from Jul 2002 to Jan 2006. We use the last two years of the trace as two separate traces in our study under the name HPC2N04 and HPC2N05. Note that, in our study, we only take into account jobs that finished successfully, i.e. jobs that have status equal to 1 in the trace. Jobs that have not finished yet should not be used in prediction because their runtimes are not actual. All traces and detailed information are available on [80]. Although a few newer traces can be found on [80] (with the newest one collected till Jun 2007), we do not select them in our experiments because these traces lack the information of user requested runtimes which is important for our predictor. Traces in Table 8.1 are - to our knowledge - the most recent ones that have the information of user estimated runtime.

<table>
<thead>
<tr>
<th>Trace</th>
<th>Period</th>
<th>Processors</th>
<th>Number of jobs</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDSC02</td>
<td>01/2002-12/2002</td>
<td>1152</td>
<td>80756</td>
</tr>
<tr>
<td>SDSC04</td>
<td>03/2004-03/2005</td>
<td>1664</td>
<td>66743</td>
</tr>
<tr>
<td>HPC2N04</td>
<td>01/2004-12/2004</td>
<td>240</td>
<td>81113</td>
</tr>
<tr>
<td>HPC2N05</td>
<td>01/2005-01/2006</td>
<td>240</td>
<td>55389</td>
</tr>
</tbody>
</table>

**8.3 Predict Application Runtime**

Our method to predict application runtimes is presented in this section. The assumption that similar jobs will have similar runtimes is applied in our predictor as in studies mentioned in Section 8.1. However, the method we use to define the job similarity and calculate the estimates is quite different. We define the job similarity by a combination of the categorization and the instance based learning methods, with a new similarity function (see Eq. (8.5) in Section 8.3.1). Moreover, when predicting the runtimes, we do take care of the problem of underestimation. We also notice different impacts of small jobs versus big jobs and separate their roles.

**8.3.1 Define Job Similarity**

Due to the limitation of information contained in traces used in experiments, we can only select and take into account seven job attributes to define job similarity. They are divided into two groups: nominal and numeric attributes. The nominal
8.3. Predict Application Runtime

group includes “user name” (u), “group name” (g), “queue name” (q) and “job name” (j). The numeric group consists of “requested number of processors” (c), “requested runtime” (r) and “requested memory” (m). Any other potentially useful attributes such as partition number and application arguments should be added to one of these two groups depending on the kind of attributes.

Our idea of defining the job similarity is a combination of the categorization and the instance based learning methods. Nominal attributes are used to categorize jobs and numeric attributes are used in determining the similarity by a new function (see Eq. (8.5)).

Given two jobs:

- \( J_1 = (u_1, g_1, q_1, j_1, c_1, r_1, m_1) \)
- \( J_2 = (u_2, g_2, q_2, j_2, c_2, r_2, m_2) \).

In order to determine whether these two jobs are similar, we first need to find a best template to classify them. The template can be any subset of the set \((u, g, q, j)\) including the empty subset\(^2\). For example, the template can be \((u), (g), (u, j),\) or \(()\). The question “how to select the best template” is answered in Section 8.3.3. Once the template is determined, these two jobs will be classified. If they fall into the same category, they are considered to be similar. Otherwise, they are not similar. For instance, if the best template found is \((u)\), \(J_1\) and \(J_2\) are similar in case they have the same user name.

Once \( J_1 \) and \( J_2 \) are determined to be similar, we need to calculate their similarity using numeric attributes. To avoid the phenomenon where an attribute can overpower the other attributes due to its large range, a linear normalization function \( f(x) \) is used to reduce numeric attributes in the range \([0, 1]\)

\[
f(x) = \frac{x - \min_x}{\max_x - \min_x}.
\] (8.4)

After normalizing \( c_1, c_2, r_1, r_2, m_1, m_2 \) by Eq. (8.4), we calculate the similarity between \( J_1 \) and \( J_2 \) by

\[
\text{Sim}(J_1, J_2) = \frac{c_1 \times c_2 + r_1 \times r_2 + m_1 \times m_2}{\sqrt{c_1^2 + r_1^2 + m_1^2} \times \sqrt{c_2^2 + r_2^2 + m_2^2}}.
\] (8.5)

The similarity is stronger if the value of \( \text{Sim}(J_1, J_2) \) is large.

\(^2\)An empty subset is symbolized as \(()\)
8.3.2 Predict the Runtime for a Future Job

We present in this section the idea to predict the runtime for a future job $J$. Firstly, we look for a set of jobs that are most similar to $J$ and then the runtime of $J$ is estimated based on the runtimes of jobs that belong to this set.

Search for a Set of Similar Jobs

First, we need to save $N$ recent finished jobs in the historical database. Once a future job $J$ arrives in the system, we search in this database $K$, $K < N$, jobs that are most similar to $J$ (these $K$ jobs are called a set of nearest neighbors). The approach to define the similarity between two jobs is described in Section 8.3.1. With this approach, it is possible to find no job similar to $J$ in the saved database using the best found template (see Section 8.3.3 for determining the template). In this case, we simply replace the best found template by the empty template\(^3\). It means all the jobs in the database are applied by Eq. (8.5) to calculate the similarity with $J$. The parameters $K$ and $N$ are determined in Section 8.3.3.

Estimate the Runtime

Once a set of $K$ nearest neighbor jobs are identified, we estimate the runtime of the future job $J$ by the average of the runtimes of these $K$ jobs

$$Est(J) = \frac{\sum_{i=1}^{K} R_i}{K}, \quad (8.6)$$

where $R_i$ is the actual runtime of job $i$th in the set of nearest neighbors. A weighted-average-based estimation similar to Eq. (8.3) can be applied, but we find in our experiments that this idea is not better than the estimation in Eq. (8.6).

A simple way proposed in [73] to reduce the possibility that $J$ is underestimated is to add $1 \frac{1}{2}$ times the standard deviation of the runtimes of these $K$ jobs. However, this method can lead to an inaccuracy and that increases the prediction error. Therefore, to reduce the inaccuracy, we will add to the estimate the standard deviation multiplied by a factor $\alpha$ as

$$Est(J) = \frac{\sum_{i=1}^{K} R_i}{K} + \alpha \times std(\{R_i\}), \quad (8.7)$$

---

\(^3\)Other choices can be applied. For example, we can remove some elements from the best found template to form a new template. However, it is not easy to generally determine elements that should be removed. Therefore, we decide to remove all elements in our study.
where \( std(\cdot) \) indicates the standard deviation and the factor \( \alpha \) will be determined in Section 8.3.3 in such a way that it decreases the inaccuracy as much as possible.

Adding the standard deviation can reduce the problem of underestimation but can also potentially cause the problem of overestimation. However, we find that the problem of overestimation can be partly limited by using the user requested runtime. This is because in backfilling parallel systems users are motivated to overestimate their jobs so that jobs will not be killed. Therefore, the user requested runtime can be used as a good upper bound for the estimate of the runtime of \( J \). Figure 8.1 gives an illustration about the ratio between the actual runtime and user requested runtime per job. We see clearly that most of the ratios are smaller than 1. This means that even also another variable such as the user requested runtime multiplied by a factor \( \beta, \beta < 1 \), can be used as the upper bound. Therefore, we use the user requested runtime multiplied by a factor \( \beta \) as the upper bound in our study as in the following equation

\[
Est(J) = \min \left( \sum_{i=1}^{K} \frac{R_i}{K} + \alpha \times \text{std}\{R_i\}, \beta \times r_J \right), \quad (8.8)
\]

where \( r_J \) is the user requested runtime of \( J \) and the factor \( \beta \) is determined in Section 8.3.3.

Figure 8.1: Ratios between actual runtimes per user requested runtimes, extracted from SDSC04.
8.3.3 Train for Best Parameters

As indicated in Sections 8.3.1 and 8.3.2, there are several parameters that need to be determined. They include the template to categorize jobs, the historical database size $N$, the number of nearest neighbor jobs $K$, the factor $\alpha$ and the factor $\beta$. However, we found that it is not reasonable if all jobs are applied with the same parameters to predict their runtimes. Instead, jobs should be divided into groups and each group should have its own parameters.

Separate Jobs

The user requested runtime is clearly an upper bound for estimating the runtime of a job. Therefore, we know that the actual runtime of a job is really small if its requested runtime is small. However, no conclusion can be made if the requested runtime is big. By using the user requested runtime as the upper bound in Eq. (8.8), we have reasons to believe that many of the prediction errors belong to jobs that have big user requested runtimes. To check this idea, we implement the most recent predicting method proposed in [104]. This is a categorization-based approach. Its idea is that the runtime estimate for a future job is calculated by the average of the actual runtime of the two most recently finished jobs submitted by the same user. We apply this predictor on the trace SDSC04 and manually select 20,000 seconds as pivot to separate jobs$^4$. This means that jobs that have a user requested runtime smaller than 20,000 seconds are considered to be small. Otherwise, they are considered as big jobs. Interestingly, the results in Table 8.2 show that although there are only a small number of big jobs (23.6\%), they contribute to a considerable prediction error (74\%). We explain this situation as follows. If a future job has a big user requested runtime, its actual runtime has a high probability to be big. Therefore, if we want to have an accurate estimate for this future job, we need to find similar big jobs. However, big jobs in fact usually have not terminated yet at the time of making the prediction. Consequently, short jobs that already terminated are often found as similar jobs. This will lead to the situation of underestimation and create a considerable prediction error. In order to solve this problem, we need to look further in the past with a hope that at the time of making the prediction similar big jobs are already completed. However, looking so far in the past is not good for small jobs because it is proven that parallel jobs are highly repetitive [15, 21]. This means that a recency characteristic is necessary for runtime prediction of small jobs. Therefore, our new idea to solve this issue is that we separate big jobs from small jobs and determine individual parameters for each group. The following is a simple example for this issue.

Assume that we have 6 jobs:

$^4$The manual selection is just to check our belief that many of the prediction errors belong to jobs that have big user requested runtimes. The possible best value for this pivot parameter will be determined in the training step of our predictor.
8.3. Predict Application Runtime

Table 8.2: Contribution of small jobs compared with big jobs in the absolute prediction error using SDSC04.

<table>
<thead>
<tr>
<th>Quantity</th>
<th>Big jobs</th>
<th>Small jobs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absolute error</td>
<td>38514 hours (74%)</td>
<td>13584 hours (26%)</td>
</tr>
</tbody>
</table>

- $J_1 = (u_1, g_1, q, app_1, 10, 500, 10, 1, 450)$,
- $J_2 = (u_2, g_2, q, app_2, 4, 110, 2, 3, 100)$,
- $J_3 = (u_1, g_1, q, app_1, 11, 520, 10, 480, 500)$,
- $J_4 = (u_2, g_2, q, app_2, 5, 50, 3, 490, 40)$,
- $J_5 = (u_2, g_2, q, app_2, 4, 90, 2, 550, 30)$,
- $J_6 = (u_1, g_1, q, app_1, 11, 500, 12, 560, 480)$,

where each job is formatted as (user name, group name, queue name, job name, requested number of processors, requested runtime, requested memory, arrival time, actual runtime). At the time $J_6$ arrives the system (560), if we do not look so far in the past to predict the runtime for $J_6$ (for example the historical database size $N = 3$ with $J_3$, $J_4$, and $J_5$ are in the database), we will not find any job that is really similar to $J_6$. Although $J_3$ is a good candidate, we do not know exactly its actual runtime since at the time we predict the runtime for $J_6$ (560), $J_3$ has not terminated yet. It is because $J_3$ arrives the system at the time 480 and runs 500 seconds, then it will terminate at the time 980. As such, we need to look further in the past by increasing $N$ to find similar jobs for $J_6$. If $N = 5$, $J_1$ can be used to yield a good runtime estimation for $J_6$ since $J_1$ already terminated. However, the estimation for $J_5$ will be inaccurate in this case. It is because $J_2$ is selected as the similar job for $J_5$ in case $N = 5$ instead of $J_4$ in case $N = 3$. This is clearly an issue: a big historical database size is better for big jobs while a smaller value of $N$ is better for small jobs due to the recency characteristic.

Train Parameters

We firstly summarize in this section all the parameters that have to be determined and then present the approach to obtain them.

All necessary parameters in our study include:
• A template to classify jobs as indicated in Section 8.3.1. This template is represented by a set \((x_1, x_2, x_3, x_4)\), where \(x_1, x_2, x_3, x_4\) represent user, group, queue and job name, respectively. If \(x_i = 1\), the corresponding job attribute will be inserted to the template (and removed from the template if \(x_i = 0\)).

• Because we need to separate jobs using the user requested runtime attribute \(r\), a pivot parameter Pivot needs to be determined, where \(\min(r) < \text{Pivot} < \max(r)\). It can be flexible to separate jobs in a more detailed way and obtain more accurate prediction. For instance, we can divide jobs into three groups: small, medium and big jobs. In that case, just simply define two pivot parameters Pivot\(_1\) and Pivot\(_2\).

• For each group of jobs, we need to determine the following parameters: the historical database size \(N\), the number of nearest neighbor jobs \(K\), the factor \(\alpha\) and the factor \(\beta\). We separate jobs into two groups in our study, hence we need \((N_1, K_1, \alpha_1, \beta_1)\) for the first group and \((N_2, K_2, \alpha_2, \beta_2)\) for the second group.

In our study, we divide each trace into two parts: the first part is used for training and the second part is used for testing. To determine the above parameters, we apply a genetic algorithm [30] on the training part and then use these parameters for the testing part to make predictions. A genetic algorithm will evolve individuals throughout a number of generations. For each generation, the algorithm will evaluate individuals in the population, select good individuals, cross and mutate them to produce the next generation. This work is repeated until a stopping condition is satisfied. We use a maximum number of generations as stopping condition.

Our individual representation for the training parameters (described above) is as follow

\[(x_1, x_2, x_3, x_4, \text{Pivot}, N_1, K_1, \alpha_1, \beta_1, N_2, K_2, \alpha_2, \beta_2).\]

A fitness function is used to evaluate each individual. It is chosen so that good individuals have higher fitness and therefore have higher chance to be selected for producing the next generation. Because our objective is to reduce the number of jobs that are underestimated as well as to decrease the prediction error as much as possible, we select the following fitness function

\[
\text{Fitness} = \frac{-\text{NormalizedError}}{e^{(1 - \text{PercentUnderestimate})^2}},
\]  

(8.9)

where \(\text{NormalizedError} = \sum_i |P_i - R_i|/\sum_i R_i\), \(P_i\) and \(R_i\) are the predicted and actual runtime of job \(i\)-th, respectively. \(\text{PercentUnderestimate}\) is the percentage of underestimated jobs, \(\text{PercentUnderestimate} \in [0, 1]\). We use \(\text{exp}\) in the denominator of the fitness function because we want to reduce the impact of \(\text{PercentUnderestimate}\) and increase the impact of \(\text{NormalizedError}\) on the function.
8.4 Experimental Results

The workloads used in our experiments are described in Section 8.2. Training parameters obtained for each workload are shown in Table 8.3. The quality of our predictor is compared with the predictors proposed in the most recent studies [50, 104]. For the categorization-based approaches, we select the predictor proposed by Tsafir et al. in [104]. It predicts the runtime of a future job by averaging the runtimes of the two most recent terminated jobs submitted by the same user. This simple predictor is demonstrated to significantly improve the prediction accuracy. For the instance-based learning approach, we select the most recent predictor proposed by Li et al. in [50]. An implementation of this predictor is available on [81]. We refer the first predictor as Tsafir’s and the second predictor as Li’s.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>SDSC02</th>
<th>SDSC04</th>
<th>HPC2N04</th>
<th>HPC2N05</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1$</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$x_2$</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$x_3$</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$x_4$</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Pivot</td>
<td>19369</td>
<td>40931</td>
<td>32257</td>
<td>122562</td>
</tr>
<tr>
<td>$N_1$</td>
<td>5046</td>
<td>6268</td>
<td>6904</td>
<td>4763</td>
</tr>
<tr>
<td>$K_1$</td>
<td>10</td>
<td>8</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>$\alpha_1$</td>
<td>0.956</td>
<td>0.976</td>
<td>0.997</td>
<td>1</td>
</tr>
<tr>
<td>$\beta_1$</td>
<td>0.949</td>
<td>0.958</td>
<td>0.976</td>
<td>0.989</td>
</tr>
<tr>
<td>$N_2$</td>
<td>8661</td>
<td>9757</td>
<td>9189</td>
<td>5610</td>
</tr>
<tr>
<td>$K_2$</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>$\alpha_2$</td>
<td>0.105</td>
<td>0.038</td>
<td>0.021</td>
<td>0.008</td>
</tr>
<tr>
<td>$\beta_2$</td>
<td>0.941</td>
<td>0.925</td>
<td>0.986</td>
<td>0.832</td>
</tr>
</tbody>
</table>

8.4.1 Metrics Used in Evaluation

We use two metrics to evaluate the accuracy and one metric to evaluate the problem of underestimation.
Underestimation

We use the percentage of underestimated jobs to evaluate the problem of underestimation since we want to reduce the number of jobs that are underestimated as much as possible.

Mean Absolute Error (MAE)

The mean absolute error of \( N \) jobs is calculated by

\[
MAE = \frac{\sum_{i=1}^{N} |P_i - R_i|}{N},
\]

(8.10)

where \( P_i \) and \( R_i \) are the predicted and actual runtime of job \( i \)th, respectively. Another metric called normalized error (NE) can also be used

\[
NE = \frac{\sum_{i=1}^{N} |P_i - R_i|}{\sum_{i=1}^{N} R_i}.
\]

(8.11)

Since we apply the same traces for all predictors, it means \( \sum_{i=1}^{N} R_i \) does not change. Hence, we only use the metric mean absolute error in our evaluation.

Weighted Absolute Error (WAE)

Proposed in [108], the metric weighted absolute error is motivated by the fact that a larger and longer job with a prediction error should have more impact on other jobs than a smaller and shorter job with the same prediction error. The weighted absolute error of \( N \) jobs is calculated by

\[
WAE = \frac{\sum_{i=1}^{N} |P_i - R_i| \times C_i \times R_i}{\sum_{i=1}^{N} C_i \times R_i},
\]

(8.12)

where \( P_i, R_i \) and \( C_i \) are the predicted runtime, actual runtime and requested number of processors of job \( i \)th, respectively.

8.4.2 Underestimation Problem

As explained in the beginning of this chapter, in backfilling using system-generated predictions, the problem of underestimation can lead to an adverse impact on system
performance. Therefore, predictors need to carefully take care of this problem and reduce the number of underestimated jobs as much as possible. Our predictor notices this problem and considerably reduces the number of underestimated jobs compared with other predictors as shown in Table 8.4. This is since we add the standard deviation as in Eq. (8.8) when we calculate the estimate. Furthermore, it is also caused by the fact that we separate the roles of big jobs from small jobs as explained in Section 8.3.3 and take into account the percentage of underestimated jobs in the objective function when training for the best parameters (see Eq. (8.9)).

<table>
<thead>
<tr>
<th>Trace</th>
<th>Tsafrir’s</th>
<th>Li’s</th>
<th>Our Predictor</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDSC02</td>
<td>47%</td>
<td>50%</td>
<td>25%</td>
</tr>
<tr>
<td>SDSC04</td>
<td>45%</td>
<td>49%</td>
<td>29%</td>
</tr>
<tr>
<td>HPC2N04</td>
<td>55%</td>
<td>52%</td>
<td>36%</td>
</tr>
<tr>
<td>HPC2N05</td>
<td>51%</td>
<td>50%</td>
<td>31%</td>
</tr>
</tbody>
</table>

Reducing the number of underestimated jobs will certainly increase the number of jobs that are overestimated. However, in backfilling systems, overestimating jobs is still better than underestimating since this gives the scheduling algorithms some flexibility that leads to better schedules [73]. Furthermore, we also take care and try not to let the overestimation problem increase the prediction error by using the upper bound for an estimation as in Eq. (8.8). This mechanism really contributes efficiently to decrease the prediction error as shown in Sections 8.4.3 and 8.4.4.

### 8.4.3 The Mean Absolute Error

Results for the mean absolute error are shown in Table 8.5. Our predictor is clearly better than Tsafrir’s in most cases. Comparing with Li’s, our predictor is much better in case of SDSC04 but not much different in the other cases. In order to explain the reason, we draw Figure 8.2, which shows the cumulative distribution functions (CDF) of the runtimes. We would like to remind that our predictor will work well if there are several big jobs in a trace since we separate the roles of big jobs from small jobs as discussed in Section 8.3.3. This is demonstrated in case of SDSC04, where nearly 14% jobs are bigger than 10,000 seconds, our predictor is 20% better than Li’s. Among all the traces, SDSC02 includes a large majority of small jobs (only 5% jobs are bigger than 10,000 seconds), therefore, our predictor yields approximately the same error as Li’s in this case. Note, we already showed in Section 8.3.3 that only a small number of big jobs will also contribute to a considerable prediction error. Although HPC2N04 and HPC2N05 also have several big jobs, the results are not much different between all three predictors. It is because their mean runtimes are
very big (see Table 8.7). This means that these traces have a lot of very long jobs and all predictors including ours do not look far enough in the past to find similar jobs. Consequently, when predicting for very long jobs they find wrong similar jobs and yield big prediction errors. Particularly, our predictor is still a little bit better than Li’s in case of HPC2N04 because its mean runtime is still not very big.

![Runtime distribution of traces.](image)

**Figure 8.2:** Runtime distribution of traces.

**Table 8.5:** Mean absolute error (in minutes).

<table>
<thead>
<tr>
<th>Trace</th>
<th>Tsafrir’s (1)</th>
<th>Li’s (2)</th>
<th>Our Predictor</th>
<th>Compared with (1)</th>
<th>Compared with (2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDSC02</td>
<td>38.7</td>
<td>30.5</td>
<td>30.7</td>
<td>21%</td>
<td>0%</td>
</tr>
<tr>
<td>SDSC04</td>
<td>60.4</td>
<td>60.1</td>
<td>48.1</td>
<td>20%</td>
<td>20%</td>
</tr>
<tr>
<td>HPC2N04</td>
<td>187</td>
<td>164</td>
<td>156</td>
<td>17%</td>
<td>5%</td>
</tr>
<tr>
<td>HPC2N05</td>
<td>474</td>
<td>436</td>
<td>438</td>
<td>8%</td>
<td>0%</td>
</tr>
</tbody>
</table>

**8.4.4 The Weighted Absolute Error**

The study in [108] demonstrates that no single error metric can fully predict scheduling performance and the metric mean absolute error is not sufficient for characterizing errors. According to [108], the metric mean absolute error is even much worse than the metric weighted absolute error in capturing the adverse impact of runtime estimate errors on scheduling performance. With respect to the metric weighted absolute error, Table 8.6 shows that our predictor works considerably better than Tsafrir’s and Li’s.
8.5 Summary

It is easy to see that the best result is from HPC2N05 because this trace has a large number of big and long jobs (see Table 8.7). However, the difference is not strong in case of SDSC02 because most of its jobs are small and short (68% jobs are serial and mean runtime is only 52 minutes).

<table>
<thead>
<tr>
<th>Trace</th>
<th>Tsafir’s (1)</th>
<th>Li’s (2)</th>
<th>Our Predictor</th>
<th>Compared with (1)</th>
<th>Compared with (2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDSC02</td>
<td>523</td>
<td>402</td>
<td>374</td>
<td>29%</td>
<td>7%</td>
</tr>
<tr>
<td>SDSC04</td>
<td>328</td>
<td>339</td>
<td>271</td>
<td>17%</td>
<td>20%</td>
</tr>
<tr>
<td>HPC2N04</td>
<td>984</td>
<td>903</td>
<td>805</td>
<td>18%</td>
<td>11%</td>
</tr>
<tr>
<td>HPC2N05</td>
<td>2134</td>
<td>2543</td>
<td>1722</td>
<td>19%</td>
<td>32%</td>
</tr>
</tbody>
</table>

Table 8.6: Weighted absolute error (in minutes).

<table>
<thead>
<tr>
<th>Trace</th>
<th>Mean Runtime (minutes)</th>
<th>Serial Jobs</th>
<th>Parallel Jobs</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDSC02</td>
<td>52</td>
<td>68%</td>
<td>32%</td>
</tr>
<tr>
<td>SDSC04</td>
<td>86</td>
<td>33%</td>
<td>67%</td>
</tr>
<tr>
<td>HPC2N04</td>
<td>246</td>
<td>63%</td>
<td>37%</td>
</tr>
<tr>
<td>HPC2N05</td>
<td>478</td>
<td>35%</td>
<td>65%</td>
</tr>
</tbody>
</table>

Table 8.7: Characteristics of traces.

8.5 Summary

In this chapter, we presented a novel approach to predict application runtimes in backfilling parallel systems. We also explained that both inaccuracy and underestimation in prediction are not good for backfilling systems. Hence, our predictor is suitable for backfilling scheduling because it obtained good accuracy and considerably reduced the number of underestimated jobs. The idea of our approach is to define job similarity by a combination of the categorization and the instance based learning methods, with a new similarity function (see Eq. (8.5) in Section 8.3.1). Moreover, another idea is to separate big jobs from small jobs (see Section 8.3.3). In future work, we will use this predictor to estimate job response times. Moreover, we also want to evaluate how well this approach really impacts the efficiency of backfilling scheduling.