Process Lifetime Prediction Using Artificial Neural Networks : PLANN

Mark Rich & Louis Oliphant

17th December 2001

Abstract

Process CPU lifetime prediction is currently very rudimentary, using a single characteristic and perhaps some *a priori* knowledge about process behavior[HBD97]. We believe that more accurate prediction of CPU lifetime through the use of multiple characteristics will improve performance and flexibility in process migration. We propose a new framework called Process Lifetime Artificial Neural Network (PLANN) to assist in process migration. A neural network will be intermittently trained off-line and then the results will be quickly incorporated into migration decisions. This paper details the viability of our concept, the difficulty of the problem, and directions for improving predictive accuracy.

1 Introduction

The amount of CPU time a process needs to run to completion is useful information but very challenging to obtain. This information could be used in migration systems. When a CPU is under load processes are migrated to other systems, relieving the local system. It is most efficient to migrate processes that will require the most CPU time in the future. These processes will be best able to amortize the cost of the move from one machine to the next.

Previous approaches to calculating a process's lifetime have used cumulative statistical approaches on large workloads, such as mean run time and standard deviation for a specific command. These data are then used to make predictions about future processes.

Our approach to this problem is slightly different. We will take each individual process from the workload and monitor it over its lifetime looking for characteristics about the process that indicate whether the process is about to terminate. We will train a neural network to observe these changes in characteristics. When the CPU is under load the neural network can be queried about each process currently running. The network will predict the lifetime of each process and the machine can then migrate the process that has the longest remaining lifetime.

2 Related Work

Process CPU lifetime prediction is currently very rudimentary, using a single characteristic and perhaps some *a priori* knowledge about process behavior. First, some researchers in off-line static prediction used records of previous job features to predict process lifetimes. Svensson [Sve90] attempted static prediction of a process's lifetime by using an average of previous run times for that particular job. He used these predictions to perform non-preemptive scheduling of jobs in a 16 node diskless SUN-3 network. His results show progress toward creating more efficient scheduling policies with rapid response times. Smith *et. al.* [SFT98] enrich their feature space beyond process name by including other parameters such as user, arguments, number of nodes requested. They used a greedy search to find a template for similar jobs and predicted the lifetime of a job by calculating the mean of the most similar cluster.

Second, efforts toward dynamic prediction incorporate on-line job behavior into their process lifetime predictions. Harcol-Balter and Downey [HBD97] use an analytic model to predict total process run time based on current run time. They incorporate this prediction into a preemptive network scheduling policy. Kapadia *et. al.* [KBFL98] look for exact matches to selective processes with their current parameters. When a match is not found, they use feature selection and locally weighted regression analysis. This prediction is real-time in nature because it must occur after the user initiates the job, but before it is scheduled. In a later paper, [KFB99], they compare nearest-neighbor, weighted average and locally-weighted polynomial regression as methods for prediction of process lifetime. These predictions are used in a large-scale network scheduler at Purdue University. Russ *et. al.* [RLC+01] propose theoretical implementations of an artificial immune system for run-time process migration. This system would adapt to process variation and evolution over time as well as system load. Their work is currently very preliminary.

Finally, opportunities for performance improvement in various systems are explored through the help of CPU lifetime prediction. Freund *et. al.* [FKM96] break up the run time of a job into pieces and compute disjoint characteristics. These predictions are part of their larger system for SmartNet, a scheduler framework applicable to many distributed system schedulers. More accurate predictions of CPU lifetime could improve their performance. Smith *et. al.* [STF99] extend their previous work to estimate queue wait times and improve scheduler performance. They find that using their run-time predictor results in lower mean wait times for their workloads in comparison to existing predictors.

3 PLANN Theory

The PLANN module will work in three different phases, normal mode, training mode, and query mode. During normal mode the system will run without modification. For a schematic of how the module will run in the two modified modes see Figure 1.

Figure 1: Schematic of the PLANN module

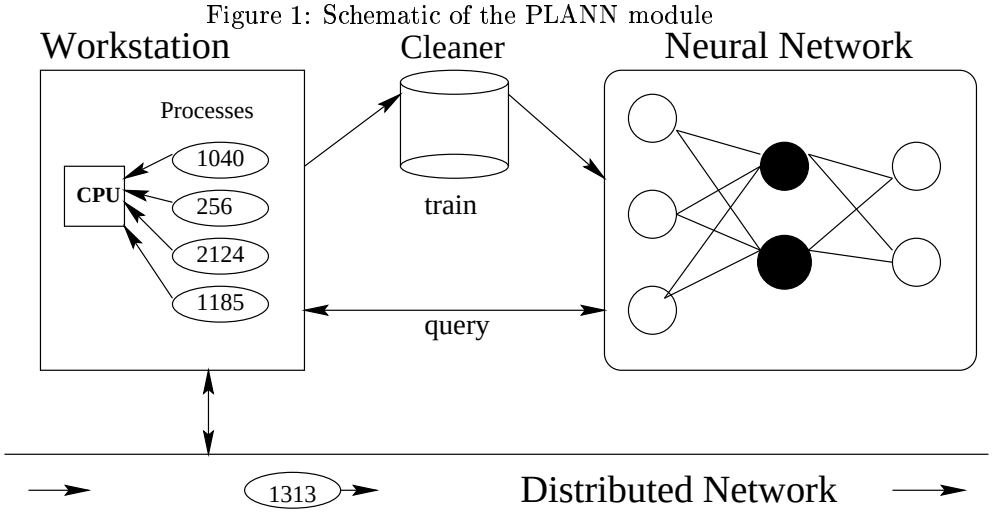


Table 1: snapshot of a process with categorization

$feat_1 = pid$	$feat_2 = current\,run\,time$	$feat_3 = \#open\,files$	...	$category$
1040	5 seconds	4	···	35 seconds

During training mode the workstation will continuously send information about each process on the system to a database. For a list of features about each process written to this database see Table 2. At some point within the training phase ($\sim \frac{1}{2}$hour into training) the workstation stops sending information and the database is cleaned. Cleaning removes any records for processes where the entire process's lifetime was not captured in the database. This will include processes that started before the training phase and processes that did not finish execution within the training phase. In addition, records that are for processes that had very short running times ($< 1\,sec.$) are removed. Each record is then labeled with the process's final run time. An example of a cleaned record is shown in Table 1 .

This database is then given to the neural network for training. The network looks at the features of each record and attempts to predict the run time. If it is incorrect in predicting the run time then the network's weights are modified to improve its prediction. The training of the neural network involves a lot of processor time. It is envisioned that this will occur when the system is not in use. During the gathering of data the system will run slightly slower.

The final phase is the query phase. When the workstation becomes overloaded and decides to migrate a job away the PLANN module is consulted. Each process that is currently running on the network has a snapshot of its features taken and sent to the module. the module will predict how long that process will run. The workstation can then decide which process to migrate.

There are many reasons for using neural networks [Mit97] in the PLANN module. Neural networks are essentially learning an evaluation function:

$$f(feat_1, feat_2, feat_3 \cdots) = category$$

Neural Networks can easily handle many features about a process. The training phase of the network is slow, but once trained the time spent calculating a prediction will be minimal, involving little more than two matrix multiplications. Essentially each node in the network performs a linear weighted sum of the input features and then passes this weighted sum through a squashing function called the sigmoid so values do not get too large

$$sigmoid\left(\sum_{i=1}^{n} w_i * feat_i\right) = output$$

where n is the number of features and w_i is the weight of feature i. The outputs are used to predict the category. Neural networks also provide good approximations and excel at real-valued features as well as real-valued output.

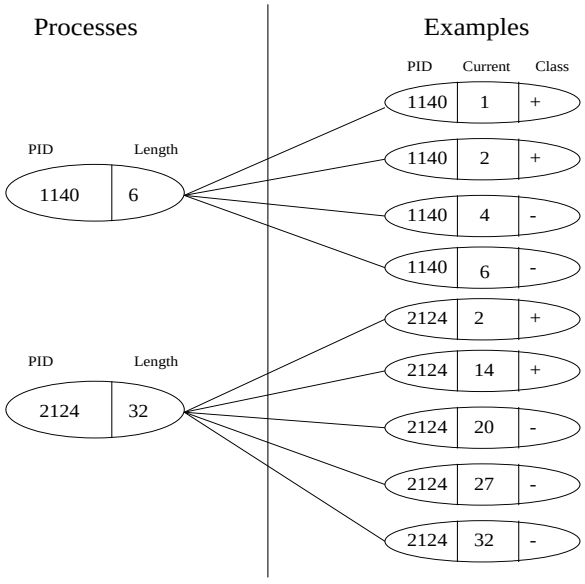
4 Methodology

Our first consideration with gathering data concerns the features, or attributes, of our dataset. What are the salient pieces of information that will help us decide how long a particular process will run? We settled on two separate types of features. First, we want to find features that will change within a process. We hope these features will give us some indication of when a process is shifting gears and closing down execution, for example, the number of open files could decrease as a process nears completion, or the instruction pointer could approach the end of the code segment. Second, we want to find features that will help us distinguish between processes. With these, we hope that there will be some generalization possible among different processes; perhaps the length of a processes data segment the personality of a process is related across processes to their CPU usage time.

But before we begin to analyze our data, we must also consider the characteristics of our workload. Harcol-Balter and Downey[HBD97] experimented with six different workloads with at least 7000 processes. They conclude that their process distributions match a 1/T curve, and use this in their migration policy. We do not have that much time nor available data, however, we must guarantee that our workload follows a 1/T distribution for our results to be relevant and worthy of analysis. Also, we wish to have the workload that we record to be under migration conditions, such that the CPU is in demand.

Previous research gathered data on a per process basis. Conclusions were then made based upon similarity to these cumulative statistics. Under our approach, we will monitor the active processes each second, take a snapshot of every process, and record all the relevant data to a file. Therefore, a process will

Figure 2: How processes become training examples for PLANN



be divided up into many examples, one for each time it is seen inside the CPU scheduler. Suppose we have two processes that will run on our CPU, with PIDs 1040 and 2124. Figure 2 shows how these processes get recorded as training examples.

Once our data is captured and divided into examples, there are two preprocessing steps before we begin training our neural network. First, all processes which did not use any user ticks are removed. These processes belong to the kernel and should not be migrated. Second, we need to add the category for each example. In our prototype PLANN, we begin by trying to answer the question "Will this process run at least as long as it has already?" The final user ticks is found for each process ID. If an example is greater than half this number, it will become a negative example, otherwise it will be positive. Again, Figure 2 demonstrates this processes.

5 Experiments

We ran our prototype system on a Pentium III 667MHz machine with 128 MB running Linux 2.4.7. In order to take the snapshot of the processes, we needed to modify one method in the Linux kernel code in the file SCHED.C. All of our data could be found in the TASK_STRUCT structure. These structures are kept inside a list in the scheduler; every second, we recorded the features for each process to a file. Data was gathered for 30 minutes with various processes running, such as MAKE, CPP, JAVA, XEMACS, BASH, KDEINIT, FIND, GIMP, etc.

Figure 3: Workload gathered from Pentium III 667MHz running Linux 2.4.7

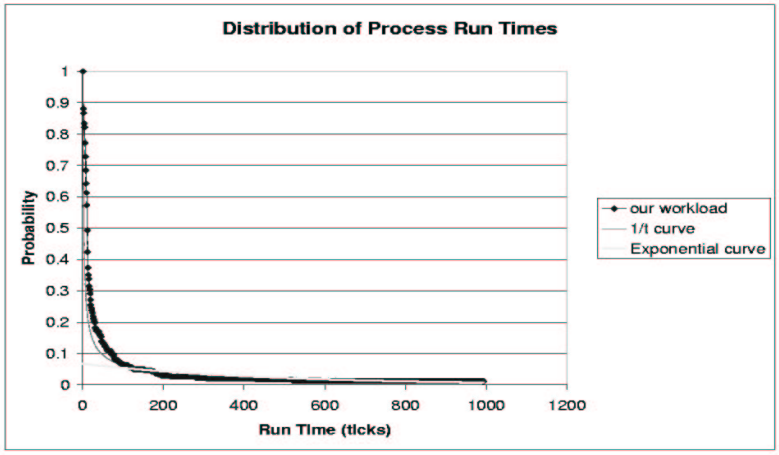


Table 2: Features recorded for each process.

Category	Maximum	Minimum
Process ID	234	5248
User ticks	0	8391
Kernel ticks	0	56942
State	0	2
Address Limit	-1073741824	0
Counter	0	20
Nice	*0*	*0*
Beginning of Code	0	134512640
End of Code	0	136203153
EIP	-1072622610	0
Length of Data	0	2906496
Flags	0	320
Personality	*0*	*0*
User ID	0	43
Group ID	0	43
Capability Type 1	-257	0
Capability Type 2	*0*	*0*
Capability Type 3	-257	0
Links	*0*	*0*
tty	0	1
Locks	0	1

Our first step was to verify that our sample run had generated reasonable data. We found a very close match to the 1/T curve, as can be seen in Figure 3. A listing of the features gathered can be found in Table 2. Notice that some features turned out to be useless, such as *Nice* and *Personality*. After cleaning, a total of 303 processes were detected, and this divided into 3584 examples. Of these examples, 449 were classified as positive examples, meaning they would run at least as long as their current user ticks value.

A common technique for assessing predictive accuracy in machine learning is 10-fold cross-validation. The data is divided into 10 "folds", or distinct subsets. Each fold is then used as a testing set while the other 9 folds become the training set. This method tries to even out the gains or losses from choosing a particular random sample for our training set. Folds were created by process rather than example for our experiments. This was necessary to prevent the neural network from inferring the classification of an example based on other examples from the same processes. We viewed this as peaking at the test set. Again, we were hoping for changes within a process to generalize across processes. Our results can be found in Table 3. Our average accuracy over all 10 folds was an astounding 88%, with a standard deviation of only 8%.

But, did we really learn anything about our data? To determine this, we must perform some statistical tests to see if we beat the null hypothesis. The null hypothesis states that the results of your two algorithms are not statistically significant. We chose to compare our algorithm with a very simple one that for every single example would predict negative. Results for this algorithm are also presented in Table 3. The predictive accuracy of these two algorithms is identical; there is no way to distinguish them statistically. So, our neural network has learned to classify all processes as negative, meaning it will recommend no processes for migration.

Why was there a distinct lack of learning? Further examination of our data showed there were only 16 processes with both positive and negative examples. A large number of the recorded processes were only seen once in our dataset, and therefore each of these examples was classified as negative. Also, many processes quickly executed their user ticks early in their CPU tenure and subsequently went to sleep. Each time the process was recorded, the user ticks were constant. The overwhelming majority of negative examples contributed to our network's incapacity to learn these classifications. And within these 16 relevant processes, we could observe none of the expected changes from start to finish. Within process changes were either random or nonexistent.

To remedy this problem, we decided to increase the number of possible classifications of an example. We label each example with $\log_2$ of its final execution time. As an example a process that ran for 64 ticks would be placed in bin 6. This grouping allowed us to have a relatively small number of bins even though the maximum final user ticks was above 8000. This approach is akin to earlier attempts of per process classification, as each example from a process will receive the same classification. This removes the possibility of within process learning, which was quite unsuccessful. The results our 10 fold cross-validation testing are shown in Table 4. For comparison, we chose an "always choose most

Table 3: Predictive Accuracy of ANN vs Null Hypothesis

Fold	ANN	Always -
1	57%	57%
2	86%	86%
3	98%	98%
4	68%	68%
5	99%	99%
6	100%	100%
7	99%	99%
8	90%	90%
9	83%	83%
10	100%	100%
Average	88%	88%
Std. dv.	8%	8%

popular bin" algorithm, which chose bin 4 for every fold except fold 6, where the bin chosen was 6.

For multi-category classification problems, it is also common to examine the confusion matrix. The rows are the correct output, while the columns are the learned output. From this matrix we can assess the relative difficulty of each category and identify any common clusters. As can be seen in Figure 4, our network is able to predict well for bin 2, 4, and 5. It is also clear that our network most often predicts bin 4. For some reason we classify examples from bin 6 as either bin 4 or bin 9. Our second attempt at process lifetime prediction did not fare much better. Although we have a higher accuracy than the "Always choose most popular bin" algorithm, our confidence level in this distinction, determined by statistical paired t-tests, is much less than 90%.

6 Conclusion

We believe the PLANN framework is viable. Only minimal changes were necessary to the Linux kernel to record the processes. The slowdown time of the CPU for this extra processing is negligible. Training the neural network, can be done off-line at the CPU's leisure, and querying the neural network for process predictions is fast.

Unfortunately, the chosen domain of process lifetime prediction is very hard. Processes expose very little of their internal details to the kernel, making useful features hard to come by. What features we do have show little to no correlation with the ultimate CPU length of the process.

Before a full implementation of PLANN can be achieved accurate results must be obtained. Other ways that may prove fruitful include looking at additional features such as the instruction pointer. Reducing redundancy in the data gathering process would also prove useful. Having each process write out

Figure 4: Confusion Matrix

Confusion Matrix for PLANN

correct classification \ PLANN prediction	1	2	3	4	5	6	7	8	9	10	11	12	13	Grand Total
1		4	3	149	79			1						236
2		427	4	3	138		1						1	574
3		18	105	196	1								1	321
4	150		90	316	140	141							1	838
5			1	150	258	1	1	111						522
6				141	1	1	6		138					287
7		3		13	157		2							175
8			147		138									285
9				136		136				1				275
10				146	90									236
11														0
12														0
13				22										22
Grand Total	150	452	350	1274	1002	279	10	112	138	1	0	0	3	3771

Table 4: Predictive Accuracy using Exponential Bin divisions

Fold	ANN	Always Popular Bin
1	9%	0%
2	46%	2%
3	6%	84%
4	9%	7%
5	4%	33%
6	3%	1%
7	4%	1%
8	50%	19%
9	21%	0%
10	56%	30%
Average	21%	18%
Std. dv	0.214	0.265

information to the database only when its user ticks increases would help reduce redundancy. In addition other workloads could be obtained. We performed most of our tests on a single workload.

Once somewhat accurate results are found, we believe implementing the PLANN module in a migration system would be straight forward. Measurements could then be taken to see if average time to completion is reduced and if processor utilization increases.

References

[FKM96] R. Freund, T. Kidd, and L. Moore. Smartnet: a scheduling framework for heterogeneous computing. In *Proceedings of the 2nd Int. Symp. Parallel Architectures, Algorithms, and Networks*, pages 514–521, 1996.

[HBD97] Mor Harchol-Balter and Allen B. Downey. Exploiting process lifetime distributions for dynamic load balancing. *ACM Transactions on Computer Systems*, 15(3):253–285, 1997.

[KBFL98] Nirav H. Kapadia, Carla E. Brodley, Jose A. B. Fortes, and Mark S. Lundstrom. Resource-usage prediction for demand-based network-computing. In *Proceedings of the Symposium on Reliable Distributed Systems*, pages 372–377, 1998.

[KFB99] N. Kapadia, J. Fortes, and C. Brodley. Predictive application-performance modeling in a computational grid environment. In *Eighth IEEE International Symposium on High Performance Distributed Computing (HPDC)*, 1999.

[Mit97] Tom Mitchell. *Machine Learning.* McGraw-Hill Companies, 1997.

[RLC+01] Samuel H. Russ, Aric Lambert, Joel Camenisch, Vijay Velusamy, Rajesh Rajan, and et al. Predictive scheduling for distributed computing, 2001.

[SFT98] Warren Smith, Ian T. Foster, and Valerie E. Taylor. Predicting application run times using historical information. In *JSSPP*, pages 122–142, 1998.

[STF99] Warren Smith, Valerie E. Taylor, and Ian Foster. Using run-time predictions to estimate queue wait times and improve scheduler performance. In *JSSPP*, pages 202–219, 1999.

[Sve90] A. Svensson. History, an intelligent load sharing filter. In *Proceedings of the 10th International Conference on Distributed Computing Systems (ICDCS)*, pages 546–553, Washington, DC, 1990. IEEE Computer Society.