Journal of Systems Architecture 95 (2019) 47–54

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Run-time timing prediction for system reconfiguration on many-core embedded systems

Zheng Li[a,*], Shuibing He[b]

[a] Computer Science Program, School of Business, Stockton University, Galloway, NJ, USA
[b] School of Computing, Wuhan University, Wuhan, Hebei, China

ARTICLE INFO

Keywords:
Timing
Prediction
System reconfiguration
Real-Time
Many-core

ABSTRACT

Many-core embedded systems usually have real-time constrains, which may work in hostile environment and operate continuously without supervision. However, system execution mode change and hardware malfunction could alter deployed applications' response time and result in the violation of system's real-time constraints. To accommodate such incidents, run-time system reconfiguration, which invokes dynamic application migration, needs to be supported on many-core embedded systems. As different migration strategies will impact system's timing behaviors in different manners, it is vital to choose an appropriate one such that the system's timing performance after the migration is still acceptable. The focus of this research is to predict system's timing change induced by any migration strategy, which can be utilized to select the optimal migration strategy among all the possible choices. To be more specific, a two-stage timing prediction approach is investigated in this paper, where the offline stage is to train the initial model using historical data and the online stage is to fine tune the model at run-time. Extensive experiments have been conducted and the results validate the effectiveness of our proposed approach.

1. Introduction

With the advance of tremendous computing capacity and high degree of explicit parallelism, many-core processors, which integrate tens or possibly hundreds of cores on the same die, are being widely considered for embedded systems [4]. To improve the system resource utilization and reduce production cost, mixed-criticality design with both critical and non-critical applications sharing the same many-core platform, is also an increasing design trend for future embedded systems [3,27,28]. Critical applications, such as flight control applications, are crucial for the entire system and a deadline miss will result in catastrophic consequences, to ensure systems safety, hardware resources such as CPU cycles and memory channel/bank are reserved under the worst case consideration. Though resource reservation can guarantee applications' performance, it is cost-ineffective. To improve resource utilization, non-critical applications are usually designed to compete the available resources and run simultaneously to provide best-effort service [11]. Non-critical applications are not safety-critical, but their performance degradation will impact system's quality-of-service (QoS) [19,20].

In traditional embedded system design, these applications are statically mapped to processing cores and all the possible use cases are explored to ensure the desired QoS can be achieved [14]. However, embedded systems may work in a continuously changing environment, hence, testing and verifying all the possibilities is time and effort unaffordable. Event if all the use-cases can be explored, if predefined cases do not match the real-world execution scenarios, the system may still suffer performance degradation.

In addition to execution scenario changes, hardware malfunction is another challenge on many-core systems. Due to technology advances, more and more cores are being integrated into a single chip. Recent study reveals that the core wear-out rate keeps increasing due to the ever-shrinking feature size and growing transistor count integrated on the chip [17]. In case of core wear-out, system must be reconfigured by migrating applications on faulty cores to functional ones in order to continue the service.

Embedded systems are expected to keep running without stopping, in order to accommodate execution scenario changes and core malfunction, application migration should be conducted at run-time. However, different migration strategies may affect the system's timing behaviors in different manners. But if such timing changes can be foreseen, upon which we are able to choose an appropriate migration strategy to meet system requirements after the reconfiguration.

Application timing behaviors are determined by various factors and heavily influenced by scheduling policies. As Operating systems based on Linux kernel are widely used in embedded systems such as consumer electronics, industrial automation and spacecraft flight [16], in this

* Corresponding author.
E-mail addresses: zheng.li@stockton.edu (Z. Li), heshuibing@whu.edu.cn (S. He).

https://doi.org/10.1016/j.sysarc.2019.03.004
Received 10 October 2018; Received in revised form 12 March 2019; Accepted 14 March 2019
Available online 14 March 2019

paper, we adopt the default scheduler in Linux kernel since 2.6.23 release, i.e. the Completely Fair Scheduler (CFS) which gives all processes fair chance to execute on processors [25] to schedule our non-critical applications.

Theoretically, the exact timing impacts induced by application migration can only be tracked after the migration process is completed, but some simple facts reveal that which may be predicted by exploring potential predictors. For instance, under the same scheduling policy, a computation-intensive task migrated to a less busy core usually can finish earlier. Inspired by these observations, in this paper, we are to explore the predictors and establish an analytical model to 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 service are guaranteed, the focus of this paper is to predict the timing of non-critical applications which simultaneously run on shared many-core processors scheduled by CFS. It's worth pointing out that application migration overhead is not considered in this paper, as recent study reveals such overhead is insignificant for most real-time applications [2,26].

The rest of the paper is organized as follows. We first summarize the related work in Section 2. An offline timing prediction model is presented in Section 3. To keep the timing prediction mode always up-to-date at run-time, our proposed online model update strategy is studied in Section 3.3. Experimental settings and results are discussed in Section 4. Finally we conclude our work in Section 5.

2. Related work

As many-core platform are being used in embedded systems, extensive research has been done on how to statically deploy applications on a many-core platform with various optimization goals. Chou [5] analyzed the impact of communication overhead on applications mapped to a many-core platform. In order to minimize on-chip communication workload, applications to many-core platforms mapping was formulated to an integer linear programming problem. Derin [8] discussed another optimal mapping strategy with the objective of minimizing applications' total execution time.

To achieve system performance optimization in case of execution scenario changes, Benini [14] proposed a semi-static approach to compute and store the application allocations and scheduling solutions for all possible use-cases offline. When system use case changes, applications will be migrated dynamically based on the stored solutions. However, the time and space complexity of this semi-static approach is exponential to the number of applications and cores. In addition, if the predefined use cases do not match the real-world execution scenarios, this strategy may suffer performance degradation. To overcome such shortcoming, application migration strategy should be determined dynamically instead.

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 [13] applied regression-based approaches for performance prediction. In [12], Ipek utilized artifical neutral network to construct a processor's performance prediction model instead of regression. To predict a workload on another hardware platform with different underlying architecture, Zheng [29] recently developed an offline cross-platform prediction approach, which predicted applications' performance on a not-yet-available target platform by running it on an available platform but with different underlying architecture. Zheng's approach assumed that application's performance lied in a linear relationship with the predictors and hence the prediction accuracy may suffer if such assumption was not invalid.

To explore the potential non-linear relation between the target variable and its predictors, kernel-based regression methods are commonly utilized [1]. Considering training data instances arrived sequentially, in order to minimize the computation overhead, online kernel-based regression techniques which learn one data instance at a time were introduced in [22]. As computer systems usually have limited storage, some existing data instances need to be pruned before the new one comes in. To address the data pruning problem, a sliding-window method which always throws away the oldest data instance was developed in [21]. To minimize the prediction error by data removal, an intelligent data pruning technique was further proposed in [6]. To predict timing behavior changes of the migrated application invoked by system reconfiguration on many-core embedded systems, our preliminary work was presented in [30]. In this paper, we extend the topic and apply kernel based regression techniques to predict the timing impact of all involved applications. In addition, extensive experimental results from a real-world many-core testing bed are also provided to validate our proposed approach.

3. Timing prediction for dynamic application migration

Consider an application migration scenario on a many-core platform depicted in Fig. 1, the application (app1) is migrated from core 1 to core N. For this migration process, core 1 and core N are the source and destination, respectively. We classify these applications involved in the migration process into three categories: source application (Sapp), i.e. the application remains on the source core before and after the migration; migrated application (Mapp), i.e. the one to be migrated; and destination application (Dapp), i.e. the one remains on the destination core before and after the migration. In this example, app1, app2 and app3 are Mapp, Sapp and Dapp, respectively.

Clearly an application migration process will alter the timing behaviors of Mapp, Sapp as well as Dapp, and hence we need to estimate the impact of all involved applications as to determine if such migration strategy is feasible. In the following, we first focus on predicting the timing change of Mapp, and then extend the discussion to Sapp and Dapp.

We adopt model-based strategy to predict the response time of impacted applications, which consists of two stages: offline model training and online model update. The first stage is to train an initial timing prediction model using historical data set; after the system is actually in service, online update stage is to dynamically tune the prediction model to match the current system status.

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 response time and trace the related data to be used for model training.

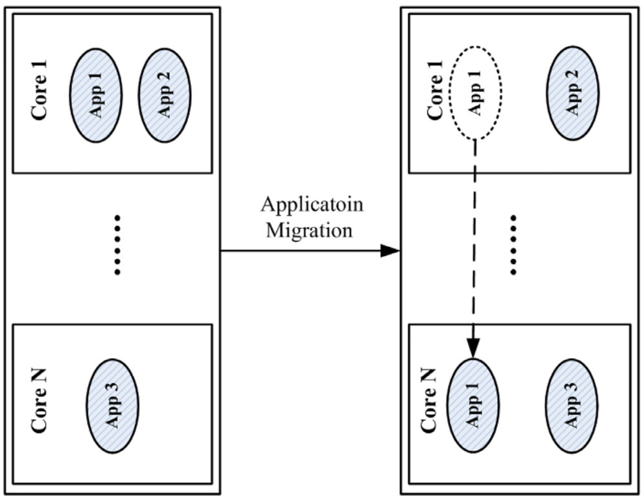

Fig. 1. Application Migration Scenario.

3.1. Training data profiling

Though an application's response time is impacted by various factors, a few simple facts reveal that it may still be predictable. For example, an application with more instructions may take longer time if deployed on the same hardware platform; while for the same computation-intensive application, it may run faster on a high-speed processor than on a low-speed one. In addition, if multiple applications deployed on the same processing unit, their response times usually will be extended. Things can be more complicated on many-core systems. All the cores may have dedicated processing units and caches but share the main memory, therefore, memory-access interferences from other cores may also impact an application's timing behaviors as well. These observations imply that, in order to accurately predict an application' response time, the status of both the target application and hardware platform need to be taken into consideration.

For human beings, their health conditions are usually indicted by heart beat, blood pressure, temperature and etc. Analogously, the running status of applications and computer systems can be reflected by indicators such as CPU utilization, cache miss rate and etc. Fortunately, a list of performance counters (shown in Table 1) are implemented in which can be profiled to track the status of a computing system as well as the deployed applications [7]. To save the space, the explanation of each counter is not included in this paper, but can be found in [7].

These counters tracked at system level indicate how well the system is performing; while counters traced at application level reveal the performance of the target application (counters 11–14 in Table 1 are application specific and can only be traced at system level). For instance, cache-references of an application indicates the number of cache references invoked by this application only; while the cache-references counter measured for a processing core indicates total cache references invoked by all the applications deployed on this core. Since an application's response time is heavily impacted by the hardware platform, we trace the performance counters for the target application and its processing core as potential prediction indicators.

Table 2 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 cores are also profiled as an indicator (i.e. 25). Our objective is to predict an application's response time after future migration process, therefore, the current response time should also be traced as a very informative predictor (i.e. 26). In addition, an application's response 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 1) of the destination core should also be traced as part of the predictors for Mapp.

As we mentioned above, model based prediction approach is to discover the relation between the outcome variable and predictors by training the traced data. More specifically, 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 variables, respectively. To predict the execution time of Mapp after migration, $\mathbf{x}_i$ collected for Mapp consists of its performance counters (i.e. counters in Table 2) on the source and destination core, while y_i is the observation of the Mapp's response time after the migration.

Such data pairs ($\mathbf{x}_i$, y_i) can be collected offline by 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 response time after migration by training these traced data pairs.

Table 1
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

Table 2
Application performance indicator.

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

3.2. Offline timing prediction model training

Mathematically, modeling the relationship between an outcome variable and its predictors belongs to regression problem. Assuming the Mapp's response time lies in the linear relation with its predictors, for a specific observation $\mathbf{x}_* = [x_*(1), x_*(2), \dots, x_*(D)]^T$, the corresponding response time can be estimated using linear regression as [1]:

$$f(\mathbf{x}_*) = w(0) + w(1)x_*(1) + \dots + w(D)x_*(D) \\ = \sum_{i=0}^{D} x_*(i)w(i) = \mathbf{x}_*^T\mathbf{w} \tag{1}$$

where $\mathbf{w} = [w(0), w(1), w(2), \dots, w(D)]^T$ is a coefficient vector to be determined, D denotes the number of predictors and $x_*(0) = 1$.

The appropriate selection of $\mathbf{w}$ will make the linear model fits the training data as close as possible. 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) with y_i as the actual response time of targeted application i, the cost function can be calculated as [9]:

$$E(\mathbf{w}) = \|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2 + \lambda\|\mathbf{w}\|^2 \tag{2}$$

where $\mathbf{y}$ is a vector with $\mathbf{y} = [y_1, y_2, \dots, y_N]^T$, $\mathbf{X}$ is a matrix with $\mathbf{X} = [\mathbf{x}_1^T, \mathbf{x}_2^T, \dots, \mathbf{x}_N^T]$, $\|\mathbf{w}\|$ indicates the L2-norm of vector $\mathbf{w}$, and λ is a regularization parameter which imposes smoothness of coefficients [9,22].

The smaller $E(\mathbf{w})$ indicates the better fit and gradient decent algorithm [1] can be used to search the best coefficient $\mathbf{w}$ to minimizes $E(\mathbf{w})$.

Though linear regression technique is easy to apply, it usually exhibits high bias and underfits the truth target due to the strict linear relationship assumption between predictors and the target [1]. Therefore, to explore the potential nonlinear models, kernel based regression techniques are adopted in this study. The key idea is to map the traced predictors to a high-dimensional space with the expectation that a nonlinear relation between the predictors and the outcome variable can be well formulated [22].

There exists 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 commonly used in regression analysis [23]. 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 [18].

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 [21]:

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

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 [22]:

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

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 [21]. 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 data pair at a time.

Denoting $\mathbf{K}_n$ as the nth 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 $\mathbf{x}_n$, therefore, according to the definition of $\mathbf{K}_n$, we have [21]:

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

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

Following algebra theory, $\mathbf{K}_n^{-1}$ can be calculated as [21]:

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

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 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, embedded systems may experience unprecedented environment change and hence new data pairs should be continuously observed and trained to keep the timing prediction model up-to-date. However, embedded systems usually have very limited hardware resources, in the next section, we are to present our online model update strategy under various resource constraints.

3.3. Online timing prediction model update

Our proposed online model update strategy can be illustrated in Fig. 2. The initial 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.

If a system has unlimited storage, new coming data instances can be saved in the storage and trained using formula (5) and (6) to update the model. However, an embedded system's storage is limited, eventually, some existing training data instance will be removed before accepting the new one. However, removing different training data instances may impact the timing prediction performance in a different manner.

Formula (4) is to predict the response 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.

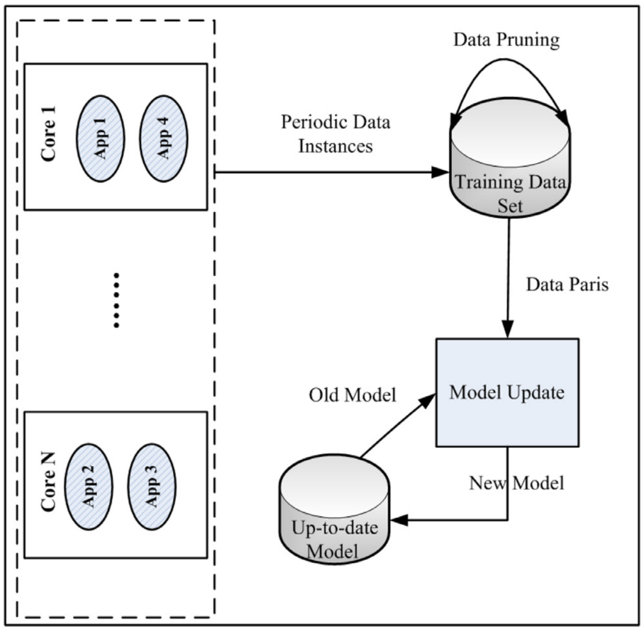

Fig. 2. Online timing model update.

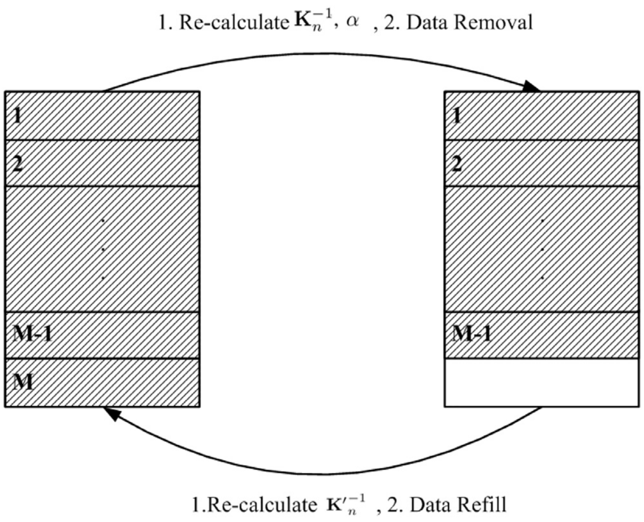
1. Re-calculate $\mathbf{K}_n^{-1}$, α , 2. Data Removal
1
2
M-1
M
1
2
M-1
1.Re-calculate $\mathbf{K'}_n^{-1}$, 2. Data Refill

Fig. 3. Data pruning procedure.

Therefore, a straightforward approach is to remove the data instance associated with minimal weight, i.e. the one with smallest absolute value of $\alpha(i)$ [6]. To further reduce the approximation error introduced by the data removal, the minimal adjusted weight strategy is also proposed in [6], 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. 3, after the storage limit (M) is reached, minimal weight and adjusted minimal weight strategy will have to take the following two steps for each incoming data instance: first, calculate $\mathbf{K}_n^{-1}$ and $\boldsymbol{\alpha}$ to determine which data pair to be removed; second, remove the selected data pair from the training set. When new data instance comes in, the corresponding matrix $\mathbf{K'}_n^{-1}$ needs to be re-calculated in order to use recursive regression method, i.e. formula (6) to train the next incoming data. We adopt a fast calculation algorithm presented in [23] 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. 3, training each incoming data instance with $O(M^2)$ time cost can keep the prediction model up-to-date.

However, as we mentioned, embedded systems have limited resources and hence we should minimize the computation overhead to the utmost.

Suppose new data pairs come in order as $\{(\mathbf{x}'_1, y'_1), (\mathbf{x}'_2, y'_2), (\mathbf{x}'_3, y'_3), \ldots, (\mathbf{x}'_N, y'_N)\}$, 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 data instances is wasted as it does not update the prediction model.

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 index of the to-be-removed data instance in the training set and the value is new incoming data pair to be added in. An example of replacement map is illustrated in Example 1.

Example 1. Suppose we have a replacement map with key-value pairs as $\Omega = [1 \to (\mathbf{x}'_1, y'_1), 3 \to (\mathbf{x}'_2, y'_2)]$, then the first and third instances in the training data set are to be replaced by the new data pairs $(\mathbf{x}'_1, y'_1)$ and $(\mathbf{x}'_2, y'_2)$, respectively.

The objective of using the replacement map is to filter out the short lived data instances before model update and avoid unnecessary computation overhead. The remaining question is, how to create such replacement map and filter out the incoming short lived data instances.

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 it is 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:

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

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

Suppose the replacement map is denoted as Ω and the key list is `key`(Ω), our proposed maximum redundancy based map update strategy for each incoming data $(\mathbf{x}'_*, y'_*)$ can be illustrated in Algorithm 1. Among

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

```
find $\psi = \{i | \text{argmax}_{i\in\{1,\ldots,M\}} \theta(i)\}$;
if $\exists k \in \psi \wedge key(\Omega)$ then
    update $\Omega(k) = (\mathbf{x}'_*, y'_*)$;
end
else
    randomly select a $k \in \psi$ and set $\Omega(k) = (\mathbf{x}'_*, y'_*)$;
end
Update $\theta(k)$ using formula (7);
for $i \in \{1 \ldots M\} \wedge i \neq k$ do
    update $\theta(i)$ using formula (7);
end
return $\{\Omega, \boldsymbol{\theta}\}$.
```

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 Ω(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.

Using Algorithm 1 to update the replacement map takes $O(M)$ computation cost, which will remove short lived data instances from the map (line 3) before actually joining the training data set, and hence it could save $O(M^2)$ time cost used for model update on these short lived data instances.

When the replacement map reaches predefined limit, we first remove all the data instances indexed by `key`(Ω) 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.

It's worth pointing out that, if the replacement map size M is super large, a waiting time threshold should be enforced in order to keep the model updated in time.

3.4. Timing prediction for Sapp and Dapp

In the above section, our proposed timing prediction strategy was detailed to predict the response time of Mapp after migration. However, the migration process not only impacts the migrated application, but also will alter the timing behaviors of applications deployed on the source and destination core, i.e. Sapp and Dapp. hence, we extend the discussion to predict response times of Sapp and Dapp after migration.

The timing change of migration process is incurred by moving Mapp from source to destination and hence the impact of Mapp should be well traced to make the prediction. Therefore, to predict the response time of Sapp after migration, the performance counters of Mapp, Sapp as well as source core will be traced as the training data set. While for Dapp, the performance counters of Mapp, Dapp, source and destination core will be traced instead.

In the next section, we are to set up a many-core experimental platform to evaluate the performance of our proposed timing prediction approaches.

4. Evaluation

4.1. Experimental setting

To evaluate our proposed timing prediction approach for Sapp, Mapp and Dapp, our testbed is configured with 20-core platform with 16GB shared main memory. Applications selected from three different benchmark suites: MiBench [10], MediaBench [15] and SD-VBS [24] are deployed. Among which, 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 applications from image processing, communications and digital signal processing; and SD-VBS is a suite of diverse vision applications.

The training data instances for Mapp are collected 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 2 of the Mapp and counters in Table 1 of the destination for one period as a $\mathbf{x}_i$;
3. Migrate the Mapp to selected destination and then trace its response time as a y_i;
4. Repeat the steps 1–3 by varying the input workload and selected applications.

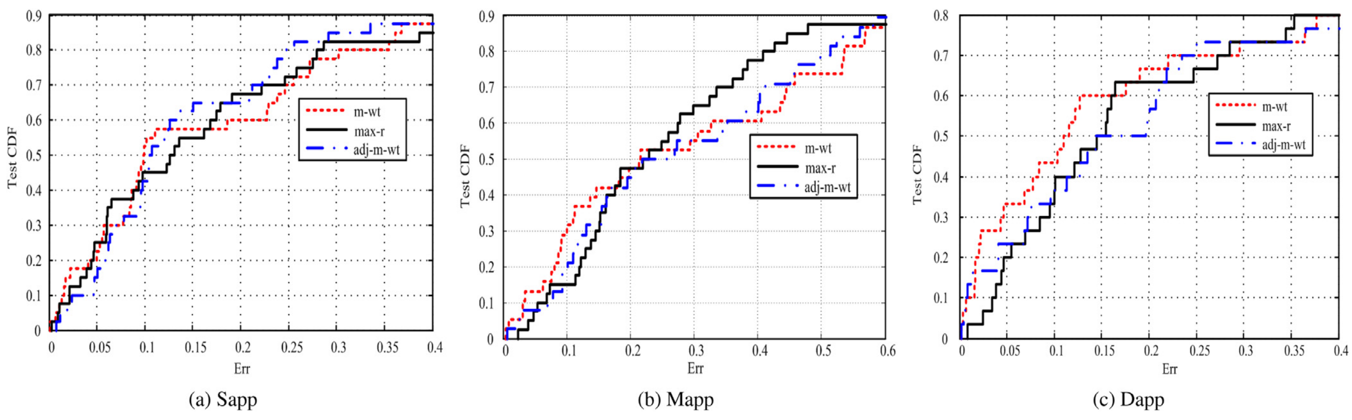

Fig. 4. Test CDF with limited storage.

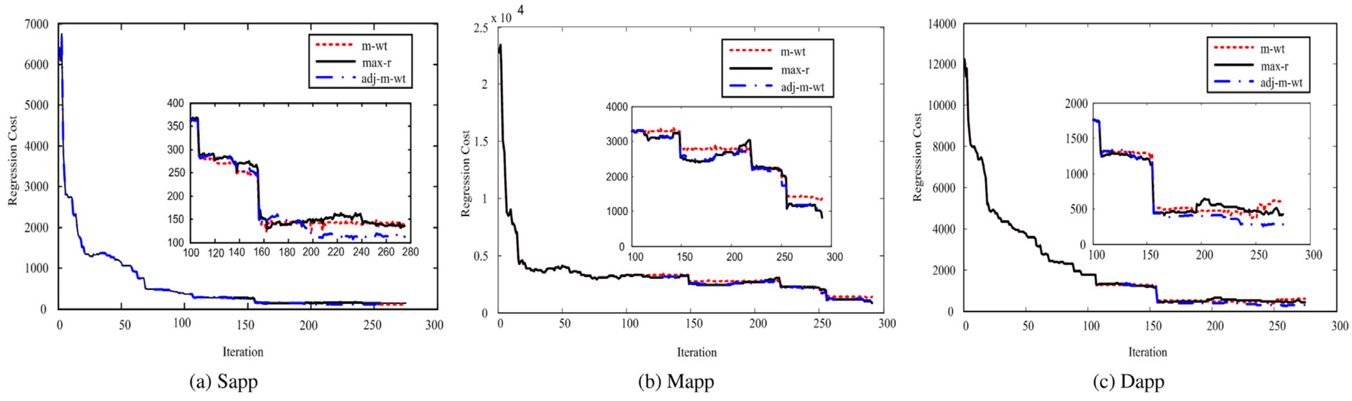

Fig. 5. Convergence of regression cost.

The counters and response times are traced using PERF [7], which is a performance analyzing tool in linux. The similar steps are adopted to collect corresponding $\mathbf{x}_i$ and y_i for Sapp and Dapp, respectively.

As we know, training data sets are used to discover the prediction model and then the obtained model will be applied to predict unseen data. Since training data instances have been learned, they can not be claimed as unseen. In order to evaluate the performance of obtained prediction models, we have to use data instances not included in the training data sets. Therefore, we partition the collected data set into two sub sets: training data set (D_r) and test data set (D_t). Among which, D_r, composed of 80% of the collected data, is used for model training; D_t, the remaining 20% of the data set, is for performance evaluation.

It is worth mentioning that the two free parameters γ (in formula (3)) and the λ (in formula 2) are problem specific, we conduct extensive experimental search and set $\gamma = 3$ and $\lambda = 0.01$.

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

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

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

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

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

4.2. Timing prediction evaluation with limited storage

To mimic the real-world scenario, we set the storage limit and replacement map size to 100 and 20 data instances, respectively. In addition, the first 100 data instances are assumed to be collected offline and trained using recursive regression method. The remaining training data are assumed to arrive sequentially when the system is actually in use, since the storage is already full, these subsequent instances are 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) [6], adjusted minimal weight (adj-m-wt) [6] and maximum redundancy based algorithm implemented in Algorithm 1 (max-r), but with different time complexity. In our experiments, all these three algorithms are implemented for performance comparison.

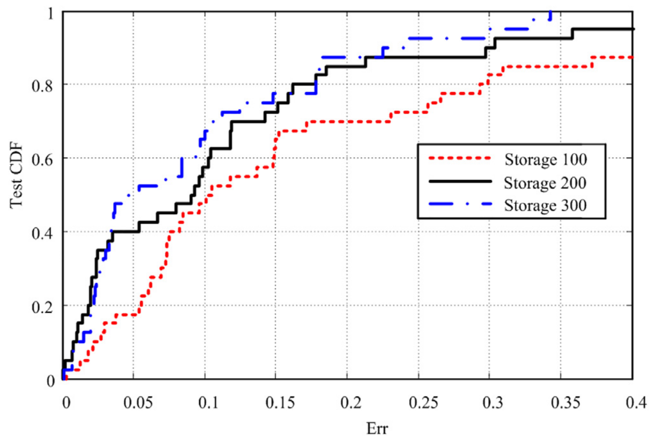

Fig. 6. Impact of different storage capacity for Sapp.

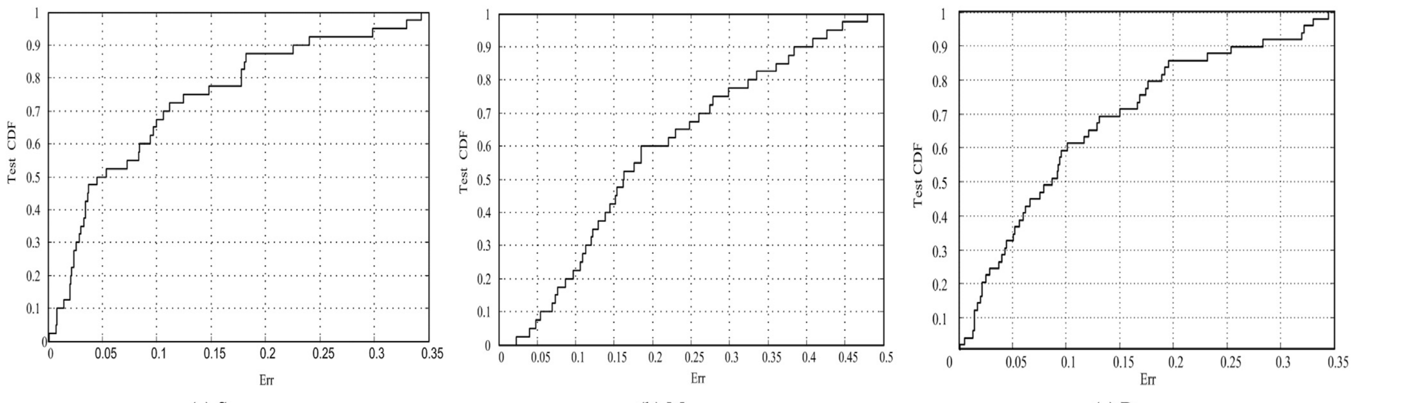

Fig. 7. Test CDF with unlimited storage.

We first train the model to predict the response time of Sapp and calculate the regression cost (i.e.formula (2)) on data set D_r. As illustrated in Fig. 5a, the overall trend of cost keeps decreasing with more data instances are trained. 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 error of obtained models on test data set D_t is depicted in Fig. 4a, from which we can see that the our proposed max-r approach has very closed prediction performance with max-wt and max-adj-wt algorithms. The similar observations are found by analyzing the experimental results for predicting the response times of Mapp and Dapp. From Figs. 5 and 6, we can conclude that, although max-r based algorithm has lower computation cost, it still can achieve similar timing prediction performance with the other two compared approaches.

To evaluate the impact of storage limit on our proposed timing prediction strategy, we vary the storage capacity from 100 to 300 data instances and re-train the prediction model for Sapp using maximum redundancy based algorithm (max-r), the results are illustrated in Fig. 6. As we expect, larger storage capacity results in more accurate model. According to Fig. 6, the prediction error is reduced by increasing the storage capacity from 100 to 200, which is further decreased when the capacity is increased to 300.

4.3. *Timing prediction evaluation with unlimited storage*

In this subsection, we assume the system has unlimited storage capacity to further evaluate our proposed prediction techniques. Due to unlimited storage, all the data instances collected online and offline are saved in the storage and trained using recursive regression method.

The response time prediction models of Sapp, Mapp and Dapp are trained and the CDF of corresponding prediction errors are illustrated in Fig. 7. According to Fig. 7a, we can see that over 60% of tested Sapps have prediction error less than 0.1, 70% of which are less than 0.15 and 90% of which are within 0.3. The timing prediction performance on Dapp is similar with that of Sapp, as shown in Fig. 7c, around 90% of tested Dapps with prediction error less than 0.3. The performance on predicting Mapps' response times is less inspiring, but still can achieve that over 75% of tested applications with prediction error less than 0.3 and 60% of which are less than 0.2 (Fig. 7b).

It is worth pointing out that this is an ideal case as the storage in embedded systems is always limited. However, these results can be used to evaluate the performance impact induced by storage limitation, which will help the system designer to allocate appropriate storage capacity without significant prediction performance degradation.

5. 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 to be traced as training data instances 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 for dynamic model update. To further reduce the time complexity of online model update, a maximum redundancy based replacement map was developed to filter short lived data instances and hence avoid unnecessary computation overhead. To evaluate the performance of our proposed approach, we conducted experiments by deploying multiple benchmarks including MIBENCH, MEDIABENCH and VD-VBS on a 20-core platform. Experimental results validate the effectiveness of our proposed approach.

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

References

[1] E. Alpaydin, Introduction to Machine Learning. [Sl], The MIT Press, 2010.
[2] J. Balasubramanian, A. Gokhale, A. Dubey, F. Wolf, C. Lu, C. Gill, D. Schmidt, Middleware for resource-aware deployment and configuration of fault-tolerant real-time systems, in: 2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium, 2010, pp. 69–78.
[3] J. Barhorst, T. Belote, P. Binns, J. Hoffman, J. Paunicka, P. Sarathy, J. Scoredos, P. Stanfill, D. Stuart, R. Urzi, A research agenda for mixed-criticality systems, Cyber-Physical Systems Week, 2009.
[4] A. Burns, R.I. Davis, Mixed criticality systems: a review, Technical Report MCC-1(b), Department of Computer Science, University of York, East Lansing, Michigan, 2013.
[5] C.-L. Chou, R. Marculescu, Contention-aware application mapping for network-on-chip communication architectures, in: Proc. of IEEE International Conference on Computer Design, in: ICCD, 2008, pp. 164–169.
[6] B.J. de Kruif, T.J.A. de Vries, Pruning error minimization in least squares support vector machines, IEEE Trans. Neural Netw. 14 (3) (2003) 696–702.
[7] A.C. de Melo, The new linuxperftools, Slides from Linux Kongress, 18, 2010.
[8] O. Derin, D. Kabakci, L. Fiorin, Online task remapping strategies for fault-tolerant network-on-chip multiprocessors, in: Proc. of 5th IEEE/ACM International Symposium on Networks on Chip, in: NoCS, 2011, pp. 129–136.
[9] J. Friedman, T. Hastie, R. Tibshirani, The elements of statistical learning, 1, Springer series in statistics Springer, Berlin, 2001.
[10] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, 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), 2001, pp. 3–14.
[11] J.L. Herman, C.J. Kenna, M.S. Mollison, J.H. Anderson, D.M. Johnson, Rtos support for multicore mixed-criticality systems, in: 2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium, 2012, pp. 197–208.
[12] E. İpek, S.A. McKee, R. Caruana, B.R. de Supinski, M. Schulz, Efficiently Exploring Architectural Design Spaces via Predictive Modeling, 40, ACM, 2006.

[13] P. Joseph, K. Vaswani, 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.
[14] D.B. L. Benini, M. Milano, Resource management policy handling multiple use-cases in mpsoc platforms using constraint programming, in: ICLP, 2008, pp. 470–484.
[15] C. Lee, M. Potkonjak, 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, in: MICRO 30, IEEE Computer Society, Washington, DC, USA, 1997, pp. 330–335.
[16] H. Leppinen, Current use of linux in spacecraft flight software, IEEE Aerosp. Electron. Syst. Mag. 32 (10) (2017) 4–13.
[17] Z. Li, F. Lockom, S. Ren, Maintaining real-time application timing similarity for defect-tolerant noc-based many-core systems, ACM Trans. Embed. Comput. Syst. (TECS) 13 (2s) (2014) 64.
[18] A. Müller, S. Guido, Introduction to Machine Learning with Python: a Guide for Data Scientists, O'Reilly Media, 2016.
[19] R.M. Pathan, Improving the quality-of-service for scheduling mixed-criticality systems on multiprocessors, in: M. Bertogna (Ed.), 29th Euromicro Conference on Real-Time Systems (ECRTS 2017), volume 76 of Leibniz International Proceedings in Informatics (LIPIcs), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2017, pp. 19:1–19:22.
[20] H. Su, D. Zhu, An elastic mixed-criticality task model and its scheduling algorithm, in: 2013 Design, Automation Test in Europe Conference Exhibition (DATE), 2013, pp. 147–152, doi:10.7873/DATE.2013.043.
[21] S.V. Vaerenbergh, J. Via, 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, 5, 2006. V–V.
[22] S. Van Vaerenbergh, I. Santamaría, Online Regression with Kernels, Machine Learning & Pattern Recognition Series, Chapman and Hall/CRC, New York, pp. 477–501.
[23] S. Van Vaerenbergh, I. Santamaría, W. Liu, 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.
[24] S.K. Venkata, I. Ahn, D. Jeon, A. Gupta, C. Louie, S. Garcia, S. Belongie, M.B. Taylor, Sd-vbs: The san diego vision benchmark suite, in: 2009 IEEE International Symposium on Workload Characterization (IISWC), 2009, pp. 55–64.
[25] C.S. Wong, I. Tan, R.D. Kumari, F. Wey, Towards achieving fairness in the linux scheduler, SIGOPS Oper. Syst. Rev. 42 (5) (2008) 34–43.
[26] Y. Zhang, C. Gill, C. Lu, Real-time performance and middleware for multiprocessor and multicore linux platforms, in: 2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, 2009, pp. 437–446.
[27] Q. Zhao, Z. Gu, H. Zeng, Design optimization for autosar models with preemption thresholds and mixed-criticality scheduling, J. Syst. Archit. Embed. Syst.Des. 72 (2017) 61–68.
[28] Q. Zhao, Z. Gu, H. Zeng, N. Zheng, Schedulability analysis and stack size minimization with preemption thresholds and mixed-criticality scheduling, J. Syst. Archit. Embed. Syst.Des. 83 (2017) 57–74.
[29] X. Zheng, P. Ravikumar, L.K. John, 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.
[30] H.W. Zheng Li, S. He, Timing prediction for dynamic application migration on multi-core embedded systems, in: The 4th IEEE International Conference on High Performance and Smart Computing, 2018.

Zheng Li received the B.S. degree in Computer Science and M.S. degree in Communication and Information System from University of Electronic Science and Technology of China, and Ph.D. degree from Illinois Institute of Technology, in USA. He is currently an assistant professor of Computer Science at Stockton University. His research interests include real-time system design, many-core computing and recongurable computing.

Shuibing He received the Ph.D. degree in computer science and technology from Huazhong University of Science and Technology, in 2009. He is currently an associate professor at the School of Computer Science, Wuhan University. His research areas include parallel I/O systems, file and storage systems, high-performance and distributed computing.