Machine Learning Based Cross-Platform Runtime Prediction

Ahsan Saeed, Falk Wurst, Michael Pressler, Arne Hamann, Dakshina Dasari
Robert Bosch GmBH
Email: {fixed-term.Ahsan.Saeed,Falk.Wurst, Michael.Pressler, Arne.Hamann, Dakshina.Dasari}@de.bosch.com

Abstract—**Early performance prediction and evaluation of applications on newer platforms is a crucial step and yet a key challenge in embedded system design. Given that application performance is highly dependent on the middleware, basic software and hardware-dependent optimizations, simply executing them on cycle accurate models is not sufficient for early performance prediction. In this work we tackle this challenge with a machine learning based approach by leveraging performance counters provided by the platform and using them as features in our model. Our model is trained to correlate the performance data for a set of benchmarks applications between the host and the target platform and then later the trained model is used to predict the performance of newer applications on the target platform. We highlight the shortcomings of existing approaches and tools in extracting these features and propose a new library for the same. Additionally we systematically identify that subset of all available features that have a higher impact on the application's runtime. We also present our observations on how the granularity of code instrumentation affects the derived runtime values. The efficiency of our approach is validated by predicting the average runtime of a new application on an identical as well as a new target platform.**

I. Introduction

Early performance prediction of applications is gaining importance given the increasing complexity of processing platforms and the need to select an efficient target platform early in the system design phase. However, selecting such a suitable target platform while considering the implications on factors like runtime, power and memory usage is non-trivial. In particular, runtime estimation in the absence of upcoming commercial-of-the-shelf hardware platforms is very difficult, given its dependence of numerous platform specific microarchitectural effects. While cycle accurate models seem like a good candidate for early performance estimation, they present some drawbacks; they can be too slow when an accurate estimation of the runtime is required and more so in the case when there is a need to estimate the performance of a wide spectrum of applications. A major blocker for evaluating real application on multiple target platforms are the dependencies of existing applications to middleware, basic software and even hardware-dependent optimizations. Porting complete applications to a new platform for performance evaluation is not suitable in an integration platform use-case were we not only have to integrate our software but also integrate software from other suppliers and the customer.

A promising alternative is the usage of machine learning techniques to estimate the runtime on a target platform [1]–[3], wherein the application is executed on a host platform, hardware performance events (features) are extracted on the host and then a machine learning model is trained on the target platform by computing a relation between the features of the host and the target platform to estimate the runtime of an application on the target. In addition to the implementation itself, OS, compiler etc. platform specific effects like pipeline hazards, instruction-set architecture, branch mispredictions, cache-misses, etc. influence the application runtime. Modern processors provide a set of programmable hardware performance counters in order to count the occurrence of platform specific hardware performance events.

The desired outlook is that by providing a set of small training benchmarks to a semiconductor vendor, and retrieving the hardware performance events, a machine learning network can be trained to predict the runtime for larger applications and therefore assist the early stages of the target selection process. The training data for smaller benchmarks can also be derived using cycle accurate models.

a) Contributions: In this work, we firstly propose a more accurate instrumentation approach to extract the hardware performance events. Secondly we overcome some of the limitations of the earlier work carried out in [1]–[3] such as either applications source code or binary executable is supported for runtime prediction but not both. We further use a methodology which allows us to systematically explore all available hardware events and choose the subset that have a higher impact on the runtime of an application; this is essential to reduce the vast exploration space arising due to the presence of numerous hardware performance events. We demonstrate the applicability of our approach for predicting the runtime when the host and target platform are i) identical ii) differ in the memory subsystem (including caches) and the core organization.

Since the worst-case execution time on a multi-core platform is highly dependent on the co-running applications and the associated interference effects, we currently only predict the runtime of an application, running on an operating system (Linux in our case) in isolation.

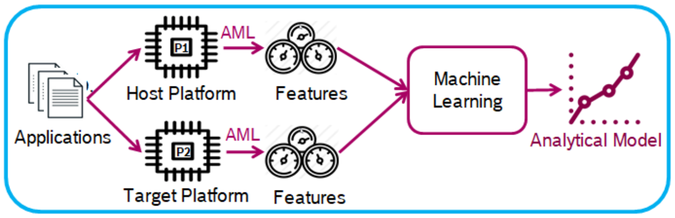
II. Related work

Learning based approaches have recently emerged for performance prediction of different architectures [1] which tends to be simpler and faster than simulation based approaches such as cycle accurate models. However, they have a non-negligible prediction error of more than 40% for small embedded benchmarks. Phase level instrumentation approach [2] uses LLVM compiler infrastructure to instrument at intermediate representation (IR) basic block level of each application during compilation. This provides fine grain performance prediction of programs and tends to reduce the prediction error to below 3%. However, upon reproducing the same approach, our analysis shows an overhead of more than 300% in the instrumented values irrespective of the phase granularity. Moreover, the approach requires availability of application source code. Further details can be found in the subsection IV-B.

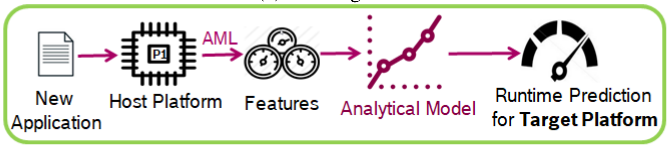
In order to overcome these mentioned problems, sampling based technique [3] was introduced in which the instrumentation is performed at fixed time intervals of an application. One of the key challenges in this approach is sample alignment between the host and target platforms during training as they correspond to different section of executed code. The paper proposed a solution to this problem by introducing stochastic dynamic coupling (SDC) heuristic. However we think the matching can be quiet challenging and could lead to inaccuracies in case the executed code differs a lot due to different libraries or additional execution units in a modern microprocessor. All the mentioned approaches are further discussed in section IV-B.

III. Proposed Approach

Our proposed machine-learning based approach is separated into the training phase and the prediction phase.



(a) Training Phase



(b) Prediction Phase

Fig. 1: Machine learning based cross-platform runtime prediction framework

Training phase: As seen in Fig. 1a, applications are executed one at a time on the host and target platforms to extract their respective features using Application Measuring Library (AML) described in subsection IV-A. In this scenario, features are set of hardware performance events which are representative of source code or binary file of an application such as instructions retired, branch predicted, L1 cache access, L2 cache access, etc. These features are then used for training statistical machine learning model in order to compute the model coefficients which forms the basis of our analytical model. We use ridge regression as the statistical machine learning model as it tends to show better results when the data suffers from multicollinearity that is independent variables are highly correlated. On completion of training phase, an analytical model is generated which essentially deduces the association of applications executed in target platforms to the one executed in host platform.

Prediction Phase: In this phase shown in Fig. 1b, a new application which has not been used for training is executed and instrumented only on the host platform. The retrieved features are then fed into the analytical model which estimates runtime of the application on the target platform.

IV. Feature Extraction

Features of an application are extracted by using hardware performance counters which are special purpose registers provided by the processor to store the counts of hardware related activities or performance events. These counters are available per core for our host and target platform. There are also events related to kernel activities that are not directly measured by hardware performance counters.

Many tools and libraries such as Linux Perf [6], PAPI [4] and Likwid [5] are available to read these performance counters. Instrumenting these events is primarily achieved in different ways i) leverage the perf_event_open system call which is part of the Linux kernel since 2.6+. Examples of tools that follow this method are Linux Perf and PAPI. ii) implementing your own kernel module or iii) in a bare-metal scenario, assembly code manipulating the right control registers, can be executed in the supervisor, thereby bypassing any kernel functionality.

However the problem with existing tools is that associated overheads and memory footprint are high. Additionally existing tools need to be installed separately and can either be used for instrumenting source code or binary files. To overcome these limitations, we propose a more efficient approach, called Application Measuring Library (AML) in Section IV-A. Even when we have an accurate instrumentation method large overheads can be introduced depending on the granularity of the instrumentation(see Section IV-B).

A. Application Measuring Library (AML)

AML is a specialized library written for efficiently reading hardware performance counters using the perf_event_open system call. The proposed AML has a small code size and memory footprint resulting in low overhead. It can be used for instrumenting portion of code or whole program as well as the binary executable.

1) Example usage of AML: In order to instrument a source code, it is required to initialize instrumentation by providing the list and number of hardware performance events (available in processor's datasheet [14]).

Listing 1: Example AML usage

```
int main(){
uint64_t event_list[] = {0x08, 0x04, 0x03};
int fd = instrument_start(event_list, 3);
<your code>
instrument_stop(fd);
}
```

A basic instrumentation outline is shown in Listing 1. The *instrument_start* is a wrapper that initializes the underlying kernel structure with options like which events to monitor, enable per core monitoring, etc., and sets the counter running.

Likewise, the counter values are finally assimilated in the *instrument_stop* function.

2) Comparison of AML with other tools: We perform various experiments in order to compare the performance of existing tools with AML. In the first experiment we calculated the minimum and maximum deviation of total cycles for instrumentation start followed by immediate instrumentation stop sequence of PAPI [4] and AML over 1,000,000 iterations. For PAPI, these values are obtained by using the PAPI built-in utility papi_cost whereas for AML, the instrumentation start/stop sequence is executed 1 million times in a for-loop. From Table I, it can been seen that the variation in minimum to maximum cycles reported by PAPI are over 120 times more than that of AML.

	Minimum Cycles	Maximum Cycles
PAPI	13,328	73,304
AML	65	580

TABLE I: Comparison of overhead between PAPI and AML

Furthermore, we investigated the accuracy of prominent hardware performance event values by using Linux Perf [6] tool and AML. In order to do so, we instrumented the small application in Listing 2. We observed that the Linux Perf is unable to instrument small applications accurately and the difference in values as compared to AML are non-negligible shown in Table II. Furthermore, it highlights the fact that AML requires negligible amount of instructions to instrument an application. The large overhead of Linux Perf can be attributed towards the high number of internal operations performed in order to make the utility usable for several general tasks such as software profiling and tracing. In order to observe if the overhead is constant, we perform another experiment by using a standard benchmark application *bitcount* from [9] which is comparatively a larger application. The instrumentation values from this benchmark application Table II shows that the difference in reported cycles by the two tools is only 1%. However, the L1 data cache miss, L2 cache access and L2 cache miss reported are drastically different and cannot be neglected.

The values in Table I and Table II use the target platform with the configurations described in Section VI-B.

B. Instrumentation Granularity

Instrumentation of an application is possible at i) program level (after the complete execution of an application [1]) ii) phase level (after intermediate representation (IR) basic block level [2]) iii) fixed time intervals [3] or iv) fixed number of instructions retired event which is detected using the hardware counter overflow interrupt.

Listing 2: Small Application

```
int main(){
int a = 1;
return 0;
}
```

Although phase level instrumentation provides fine-grained information regarding an application execution, it does so at the cost of higher instrumentation overhead. On comparing the overhead of instrumenting the *bitcount* benchmark [9] at the program level and the phase level, there was an increase of 300-700% in total cycles consumed by the phase level approach as seen in Fig. 2. This is attributed to the addition of instrumentation function calls after each IR basic block using the LLVM compiler infrastructure. For example when the granularity is 1000, the instrumentation is performed after execution of 1000 basic blocks, however, the instrumentation function is still called after every basic block in order to check granularity condition of whether 1000 basic blocks have been executed. Moreover it can been seen in the Fig. 2, the reduction in granularity results in further increase of overhead due to the increased instrumentation needed.

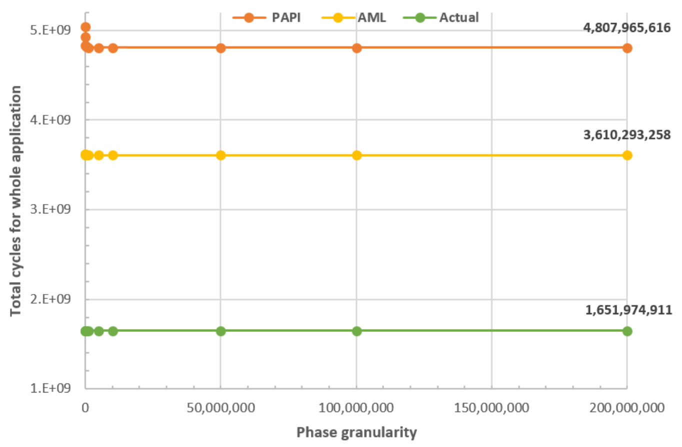


Fig. 2: Comparison of program level and phase level instrumentation using different tools

A sampling based approach [3] was introduced to overcome the problems caused by phase level instrumentation. In this approach, the hardware performance counters are read at fixed time intervals. However, a challenge of this approach is the one to one correspondence of host and target platforms samples as they correspond to different sections of the executed code due to different execution times of the two platforms. The paper [3] proposed a solution to this problem by introducing stochastic dynamic coupling (SDC) heuristic assuming that total instructions for an application in two architectures are micro-architecture independent and dependent only on the Instruction Set Architecture (ISA). However, the approach is complex and can introduce inaccuracies in the measured values after alignment. In case special instructions (e.g. SIMD) are only available in one platform, the code may drift away during execution, leading to inaccuracies. Therefore, after reviewing all the possible granularities we concluded to use instrumentation of applications at the entire program level together with the assumption that the analytical model is able to predict larger applications based on the information learned from several smaller applications.

V. Feature Selection

Incorporating all available features into the machine learning algorithm does not necessarily generate the best analytical

Application	Tool	Cycles	Instructions	Branches	L1D_CA	L1D_CM	L2_CA	L2_CM
Code in Listing 2	Linux Perf	1,269,511	477,096	82,977	211,552	41,778	46,166	4,393
Code in Listing 2	AML	253	32	6	12	3	9	2
MiBench Bitcount	Linux Perf	1,667,608,360	1,401,635,278	138,813,834	762,541,168	10,928	161,008	7,228
MiBench Bitcount	AML	1,650,329,450	1,396,698,720	137,814,203	760,505,252	233	1,405	160

TABLE II: Comparison of PMU event values between Linux PERF and AML

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 multicollinearity can also possible exist in our scenario as multiple features theoretically have impact on each other. One example of such a case is number of branches executed is related to total number of instructions executed which on turn have relation to L1 instruction cache access.

The setup described in Section VI-B comprises ARM Cortex A53 cores [14] which have 58 measurable events, but provides only 6 32-bit hardware performance counters and a dedicated 64-bit cycle counter. This adds to another level of complexity as we can only read 7 out of total 58 hardware performance events [14] at any given time. We solve this limitation by re-executing the same benchmark application in isolation, each time reading 6 different set of hardware performance events. The small size of the training applications, execution in isolation and using AML enables insignificant variation in measured values and thus they can correspond to each other as if they are read at the same instance.

Furthermore, it is important to understand the impact of prediction accuracy on the number of features used for generating analytical model. In order to do so, we propose an *Exhaustive search technique* to learn and identify the hardware performance events that best reflects the runtime of applications. The exhaustive search technique involves the use of every possible combination of hardware performance events as input and fixing the event to be predicted as output in to the machine learning model (ridge regression algorithm). Thus, one of the key challenge is the enormous computation time required to compute all the combinations. Therefore the goal is to reduce the number of possible combinations.

a) Feature reduction: The reduction of features are done in two-step process. Firstly, we computed the correlation matrix in Fig. 3 of the features which is a standard practice to find the level of dependency and association among all features. Multicollinearity can result in increases of prediction error, therefore, we can eliminate one of the features whose correlation coefficient is close to 1 with the combination of expert knowledge [13]. For example the correlation coefficient is 1 between L1 data cache refill and L2 cache access shown in Fig. 3, but we cannot remove it for our hardware setup (described in subsection VI-B) in which each core has a private L1 data cache private whereas the L2 cache is shared. Thus, these two hardware performance events provide different insights into the executing application and therefore cannot be eliminated. However, the bus cycles event whose correlation coefficient is also 1 shown in Fig. 3 can be eliminated as the occurrence of this event is exactly half the occurrence of total cycles event because the bus frequency is half the CPU frequency. This implies that the event *bus cycles* provides redundant information and need not be used as input feature to the machine learning algorithm as long as the event *cycles* is provided as output. In this way at the first step of feature reduction process, we eliminated 12 features.

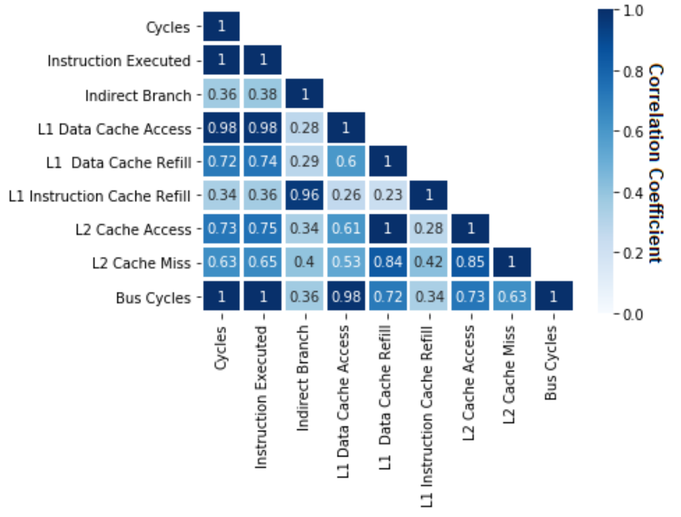


Fig. 3: Heatmap showing the correlation coefficient values of selected hardware performance events with each other.

Secondly, features reporting a value of 0 are also eliminated as their contribution to the analytical model is insignificant. We find 13 hardware performance events showing zero numeric values for all our applications (described in subsection VI-A) and hence they are removed from the feature set. After following these two steps, we are left with only 33 out of the 58 available features which reduces the computation time required for exhaustive search technique by a factor of 78. We can now continue by applying the exhaustive search technique on the reduced set of these 33 features.

VI. Evaluation of Runtime Prediction

A. DataSet

We use applications written in C and C++ taken from various embedded benchmarks (MiBench [9], Mälardalen WCET benchmark [10]) and solutions of various programming contests (ACM-ICPC International Collegiate Programming Contest [11]). In total, we use 226 applications where 202 are used for training and the rest for prediction. We use smaller programs (taking 25 seconds on average on the host platform) for the training set and longer programs (taking 600 seconds on the host platform) for the final test set. The idea is that the longer applications can be predicted based on the learned information from several smaller applications.

B. Setup

To evaluate our approach we use a Raspberry Pi 3B [8] as the host platform and the NXP S32V234 [12] as the target platform. Both platforms have 4 ARM-Cortex A53 cores [14] with different clock frequencies. They differ in the core organization and the memory hierarchy. The Raspberry Pi 3B has one cluster consisting of 4 cores each with their private L1 data and instruction caches and a shared L2 cache. The NXP S32V234 on the other hand is organized into 2 clusters with 2 cores in each cluster. Each core within a cluster has its own private L1 data and instruction cache and the 2 cores share a L2 cache. In our case the observed application and the OS are running together in one cluster on different cores. The host platform runs a 64 bit openSUSE Linux OS while the target platform runs 64 bit Linux Board Support Package (BSP) version 19.0 provided officially by NXP. Applications are cross-compiled using the gcc-linaro-6.3.1-2017.05 targeted for aarch64-linux-gnu.

C. Prediction on the Same Platform

As a first test, we considered the problem of predicting the execution time (cycles) on the same platform (Raspberry Pi). The analytical model is computed by executing 202 training applications and later tested with 24 test applications. Only 33 selected features (hardware performance events) are extracted from these applications using AML at program level granularity. Cycles count of the platform is used as output in the machine learning model (ridge regression algorithm) while the input features are identified using the exhaustive search technique that results in the least observed worst case prediction error for the test applications.

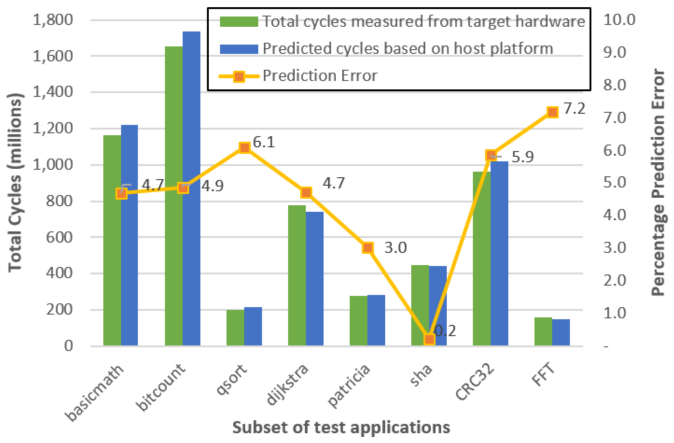
Runtime prediction results for subset (Mibench [9]) of test applications are shown in Fig. 4a. By using only six features which are the result of the exhaustive search technique shown in Table 4c, an average error of 3.8% for all the test applications is obtained. By increasing the number of features used for generating the analytical model, the observed worst case percentage prediction error can be reduced as shown in Fig. 4b. The main reason for prediction improvement is due to the availability of more information we incorporate in the form of features during the training phase. This results in the generation of a more generalized analytical model that improves the prediction of new and previously unseen applications. Furthermore, it highlights the fact that it is reasonable to use hardware performance events as feature representative of the application.

Table 4c shows one of the best combinations of input features while fixing the number of hardware performance event to six. Furthermore, it can be observed that the obtained combination consist of a mixture of prominent and some unexpected (such as event numbers 0xE4, 0xE5, 0xE7) hardware performance events, which are not previously used in machine learning based techniques [1]–[3].

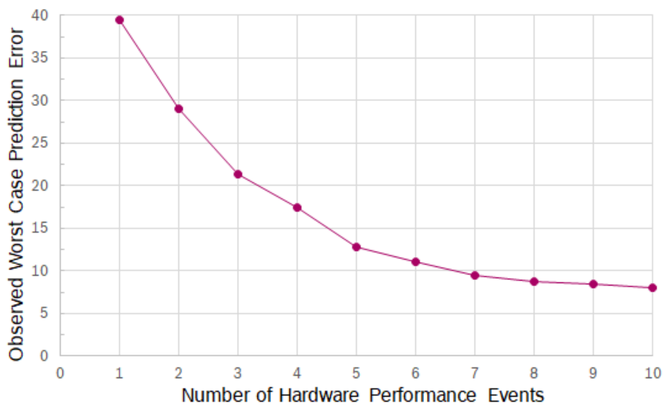
D. Cross Platform Runtime Prediction

We next predict the runtime (cycles) on the target (NXP S32V234) while running the application only on the host (Raspberry Pi). We follow the same methodology as used earlier except that we use all the 32 features plus cycles on the host as input and the cycles on the target as output of the ML algorithm. Features are extracted using AML at program level and subsequently selected using the exhaustive search technique described in Section V.

Cross-platform runtime prediction results for subset (Mibench [9]) of test applications are shown in Fig. 5a with an average error for all the test application to be 1.1%. From Fig. 5b, it can be observed that the prediction results for cross platform shows similar pattern as for same platform Fig. 4b. However, the observed worst case prediction error is much less than as that of same platform due to the availability of an additional input feature (total number of cycles executed in the host platform). Cycles are the most comprehensive repre-



(a) Runtime prediction for individual test applications by using only 6 hardware performance events for subset of test applications

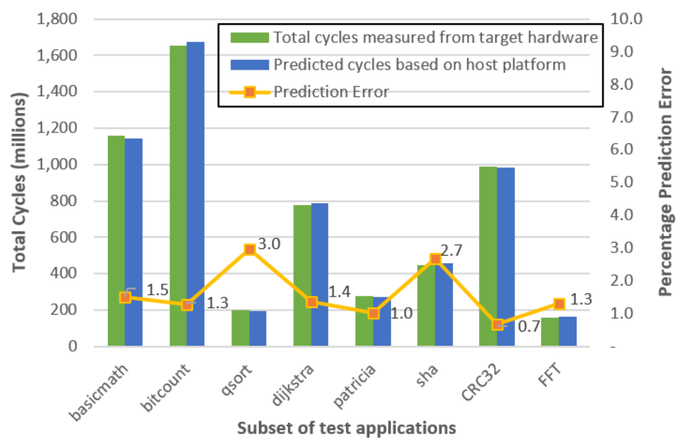


(b) Observed worst case runtime prediction error over all the test applications

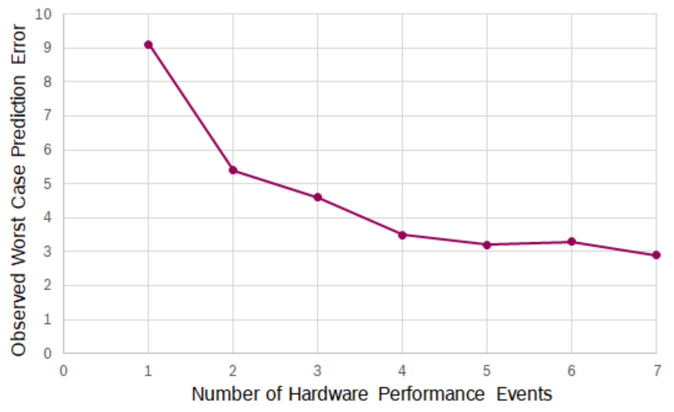
Event Number	Event mnemonic
0x07	ST_RETIRED
0x12	BR_PRED
0x14	L1I_CACHE
0xE4	-
0xE5	-
0xE7	-

(c) Best observed combination of input features for six events

Fig. 4: Same platform results



(a) Runtime prediction for individual test applications by using only 6 hardware performance events plus one cycle event for subset of test applications



(b) Observed worst case runtime prediction error over all the test applications

Event Number	Event mnemonic
0x11	CPU_CYCLES
0x02	L1I_TLB_REFILL
0x86	EXC_IRQ
0x0C	PC_WRITE_RETIRED
0XC0	-
0XCA	-
0xE7	-

(c) Best observed combination of input features for seven events

Fig. 5: Cross platform results

sentative of an application and incorporates all the information about an application execution state. Therefore, we can also observe in Table 5c a different combinations of input features to the one shown in Table 4c.

VII. Limitations and Future Work

At present, we cross compile our application using the same compiler with same configurations. Since the efficiency of the language implementation or a different compiler directly affects the number and combination of instructions and the resulting runtime, we plan to extend our learning model to applications compiled with a different compiler version or implemented in another language like Python or Java. Another direction is to predict parameters other than the runtime like memory bandwidth utilization, cache behaviour etc. Further currently, the prediction is made for single application executing in isolation in the multicore domain. However, we plan to extend our work for the prediction of multiple concurrent applications on the target platform by providing additional information of per core performance events during the training phase.

VIII. Conclusion

This paper discuss the limitation of previous approaches and presents a novel machine learning based cross-platform runtime prediction methodology to circumvent these flaws. The key components of our approach consists of the use of AML for feature extraction, the two-step elimination process for feature reduction and exhaustive search techniques for final feature selection. Our specialized AML tooling extracts the features of an application with negligible overhead and minimum disturbance on the state of a hardware platform. The two-step elimination process adopted for features reduction decreased the total feature space from 58 to 33. This helped in the reduction of computation time for exhaustive search technique, thus making it possible to identify the key features and quantify the impact of number of features on the prediction accuracy. Hence, by using only 7 features (six hardware performance events in addition to cycles event) of the host platform as input features (which are also within the hardware limitation), we are able to predict the runtime of an application on the target platform with an average prediction accuracy of 98.9% and observed worst case prediction accuracy of 96.7%.

References

[1] X. Zheng et al. Learning-based analytical cross-platform performance prediction. In SAMOS, 2015.

[2] X. Zheng et al. Accurate phase-level cross-platform power and performance estimation. In DAC, 2016.

[3] X. Zheng et al. Sampling-Based Binary-Level Cross-Platform Performance Estimation. In DATE, 2017

[4] Moore, S., Terpstra, D., London, K., Mucci, P., Teller, P., Salayandia, L., Bayona, A. and Nieto, M., 2003, June. PAPI deployment, evaluation, and extensions. In User Group Conference, 2003. Proceedings (pp. 349-353). IEEE.

[5] Treibig, J., Hager, G. and Wellein, G., 2010, September. Likwid: A lightweight performance oriented tool suite for x86 multicore environments. In Parallel Processing Workshops (ICPPW), 2010 39th International Conference on (pp. 207-216). IEEE.

[6] De Melo, A.C., 2009, September. Performance counters on Linux. In Linux Plumbers Conference.

[7] RAMspeed/SMP. https://github.com/rkreutz/cache/tree/master/ramsmp-3.5.0

[8] Raspberry Pi 3B single board computer. https://www.raspberrypi.org/products/raspberry-pi-3-model-b/

[9] 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 IISWC, 2001.

[10] Mälardalen WCET benchmarks homepage. http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

[11] The ACM-ICPC International Collegiate Programming Contest. http://icpc.baylor.edu/.

[12] S32V Vision and Sensor Fusion Evaluation Board. https://www.nxp.com/products/processors-and-microcontrollers/arm-based-processors-and-mcus/s32-automotive-platform/s32v-vision-and-sensor-fusion-evaluation-board:SBC-S32V234

[13] ARM Architecture Reference Manual ARMv8-A, https://developer.arm.com/docs/ddi0487/latest/arm-architecture-reference-manual-armv8-for-armv8-a-architecture-profile

[14] ARM Cortex-A53 MPCore Processor - Technical Reference Manual http://infocenter.arm.com/help/topic/com.arm.doc.ddi0500j