Supplemental: Improving the Accuracy of Energy Predictive Models for Multicore CPUs Using *Additivity* of Performance Monitoring Counters

Arsalan Shahid[1][0000−0002−3748−6361], Muhammad Fahad[1][0000−0002−3595−8484], Ravi Reddy Manumachu[1][0000−0001−9181−3290], and Alexey Lastovetsky[1][0000−0001−9460−3897]

School of Computer Science, University College Dublin, Belfield, Dublin 4, Ireland
{`arsalan.shahid,muhammad.fahad`}`@ucdconnect.ie`,
{`ravi.manumachu,alexey.lastovetsky`}`@ucd.ie`

1 Three Energy Predictive Models

We select three predictive models for our experiments: 1). Linear Regression Model (*LM*), 2). Random Forest (*RF*), and 3). Neural Networks (*NN*). They are explained as below:

Linear Regression Model A *LM* can be represented as $Y_i = \sum_{j=0}^{M} \beta_j X_{ij} + \epsilon_i$. where, $i = 1, 2, ..., N$ represent the number of observations and $j = 1, 2, ..., M$ represent the number of independent variables. In our case, Y_i are dynamic energy measurements from HCLWattsUp X_i are PMCs. ϵ is the error term or it can be considered as fault in measurement. We solve *LM* using regression technique by estimating the value of coefficients, i.e., β.

Since dynamic energy attributes the energy consumption of an application run, we restrict our *LM* to have only positive coefficients and zero intercept. These restrictions are realized from the energy conservation of computing.

Random Forest A *RF* technique is a supervised learning algorithm considered for its accuracy in classification as well as regression based tasks [3]. It uses decision tree based approach to train on a data-set and outputs mean prediction from individual trees. It is a non-linear model build by constructing many linear boundaries. The overall non-linearity is because a single linear function can not be used to classify and regress on each iteration of the decision tree.

We apply the *RF* based regression to our data-set and determine its accuracy when applied on *additive* and *non-additive* PMCs in different experimental configurations.

Neural Networks A *NN* contains an interconnected group of nodes inspired by neurons of a human brain and each node computes weights and biases to give an

Table 1. Neural Network Parameters

Network type	Feed-forward back propagation
Input parameters	PMCs
Output parameters	Dynamic energy
Training algorithm	Bayesian regularization
Performance function	Mean-Squared Error (MSE)
Number of neurons	10
Network layers	2
Transfer function	Linear

output prediction. We observe that the input data-sets best fits a linear function so we set it as the transfer function. The learning function is bayesian regularization that gives optimal regularization parameters in an automated fashion [1]. Bayesian regularization updates the weight and biases by using Levenberg-Marquardt [4], which is an algorithm to train the *NN* upto 100 times quicker in comparison with the commonly used gradient-descent and back-propagation method. The *NN* training parameters are given in Table 1. We keep these parameters to train the network as they prevent an over-fit or an under-fit.

2 Rationale Behind Using Dynamic Energy Consumption Instead of Total Energy Consumption

We consider only the dynamic energy consumption in our problem formulations and algorithms for several reasons, which are listed below:

1. Static energy consumption is a hard constant (or a inherent property) of a platform that can not be optimized. That is, it does not depend on the application configuration and will be the same for different application configurations.
2. Although static energy consumption is a major concern in embedded systems, it is becoming less compared to the dynamic energy consumption due to advancements in hardware architecture design in HPC systems.
3. We target applications and platforms where dynamic energy consumption is the dominating energy dissipator.
4. Finally, we believe its inclusion can underestimate the true worth of an optimization technique that minimizes the dynamic energy consumption. We elucidate using two examples from published results.
   – In our first example, consider a model that reports predicted and measured total energy consumption of a system to be 16500J and 18000J, respectively. It would report the prediction error to be 8.3%. However, if it is known that the static energy consumption of the system is 9000J, then the actual prediction error (based on dynamic energy consumptions only) would be 16.6% instead.

– In our second example, consider two different energy prediction models (M_A and M_B) with same prediction errors of 5% for an application execution on two different machines (A and B) with same total energy consumption of 10000J. One would consider both the models to be equally accurate. But supposing it is known that the dynamic energy proportions for the machines are 30% and 60%. Now, the true prediction errors (using dynamic energy consumptions only) for the models would be 16.6% and 8.3% respectively. Therefore, the second model M_B should be considered more accurate than the first.

3 Steps to Ensure Reliable Experiments

To ensure the reliability of our results, we follow a statistical methodology where a sample mean for a response variable is obtained from several experimental runs. The sample mean is calculated by executing the application repeatedly until it lies in the 95% confidence interval and a precision of 0.025 (2.5%) has been achieved. For this purpose, Student's t-test is used assuming that the individual observations are independent and their population follows the normal distribution. We verify the validity of these assumptions by plotting the distributions of observations.

The server is fully dedicated for the experiments. To ensure reliable energy measurements, we took following precautions:

1. *HCLWattsUp* API [2] gives the total energy consumption of the server during the execution of an application using physical measurements from the external power meters. This includes the contribution from components such as NIC, SSDs, fans, etc. To ensure that the value of dynamic energy consumption is purely due to CPUs and DRAM, we verify that all the components other than CPUs and DRAM are idle using the following steps:
   - Monitoring the disk consumption before and during the application run. We ensure that there is no I/O performed by the application using tools such as *sar*, *iotop*, etc.
   - Ensuring that the problem size used in the execution of an application does not exceed the main memory, and that swapping (paging) does not occur.
   - Ensuring that network is not used by the application using monitoring tools such as *sar*, *atop*, etc.
   - Bind an application during its execution to resources using cores-pinning and memory-pinning.
2. Our platform supports three modes to set the fans speed: *minimum*, *optimal*, and *full*. We set the speed of all the fans to *optimal* during the execution of our experiments. We make sure there is no contribution to the dynamic energy consumption from fans during an application run, by following the steps below:

- We continuously monitor the temperature of server and the speed of fans, both when the server is idle, and during the application run. We obtain this information by using Intelligent Platform Management Interface (IPMI) sensors.
- We observed that both the temperature of server and the speeds of the fans remained the same whether the given application is running or not.
- We set the fans at *full* speed before starting the application run. The results from this experiment were the same as when the fans were run at *optimal* speed.
- To make sure that pipelining, cache effects, etc, do not happen, the experiments are not executed in a loop and sufficient time (120 seconds) is allowed to elapse between successive runs. This time is based on observations of the times taken for the memory utilization to revert to base utilization and processor (core) frequencies to come back to the base frequencies.

4 Brief overview of *SLOPE-PMC* and *AdditivityChecker*

SLOPE-PMC is developed on top of Likwid tool to automate the process of PMC collection. It takes an input application and operates in three steps. First, it identify the available PMCs on a given platform and list them in a file. In second step, the input application is executed several times as in a single invocation of an application only 4 PMCs can be collected. To ensure reliable results, we also take an average of each PMC count using multiple executions (atleast 3) of an application. In the final step, the PMCs are extracted with labels in a stats file. Figure 4 summarizes the work-flow of *SLOPE-PMC*.

Figure 4 describes the *AdditivityChecker* where it takes as an input: 1). PMCs of two base applications A and B, and a compound application (AB) composed of base applications and 2). user-specified tolerance in percentage. It returns a list of *additive* and *non-additive* PMCs along with their percentage errors.

Acknowledgement

This publication has emanated from research conducted with the financial support of Science Foundation Ireland (SFI) under Grant Number 14/IA/2474.

References

1. Foresee, F.D., Hagan, M.T.: Gauss-newton approximation to bayesian learning. In: Proceedings of the 1997 international joint conference on neural networks. vol. 3, pp. 1930–1935. Piscataway: IEEE (1997)
2. HCL: HCLWattsUp: API for power and energy measurements using WattsUp Pro Meter (2016), `http://git.ucd.ie/hcl/hclwattsup`
3. Liaw, A., Wiener, M., et al.: Classification and regression by randomforest. R news **2**(3), 18–22 (2002)

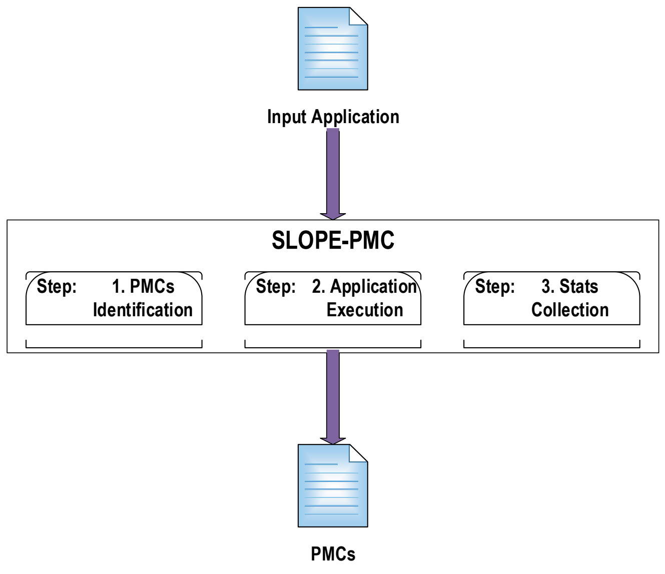
Input Application
SLOPE-PMC
Step: 1. PMCs Identification
Step: 2. Application Execution
Step: 3. Stats Collection
PMCs

Fig. 1. *SLOPE-PMC*: Towards the automation of PMC collection on Modern Computing Platforms

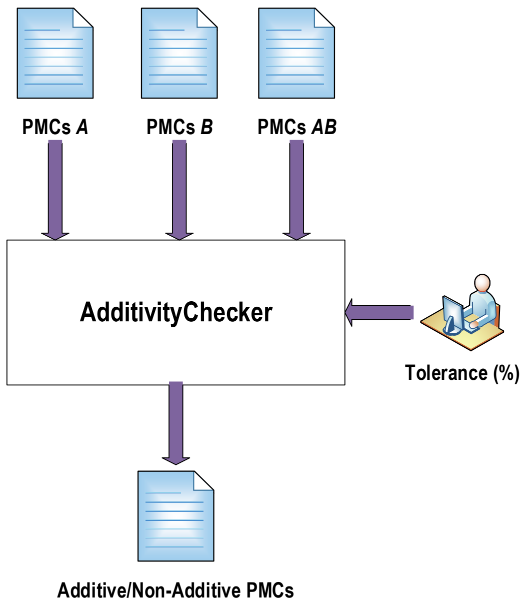

Fig. 2. *AdditivityChecker*: Test PMCs for *Additivity*

4. Moré, J.J.: The levenberg-marquardt algorithm: implementation and theory. In: Numerical analysis, pp. 105–116. Springer (1978)