2017 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC)

Prediction Based Run-Time Reconfiguration on Many-core Embedded Systems

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

Shuibing He
School of Computing
Wuhan University
Wuhan, China

Li Wang
LinkedIn Corporation
Sunnyvale, CA

***Abstract*—This paper studies prediction based run-time system reconfiguration strategy to tolerate environment change and hardware malfunction on many-core embedded systems. System reconfiguration will invoke application migration, which may significantly impact system's timing behaviors, therefore, it is vital important to select an appropriate migration strategy after which the system's performance is still acceptable. The essence of our prediction based approach is to pre-estimate the impact of possible migration strategies and upon which to choose the optimal one. Our proposed approach includes data profiling, model training and execution time prediction phases. The initial data profiling and model training are conducted in the design stage, and will be continuously updated as time goes on after the system is in use. When system reconfiguration is invoked, the most recently trained models will be used for prediction at run-time. Extensive experiments have been set up by running multiple benchmarks on a four-core hardware platform and experimental results evaluate and validate our proposed approach.**

***Index Terms*—System Reconfiguration, Performance Counters, Application Migration, Timing Prediction**

I. Introduction

Embedded systems are widely used in the consumer, automotive, commercial and military industries. According to the white paper from GrammaTech, it is expected that the global market of embedded systems to exceed over 190 billion dollars by 2018. Such embedded systems integrated with a fixed set of applications are usually considered to be used in predictable environments [1]. In addition, as technology advances, many-core processors are being adopted in embedded systems due to the tremendous computing capacity and high degree of explicit parallelism. In traditional embedded system design, applications are statically deployed and fine-tuned on different processing cores to meet the desired system specification. However, the operating environments sometimes may change in unpredictable manners. If such modified environment changes are neglected, the systems may suffer degraded performance. To accommodate these unknown changes, traditionally, for the sake of performance guarantee, the resources are reserved for deployed applications under the consideration of extremely worst case cases. However, if these extreme cases not always happen, the whole system will be underutilized [2].

To overcome these shortcomings, run-time system reconfiguration which supports dynamic application migration can optimize the resource management and improve system's performance. If some application happens to workload burst due to unprecedented environment change and the provisioned resources are not sufficient, migrating such application to an underutilized processing core can keep the system still running with desired performance.

Besides the unknown upcoming environment change, hardware malfunctions will also result in the application re-deployment in the system. Recent study reveals that multi-core processors are being used in embedded systems while the core wear-out rate keeps increasing due to the ever-shrinking semiconductor process size [3], [4], [5]. Therefore, core malfunction will not be a rare case any longer and run-time system reconfiguration has to be implemented for reliability purpose.

However, migrating applications to a new host will not only change timing behaviors of the migrated application, but also alter the performance of all host applications resident on the destination core. In addition, the running status of different cores are varied, different application migration strategy may impact system's performance in different manners. Therefore, it is essential to pre-estimate the impact incurred by the migration process and upon on which to choose an appropriate destination cores such that the system's timing change is acceptable.

The exact timing change only can be obtained after the migration process, but some basic phenomenons give us the intuition that there may exist potential relationships between the performance of an application before and after the migration. For instance, the execution time of host applications are expected to be extended since migrated applications will compete the resources with them. In addition, comparing with fully utilized processing cores, a computation-intensive application migrated to the underutilized core may shorten its execution time. With the above observations, in this paper we are to establish analytical models to reveal these potential relations and utilize them to predict the timing change of impacted applications.

The rest of the paper is organized as follows. We first summarize the related work in Section II. Our proposed run-time system reconfiguration strategy is presented in Section III. Experimental settings and results are discussed in Section IV. Finally we conclude our work in Section V.

978-1-5386-3221-5/17 $31.00 © 2017 IEEE
DOI 10.1109/CSE-EUC.2017.210

IEEE computer society

Authorized licensed use limited to: Zhejiang University. Downloaded on March 13,2023 at 01:46:01 UTC from IEEE Xplore. Restrictions apply.

II. Related work

In traditional embedded system design, applications are statically deployed on processing units and their timing behaviors are fine-tuned in order to meet the requirements. As many-core platform are being used in embedded systems, extensive research has been done about how to map applications to many-core platform under various optimization goals. For instance, Chou [6] analyzed the impact of communication overhead on applications mapped to a many-core platform. In order to minimize on-chip communication workload, the applications to many-core platforms mapping was formulated to an integer linear programming problem. Derin [7] discussed another optimal mapping strategy with the objective of minimizing applications' total execution time.

With reducing feature size and increasing transistor count, chips are becoming susceptible to permanent faults [8]. When core wear-out happens, system reconfiguration process will be invoked and the application deployed on the defect cores has to be re-deployed on functional cores in order to continue the system's service. With the objective to maximize system's throughput after reconfiguration, Zhang [9] proposed a heuristic Row Rippling Column Stealing (RRCS) approach to select proper backup cores for faulty core replacement. From the energy consumption perspective, Das [10] proposed an integer linear programming approach to minimize communication energy consumption after the reconfiguration. To minimize the application's timing behavior change from its initial deployment after reconfiguration, several heuristics were also developed in [11], [3]. To support run-time system reconfiguration, mixed integer quadratic programming based approach was proposed in [12], which pre-calculated the reconfiguration strategies of all failure scenarios and directly used it at run-time.

The above mentioned reconfiguration strategies were based on the assumption that enough resource was reserved under the extreme worst-case scenario. If extreme case is rare, the whole system will be underutilized. To address these issues, this paper is to study run-time system reconfiguration under continuously changing environment and the essence of is to predict the system's timing change incurred by application migration process based on historical profiled data. There were also some existing work to predict a processor's performance. Joesph [13], [14] applied regression-based approaches to model processor performance. In [15], Ipek utilized artifical neutral network to construct processor performance model instead of regression. To predict workload's performance on another hardware platform with different underlying architecture, Zheng [16] recently developed an offline cross-platform prediction approach, which predicted applications' performance on a not-yet-available target platform by running it on another host platform with different underlying architecture. Different with the study in [16], our research is to predict the timing behavior change of impacted applications invoked by the application migration process, in addition, such prediction strategy has to be performed at run-time.

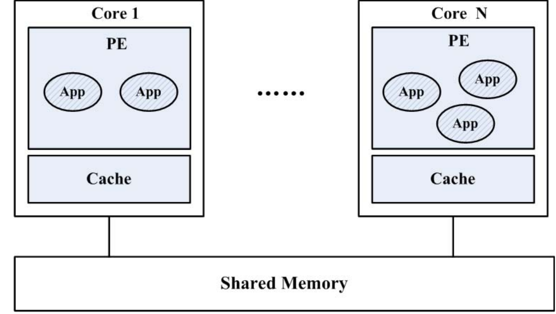

Fig. 1: Multi-core Processor

TABLE I: Performance Counters

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

III. Run-Time System Re-configuration Strategy

As stated above, system reconfiguration will be triggered when operating environment changes or core malfunction, both of which could happen at run-time. Therefore, the system re-configuration is expected to be invoked at run-time and the proposed timing prediction should be able to finish in short time. In addition, the execution time of both the migrated and host applications are impacted by migration process, therefore, both of which need analytical models to predict their potential timing change.

With these observations, our proposed timing predication strategy is illustrated in Fig. 2, which can be divided into the following steps: 1) Data Profiling 2) Analytical Model Training and 3) Run-Time Timing Prediction.

A. Data profiling

Learning based approaches are heavily dependent on the training data set [16]. As our prediction targets at execution time cost, the features which potentially impact an application's running time should be traced. Obviously, some specific features of the running application such as number of instructions and cache-references are the important factors. In addition, status of the running cores may also impact the deployed application's timing behavior. As shown in Fig. 1, all the cores on many-core platform have dedicated processing elements and caches but share the main memory. Therefore, for a specific application, it will compete with the other applications deployed on the same core for computing element, cache and memory, while at the same time, it will also compete the memory-access from all the other cores.

As application migration process impacts the execution times of both migrated applications and host applications,

Authorized licensed use limited to: Zhejiang University. Downloaded on March 13,2023 at 01:46:01 UTC from IEEE Xplore. Restrictions apply.

TABLE II: Profiled Features for Migrated Application Prediction

Mapp-1	Mapp-8	DC-1/SC-1	DC-12/SC-12
Mapp-2	Mapp-9	DC-2/SC-2	DC-13/SC-13
Mapp-3	Mapp-10	DC-3/SC-3	DC-14/SC-14
Mapp-4	Mapp-11	DC-4/SC-4	ID-4 of other cores
Mapp-5	Mapp-12	DC-9/SC-9	
Mapp-6	Mapp-13	DC-10/SC-10	
Mapp-7	Mapp-14	DC-11/SC-11	

TABLE III: Profiled Features for Host Application Prediction

Mapp-1	Mapp-11	Happ-7	DC-3/SC-3
Mapp-2	Mapp-12	Happ-8	DC-4/SC-4
Mapp-3	Mapp-13	Happ-9	DC-9/SC-9
Mapp-4	Mapp-14	Happ-10	DC-10/SC-10
Mapp-5	Happ-1	Happ-11	DC-11/SC-11
Mapp-6	Happ-2	Happ-12	DC-12/SC-12
Mapp-7	Happ-3	Happ-13	DC-13/SC-13
Mapp-8	Happ-4	Happ-14	DC-14/SC-14
Mapp-9	Happ-5	DC-1/SC-1	ID-4 of other cores
Mapp-10	Happ-6	DC-2/SC-2	

next, we discuss the data to be profiled for both of these applications, respectively.

1) Data Profiled for Migrated Application Timing Prediction:
Table I lists some performance counters supported by system kernel [17], which may impact an application's execution time. These counters can be measured at both application level and core level. For example, the counter cache-references measured at application level indicates the number of cache references only invoked by this application during its execution duration; while at core level means the total cache references on this core within the duration of running the application.

Intuitively, an application's execution time will be extended if it is migrated to a busier core, while it will be shorten if the destination core is free. Therefore, we trace the performance counters of the migrated application, source core as well as the destination core to make the prediction. Table II lists all the counters to be measured for timing prediction of migrated applications. Among which, Mapp-i, SC-i, DC-i and are the ith performance counter measured for migrated application, source and destination core, respectively. ID-4 of other cores are the cache miss rate of the processing cores rather than the source and destination, which potentially indicate the inter-core memory-access interference. To reduce the feature dimensions without losing information contained therein, we use the ratio of DC-i and SC-i, i.e. DC-i/SC-i, to represent the change of the running platform. Counters 5 - 8 are not traced at the core level as they are application specific counters.

2) Data Profiled for Host Application Timing Prediction:
The host applications are expected to run slower after the migration process. How much impact on the host applications depends on the degree of resource competition from migrated applications. Therefore, we traced the performance counters listed in Table III. It is not hard to find that, in addition to all the counters included in Table II, the performance counters of host applications are also profiled.

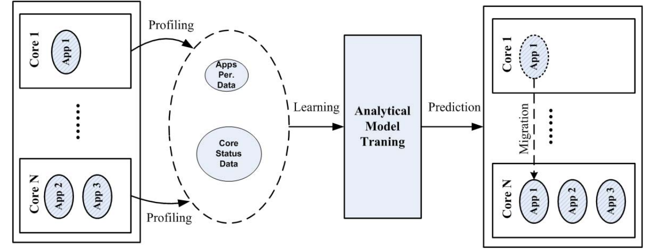

Fig. 2: Timing Prediction Strategy

Having the data collected, our next step is utilizing these profiled features to train analytical models, in order to predict the execution time of both migrated and host applications.

B. Analytical Model Prediction

The intuition behind model based prediction is to find some hidden pattern of "similar" applications. For example, To predict the execution time of a unknown heavy computing application, it is more reasonable to use the profiled data from computation-intensive applications rather than memory-intensive ones. Therefore, before training the analytical models for execution time prediction, we first discuss how to cluster the applications.

1) Application Clustering: Applications deployed in embedded systems can be classified into different categories (We assume Q categories total and Q is yet to be determined) based on their profiled performance counters. To classify an unknown application, we have to find the centeroid of each category and the closest one will determine its category.

To find the centeroid of each category, iterative K-mean clustering algorithm [18] is adopted and the major steps can be highlighted as follows:

1) Initialize the centroid of each category randomly;
2) Classify the profiled applications based on the current centeroids;
3) Update the centeroids of currently classified clusters;
4) Repeat Step 2 - 3 until reaching the maximum iteration times.

Application Classification:
Suppose $r^k = \{r_1^k, r_2^k, ..., r_n^k\}$ is the centeroid of category k, where r_i^k is the value of the corresponding feature. An application $x^i = \{x_1^i, x_2^i, ..., x_n^i\}$ will be classified to the category with nearest centeroid, i.e.

$$G(x^i) = k \text{ minimizes } \sqrt{\sum_{j=1}^{N}(x_j^i - r_j^k)^2}, k \in \{1..Q\} \quad (1)$$

where $G(x^i)$ indicates the category of application x_i and Q is the total number of categories yet to be determined from the profiled application features[1].

Centeroid Update:
After the application classification, the current centeroids may

[1]The features of applications are their profiled perfomrance counters, both terms are used in this paper and they are interchangeable.

Authorized licensed use limited to: Zhejiang University. Downloaded on March 13,2023 at 01:46:01 UTC from IEEE Xplore. Restrictions apply.

not still be the center points of newly classified clusters. Therefore, each centeroid needs to be updated as the center of all the applications clustered in the corresponding category, i.e. $r^k = \{r_i^k, i \in \{1..n\}\}$ where

$$r_i^k = \frac{\sum_{\cup\{G(x_i)=k\}} x_i^k}{|\cup\{G(x_i) = k\}|}$$

where $\cup\{G(x_i) = k\}$ indicates the set of applications classified to category k.

2) Analytical Model Training: We assume that the execution time of the running application lies in linear relation with selected features and and regularized linear regression algorithm is adopted to train the model. More specifically, suppose x_i^k is the ith profiled feature of the target application k, then its execution time can be predicted as [19]:

$$\begin{aligned} h^k(\theta, x) &= \theta_0 + \theta_1 x_1^k + ... + \theta_n x_n^k \\ &= \sum_{i=0}^{n} \theta_i x_i^k \end{aligned} \quad (2)$$

where n is the total number of features and $\theta = \{\theta_1, ..., \theta_n\}$ is the hypothesis yet to be determined.

The appropriate selection of θ will make the profiled application data fits the linear model as close as possible. We use y^k to denote the actual execution time of targeted application k and define the cost function of a specific hypothesis $h^k(\theta, x)$ as [20]:

$$E(\theta) = \frac{1}{2}\sum_{k=1}^{m}(h^k(\theta, x) - y^k)^2 + \frac{\lambda}{2m}(\sum_{i=1}^{n}\theta_i^2) \quad (3)$$

where m is the total number of profiled data records. λ is a regularization parameter, which inherently performs feature selection to prevent overfitting [20].

The smaller $E(\theta)$ indicates the better fit of profiled data. The objective of the model training is to find the best θ which minimizes $E(\theta)$ and gradient decent algorithm [19] is adopted to search the hypothesis θ.

For self-containment, we brief the gradient decent algorithm in the following steps:

1) Initialize θ randomly
2) Update $\theta_i = \theta_i - \alpha\frac{\partial E(\theta)}{\partial \theta_i}$
3) Repeat the above two steps until convergence

where

$$\frac{\partial E(\theta)}{\partial \theta_i} = \begin{cases} \frac{1}{m}\sum_{k=1}^{m}(y^k - h(\theta, x))x_i^k & \text{if } i = 0 \\ \frac{1}{m}\sum_{k=1}^{m}(y^k - h(\theta, x))x_i^k + \frac{\lambda}{m}\theta_i & \text{if } i \geq 1 \end{cases}$$

and α is the learning rate, which impacts the speed of convergence.

It is not hard to find that the value of performance counters are not at the same scale. For example, the number of instructions could be millions of times the context-switches. When features differ in orders of magnitude, feature normalization can speed up the convergence of gradient descent algorithm. Therefore, before applying the above gradient descent algorithm, we normalize the features by subtracting the mean value first and then dividing the range, i.e.

$$x_i^k = \frac{x_i^k - \frac{1}{m}\sum_{k=1}^{m} x_i^k}{\max\limits_{k\in\{1..m\}} x_i^k - \min\limits_{k\in\{1..m\}} x_i^k}$$

Given the number of application category Q, the regularization parameter λ and the learning rate α, we can follow the above steps to obtain the prediction model for each application category. However, different selections of Q, λ and α may result in different prediction models. As we know, model training is to select the one minimizing cost function using the training data set. However, best fit of the training data does not mean the best prediction of future data. Therefore, we adopt cross-validation approach to evaluate the trained models [20]. In particular, the whole data set D is split into two partitions: D_t and D_v, where D_t is used for model training and the D_v is to validate the performance of selected model. Typically, D_v is around 10% of total whole data set and should be randomly selected.

The applications deployed on a specific embedded system are relatively fixed and the total number of category Q_{max} can be pre-estimated by screening all deployed applications. In addition, we set $0 \leq \lambda \leq \lambda_{\text{max}}$ and $0.1 \leq \alpha \leq \alpha_{\text{max}}$ where $\lambda_{\text{max}} = \alpha_{\text{max}} = 3$ are the typical range of regularization parameter and learning rate. To obtain the best selection of these parameters, iterative cross-validation approach is developed and described in Algorithm 1, where $step$ could be set at 0.1 scale.

ALGORITHM 1: Iterative Cross-Validation

```
1 for Q = 1; Q ≤ Q_max; k++ do
2 |  for λ = 0; λ ≤ λ_max; λ = λ + step do
3 |  |  for α = 0.1; α ≤ α_max; α = α + step do
4 |  |  |  Obtain the hypothesis θ using regualization
   |  |  |    linear regression on D_s
5 |  |  |  Calculate the cost E(θ) (formula (3)) on D_v
6 |  |  end
7 |  end
8 end
9 return The hypothesis θ with the minimum E(θ)
```

The training procedures of execution time prediction models for both migrated and host applications are the same except using different features. More specifically, profiled features included in Table II are used for migrated applications, while Table III is used instead for host ones.

As the profiled performance counters for applications in different category may vary vastly, the presented algorithm will be used to train the execution time prediction models separately for each category.

C. Run-Time Timing Prediction

Both data profiling and model training involve a large amount of the data and usually are time consuming, which can

Authorized licensed use limited to: Zhejiang University. Downloaded on March 13,2023 at 01:46:01 UTC from IEEE Xplore. Restrictions apply.

not be applied at run-time if these two stages started after the time instant when system reconfiguration is actually triggered. Notice that applications deployed on embedded systems are usually periodic, therefore, our developed approach can be implemented in the following steps to make it applicable at run-time:

1) Conduct initial experimental test and data profiling before system is actually in use, train the models based on the collected data;
2) Continuously collect the selected performance counters in Table II and Table III periodically (sampling if too much overhead) after system is actually in use;
3) Conduct model training when enough new data is collected and update the existing models;
4) Repeat the steps 2) - 3) to update the existing models as the time goes on;
5) If environment change or core malfunction happens: a) cluster the migrated and host applications based on their most recent performance counters; b) Use the current models to predict their execution times and upon which to determine the application migration strategy.

Following the above steps, the timing prediction models will be continuously updated and adapted to ever-changing environment. When the system re-configuration is triggered, the analytical models will be ready and can be adopted immediately for timing prediction.

IV. Evaluation

In this section, we set up experiments to evaluate our proposed timing prediction approach.

A. Experimental Setting

Our testbed is configured with four Intel i5-520M processing cores (each has 3MB cache), 6 GB main memory shared by all the processing cores and 500 GB solid state drive. Ubuntu operation system version 16.04 is installed on the testbed. In addition, the applications we profiled come from three commercially representative benchmark suites: MiBench, MediaBench and SD-VBS. MiBench [21] is a set of commercially embedded applications which can be divided into six categories: Automotive and Industrial Control, Consumer Devices, Office Automation, Networking, Security, and Telecommunications. Each of these categories represents a specific area of the embedded market. MediaBench [22] contains 19 applications selected from image processing, communications and DSP applications. SD-VBS [23] is a suite of applications from the vision domain, including nine applications with over 28 kernels, three different sets of configurations per application and five distinct input data sets per configuration.

The performance counters of migrated applications are profiled in the following steps:

1) Randomly select some applications and deploy them on different cores, keep all of them running as background processes;
2) Choose one core as the source and another one as the destination, deploy and launch one more application (to be migrated) on the source core, trace the performance counters in Table II and the execution time of this application using PERF [24]: a performance analyzing tool in Linux.
3) Migrate the application to a destination core and profile its execution time after the migration.

By varying the input workload of the applications and background processes running on different cores, we traced about 600 different data records. The same steps are followed to generate the training data set for host applications except that the performance counters in Table III instead.

The performance of our proposed approach are evaluated in the following steps:

- Partition the profiled data set into three sub sets training data set (D_t), validation data set (D_v) and prediction data set (D_p). Among which, D_t is composed of 80% of the whole data set, D_v and D_p are 10% each;
- Train the models using Algorithm 1 using D_t and D_v;
- Apply the obtained models to predict execution times of applications included in D_p and compare with their actually profiled execution times.

B. Migrated Application Timing Prediction

In this subsection, we analyze the experimental results to predict execution times of migrated applications. Table. IV lists a few applications randomly selected and included in D_p. The comparisons of predicted execution times and actually profiled data are depicted in Fig. 3. From which we can find that, for most of the applications, such as CRC32, JPGDEC, MAD and JDPEG, our predictions are closed to the profiled data.

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

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

In order to give an overall statistics of our prediction, we calculate the cumulative distribution function (CDF) of prediction error on the whole data set D_p 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.

From the results given in Fig. 4, we can see that 70% of the applications have prediction error less than 0.3, 90% of which are less than 0.4 and almost all the prediction errors are within 0.5.

C. Host Application Timing Prediction

The execution time prediction model of host applications is also trained and their CDF of the prediction error is illustrated in Fig. 5. According to Fig. 5, 80% of the prediction errors are within 0.3, 95% of which are less than 0.4 and the maximum value is 0.45.

Authorized licensed use limited to: Zhejiang University. Downloaded on March 13,2023 at 01:46:01 UTC from IEEE Xplore. Restrictions apply.

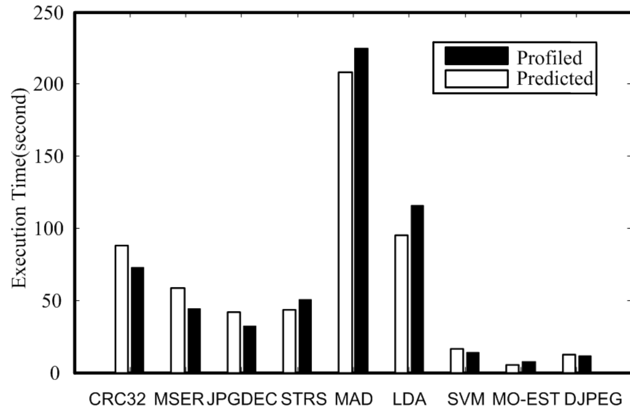

Fig. 3: Migrated Application Timing Prediction

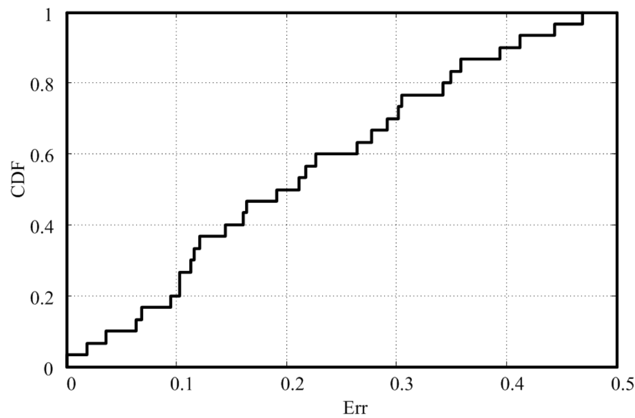

Fig. 4: CDF of Migrated Application Execution Time Prediction

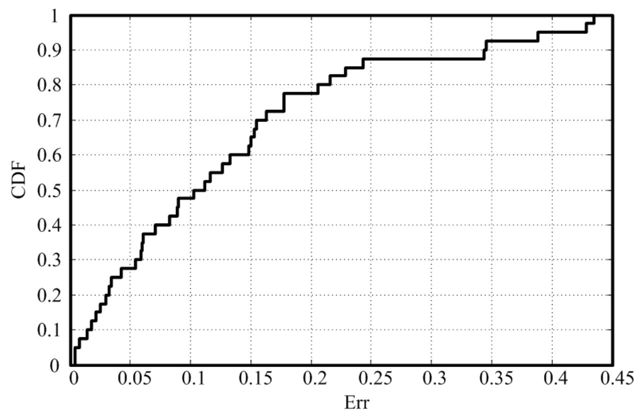

Fig. 5: CDF of Host Application Execution Time Prediction

TABLE IV: Selected Migrated Applications for Prediction

ID	Benchmark	Application
1	MIBENCH	CRC32
2	SD-VBS	MSER
3	MEDIABENCH	JPGDEC(Jpg2000dec)
4	MIBENCH	STRS(StringSearch)
5	MIBENCH	MAD
6	SD-VBS	LDA
7	SD-VBS	SVM
8	SD-VBS	MO-EST(Motion-Estimation)
9	MEDIABENCH	DJPEG

V. Conclusion

In this paper, we present prediction based run-time system reconfiguration approach on many-core embedded systems. To select an appropriate application migration strategy such that the system's timing behaviors after the migration are still acceptable, a run-time timing prediction strategy, including data profiling, model training and execution time prediction, are proposed. Initial data profiling and model training are conducted before the system put in use, the obtained models will be continuously updated and adjusted to the new environment after system is actually in use. The most recent models will be used for timing prediction when system reconfiguration is invoked at run-time. To evaluate the performance of our proposed approach, we conducted extensive experiments by deploying multiple benchmarks including MIBENCH, MEDIABENCH and VD-VBS on a multi-core platform. Experimental results validate the effectiveness of our proposed approach.

Our future work will investigate non-linear analytical models and online regression strategies to further improve the prediction performance and reduce the online computation complexity.

Acknowledgement

This work was supported by Western Illinois University URC grant 330315.

References

[1] F. J. Rammig, S. Grösbrink, K. Stahl, and Y. Zhao, "Designing self-adaptive embedded real-time software–towards system engineering of self-adaptation," in *2014 Brazilian Symposium on Computing Systems Engineering*. IEEE, 2014, pp. 37–42.

[2] L. Almeida, S. Fischmeister, M. Anand, and I. Lee, "A dynamic scheduling approach to designing flexible safety-critical systems," in *Proceedings of the 7th ACM & IEEE international conference on Embedded software*. ACM, 2007, pp. 67–74.

[3] Z. Li, F. Lockom, and S. Ren, "Maintaining real-time application timing similarity for defect-tolerant noc-based many-core systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 2s, p. 64, 2014.

[4] Z. Li, S. Li, X. Hua, H. Wu, and S. Ren, "Run-time reconfiguration to tolerate core failures for real-time embedded applications on noc many-core platforms," in *High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on*. IEEE, 2013, pp. 1990–1997.

[5] C. Wu, C. Deng, L. liu, J. Han, J. Chen, S. Yin, and S. Wei, "A multi-objective model oriented mapping approach for noc-based computing systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. PP, no. 99, pp. 1–1, 2016.

Authorized licensed use limited to: Zhejiang University. Downloaded on March 13,2023 at 01:46:01 UTC from IEEE Xplore. Restrictions apply.

[6] C.-L. Chou and R. Marculescu, “Contention-aware application mapping for network-on-chip communication architectures,” in *Proc. of IEEE International Conference on Computer Design*, ser. ICCD, oct. 2008, pp. 164–169.
[7] O. Derin, D. Kabakci, and 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*, ser. NoCS, may 2011, pp. 129–136.
[8] C. Constantinescu, “Trends and challenges in vlsi circuit reliability,” *Micro*, vol. 23, no. 4, pp. 14–19, 2003.
[9] L. Zhang, Y. Han, Q. Xu, and X. Li, “Defect tolerance in homogeneous manycore processors using core-level redundancy with unified topology,” in *Proc. of IEEE/ACM Design, Automation and Test in Europe Conference*, ser. DATE, march 2008, pp. 891–896.
[10] A. Das, A. Kumar, and B. Veeravalli, “Energy-aware communication and remapping of tasks for reliable multimedia multiprocessor systems,” in *Proc. of 18th International Conference on Parallel and Distributed Systems*, 2012, pp. 564–571.
[11] K. Yue, F. Lockom, Z. Li, S. Ghalim, S. Ren, L. Zhang, and X. Li, “Hungarian algorithm based virtualization to maintain application timing similarity for defect-tolerant noc,” in *Proc. of 17th Asia and South Pacific Design Automation Conference*, ser. ASP-DAC, 2012, pp. 493 –498.
[12] Z. Li, S. Li, X. Hua, H. Wu, and S. Ren, “Run-time reconfiguration to tolerate core failures for real-time embedded applications on noc manycore platforms,” in *2013 IEEE 10th International Conference on High Performance Computing and Communications 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, Nov 2013, pp. 1990–1997.
[13] P. Joseph, K. Vaswani, and M. J. Thazhuthaveetil, “Construction and use of linear regression models for processor performance analysis,” in *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*. IEEE, 2006, pp. 99–108.
[14] ——, “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.
[15] E. Ïpek, S. A. McKee, R. Caruana, B. R. de Supinski, and M. Schulz, *Efficiently exploring architectural design spaces via predictive modeling*. ACM, 2006, vol. 40, no. 5.
[16] 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.
[17] A. C. de Melo, “Performance counters on linux,” in *Linux Plumbers Conference*, 2009.
[18] P.-N. Tan *et al.*, *Introduction to data mining*. Pearson Education India, 2006.
[19] E. Alpaydin, *Introduction to Machine Learning. [Sl]*. The MIT Press, 2010.
[20] J. Friedman, T. Hastie, and R. Tibshirani, *The elements of statistical learning*. Springer series in statistics Springer, Berlin, 2001, vol. 1.
[21] 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.
[22] 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.
[23] 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.
[24] A. C. de Melo, “The new linuxperftools,” in *Slides from Linux Kongress*, vol. 18, 2010.

Authorized licensed use limited to: Zhejiang University. Downloaded on March 13,2023 at 01:46:01 UTC from IEEE Xplore. Restrictions apply.