2018 4th IEEE International Conference on Big Data Security on Cloud

Timing Prediction for Dynamic Application Migration on Multi-Core Embedded Systems

Zheng Li
School of Computer Sciences
Western Illinois University
Macomb, USA

Hao Wu
Department of Computer Science
Southern Connecticut State University
New Haven, USA

Shuibing He
School of Computing
Wuhan University
Wuhan, China

***Abstract*—To accommodate execution mode change and hardware malfunction, dynamic system reconfiguration, which invokes application migration across different processing cores, needs to be supported on multi-core embedded systems. Different application migration strategies will impact system's timing behaviors in different manners, it is important to select an appropriate one such that the system's timing performance after the migration process is still acceptable. The focus of our research is to predict the system's timing change of possible migration strategies and upon which to choose the optimal one. Extensive experiments have been set up by running multiple benchmarks and experimental results validate the effectiveness of our proposed approach.**

I. Introduction

Multi-processor system-on-chip (MPSoC) being widely considered for embedded systems, such as automotive units and avionics. To utilize the advance of tremendous computing capacity and high degree of explicit parallelism of MPSoC, mixed-criticality design with critical and non-critical applications integrated on shared hardware platform is deemed to be the trend [1]. For safety and cost-efficiency purposes, hardware resources are reserved for critical applications under the worst case consideration, while non-critical applications are designed to compete the available resources and run simultaneously to provide best-effort service [2]. Though non-critical applications are not safety-critical, their performance degradation will impact system's quality-of-service (QoS). In traditional embedded system design, these applications are statically mapped to processing cores and tremendous efforts are spent on testing and verifying the possible use-cases in order to achieve the desired system QoS.

However, embedded systems may work in hostile environment, if predefined use cases do not match the real-world execution scenarios, the system may suffer performance degradation. In addition to unprecedented execution scenario, hardware malfunction is another challenge. In case of core wear-out, the entire system may not function any more and hence needs to be redesigned and reconfigured. If applications can be dynamically migrated among different processing cores, embedded systems will be able to reconfigured at run-time to accommodate unprecedented execution scenario changes and core malfunction.

However, different application migration strategies will change affect the system's timing behaviors in different manners. If such timing changes can be foreseen, upon which we can choose an appropriate migration strategy such that the system performance after the migration process is still acceptable. Application execution times are determined by various factors and heavily influenced by scheduling policies. As Operating systems based on the Linux kernel are widely used in embedded systems, in this paper, we assume the default scheduler in Linux kernel since 2.6.23 release, i.e. the Completely Fair Scheduler (CFS) [3], is used to schedule non-critical applications.

With the above discussion, we are to explore the predictors and establish analytical models to dynamically predict system's timing changes incurred by an application migration process on many-core embedded systems. Since critical applications run on dedicated resources and their services are guaranteed, the focus of this paper is to predict the timing of non-critical applications which simultaneously run on shared multi-core processors scheduled by CFS.

The rest of the paper is organized as follows. We first summarize the related work in Section II. An offline timing prediction model is presented in Section III. To keep the timing prediction mode always up-to-date at run-time, our proposed online model update strategy is studied in Section IV. Experimental settings and results are discussed in Section VI. Finally we conclude our work in Section VII.

II. Related work

Multiple applications may be active simultaneously on multi-core processors, to achieve system performance optimization, Benini [4] proposed a semi-static approach to compute and store the applications allocation and scheduling solutions for all possible use-cases offline. In order to dynamically determine the application migration, the related timing prediction techniques need to be studied first. There were some existing work regarding to applying learning-based prediction techniques to estimate a processor's performance. Joesph [5] applied regression-based approaches for performance prediction. To predict a workload on another hardware platform with different underlying architecture, Zheng [6] recently developed an offline cross-platform prediction approach under the assumption that application's performance lied in a linear relationship with the predictors. However, the prediction performance may suffer if such assumption was not invalid.

978-1-5386-4399-0/18/$31.00 ©2018 IEEE
DOI 10.1109/BDS/HPSC/IDS18.2018.00044

TABLE I: Performance Counters

ID	Counter	ID	Counter
1	CPU-cycles	8	LLC-store-misses
2	Instructions	9	Page-faults
3	Cache-references	10	Context-switches
4	Cache-misses	11	Branch-instructions
5	LLC-loads	12	Branch-misses
6	LLC-load-misses	13	Stalled-cycles-frontend
7	LLC-stores	14	Stalled-cycles-backends

To explore the potential non-linear relation between the target variable and its predictors, kernel-based regression methods are commonly utilized [7], [8]. In this paper, we will study how to apply kernel based regression techniques to accurately predict timing behavior changes invoked by an application migration on many-core embedded systems.

III. Offline Timing Prediction Model Training

Consider an application migration scenario on a multi-core system with app1 is being migrated from core i to core j, not only the timing behavior of migrated application app1 (Mapp) will be significantly altered, execution times of all the applications on destination core j (abbreviated as Dapp) will inevitably be extended. In the following discussion, we first focus on predicting the timing change of Mapp, and then extend the discussion to Dapps.

Our proposed model-based timing prediction strategy consists of two phases: offline model training and online model update. The offline stage is to train an initial prediction model during the system design stage; after the system is actually in service, online update stage is to dynamically tune the prediction model to keep it up-to-date. In the following, we first present our proposed offline model training in details.

A. Training Data Profiling

A model-based prediction strategy is to discover the relation between the outcome variable and predictors by learning from collected data instances. Therefore, the foremost step is to investigate potential predictors which may impact Mapp's execution time and trace the related data to be used for model training.

A few simple facts reveal that an application's execution time is determined by various factors. For example, if multiple applications scheduled by CFS on the same processing unit, each execution time will be extended. Things can be more complicated on multi-core systems. As cores may have dedicated processing elements and caches but share the main memory, memory-access interferences from other cores will also impact application's timing behaviors as well. These observations imply that, in order to accurately predict an application' execution time, the status of both the target application and the deployed hardware platform need to be taken into consideration.

As we know, performance counters listed in Table I are implemented in computer systems, which can be profiled to track the performance of a system as well as deployed applications[1] [9]. The counters measured at system level provide information as to how well the system is performing; while application level counters reveal the performance of the target application[2]. For instance, the cache-references of an application indicates the number of cache references only invoked by this application; while the cache-references counter measured for a processing core indicates total cache references invoked by all the applications deployed on this core. As an application's execution time is determined by both the application itself and how much resource to be provided by the system, we trace performance counters for both the target application and its processing core.

TABLE II: Application Performance Indicator

ID	Counter	ID	Counter
1	CPU-cycles-app	14	Stalled-cycles-backends-app
2	Instructions-app	15	CPU-cycles-core
3	Cache-references-app	16	Instructions-core
4	Cache-misses-app	17	Cache-references-core
5	LLC-loads-app	18	Cache-misses-core
6	LLC-load-misses-app	19	LLC-loads-core
7	LLC-stores-app	20	LLC-load-misses-core
8	LLC-store-misses-app	21	LLC-stores-app-core
9	Page-faults-app	22	LLC-store-misses-core
10	Context-switches-app	23	Page-faults-core
11	Branch-instructions-app	24	Context-switches-core
12	Branch-misses-app	25	Cache-misses-other-cores
13	Stalled-cycles-frontend-app	26	App-execution-time

Table II lists the counters we have identified to quantify the performance of an application running on a specific platform. Among which, 1 - 14 are the counters of the target application. As the application's counters depend on its deployed platform, the counters of its processing core (i.e. 15 - 24) are also traced. To take the inter-core memory interferences into consideration, the cache-miss rates of other processing cores are also profiled as an indicator (i.e. 25). Our objective is to predict an application's execution time after future migration process, the current execution time should also be a very informative predictor to be traced (i.e. 26). For an Mapp, its execution time after migration will also be determined by how much resource the destination core can provide, therefore, performance counters (ID 1- 10 listed in Table I) of the destination core are also profiled as part of the predictors for Mapp.

Model based prediction approach is to discover the relation between the outcome variable and predictors by training traced data. Therefore, the traced data instances should be collected as input-output data pairs $(\mathbf{x}_i, y_i)$, where $\mathbf{x}_i$ and y_i are the observations for predictors and outcome variable, respectively. As we discussed above, $\mathbf{x}_i$ collected for Mapp consists of its performance indicators on source core (i.e. counters in Table II) and performance counters of the destination core, while y_i is the observation of the Mapp's execution time after the migration. Such $\mathbf{x}_i$ and y_i can be collected offline by

[1]To save the space, explanation of each counter is not included in this paper, which can be found in [9].

[2]Performance counters with ID 11-14 in Table I are application specific and will not be traced at system level.

conducting extensive testing of application deployment and migration on the system before it is actually in service.

With the collected data instances $(\mathbf{x}_i, y_i)$, our next step is to establish an analytical model to predict the Mapp's execution time after migration.

B. Offline Timing Prediction Model Training

Mathematically, modeling the relationship between an outcome variable and its predictors belongs to regression problem. Assuming the Mapp's execution time lies in the linear relation with its predictors, for an observation of the predictors $\mathbf{x}_*$, the corresponding execution time can be estimated using linear regression technique [7], i.e.:

$$f(\mathbf{x}_*) = \sum_{m=1}^{D} x_*(m)w(m) = \mathbf{x}_*^T\mathbf{w} \tag{1}$$

where $\mathbf{x}_* = [x_*(1), x_*(2), ..., x_*(D)]$, and $\mathbf{w} = [w(1), w(2), ..., w(D)]$ is a coefficient vector to be determined by training the collected observations.

Suppose the training data set consists of N collected data pairs $(\mathbf{x}_1, y_1)$, $(\mathbf{x}_2, y_2)$, ..., $(\mathbf{x}_N, y_N)$, the coefficient vector $\mathbf{w}$ can be obtained by minimizing the regression cost:

$$\|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2 + \lambda\|\mathbf{w}\|^2 \tag{2}$$

where $\mathbf{y}$ is a vector with $\mathbf{y} = [y_1, y_2, ..., y_N]^T$, $\mathbf{X}$ is matrix with $\mathbf{X} = [\mathbf{x}_1^T, \mathbf{x}_2^T, ..., \mathbf{x}_N^T]$, $\|\mathbf{w}\|$ indicates the L2-norm of vector $\mathbf{w}$, and λ is a regularization constant which imposes smoothness of coefficients [10].

However, linear model assumption is too restrictive and timing prediction performance may suffer. To explore the potential nonlinear models, kernel based regression techniques are commonly used and the key idea is to map the traced predictors to a high-dimensional space with the expectation that a non-linear relation between the predictors and the outcome variable can be well formulated [10].

There exist different kernel functions, but Gaussian kernel

$$\kappa(\mathbf{x}_i, \mathbf{x}_j) = \exp(-\gamma\|\mathbf{x}_i - \mathbf{x}_j\|^2) \tag{3}$$

which implies infinite dimension mapping, is widely used in regression analysis [11]. Among which, γ is a problem-specific parameter to indicate the covariance of Gaussian kernel, i.e. how far of a single training data can influence its surroundings [12].

With the kernel-based technique, given the training data set as pairs $(\mathbf{x}_i, \mathbf{y}_i)$, the unknown output corresponding to a new observation $\mathbf{x}_*$ can be estimated as [13]:

$$f(\mathbf{x}_*) = {\mathbf{k}_*}^T\boldsymbol{\alpha} \tag{4}$$

where $\mathbf{k}_* \in R^{N\times 1}$ and $\mathbf{k}_*(i) = \kappa(\mathbf{x}_*, \mathbf{x}_i)$ and vector $\boldsymbol{\alpha} \in R^{N\times 1}$ are the coefficients yet to be determined.

By training the collected data instances, the optimal $\boldsymbol{\alpha}$ can be obtained as [10]:

$$\boldsymbol{\alpha} = (\mathbf{K} + \lambda\mathbf{I})^{-1}\mathbf{y} \tag{5}$$

where $\mathbf{I}$ indicates an identity matrix, $\mathbf{K} \in R^{N\times N}$ is called the kernel matrix with $K_{i,j} = \kappa(\mathbf{x}_i, \mathbf{x}_j)$.

Solving formula (5) involves inverse of an $N \times N$ matrix, which is of $O(N^3)$ computation complexity and N is usually large. To avoid the heavy computational matrix inversion, recursive kernel-based regression method is often adopted instead [13]. Rather than directly re-calculating $(\mathbf{K} + \lambda\mathbf{I})^{-1}$, recursive kernel-based regression method recursively updates $(\mathbf{K} + \lambda\mathbf{I})^{-1}$ by training only one observations at a time.

Denoting $\mathbf{K}_n$ as the n-th iteration of $\mathbf{K} + \lambda\mathbf{I}$, the general idea of recursive regression method is to obtain $\mathbf{K}_n^{-1}$ from previous calculated $\mathbf{K}_{n-1}^{-1}$ with minimal extra effort. Recursive regression method trains one data instance at a time and the nth iteration is to train $\mathbf{x}_n$, according to the definition of $\mathbf{K}_n$, we have [13]:

$$\mathbf{K}_n = \begin{bmatrix} \mathbf{K}_{n-1} & \mathbf{b} \\ \mathbf{b}^T & d \end{bmatrix}$$

where $\mathbf{b} = \{\kappa(\mathbf{x}_n, \mathbf{x}_1), \kappa(\mathbf{x}_n, \mathbf{x}_2), ..., \kappa(\mathbf{x}_n, \mathbf{x}_{n-1})\}^T$, $d = \kappa(\mathbf{x}_n, \mathbf{x}_n) + \lambda$.

Following algebra theory, $\mathbf{K}_n^{-1}$ can be calculated as [13]:

$$\mathbf{K}_n^{-1} = \begin{bmatrix} \mathbf{K}_{n-1}^{-1} + g\mathbf{e}\mathbf{e}^T & -g\mathbf{e} \\ -g\mathbf{e}^T & g \end{bmatrix} \tag{6}$$

where $e = \mathbf{K}_{n-1}^{-1}\mathbf{b}$ and $g = 1/(d - \mathbf{b}^T\mathbf{e})$.

By applying formula (6) recursively on collected data instances $\mathbf{x}_1$, $\mathbf{x}_2$, ..., $\mathbf{x}_N$ one at a time, the finally obtained $\mathbf{K}_N^{-1}$ will be plugged in to formula (5) to get the $\boldsymbol{\alpha}$, which is used in formula (4) for timing prediction.

The above recursive regression method can be adopted to establish an initial prediction model by training the offline traced data instances. However, after the embedded system is in service, it may experience unprecedented environment change and hence new observations should be continuously collected and trained to keep the timing prediction model up-to-date. However, embedded systems have limited hardware resources, in the next section, we are to present our online model update strategy under storage and computing constrains.

IV. Online Timing Preidction Model Update

The initial prediction model and training data instances obtained during the offline stage are saved in the storage first. As embedded applications are usually periodic, after the system is in service, the new training data instances will be traced periodically to update the model at run-time. However, an embedded system's storage is limited, eventually, some training data instances must be removed before accepting new observations. However, removing different training data instances may impact the timing prediction performance in a different manner.

Formula (4) is to predict the execution time, which can also be viewed as a linear combination of $\kappa(\mathbf{x}_*, \mathbf{x}_i)$ and $\alpha(i)$. Larger $\alpha(i)$ indicates more contribution of the observed data instance $\mathbf{x}_i$ to the timing prediction. Therefore, a straightforward approach is to remove the data instance associated with minimal weight, i.e. the one with smallest absolute value of

$\alpha(i)$ [8]. To further reduce the approximation error introduced by the data removal, the adjusted minimal weight strategy is also proposed in [8], which is to remove the one with smallest value of $\alpha_i/[(\mathbf{K}_n^{-1})]_{ii}$, where $[(\mathbf{K}_n^{-1})]_{ii}$ is the ith diagonal element of $\mathbf{K}_n^{-1}$.

As illustrated in Fig. 1, after the storage limit M is reached, either minimal weight or adjusted minimal weight strategy will have to take the following two steps for each incoming data instance: first, calculate $\mathbf{K}_n^{-1}$ and α to determine which data pair to be removed; second, remove the selected data pair from the training set and recalculate the corresponding matrix $\mathbf{K'}_n^{-1}$ in order to use recursive regression method, i.e. formula (6) to train the next incoming data. We use a fast calculation algorithm presented in [11] to calculate $\mathbf{K'}_n^{-1}$ based on $\mathbf{K}_n^{-1}$. The time complexity of this calculation is of $O(M^2)$. Following the procedure in Fig. 1, training each incoming data instance with $O(M^2)$ time cost can keep the prediction model always up-to-date.

As we mentioned, embedded systems have limited resources and we should also minimize the computation overhead to the utmost. Suppose new data pairs come in order as $\{(\mathbf{x}'_1, y'_1), (\mathbf{x}'_2, y'_2), (\mathbf{x}'_3, y'_3), ..., (\mathbf{x}'_N, y'_N)\}$, following the above data pruning procedure, it is possible that $(\mathbf{x}'_1, y'_1)$ is first trained and added in the training data set but immediately replaced by $(\mathbf{x}'_2, y'_2)$, and then replaced by $(\mathbf{x}'_3, y'_3)$ and so on, until $(\mathbf{x}'_N, y'_N)$. If such case happens, the first $N-1$ data instances are trained first but then removed immediately from the training data set, therefore, the computing effort spent on these short lived data instances is wasted as it does not contribute to the model update at all.

To save the computation cost for these short lived data instances, our delayed model update approach is proposed. Instead of directly updating the model for each incoming data instance, we save these data instances to a replacement map first and delay the model update process until the map size reach a predefined limit. In a replacement map, the key is the index of the to-be-removed data instance in the training set and the value is the new observation to be added in.

The objective of using replacement map is to filter out short lived data instances before model update and hence avoid unnecessary computation overhead. The remaining question is, how to update such replacement map and filter out the incoming short lived data instances. Obviously, the above mentioned minimal weight and adjusted minimal weight approaches can be adopted, but with tremendous computation overhead. In following, we present a fast map update strategy instead.

As we know, the data instance selected for removal is expected to cause least effect on model training. Since redundant data is usually least informative and hence it should be the optimal candidate. Recapping the definition of Gaussian kernel function given in formula (3), a larger value of $\kappa(\mathbf{x}_i, \mathbf{x}_j)$ indicates $\mathbf{x}_i$ and $\mathbf{x}_j$ are more similar to each other, i.e. more redundancy between $\mathbf{x}_i$ and $\mathbf{x}_j$.

To quantify how much a data instance $\mathbf{x}_i$ can be represented by the rest in the training data set, we define a new metric $\theta(i)$ as follows:

$$\theta(i) = \max_{j \in \{1,...,M\} \wedge j \neq i} \kappa(\mathbf{x}_i, \mathbf{x}_j) + \frac{\sum_{j \in \{1,...,M\} \wedge j \neq i} \kappa(\mathbf{x}_i, \mathbf{x}_j)}{M-1} \tag{7}$$

The first term of the right hand side indicates the maximum similarity of $\mathbf{x}_i$ with a single instance and the second term indicates the average similarity of $\mathbf{x}_i$ with the other training instances. The data instance with the largest $\theta(i)$ is the least informative and hence will be replaced by the incoming one.

To simplify the representation, we introduce a new notation $\widetilde{\mathbf{x}_i}$, which indicates the valid training data instance at index i. Suppose the replacement map is denoted as Ω and the key list is $\texttt{key}(\Omega)$, if $i \in \texttt{key}(\Omega)$, $\widetilde{\mathbf{x}_i}$ is the new data instance with key value as i in the replacement map; if $i \notin \texttt{key}(\Omega)$, $\widetilde{\mathbf{x}_i}$ is ith data instance in the training data set, i.e. $\mathbf{x}_i$.

With the above notations, our proposed maximum redundancy based map update strategy for each incoming data $(\mathbf{x}'_*, y'_*)$ is illustrated in Algorithm 1. Among which, we first find the indexes of all the data instances (ψ) having the maximum redundant value (line 1). If any $k \in \psi$ is already set as a key in the replacement map, the corresponding value $\Omega(k)$ will be a short lived data instance and should be replaced by $(\mathbf{x}'_*, y'_*)$ (line 2); otherwise, add a new mapping entry to store $(\mathbf{x}'_*, y'_*)$ in the replacement map (line 6). Lines 8 - 11 are to update the array θ, which will be used as the input of the algorithm to process next incoming data instance.

ALGORITHM 1: Max_R_Map_Update(Ω,S,θ, $(\mathbf{x}'_*, y'_*)$)

```
1  find ψ = {i|argmax_{i∈{1,...,M}} θ(i)};
2  if ∃k ∈ ψ ∧ key(Ω) then
3  |  update Ω(k) = (x'_*, y'_*);
4  end
5  else
6  |  randomly select a k ∈ ψ and set Ω(k) = (x'_*, y'_*);
7  end
8  Update θ(k) using formula (7);
9  for  i ∈ {1...M} ∧ i ≠ k do
10 |  update θ(i) using formula (7);
11 end
12 return {Ω, θ}.
```

Though updating the replacement map may take $O(M)$ computation cost, short lived data instances will be removed from the map (line 2) before actually joining the training data set, which could save $O(M^2)$ time cost.

When the replacement map reaches predefined limit, we first remove all the data instances indexed by $\texttt{key}(\Omega)$ from training data set. After that, recursive regression method can be used to update the model by training the new instances stored as value in the replacement map Ω.

V. Discussion

Our proposed timing prediction strategy can also be applied to predict the execution times of Dapp, but with different

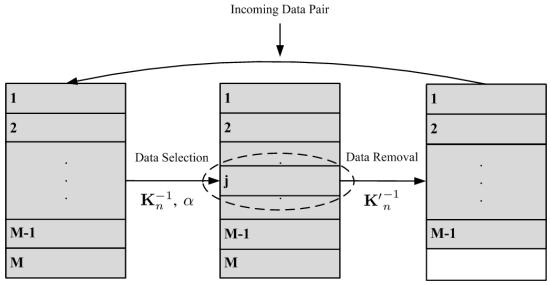


Fig. 1: Data Pruning Procedure

predictors traced in the training data instances. The predictors for Mapp have been explained in Section III. To predict the execution time of Dapp, the performance indicators of both the Mapp and targeted Dapp are traced as the training data.

VI. Evaluation

In this section, we evaluate our proposed strategy to predict application execution times.

A. Experimental Settings

Our testbed is configured with four Intel processing cores with 3MB cache each, 6 GB shared main memory and 500 GB solid state drive. The applications deployed on the testbed come from three commercially representative benchmark suites: MiBench [14], MediaBench [15] and SD-VBS [16]. The MiBench consists of selected applications representing six specific areas of embedded market: Automotive and Industrial Control, Consumer Devices, Office Automation, Networking, Security, and Telecommunications; MediaBench contains 19 applications selected from image processing, communications and digital signal processing; SD-VBS is a suite of diverse vision applications, which includes 9 applications and each has 3 different configurations and 5 distinct input testing data sets. Ubuntu operating system with the 3.13 kernel version is installed on the testbed and deployed applications are scheduled by CFS scheduler.

The training data instances for Mapp can be traced through the following steps:

1) Randomly select some of the above applications and deploy them on different cores, keep all of them running periodically on the deployed cores;
2) Randomly choose one application as the Mapp and a processing core other than the deployed one as the destination, trace the counters listed in Table II of the Mapp and the ones in Table I of the destination for one period as a $\mathbf{x}_i$;
3) Migrate the Mapp to the selected destination and then trace its execution time as a y_i.

The counters and execution times are traced using PERF [9], which is a performance analyzing tool in linux. By repeating the above steps and varying the input workload of selected applications, about 300 different data instances are collected.

A training set is used to discover the prediction model, which is to predict unseen data. Training data pairs have been learned and hence can not be claimed as unseen, to evaluate the performance of obtained prediction models, we have to use data instances not included in the training data sets. In addition, two hyper-parameters γ (in formula (3)) and λ (in formula 2) are problem specific, which need to be chosen by extensive experimental search. Therefore, we partition the collected data set into three sub sets: training data set (D_r), verification data set (D_v) and test data set (D_t). Among which, D_r, composed of 80% of the collected data, is used for model training; D_v, 10% of the data set, is utilized to iteratively search the optimal γ and λ; and D_t, the remaining 10% of the data set, is for performance evaluation.

B. Offline Timing Prediction Performance Evaluation

To quantify the prediction accuracy, the following metric is to measure the prediction error:

$$\texttt{Err} = \frac{|\texttt{Traced-Exe-Time - Predicted-Exe-Time}|}{\texttt{Traced-Exe-Time}}$$

In order to give an overall statistics of our prediction, we calculate the cumulative distribution function (CDF) of prediction error on the entire test data set D_t as follows:

$$\texttt{CDF}_{\texttt{Err}}(x) = P(\texttt{Err} < x)$$

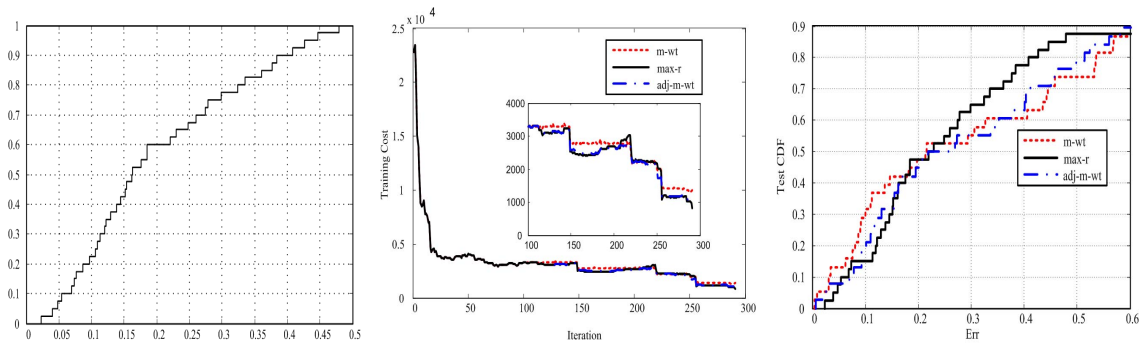
where $\texttt{CDF}_{\texttt{Err}}(x)$ represents the cumulative probability that the prediction error less than or equal to x.

The timing prediction model is trained and CDF of the corresponding prediction error is illustrated in Fig. 2a. According to Fig. 2a, we can see that over 60% of tested Mapps have prediction error less than 0.2, 90% of which are less than 0.45.

C. Online Timing Prediction Performance

To evaluate the performance of our proposed online modeling strategy, we assume the collected data instances arrive sequentially. In addition, the storage limit and replacement map size are set to 100 and 20, respectively. Therefore, the first 100 data instances will be trained one by one using recursive regression method. The subsequent ones will be filtered by the replacement map first and then trained recursively once the map is full. As we mentioned, the replacement map can be updated by three algorithms, i.e. minimal weight (m-wt) [8], adjusted minimal weight (adj-m-wt) [8] and maximum redundancy based algorithm implemented in Algorithm 1 (max-r). In our experiments, all these three algorithms are implemented for performance comparison.

We first train the model to predict the execution time of Mapp and calculate the training cost formula (2) on data set D_r. As illustrated in Fig. 2b, the overall trend of training cost keeps decreasing by processing more observations. Different algorithms will be applied to update the replacement map after the storage limit is reached, therefore, only after 100 iterations the three lines exhibit differently, but within a very small range. After training all the data instances in D_r, the prediction



(a) Offline Test CDF (b) Online Convergence of Cost (c) Online Test CDF

Fig. 2: Experimental Results

error of obtained models on test data set D_t is depicted in Fig. 2c, from which we can see that the our proposed max-r approach has very close prediction performance with max-wt and max-adj-wt algorithms. From Fig. 2b and Fig. 2c, we can conclude that, although max-r based algorithm has much lower computation cost, it still can achieve similar timing prediction performance with the other two compared approaches. It is worth pointing out that we also conducted experiments to validate the performance of our proposed strategy to predict the execution times of Dapp, however, the results are not presented due to page limitation.

VII. Conclusion

In this paper, we studied kernel regression based prediction approach to predict the system's timing change incurred by a given application migration process. We first investigated potential predictors and then proposed a two-stage approach to set up timing prediction models. The offline stage was used to train the initial model and online stage was to dynamically update the model to keep it up-to-date.

The experiments were conducted on Intel CPU and our next step is to extend the evaluation to AMD and ARM processors equipped with more processing cores. In addition, further improvement of our proposed regression strategies to achieve timing prediction accuracy and less computation complexity will also be part of the future work.

References

[1] A. Burns and R. I. Davis, "Mixed criticality systems: A review," Department of Computer Science, University of York, East Lansing, Michigan, Tech. Rep. MCC-1(b), February 2013.

[2] J. L. Herman, C. J. Kenna, M. S. Mollison, J. H. Anderson, and D. M. Johnson, "Rtos support for multicore mixed-criticality systems," in *2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium*, April 2012, pp. 197–208.

[3] C. S. Wong, I. Tan, R. D. Kumari, and F. Wey, "Towards achieving fairness in the linux scheduler," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, pp. 34–43, Jul. 2008.

[4] D. B. L. Benini and M. Milano, "Resource management policy handling multiple use-cases in mpsoc platforms using constraint programming," in *ICLP*, March 2008, pp. 470–484.

[5] P. Joseph, K. Vaswani, and M. J. Thazhuthaveetil, "A predictive performance model for superscalar processors," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2006, pp. 161–170.

[6] X. Zheng, P. Ravikumar, L. K. John, and A. Gerstlauer, "Learning-based analytical cross-platform performance prediction," in *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS), 2015 International Conference on*. IEEE, 2015, pp. 52–59.

[7] E. Alpaydin, *Introduction to Machine Learning. [Sl]*. The MIT Press, 2010.

[8] B. J. de Kruif and T. J. A. de Vries, "Pruning error minimization in least squares support vector machines," *IEEE Transactions on Neural Networks*, vol. 14, no. 3, pp. 696–702, May 2003.

[9] A. C. de Melo, "The new linuxperftools," in *Slides from Linux Kongress*, vol. 18, 2010.

[10] S. Van Vaerenbergh and I. Santamaría, *Online Regression with Kernels*. New York: Chapman and Hall/CRC, 2014, no. Machine Learning & Pattern Recognition Series, ch. 21, pp. 477–501.

[11] S. Van Vaerenbergh, I. Santamaría, W. Liu, and J. C. Príncipe, "Fixed-budget kernel recursive least-squares," in *Acoustics Speech and Signal Processing (ICASSP), 2010 IEEE International Conference on*. IEEE, 2010, pp. 1882–1885.

[12] A. Müller and S. Guido, *Introduction to Machine Learning with Python: A Guide for Data Scientists*. O'Reilly Media, 2016.

[13] S. V. Vaerenbergh, J. Via, and I. Santamaria, "A sliding-window kernel rls algorithm and its application to nonlinear channel identification," in *2006 IEEE International Conference on Acoustics Speech and Signal Processing Proceedings*, vol. 5, May 2006, pp. V–V.

[14] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, Dec 2001, pp. 3–14.

[15] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems," in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 30. Washington, DC, USA: IEEE Computer Society, 1997, pp. 330–335.

[16] S. K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, and M. B. Taylor, "Sd-vbs: The san diego vision benchmark suite," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, Oct 2009, pp. 55–64.