Learning based Memory Interference Prediction for Co-running Applications on Multi-Cores

Ahsan Saeed[1,2], Daniel Mueller-Gritschneder[2], Falk Rehm[1], Arne Hamann[1],
Dirk Ziegenbein[1], Ulf Schlichtmann[2] and Andreas Gerstlauer[3]
[1]Robert Bosch GmbH, [2]Technical University of Munich, [3]The University of Texas at Austin
{ahsan.saeed,falk.rehm,arne.hamann,dirk.ziegenbein}@de.bosch.com,
{daniel.mueller,ulf.schlichtmann}@tum.de, gerstl@ece.utexas.edu

***Abstract*—Early run-time prediction of co-running independent applications prior to application integration becomes challenging in multi-core processors. One of the most notable causes is the interference at the main memory subsystem, which results in significant degradation in application performance and response time in comparison to standalone execution. Currently available techniques for run-time prediction like traditional cycle-accurate simulations are slow, and analytical models are not accurate and time-consuming to build. By contrast, existing machine-learning-based approaches for run-time prediction simply do not account for interference. In this paper, we use a machine learning-based approach to train a model to correlate performance data (instructions and hardware performance counters) for a set of benchmark applications between the standalone and interference scenarios. After that, the trained model is used to predict the run-time of co-running applications in interference scenarios. In general, there is no straightforward one-to-one correspondence between samples obtained from the standalone and interference scenarios due to the different run-times, i.e. execution speeds. To address this, we developed a simple yet effective sample alignment algorithm, which is a key component in transforming interference prediction into a machine learning problem. In addition, we systematically identify the subset of features that have the highest positive impact on the model performance. Our approach is demonstrated to be effective and shows an average run-time prediction error, which is as low as 0.3% and 0.1% for two co-running applications.**

***Index Terms*—run-time, prediction, machine learning, interference, co-running, microprocessor, sample alignment, memory interference**

I. Introduction

Early performance prediction is one of the major challenges for co-running or co-locating independent (e.g. virtualized) applications on multi-core processors or on the same server in the cloud. In the automotive industry, for example, the current trend is to combine multiple distributed electrical/electronic (E/E) systems towards a more centralized design where multiple functions are consolidated onto powerful multi-core and multi-processor systems-on-chip (MPSoCs).

Building such multi-core processor platforms imposes new challenges for a system designer as well application developers, given that these powerful platforms typically feature multiple processing elements, sharing some hardware resources like the interconnect and the main memory. Emerging applications, such as assisted driving, require a lot of memory, and the additional delay caused by memory contention cannot be ignored because it has a significant impact on run-time. Modern multi-core processor platforms, on the other hand, are extremely complex and are designed for performance rather than predictability. As a consequence, when applications are co-run, interference through shared resources leads to undesired application performance coupling [1] and as a result, non-negligible context-dependent execution-time variability.

With such effects, prediction of the response time of an application is non-trivial since these contention effects are very difficult to analyze. Contention effects can be evaluated by simply co-running the applications. However, in many cases, different parties develop applications independently, and sharing applications between them is not possible due to the involvement of intellectual property (IP). Similarly, in a server or cloud context, an operating system (OS) should be able to estimate the performance impact of co-location decisions before committing them. As such, it is highly desirable to evaluate the timing characteristics of the entire platform (e.g., application response times) before co-locating tasks.

Widely adopted simulation-based approaches, such as cycle accurate models, and traditional analytical models are not an option for interference evaluation due to the aforementioned application unavailability due to the involvement of IP in addition to lacking significantly in speed and accuracy respectively. Machine-learning based approaches have emerged for application performance prediction [2]–[6], but existing approaches do not take interference into account. Therefore, there is a need for new methods predicting the effects of interference on run-time of applications while representing interfering applications in a way that does not require the sharing of actual application themselves.

In this paper, we demonstrate how to formulate the problem of run-time prediction of co-running applications in the multi-core processor platform into a machine learning-based approach and propose an Interference-Aware Runtime Predictor framework for generating the predictive model. One-to-one correspondence of samples for machine learning between the standalone and interference scenarios is not possible due to the different rates of application progress in the respective scenarios. To address this issue, we propose a novel, efficient, and effective sample alignment algorithm that aligns the samples so that they roughly correspond to the same section of

978-1-6654-3166-8/21/$31.00 ©2021 IEEE

executed code. The key contributions of this work are:

1) We propose a machine learning based approach that can predict the run-time of co-running applications in multi-core processors. It combines an efficient sample alignment algorithm based on correspondence of executed instructions in standalone and interference scenarios with a regression-based learning formulation to predict the number of instructions executed per sampling period in interference.
2) We identify and select features based on actual DRAM utilization and hardware performance events of the memory controller and cores to accurately represent the execution characteristics of an application for interference prediction. Additionally, we explore different linear regression models for their suitability in modeling interference behavior.
3) We validate our approach on a multi-core (NXP S32V234) platform and evaluate it on an extensive set of realistic benchmarks from the San Diego Vision Benchmark set [7], and demonstrate its effectiveness by predicting the average run-time of applications in interference scenario while executing applications in standalone only. Results show that our approach is able to predict the average run-time with an error as low as 0.3% and 0.1% for two co-running applications, respectively.

II. Related Work

Widely adopted simulation-based approaches estimate performance of an application by executing it on cycle-accurate or cycle-approximate instruction set simulators (ISSs) [8]–[10], which excel in accuracy yet have substantial speed limitations. Traditional analytic models for performance prediction of different architectures are difficult and time-consuming to build, and are frequently incapable of capturing the full system and application complexity.

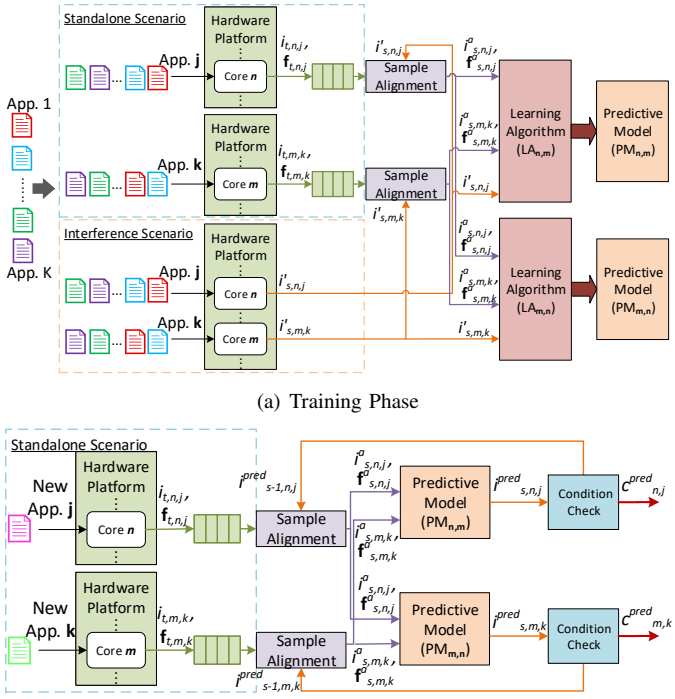
By contrast, learning based approaches have recently emerged [2]–[6], which tend to be simple, fast and accurate. However, they only execute a single task on a single core and thus do not take interference into account. While [11] claims to predict parallel application performance, the prediction is intended to aid in tuning an application's input parameters and algorithm parameters rather than to estimate interference for co-running applications. In general, all existing machine learning approaches rely solely on hardware performance events within the cores, which provides insufficient information about the underlying memory saturation and thus interference. This prevents the predictive models from learning important application characteristics relevant to interference scenarios. By contrast, we propose an extensive set of features based on actual DRAM utilization and hardware performance events of memory controllers in addition to cores.

III. Interference Prediction

An overview of the proposed approach for interference prediction is shown in Fig. 1. The set of symbols used throughout the paper is summarized in Table I. The learning-based formulation of the run-time prediction of co-running applications consists of two phases: a training phase (Fig. 1(a)) and a prediction phase (Fig. 1(b)). During the training phase, a pair of applications (j,k) from a benchmark set (training applications) are run individually (standalone scenario) and then co-run (interference scenario) on cores n and m of the hardware platform. For each of these scenarios, we collect, at a fixed sampling period, the number of instructions i executed as well as various hardware performance events (features) $\mathbf{f}$ per period. The standalone instructions and features are then passed through a sample alignment algorithm to obtain aligned samples i^a and $\mathbf{f}^a$ that correspond to sections of code

TABLE I
Symbol Table.

Symbols	Descriptions
N	Number of cores
K	Number of applications
T	Number of samples in an application in standalone
j	j-th application running on nth core
k	k-th application running on mth core
t	t-th sample in standalone scenario
s	s-th sample in interference scenario
τ	Sample period
i	Instructions per τ in standalone scenario
$\mathbf{f}$	Vector of features per τ in standalone scenario
i^a	Aligned instructions per τ
$\mathbf{f}^a$	Aligned vector of features per τ
$i^{'}$	Instructions per τ in interference scenario
i^{pred}	Predicted instructions
a	Accumulated i^{pred}
c^{pred}	Predicted run-time



(a) Training Phase

(b) Prediction Phase

Fig. 1. Interference prediction framework

on both cores running in contention during interference. Our goal is to extract the latent relationship between the hardware performance events and the number of instructions executed in interference per sample period for applications running on core n when they are co-executed with another application on core m. We formulate this problem into a machine learning setting, and derive a predictive model $PM_{n,m}$ by using a learning algorithm $LA_{n,m}$.

It is worth to note that the number of needed *LA* instances, and therefore the corresponding *PM*, is equal to the total number of unique target and interfering core pairs on which applications in the hardware platform co-run. This is due to the fact that all cores with an operating system running on top of it are not always same in terms of execution characteristics. Hence, the order is important, and $PM_{n,m}$ is not equal to $PM_{m,n}$.

During the prediction phase, a new pair of applications is run standalone. A set of hardware performance events is obtained at a fixed sampling period and passed through sample alignment to be used as inputs to the *PM* in order to produce an estimate of the number of instructions i^{pred} executed per sampling period in interference. The number of instructions executed per sampling period is then translated into a predicted run-time c^{pred} of the applications.

A. Interference-Aware Runtime Predictor

Algorithm 1 sketches the working of our proposed interference-aware run-time prediction in training and prediction phases to learn and apply on two cores, but can be generalized to more cores. During the training phase, training applications are profiled in two different scenarios: standalone and interference. At first, pairs of training applications j and k are run standalone on cores m and n, respectively, to collect T_j and T_k number of samples of instructions i and features $\mathbf{f}$, which are stored in global buffers $\mathbf{i}^{buf}$ and $\mathbf{F}^{buf}$. Next, the same pairs of training applications (j,k) are then co-run on cores (n, m) to collect only instructions $i^{'}$ in interference. $i^{'}$ samples are passed through the *sampAlign* algorithm (discussed in Section III-B) together with $\mathbf{i}^{buf}$, $\mathbf{F}^{buf}$ to obtain the aligned samples i^{a} and $\mathbf{f}^{a}$. The sample collection is performed until the first application finishes execution and there is no more contention, which is determined by comparing the accumulated instructions a with the total number of instructions i^{total} in each application. The arrays $\mathbf{X}$ and $\mathbf{y}$ are used to store all of the aligned samples and their corresponding instructions in interference, respectively, from all the pairs of applications. A learning algorithm *LA* is then applied to derive a predictive model *PM*, which learns the relationship between the arrays of aligned features $\mathbf{X}$ and instructions $\mathbf{y}$ executed during each sample period τ.

During the prediction phase, a pair of new applications is run standalone on cores (m,n). The collected samples are stored, passed through the sample alignment algorithm (similar to the training phase), and fed into the *PM* to obtain the predicted instructions i^{pred} during each sample period τ. The total number of predicted instructions a is kept track of by

Algorithm 1: Interference-Aware Run-time Predictor

Training Phase:
input : n, m
output: $PM_{n,m}$, $PM_{m,n}$
foreach *pair of App (j,k) running on Core (n,m)* **do**
  foreach *sample* $t < T_j$ **do**
    Append $i_{t,n,j}$ to $\mathbf{i}^{buf}_{n,j}$, Append $\mathbf{f}_{t,n,j}$ to $\mathbf{F}^{buf}_{n,j}$
  foreach *sample* $t < T_k$ **do**
    Append $i_{t,m,k}$ to $\mathbf{i}^{buf}_{m,k}$, Append $\mathbf{f}_{t,m,k}$ to $\mathbf{F}^{buf}_{m,k}$
  $i^{total}_{n,j} = \sum_{t=1}^{T_j} i^{buf}_{t,n,j}$, $i^{total}_{m,j} = \sum_{t=1}^{T_k} i^{buf}_{t,m,k}$
  $a_{n,j}$ = 0, $a_{m,k}$ = 0, s = 1
  while $a_{n,j} < i^{total}_{n,j}$ *and* $a_{m,k} < i^{total}_{m,k}$ **do**
    if $s == 1$ **then**
      $i^{a}_{s,n,j} = i^{buf}_{1,n,j}$, $\mathbf{f}^{a}_{s,n,j} = \mathbf{f}^{buf}_{1,n,j}$
      $i^{a}_{s,m,k} = i^{buf}_{1,m,k}$, $\mathbf{f}^{a}_{s,m,k} = \mathbf{f}^{buf}_{1,m,k}$
    else
      $i^{a}_{s,n,j}$, $\mathbf{f}^{a}_{s,n,j}$ = sampAlign($\mathbf{i}^{buf}_{n,j}$, $\mathbf{F}^{buf}_{n,j}$, $i^{'}_{s,n,j}$)
      $i^{a}_{s,m,k}$, $\mathbf{f}^{a}_{s,m,k}$ = sampAlign($\mathbf{i}^{buf}_{m,k}$, $\mathbf{F}^{buf}_{m,k}$, $i^{'}_{s,m,k}$)
    Append $[i^{a}_{s,n,j}, \mathbf{f}^{a}_{s,n,j}, i^{a}_{s,m,k}, \mathbf{f}^{a}_{s,m,k}]$ to $\mathbf{X}$
    Append $i^{'}_{s,n,j}$ to $\mathbf{y}_n$, Append $i^{'}_{s,m,k}$ to $\mathbf{y}_m$
    $a_{n,j} = a_{n,j} + i^{'}_{s,n,j}$, $a_{m,k} = a_{m,k} + i^{'}_{s,m,k}$
    s = s + 1
$PM_{n,m} \leftarrow LA_{n,m}(\mathbf{X}, \mathbf{y}_n)$
$PM_{m,n} \leftarrow LA_{m,n}(\mathbf{X}, \mathbf{y}_m)$

Prediction Phase:
input : n, m, j, k, τ
output: $c^{pred}_{n,j}$, $c^{pred}_{m,k}$
foreach *sample* $t < T_j$ **do**
  Append $i_{t,n,j}$ to $\mathbf{i}^{buf}_{n,j}$, Append $\mathbf{f}_{t,n,j}$ to $\mathbf{F}^{buf}_{n,j}$
foreach *sample* $t < T_k$ **do**
  Append $i_{t,m,k}$ to $\mathbf{i}^{buf}_{m,k}$, Append $\mathbf{f}_{t,m,k}$ to $\mathbf{F}^{buf}_{m,k}$
$i^{total}_{n,j} = \sum_{t=1}^{T_j} i^{buf}_{t,n,j}$, $i^{total}_{m,j} = \sum_{t=1}^{T_k} i^{buf}_{t,m,k}$
$a_{n,j}$ = 0, $a_{m,k}$ = 0, s = 1
while $a_{n,j} < i^{total}_{n,j}$ *and* $a_{m,k} < i^{total}_{m,k}$ **do**
  if $s == 1$ **then**
    $i^{a}_{s,n,j} = i^{buf}_{1,n,j}$, $\mathbf{f}^{a}_{s,n,j} = \mathbf{f}^{buf}_{1,n,j}$
    $i^{a}_{s,m,k} = i^{buf}_{1,m,k}$, $\mathbf{f}^{a}_{s,m,k} = \mathbf{f}^{buf}_{1,m,k}$
  else
    $i^{a}_{s,n,j}$, $\mathbf{f}^{a}_{s,n,j}$ = sampAlign($\mathbf{i}^{buf}_{n,j}$, $\mathbf{F}^{buf}_{n,j}$, $i^{pred}_{s-1,n,j}$)
    $i^{a}_{s,m,k}$, $\mathbf{f}^{a}_{s,m,k}$ = sampAlign($\mathbf{i}^{buf}_{m,k}$, $\mathbf{F}^{buf}_{m,k}$, $i^{pred}_{s-1,m,k}$)
  $i^{pred}_{s,n,j} = PM_{n,m}([i^{a}_{s,n,j}, \mathbf{f}^{a}_{s,n,j}, i^{a}_{s,m,k}, \mathbf{f}^{a}_{s,m,k}])$
  $i^{pred}_{s,m,k} = PM_{m,n}([i^{a}_{s,n,j}, \mathbf{f}^{a}_{s,n,j}, i^{a}_{s,m,k}, \mathbf{f}^{a}_{s,m,k}])$
  $a_{n,j} = a_{n,j} + i^{pred}_{s,n,j}$, $a_{m,k} = a_{m,k} + i^{pred}_{s,m,k}$
  $s = s + 1$
Find largest β_j for which $i^{buf}_{1,n,j} + \cdots + i^{buf}_{\beta_j,n,j} \leq a_{n,j}$
Find largest β_k for which $i^{buf}_{1,m,k} + \cdots + i^{buf}_{\beta_k,m,k} \leq a_{m,k}$
$c^{pred}_{n,j} = s \times \tau + (T_j - \beta_j) \times \tau$
$c^{pred}_{m,k} = s \times \tau + (T_k - \beta_k) \times \tau$
return $c^{pred}_{n,j}$, $c^{pred}_{m,k}$

Algorithm 2: Sample Alignment

1 sampAlign:
input : $\mathbf{i}^{buf}$, $\mathbf{F}^{buf}$, $i^{'}$
output: i^{a}, $\mathbf{f}^{a}$
2 Find largest β for which i_1^{buf} + ... + $i_\beta^{buf} \leq i^{'}$
3 $\gamma = \frac{i_{\beta+1}^{buf} - i^{'}}{i_{\beta+1}^{buf}}$
4 **if** γ == *1* **then**
5 $\quad i^{a} = i_\beta^{buf}$
6 $\quad \mathbf{f}^{a} = \mathbf{f}_\beta^{buf}$
7 **else**
8 $\quad i^{a} = i_\beta^{buf} \times (1-\gamma)$ + $i_{\beta+1}^{buf} \times \gamma$
9 $\quad \mathbf{f}^{a} = \mathbf{f}_\beta^{buf} \times (1-\gamma)$ + $\mathbf{f}_{\beta+1}^{buf} \times \gamma$
10 **return** i^{a}, $\mathbf{f}^{a}$

accumulating i^{pred}. As soon as one of the two application finishes its execution, there is no more interference, which is again detected by comparing a against the total number of instructions i^{total} of each application. Finally, the run-time of the application that finished is calculated by multiplying the number of predicted iterations s with the sample period τ. The run-time of the unfinished application is estimated by converting its remaining instructions into samples and adding the additional time left for running standalone.

B. Sample Alignment

Co-running applications tend to experience delays in run-time due to interference. As a result, applications have different numbers of instructions executed per sample period in standalone vs. interference scenarios. This poses an issue for prediction and training, where features are collected in a standalone setting but the predictive model *PM* needs to be provided inputs that correspond to sections of code that execute in contention and hence run simultaneously on both cores in interference. To address this issue, we propose an efficient sample alignment algorithm (see Algorithm 2), which aligns samples of applications on each core in standalone scenarios to approximately correspond to the same section of executed code in interference. The algorithm is based on the fact that the number of instructions in an application remains the same irrespective of whether they are executed in standalone or interference scenarios.

Algorithm 2 illustrates our proposed sample alignment algorithm. Given the buffer $\mathbf{i}^{buf}$ holding the numbers of instructions executed per standalone sample and a target number of instructions $i^{'}$ executed in interference to align to, the algorithm first scans the buffer to find the standalone sample β that corresponds to i', i.e. that has executed the same number of instructions (line 2). In general, the corresponding element in $\mathbf{i}^{buf}$ and the corresponding sample of hardware performance events in $\mathbf{F}^{buf}$ are then returned as aligned samples i^{a} and $\mathbf{f}^{a}$ (lines 5 and 6). However, if the adjustment is made at sample level, the alignment can potentially be very coarse. Although sampling with a small period is theoretically possible, it entails an overhead due to the generation of more frequent timer interrupts. We instead align samples by interpolating between them. The algorithm calculates a ratio γ to determine the position of $i^{'}$ in between the current and next sample in i^{buf} (line 3). If i' falls exactly on a sample boundary (line 4), i^{a} and $\mathbf{f}^{a}$ are returned directly. Otherwise, the aligned samples i^{a} and $\mathbf{f}^{a}$ are calculated by proportionally combining parts of the β-th and $\beta+1$ samples (lines 8 and 9).

TABLE II
SB-VDS BENCHMARK CHARACTERISTICS IN STANDALONE.

Applications	Avg. IPC	Avg. Bandwidth
multi_ncut	0.88	450 MB/s
disparity	0.50	441 MB/s
tracking	0.59	406 MB/s
mser	0.67	328 MB/s
sift	0.69	126 MB/s
stitch	0.90	124 MB/s

IV. EVALUATION AND RESULTS

We evaluate our approach on the NXP S32V234 [12] embedded platform. The SoC features 4 ARM Cortex A53 [13] CPUs, organized into 2 clusters each having 2 cores. Each core has its own private L1 data and instruction cache whereas the 2 cores within a cluster share a unified L2 cache. We use two cores from two distinct clusters to perform our analysis.

All the hardware performance events are collected at a fixed sampling period of 0.2ms using a System Level Instrumentation Framework (SLIF). Hardware performance events of an application are obtained from Performance Monitoring Units (PMUs) of the cores and Profiling Unit (PU) of the memory controller. SLIF is implemented in Linux version 4.19 as a loadable kernel module. The collected samples are stored in SRAM memory to avoid additional traffic to the main memory.

A subset of benchmarks in the San Diego Vision Benchmark Suite (SD-VBS) [7] are used to gain insight into the platform and evaluate the proposed approach. Since we are interested in applications that are DRAM-bound, we use the ones with the largest input data size (named *FullHD*). The characteristics of each benchmark included in our evaluations are summarized in Table II. Unless otherwise noted, we use *multicut*, *mser*, *stitch* and *sift* applications for the training set, and *disparity* and *tracking* as the test set in all experiments.

A. Feature Selection

The evaluation platform has 58 measurable hardware performance events, but provides only six generic 32-bit hardware performance counters and a dedicated 64-bit cycle counter. As such, we can only read 6 of the total 58 hardware performance events in addition to the dedicated cycle counter at any given time. This limitation is overcome by re-running the benchmark applications each time reading a different set of 6 hardware performance events out of 58. The values of all hardware performance events are averaged over 50 iterations.

Apart from hardware performance events inside the cores, the memory controller exposes a set of memory-mapped

performance counters that report: (1) the number of DDR cycles elapsed, (2) the number of busy DDR cycles, the total number of memory accesses in terms of (3) reads and (4) writes, and the total number of bytes transferred in (5) read and (6) write transactions.

All combined, we measured a total of 128 hardware performance events from the system, 64 for each core. However, incorporating all available features into the machine learning algorithm does not necessarily generate the best analytical model. In fact, the prediction error can become large in case of severe multicollinearity as it increases the variance of the regression coefficients, making them unstable. This issue of correlation may also arise in our setting because different features theoretically have an impact on each other. One example of such a case is that the number of branches executed is related to the total number of instructions executed, which in turn have relation to L1 instruction cache access.

To address such concerns while also making the learning formulation more efficient, we apply a systematic feature selection process. The reduction of features is accomplished in three steps. First, features consistently reporting a value of 0 are eliminated as their contribution to the analytical model is insignificant. We discover 26 such hardware performance events with zero numeric values for all of our applications.

Second, we computed the correlation matrix using Spearman rank-order correlations of the remaining features between each other. On top of this computed correlation matrix, we performed hierarchical clustering and manually selected a threshold by visual inspection of the branch diagram to group our features into clusters and keep a single feature from each cluster. For example, L2 cache accesses, DDR busy cycles and non-cacheable external memory requests are clustered closely together. By visual inspection, we chose a threshold of 0.4, which reduced the overall feature set to 71.

Finally, we arrange these features in the order of their "importance" by using a permutation importance [14] technique with the Ridge Linear Regression [15] method. Permutation importance calculates the increase in the model's prediction error after permuting the features in the training set. A feature is "important" if changing its values raises the model error, because the model depended on the feature for the prediction. The impact of the number of ordered features on the mean absolute error between the measured and predicted instructions per sample is calculated and the set of features with minimum mean absolute error is selected. This resulted in 14 final features per core being selected for prediction as shown in Table III. It is critical to address multicollinearity before applying permutation importance, because otherwise the results will show that none of the features are "important".

B. Interference-Aware Run-time Prediction

We evaluate four different linear regression models: Ridge [15], Lasso [16], Elastic-Net [17] and Linear Support Vector Regression (SVR) [18], which is SVR with a linear kernel. For this particular experiment, we use the measured instructions $i^{'}$ in interference rather than the predicted instructions i^{pred} for sample alignment to avoid prediction error propagation to the next sample. Fig. 2 compares the mean absolute error between the measured and predicted instructions per sample for these models. The ridge method has the minimum mean absolute error of 2.8% and 2.2% for core 1 and core 2, respectively.

TABLE III
SET OF 14 FEATURES SELECTED FOR EACH CORE.

Instructions	L1 instruction cache access
DRAM write-bytes	L1 instruction TLB refill
DRAM read-bytes	Load instructions
DRAM busy-cycles	Linefill because of prefetch
Conditional branches	Read allocate mode
Unconditional branches	L2 Data cache access
Exception taken	pipeline stall because the store buffer is full

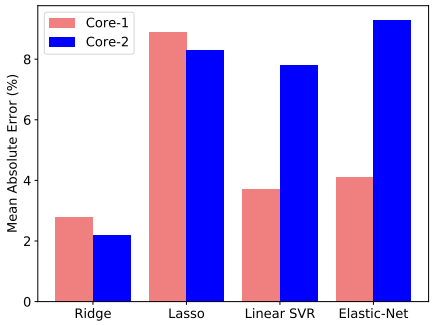


Fig. 2. Comparison of different linear regression models

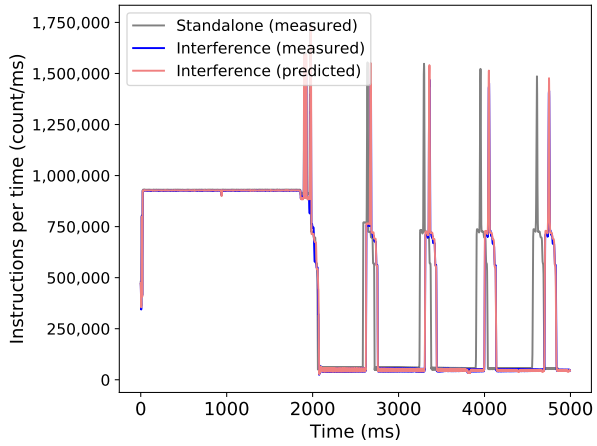
Table IV shows the summary of various combinations of training and test sets, and the corresponding run-time of the test set in standalone and interference scenarios, along with predicted run-time in interference from Algorithm 1 using Ridge Linear Regression as the predictive model. When the *disparity* and *tracking* applications are co-run for 5,000 ms, the corresponding run-time in the standalone scenario is significantly smaller, at 4,666 ms and 4,878 ms, respectively. By using the Interference-Aware Run-time Predictor (Algorithm 1), we are able to predict the run-time with an average percentage error as low as 0.3% and 0.1% for two co-running applications, respectively.

It is worth to note that the average prediction error in Table IV (e.g. 2.2% for *disparity*) can at times be higher the mean absolute prediction error (e.g. -3.6% for *disparity*) shown in Fig. 2. This is due to the propagation of prediction errors, which introduces additional inaccuracy into the sample alignment algorithm (Algorithm 2).

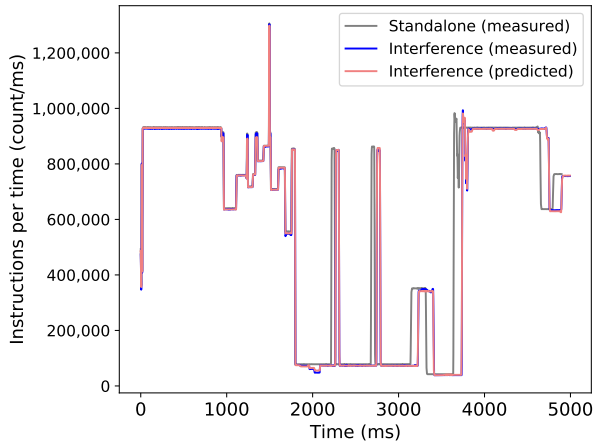
Fig. 3 shows per sample execution characteristics of the two co-running test set applications (*disparity* and *tracking*). It can be observed that the two applications start to suffer from interference and a delay in execution time roughly after 2,000 ms, which is also the portion where the instructions per time of both applications falls to extremely low values. It is also worth noting that the measured and predicted interference values are following a similar pattern, which shows the accuracy of our approach.

TABLE IV
COMPARISON OF MEASURED AND PREDICTED TOTAL RUN-TIME OF APPLICATIONS USING THE RIDGE LINEAR REGRESSION

Training Set	**Test Set**		**Standalone (measured)**		**Interference (measured)**		**Interference (predicted)**	
	Core 1	*Core 2*	*Core 1 (ms)*	*Core 2 (ms)*	*Core 1 (ms)*	*Core 2 (ms)*	*Core 1 (ms)*	*Core 2 (ms)*
multicut, *mser*, *stitch*, *sift*	*disparity*	*tracking*	4666	4878	5000	5000	4818 (-3.6%)	4994 (-0.1%)
stitch, *sift*, *disparity*, *tracking*	*multicut*	*mser*	4912	4898	5000	5000	4981 (-0.4%)	4943 (-1.1%)
disparity, *tracking*, *multicut*, *mser*	*stitch*	*sift*	4952	4985	5000	5000	4986 (-0.3%)	4997 (-0.1%)



(a) Core 1 (*disparity*)



(b) Core 2 (*tracking*)

Fig. 3. Comparison of measured and predicted per sample run-time in interference along with standalone values

V. SUMMARY AND CONCLUSIONS

In this paper, we presented a novel learning based approach to predict interference-aware run-time. The key component of our approach consists of the use of a sample alignment algorithm and a formulation of interference prediction as a machine learning problem. We used hardware performance events present inside the cores as well as the memory controller to extract key features used for prediction. The training data is then utilized to generate a predictive model that can predict the performance of previously unseen applications in interference scenario. By picking the 28 most important features out of 128 and employing the most effective model, Ridge Regression, we were able to achieve an average run-time prediction error of less than 0.3%, and a mean absolute per-sample prediction error of less than 3% on two cores.

ACKNOWLEDGEMENT

This work has received funding from the European Unions Horizon 2020 research and innovation programme under grant agreement No 871669.

REFERENCES

[1] R. Cavicchioli *et al.*, "Memory interference characterization between CPU cores and integrated GPUs in mixed-criticality platforms," in *ETFA*, 2017.
[2] X. Zheng *et al.*, "Learning-based analytical cross-platform performance prediction," in *SAMOS*, 2015.
[3] X. Zheng *et al.*, "Accurate Phase-Level Cross-Platform Power and Performance Estimation," in *DAC*, 2016.
[4] X. Zheng *et al.*, "Sampling-based binary-level cross-platform performance estimation," in *DATE*, 2017.
[5] A. Saeed *et al.*, "Machine Learning Based Cross-Platform Runtime," in *ECRTS, Waters Workshop*, 2019.
[6] C. Mendis *et al.*, "Ithemal:Accurate, Portable and Fast Basic Block Throughput Estimation using Deep Neural Networks," in *ICML*, 2018.
[7] S. K. Venkata *et al.*, "SD-VBS: The San Diego Vision Benchmark Suite," in *IISWC*, 2009.
[8] N. Binkert *et al.*, "The gem5 simulator," *SIGARCH Computer Architecture News*, vol. 39, no. 2, 2011.
[9] T. Carlson *et al.*, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *SC*, 2011.
[10] P. Magnusson *et al.*, "Simics: A full system simulation platform," *IEEE Computer*, vol. 35, no. 2, 2002.
[11] K. Singh *et al.*, "Predicting parallel application performance via machine learning approaches," *Concurrency and Computation: Practice and Experience*, 2007.
[12] "S32V Vision and Sensor Fusion Evaluation Board."
[13] "ARM Cortex-A53 MPCore Processor - Technical Reference Manual."
[14] A. Fisher *et al.*, "Model Class Reliance: Variable Importance Measures for any Machine Learning Model Class, from the "Rashomon" Perspective," in *arXiv Prepr*, 2018.
[15] D. Marquardt *et al.*, "Ridge Regression in Practice," *American Statistician - AMER STATIST*, 1975.
[16] R. Tibshirani, "Regression shrinkage and selection via the lasso," *Journal of the Royal Statistical Society Series B*, 1996.
[17] H. Zou *et al.*, "Regularization and variable selection via the elastic net," *Journal of the Royal Statistical Society Series B*, 2005.
[18] D. Basak *et al.*, "Support Vector Regression," *Neural Information Processing Letters and Reviews*, 2007.