Additivity: A Selection Criterion for Performance Events for Reliable Energy Predictive Modeling

Arsalan Shahid1, Muhammad Fahad1, Ravi Reddy1, Alexey Lastovetsky1

© The Authors 2017. This paper is published with open access at SuperFri.org

Performance events or performance monitoring counters (PMCs) are now the dominant predictor variables for modeling energy consumption. Modern hardware processors provide a large set of PMCs. Determination of the best subset of PMCs for energy predictive modeling is a non-trivial task given the fact that all the PMCs can not be determined using a single application run. Several techniques have been devised to address this challenge. While some techniques are based on a statistical methodology, some use expert advice to pick a subset (that may not necessarily be obtained in one application run) that, in experts’ opinion, are significant contributors to energy consumption. However, the existing techniques have not considered a fundamental property of predictor variables that should have been applied in the first place to remove PMCs unfit for modeling energy. We address this oversight in this paper.

We propose a novel selection criterion for PMCs called additivity, which can be used to determine the subset of PMCs that can potentially be used for reliable energy predictive modeling. It is based on the experimental observation that the energy consumption of a serial execution of two applications is the sum of energy consumptions observed for the individual execution of each application. A linear predictive energy model is consistent if and only if its predictor variables are additive in the sense that the vector of predictor variables for a serial execution of two applications is the sum of vectors for the individual execution of each application. The criterion, therefore, is based on a simple and intuitive rule that the value of a PMC for a serial execution of two applications is equal to the sum of its values obtained for the individual execution of each application. The PMC is branded as non-additive on a platform if there exists an application for which the calculated value differs significantly from the value observed for the application execution on the platform. The use of non-additive PMCs in a model renders it inconsistent.

We study the additivity of PMCs offered by the popular state-of-the-art tools, Likwid and PAPI, by employing a detailed experimental methodology on a modern Intel Haswell multicore server CPU. We show that many PMCs in Likwid and PAPI that are widely used in models as key predictor variables are non-additive. This brings into question the reliability and the reported prediction accuracy of these models.

Keywords: performance events, PMC, energy predictive models, Likwid, PAPI.

Introduction

Performance events or performance monitoring counters (PMCs) are special-purpose registers provided in modern microprocessors to store the counts of software and hardware activities. We will use the acronym PMCs to refer to software events, which are pure kernel-level counters such as page-faults, context-switches, etc. as well as micro-architectural events originating from the processor and its performance monitoring unit called the hardware events such as cache-misses, branch-instructions, etc. They have been developed primarily to aid low-level performance analysis and tuning. Remarkably while PMCs have not been used for performance modeling, over the years, they have become dominant predictor variables for energy predictive modeling.

Modern hardware processors provide a large set of PMCs. Consider the Intel Haswell multicore server CPU whose specification is shown in Tab. 1. On this server, the PAPI tool [19] provides 53 hardware performance events. The Likwid tool [17, 23] provides 167 PMCs. This

1School of Computer Science, University College Dublin, Dublin, Ireland
Table 1. Specification of the Intel Haswell Multicore CPU

<table>
<thead>
<tr>
<th>Technical Specifications</th>
<th>Intel Haswell Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Intel E5-2670 v3 @2.30GHz</td>
</tr>
<tr>
<td>OS</td>
<td>CentOS 7</td>
</tr>
<tr>
<td>Micro-architecture</td>
<td>Haswell</td>
</tr>
<tr>
<td>Thread(s) per core</td>
<td>2</td>
</tr>
<tr>
<td>Cores per socket</td>
<td>12</td>
</tr>
<tr>
<td>Socket(s)</td>
<td>2</td>
</tr>
<tr>
<td>NUMA node(s)</td>
<td>2</td>
</tr>
<tr>
<td>L1d cache</td>
<td>32 KB</td>
</tr>
<tr>
<td>L11 cache</td>
<td>32 KB</td>
</tr>
<tr>
<td>L2 cache</td>
<td>256 KB</td>
</tr>
<tr>
<td>L3 cache</td>
<td>30720 KB</td>
</tr>
<tr>
<td>Main memory</td>
<td>64 GB DDR4</td>
</tr>
<tr>
<td>Memory bandwidth</td>
<td>68 GB/sec</td>
</tr>
<tr>
<td>TDP</td>
<td>240 W</td>
</tr>
<tr>
<td>Idle Power</td>
<td>58 W</td>
</tr>
</tbody>
</table>

includes events for uncore and micro-operations (µops) of CPU cores specific to Haswell architecture that are not provided by PAPI. However, all the PMCs can not be determined using a single application run since only a limited number of registers is dedicated to collecting them. For example, to collect all the Likwid PMCs for a single runtime configuration of an application on the server, the application must be executed 53 times. It must be also pointed out that energy predictive models based on PMCs are not portable across a wide range of architectures. While a model based on either Likwid PMCs or PAPI PMCs may be portable across Intel and AMD architectures, it will be unsuitable for GPU architectures.

Therefore, there are three serious constraints that pose difficult challenges to employing PMCs as predictor variables for energy predictive modeling. First, there is a large number of PMCs to consider. Second, tremendous programming effort and time are required to automate and collect all the PMCs. This is because all the PMCs cannot be collected in one single application run. Third, a model purely based on PMCs lacks portability. In this paper, we focus mainly on techniques employed to select a subset of PMCs to be used as predictor variables for energy predictive modeling. We now present a brief survey of them.

O’Brien et al. [18] survey the state-of-the-art energy predictive models in HPC and present a case study demonstrating the ineffectiveness of the dominant PMC-based modeling approach for accurate energy predictions. In the case study, they use 35 carefully selected PMCs (out of a total of 390 available in the platform) in their linear regression model for predicting dynamic energy consumption. [1, 9, 10] select PMCs manually, based on in-depth study of architecture and empirical analysis. [8, 15, 18, 21, 22, 27, 30] select PMCs that are highly correlated with energy consumption using Spearman’s rank correlation coefficient (or Pearson’s correlation coefficient) and principal component analysis (PCA). [1, 2, 15] use variants of linear regression to remove PMCs that do not improve the average model prediction error.

From the survey, we can classify the existing techniques into three categories. The first category contains techniques that consider all the PMCs with the goal to capture all possible
contributors to energy consumption. To the best of our knowledge, we found no research works that adopt this approach. This could be due to several reasons: a) Gathering all PMCs requires huge programming effort and time; b) Interpretation (for example, visual) of the relationship between energy consumption and PMCs is difficult especially when there is a large number of PMCs; c) Dynamic or runtime models must choose PMCs that can be gathered in just one application run; d) Typically, simple models (those with less parameters) are preferred over complex models not because they are accurate but because simplicity is considered a desirable virtue.

The second category consists of techniques that are based on a statistical methodology. The last category contains techniques that use expert advice or intuition to pick a subset (that may not necessarily be determined in one application run) and that, in experts’ opinion, is a dominant contributor to energy consumption. However, the existing techniques have not considered one fundamental property of predictor variables that should have been considered in the first place to remove PMCs unfit for modeling energy. We address this oversight in this paper.

We propose a novel selection criterion for PMCs called additivity, which can be used to determine the subset of PMCs that can potentially be used for reliable energy predictive modeling. It is based on the experimental observation that the energy consumption of a serial execution of two applications is the sum of energy consumptions observed for the individual execution of each application. We define a compound application to represent a serial execution of a combination of two or more individual applications. The individual applications are also termed as base applications.

A linear predictive energy model is consistent if and only if its predictor variables are additive in the sense that the vector of predictor variables for a compound application is the sum of vectors for the individual execution of each application. The additivity criterion, therefore, is based on simple and intuitive rule that the value of a PMC for a compound application is equal to the sum of its values for the executions of the base applications constituting the compound application.

We brand a PMC non-additive on a platform if there exists a compound application for which the calculated value significantly differs from the value observed for the application execution on the platform (within a tolerance of 5.0%). If we fail to find a compound application (typically from a sufficiently large suite of compound applications) for which the additivity criterion is not satisfied, we term the PMC as potentially additive, which means that it can potentially be used for reliable energy predictive modeling. By definition, a potentially additive PMC must be deterministic and reproducible, that is, it must exhibit the same value (within a tolerance of 5.0%) for different executions of the same application with same runtime configuration on the same platform.

The use of a non-additive PMC as a predictor variable in a model renders it inconsistent and therefore unreliable.

We study the additivity of PMCs offered by two popular tools, Likwid and PAPI, by employing a detailed statistical experimental methodology on a modern Intel Haswell multicore server CPU. We observe that all the Likwid PMCs and PAPI PMCs are reproducible. However, we show that while many PMCs are potentially additive, a considerable number of PMCs are not. Some of the non-additive PMCs are widely used in energy predictive models as key predictor variables.

For each non-additive PMC, we determine the maximum percentage error (averaged over several runs) observed experimentally. This is the ratio of the difference between the PMC of
a compound application and the sum of the PMCs of the base applications and the sum of
the PMCs. We show that there is a PMC where the error is as high as 3075% and there are
several PMCs where the error is over 100%. This brings into question the reliability and reported
prediction accuracy of models that use these PMCs.

Our key contribution in this work is that we propose a novel criterion called \textit{additivity}
that can be used to identify PMCs not suitable for energy predictive modeling. PMCs offered by
popular tools such as \textit{Likwid} and \textit{PAPI} are classified based on this criterion using a detailed
experimental methodology on a modern Intel Haswell multicore server CPU. In our future work,
we plan to classify the \textit{non-additivity} of PMC into application-specific and platform-specific
categories.

The rest of the paper is structured as follows. Section 1 surveys popular tools that provide
programmatic and command-line interfaces to obtain PMCs. In Section 2, we define the property
of \textit{additivity} and explain why it is important for reliable energy predictive modeling. Section 3
presents the experimental methodology used to determine the \textit{additivity} of Likwid and PAPI
PMCs. Sections 4 and 5 present a classification of Likwid and PAPI PMCs respectively based
on the criterion of \textit{additivity}. Finally, Section 6 concludes the paper.

1. Related Work

This section is divided into two parts. In the first part, we present tools widely used to
obtain PMCs. In the second part, we survey notable research on selection of PMCs for power
and energy modeling from a large set supplied by a tool.

1.1. Tools to Determine PMCs

\textit{PAPI} [19] provides a standard API for accessing PMCs available on most modern micropro-
cessors. It provides two types of events, \textit{native events} and \textit{preset events}. \textit{Native events} correspond
to PMCs native to a platform. They form the building blocks for \textit{preset events}. A \textit{preset event}
is mapped onto one or more native events on each hardware platform. While \textit{native events} are
specific to each platform, \textit{preset events} obtained on different platforms can not be compared.

\textit{Likwid} [22] provides command-line tools and an API to obtain PMCs for both Intel and
AMD processors on the Linux OS.

For Nvidia GPUs, \textit{CUDA Profiling Tools Interface (CUPTI)} [3] can be used for obtaining
the PMCs. \textit{Intel PCM} [14] is used for reading PMCs of core and uncore (which includes the
QPI) components of an Intel processor. \textit{Perf} [25] also called \textit{perf.events} can be used to gather
the PMCs for CPUs in Linux.

1.2. Techniques for Selection of PMCs for Energy Predictive Modeling

All the models surveyed in this section are linear energy predictive models.

Singh et al. [20] use PMCs provided by AMD Phenom processor. They divide the PMCs
into four categories and rank them in the increasing order of correlation with power using the
Spearman’s rank correlation. Then they select the top PMC in each category (four in total) for
their energy prediction model.

Goel et al. [8] divide PMCs into event categories that they believe capture different kinds
of microarchitectural activity. The PMCs in each category are then ordered based on their
correlation to power consumption using the Spearman’s rank correlation. The PMCs with less correlation are then investigated by analyzing the accuracy of several models that employ them.

Kadayif et al. [16] present a PMC-based model for predicting energy consumption of programs on an UltraSPARC platform. The platform provides 30 different PMCs. However, they use only eight and do not specify how they have selected them.

Lively et al. [17] employ 40 PMCs in their predictive model. They use an elaborate statistical methodology to select PMCs. They compute the Spearman’s rank correlation for each PMC and remove those below a threshold. They compute the principal components (PCA) of the remaining PMCs and select those with the highest PCA coefficients. Bircher et al. [1] employ an iterative linear regression modeling process where they add a PMC at each step and stop until desired average prediction error is achieved.

Song et al. [21] select a group of PMCs (for their energy model of Nvidia Fermi C2075 GPU) that are strongly correlated to power consumption based on the Pearson correlation coefficient.

Witkowski et al. [26] use PMCs provided by the Perf tool for their model. They use the correlation (Pearson correlation coefficient) between a PMC and the measured power consumption and select those PMCs, which have high correlation coefficients. Although they find that the PMCs related to DRAM have a low correlation with power consumption, they still use them since these variables signify intensity of DRAM operations, which contribute significantly to power consumption.

Gschwandtner et al. [9] deal with the problem of selecting the best subset of PMCs on the IBM POWER7 processor, which offers over 500 different PMCs. They first manually select a medium number of hardware counters that they believe are prominent contributors to energy consumption. Then they empirically select a subset from their initial selection. Jarus et al. [15] use PMCs provided by the Perf tool for their models. The PMCs employed differ for different models and are selected using two-stage process. In the first stage, PMCs that are correlated 90% or above are selected. In the second stage, stepwise regression with forward selection is used to decide the final set of PMCs.

Haj-Yihia et al. [10] start with a set of 23 PMCs (offered by Likwid) based on expert knowledge of the Intel architecture. Then they perform linear regression iteratively where they drop PMCs (one by one) that do not impact the average prediction error of their model.

Wu et al. [29] use the Spearman correlation coefficient and PCA to select the subset of PMCs, that are highly correlated with power consumption. Chadha et al. [2] select a particular PMC from the list of PAPI PMCs available for their platform and check if it fits well with linear regression model. If it does, they select it as a key parameter for their modeling and experimental study. Otherwise, they skip it.

2. Additivity: Definition

The additivity criterion is based on simple and intuitive rule that the value of a PMC for a compound application is equal to the sum of its values for the executions of the base applications constituting the compound application.

We brand a PMC non-additive on a platform if there exists a compound application for which the calculated value significantly differs from the value observed for the application execution on the platform (within a tolerance of 5.0%). If the experimentally observed PMCs (sample means) of two base applications are $\bar{e}_1$ and $\bar{e}_2$ respectively, then a non-additive PMC of the compound...
Table 2. List of Applications

<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NPB IS</td>
<td>Integer Sort, Kernel for random memory access</td>
</tr>
<tr>
<td>NPB LU</td>
<td>Lower-Upper Gauss-Seidel solver</td>
</tr>
<tr>
<td>NPB EP</td>
<td>Embarrassingly Parallel, Kernel</td>
</tr>
<tr>
<td>NPB BT</td>
<td>Block Tri-diagonal solver</td>
</tr>
<tr>
<td>NPB MG</td>
<td>Multi-Grid on a sequence of meshes</td>
</tr>
<tr>
<td>NPB FT</td>
<td>Discrete 3D fast Fourier Transform</td>
</tr>
<tr>
<td>NPB DC</td>
<td>Data Cube</td>
</tr>
<tr>
<td>NPB UA</td>
<td>Unstructured Adaptive mesh, dynamic and irregular memory access</td>
</tr>
<tr>
<td>NPB CG</td>
<td>Conjugate Gradient</td>
</tr>
<tr>
<td>NPB SP</td>
<td>Scalar Penta-diagonal solver</td>
</tr>
<tr>
<td>NPB DT</td>
<td>Data traffic</td>
</tr>
<tr>
<td>MKL FFT</td>
<td>Fast Fourier Transform</td>
</tr>
<tr>
<td>MKL DGEMM</td>
<td>Dense Matrix Multiplication</td>
</tr>
<tr>
<td>HPGCG</td>
<td>High performance conjugate gradient</td>
</tr>
<tr>
<td>stress</td>
<td>CPU, disk and I/O stress</td>
</tr>
<tr>
<td>Naive MM</td>
<td>Naive Matrix-matrix multiplication</td>
</tr>
<tr>
<td>Naive MV</td>
<td>Naive Matrix-vector multiplication</td>
</tr>
</tbody>
</table>

application will experimentally exhibit a count that does not lie between $(e_1 + e_2) \times (1 - \epsilon)$ and $(e_1 + e_2) \times (1 + \epsilon)$, where the tolerance $\epsilon = 0.05$.

If we fail to find a compound application (typically from a large set of diverse compound applications) for which the additivity criterion fails, we term the PMC as potentially additive, which means that it can potentially be used for reliable energy predictive modeling. By definition, a potentially additive PMC must be deterministic and reproducible, that is, it must exhibit the same value (within a tolerance of 5.0%) for different executions of the same application with the same runtime configuration on the same platform.

The use of a non-additive PMC as a predictor variable in a model renders it inconsistent and therefore unreliable. We explain this point using a simple example. Consider an instance of an energy prediction model that uses a non-additive PMC as a predictor variable. A natural and intuitive approach to predict the energy consumption of an application that executes two base applications one after the other is to substitute the sum of the PMCs for the base applications in the model. However, since the PMC is non-additive, the prediction would be very inaccurate.

Therefore, using non-additive PMCs in energy predictive models adds noise and can significantly damage the predicting power of energy models based on them.

We now present a test to determine if a PMC is non-additive or potentially additive. We call it the additivity test.

2.1. Additivity Test

The test consists of two stages. A PMC must pass both stages to be declared additive for a given compound application on a given platform. At the first stage, we determine if the PMC is deterministic and reproducible.
At the second stage, we examine how the PMC of the compound application relates to its values for the base applications. At first, we collect the values of the PMC for the base applications by executing them separately. Then, we execute the compound application and obtain its value of the PMC. Typically, the core computations for the compound application consist of the core computations of the base applications programmatically placed one after the other. This has to be the case for PAPI PMCs. However, for Likwid PMCs, one can use the system call to invoke the base application. It must also be ensured that the execution of the compound application takes place under platform conditions similar to those for the execution of its constituent base applications.

If the PMC of the compound application is equal to the sum of the PMCs of the base applications (with a tolerance of 5.0%), we classify the PMC as potentially additive. Otherwise, it is non-additive. We call the PMC that passes the additivity test potentially additive. For it to be called absolutely additive on a platform, ideally it must pass the test for all conceivable compound applications on the platform. Therefore, we avoid this definition.

In our experiments, we observed that all the PMCs were deterministic and reproducible. For each PMC, we determine the maximum percentage error. For a compound application, the percentage error (averaged over several runs) is calculated as follows:

$$\text{Error(\%)} = \left(\frac{|\bar{e}_c - (\bar{e}_{b1} + \bar{e}_{b2})|}{\bar{e}_{b1} + \bar{e}_{b2}}\right) \times 100$$

where $\bar{e}_c$, $\bar{e}_{b1}$, $\bar{e}_{b2}$ are the sample means of predictor variables for the compound application and the constituent base applications respectively. The maximum percentage error is then calculated as the maximum of the errors for all the compound applications in the experimental testsuite.

### 3. Experimental Methodology to Obtain Likwid and PAPI PMCs

In this section, we present our experimental setup to determine the additivity of PMCs.

The experiments are performed on the Intel Haswell multicore CPU platform (specifications given in Tab. 1). We used diverse range of applications (both compute-bound and memory-bound) in our testsuite composed of NAS parallel benchmarking suite (NPB), Intel math kernel library (MKL), HPCG [13], and stress [24] (description given in Tab. 2). The experimental workflow is shown in Fig. 1 where the internals of the server are shown in great detail.

For each run of a application in our testsuite, we measure the following: 1) Dynamic energy consumption, 2) Execution time, and 3) PMCs. The dynamic energy consumption and the application execution time are obtained using the HCLWattsUp interface [11]. We would like to mention that the output variables (or response variables) in the performance and energy predictive models, i.e. energy consumption and execution time, are additive. We confirm this via thorough experimentation and therefore we will not discuss them hereafter.

We now present our experimental methodologies for determining Likwid and PAPI PMCs.

#### 3.1. LIKWID PMCs

In this section, we explain the experimental methodology to obtain Likwid PMCs.
A sample Likwid command-line invocation is shown below where \texttt{EVENTS} represents one or more PMCs, which are collected during the execution of the given application \texttt{APP}:

\texttt{likwid-perfctr -f \!-\!C S0:0\!-\!11@S1:12\!-\!23 -g EVENTS APP}

Here, the application (\texttt{APP}) during its execution is pinned to physical cores (0-11, 12-23) in our platform. Since Likwid does not provide option to bind application to memory, we have used \texttt{numactl}, i.e. a command-line linux tool, with option \texttt{--membind} to pin our applications to memory blocks (for our platform \texttt{numactl} gives 2 memory blocks, 0 and 1). The list of comma-separated PMCs is specified in \texttt{EVENTS}. For example, the following command:

\texttt{likwid-perfctr -f \!-\!C S0:0\!-\!11@S1:12\!-\!23 -g ICACHE\_ACCESSSES\:PMC0,ICACHE\_MISSES\:PMC1 numactl \!--membind\!=\!0,1 APP}

determines the counts for two PMCs, \texttt{ICACHE\_ACCESSSES\:PMC0} and \texttt{ICACHE\_MISSES\:PMC1}.

Collection of all PMCs requires significant programming efforts and execution time because only a limited number of PMCs can be obtained in a single application run due to the limited
number of registers dedicated to collecting PMCs. In addition, to ensure the reliability of our results, we follow a detailed statistical methodology where sample mean of a PMC is used. It is calculated by executing the application repeatedly until it lies in the 95% confidence interval and a precision of 0.050 (5.0%) 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.

Likwid provides 167 PMCs for our platform. In order to collect all of them for an application, we have to run the application 53 times. We wrote a software tool to automate this collection process, SLOPE-PMC-LIKWID [12].

Before we apply the additivity test, we remove few PMCs such as I/O CREDIT (related to I/O and QPI), and OFFCORE_RESPONSE since they exhibit zero counts. We also remove PMCs having very low count (less than 10). The resulting dataset contained 151 performance events, which are then input to the additivity test.

3.2. PAPI PMCs

In this section, we explain the experimental methodology to obtain PAPI PMCs.

We check the available PAPI PMCs for our Intel Haswell platform using the command-line invocation, 'papi_avail − a'. We found that a total of 53 PMCs are available. The number of PMCs that can be gathered in a single application run varies. While gathering a set of 4 PMCs is common, there are a few event sets, which can contain up to 2 or 3 PMCs. Therefore, we found that the application has to be executed 14 times in order to collect all the PMCs for the application on our platform.

We wrote a software tool to automate the process of collection of PMCs, SLOPE-PMC-PAPI [12]. It is to be noted that for ensuring the reliability of our experimental results, we follow the same statistical methodology that was followed for determining Likwid PMCs.

4. Additivity of Likwid PMCs

In this section, we determine the additivity of Likwid PMCs. We execute all the compound applications where each application is composed of two base applications in our test suite (shown in Tab. 2).

The list of potentially additive PMCs is shown in the Tab. 3. The list of non-additive PMCs is presented in Tab. 4, which also reports the maximum percentage error for each PMC.

It is noteworthy that some non-additive PMCs are used as predictor variables in many energy predictive models [5, 6, 10, 20, 23]. These are ICache events, L2 Transactions, and L2 Requests.

5. Additivity of PAPI PMCs

In this section, we determine the additivity of PAPI PMCs. We again execute all the compound applications where each application is composed of two base applications in our test suite (shown in Tab. 2).

The list of potentially additive PMCs is shown in Tab. 5. The list of non-additive PMCs is shown in Tab. 6, which also reports the maximum percentage error for each PMC.
Table 3. List of Potentially Additive Likwid PMCs

| BR_INST_EXEC_ALL_BRANCHES | IDQ.UOPS_NOT_DELIVERED.CYCLES | BR_INST_NOT_DELIVERED_BRANCHES | IDQ.UOPS_NOT_DELIVERED.CYCLES.PE.WAS_OK | BR_INST_RETIRED_ALL_BRANCHES | IDQ.UOPS_EXECUTED_PORT.PORT_0 | DRAM.CLOCK_TICKS | UOPS_EXECUTED_PORT.PORT_1 | SNOOPS_RSP_AFTER_DATA_LOCAL | UOPS_EXECUTED_PORT.PORT_2 | SNOOPS_RSP_AFTER_DATA_REMOTE | UOPS_EXECUTED_PORT.PORT_3 | RXI_FLITS_G1.DIS_NON_DATA | UOPS_EXECUTED_PORT.PORT_4 | RXI_FLITS_G0_NON_DATA | UOPS_EXECUTED_PORT.PORT_5 | TXI_FLITS_G1_NON_DATA | UOPS_EXECUTED_PORT.PORT_6 | CPU_CLK_UNHALTED_ANY | UOPS_EXECUTED_PORT.PORT_7 | CPU_CLOCK_UNHALTED_THREAD_PORT | UOPS_EXECUTED_PORT.PORT_8 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_9 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_10 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_11 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_12 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_13 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_14 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_15 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_16 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_17 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_18 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_19 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_20 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_21 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_22 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_23 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_24 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_25 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_26 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_27 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_28 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_29 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_30 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_31 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_32 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_33 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_34 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_35 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_36 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_37 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_38 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_39 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_40 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_41 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_42 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_43 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_44 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_45 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_46 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_47 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_48 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_49 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_50 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_51 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_52 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_53 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_54 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_55 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_56 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_57 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_58 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_59 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_60 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_61 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_62 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_63 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_64 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_65 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_66 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_67 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_68 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_69 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_70 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_71 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_72 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_73 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_74 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_75 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_76 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_77 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_78 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_79 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_80 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_81 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_82 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_83 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_84 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_85 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_86 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_87 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_88 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_89 | CPU_CLOCK_UNHALTED_THREAD_PORT_ANY | UOPS_EXECUTED_PORT.PORT_90 |

It should be mentioned that some of these non-additive PMCs such as \textit{PAPI.L1.ICM} and \textit{PAPI.L2.ICM} have been widely used in energy and performance predictive models [2, 4, 7, 17, 27, 28]. These represent L1 and L2 instruction cache misses.
5.1. Core and Memory Pinning

We ran two sets of experiments, one with the application pinned to the cores and the other with the application pinned to cores and memory. While the percentage errors were reduced slightly when the application is pinned to both the cores and the memory, we observed that memory pinning has no effect on additive PMCs but, most importantly, non-additive PMCs remained non-additive (within a tolerance of 5%).

6. Discussion

From Tab. 3 and Tab. 4 showing potentially additive and non-additive Likwid PMCs respectively, one can observe that out of a total of 151 PMCs, 43 PMCs are non-additive.
Table 5. List of potentially additive PAPI PMCs

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Maximum Percentage Error (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAPI_L1_DCM</td>
<td>PAPI_FUL_CCY</td>
</tr>
<tr>
<td>PAPI_L2_DCM</td>
<td>PAPI_BR_UCN</td>
</tr>
<tr>
<td>PAPI_CA_SHR</td>
<td>PAPI_BR_CN</td>
</tr>
<tr>
<td>PAPI_CA_CLN</td>
<td>PAPI_BR_TKN</td>
</tr>
<tr>
<td>PAPI_CA_INV</td>
<td>PAPI_BR_NTK</td>
</tr>
<tr>
<td>PAPI_CA_ITV</td>
<td>PAPI_BR_MSP</td>
</tr>
<tr>
<td>PAPI_L1_STM</td>
<td>PAPI_BR_PRC</td>
</tr>
<tr>
<td>PAPI_L2_LDM</td>
<td>PAPI_TOT_INS</td>
</tr>
<tr>
<td>PAPI_L2_STM</td>
<td>PAPI_L2_DCR</td>
</tr>
<tr>
<td>PAPI_PRF_DM</td>
<td>PAPI_L3_DCR</td>
</tr>
<tr>
<td>PAPI_TOT_CYC</td>
<td>PAPI_L2_DCA</td>
</tr>
<tr>
<td>PAPI_L2_TCA</td>
<td>PAPI_L2_TCR</td>
</tr>
</tbody>
</table>

Table 6. List of non-additive PAPI PMCs

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Maximum Percentage Error (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAPI_CA_SN</td>
<td>40.23</td>
</tr>
<tr>
<td>PAPI_TLB_DM</td>
<td>31.54</td>
</tr>
<tr>
<td>PAPI_TLB_IM</td>
<td>23.70</td>
</tr>
<tr>
<td>PAPI_STL_CCY</td>
<td>31.43</td>
</tr>
<tr>
<td>PAPI_LD_INS</td>
<td>32.06</td>
</tr>
<tr>
<td>PAPI_SR_INS</td>
<td>21.98</td>
</tr>
<tr>
<td>PAPI_LST_INS</td>
<td>45.87</td>
</tr>
<tr>
<td>PAPI_L1_ICM</td>
<td>37.28</td>
</tr>
<tr>
<td>PAPI_L2_ICM</td>
<td>37.50</td>
</tr>
<tr>
<td>PAPI_L2_ICH</td>
<td>107.12</td>
</tr>
<tr>
<td>PAPI_L2_ICA</td>
<td>30.65</td>
</tr>
<tr>
<td>PAPI_L3_ICA</td>
<td>30.2</td>
</tr>
<tr>
<td>PAPI_L2_ICR</td>
<td>30.65</td>
</tr>
<tr>
<td>PAPI_L3_TCM</td>
<td>14.54</td>
</tr>
<tr>
<td>PAPI_L3_LDM</td>
<td>74.68</td>
</tr>
<tr>
<td>PAPI_L1_LDM</td>
<td>200.82</td>
</tr>
<tr>
<td>PAPI_L3_ICR</td>
<td>19.48</td>
</tr>
</tbody>
</table>

The event ARITH_DIVIDER_UOPS exhibits the highest maximum percentage error of about 3075%. This event belongs to the µOPS group of Likwid PMCs responsible for gathering PMCs related to the instruction pipeline.

Several PMCs (HA_R2_BL CREDITS EMPTY HI HA0, HA_R2_BL CREDITS EMPTY HI HA1, HA_R2_BL CREDITS EMPTY HI R2 NCS) show maximum percentage error of about 200%. These events specifically belong to the Home Agent (HA) group of Likwid PMCs. HA is central unit that is responsible for providing PMCs from protocol side of memory interactions.
There are several PMCs that show maximum percentage error of about 100%. They are mainly from the QPI group of Likwid PMCs responsible for packetizing requests from the caching agent on the way out to the system interface.

Similarly, from Tab. 5 and Tab. 6 showing potentially additive and non-additive PAPI PMCs respectively, 17 PMCs out of a total of 53 PMCs are non-additive. One PMC, PAPI.L1.LDM, demonstrates the highest maximum percentage error of about 200%. It represents L1 load misses. Another PMC, PAPI.L2.ICH, demonstrates a maximum percentage error of over 100%. It represents L2 instruction cache hits.

If we increase the tolerance to about 20%, then only 8 non-additive Likwid PMCs will become potentially additive. For PAPI, only two non-additive PMCs will become potentially additive, PAPI.L3.TCM and PAPI.L3.ICR. They represent L3 cache misses and L3 instruction cache reads respectively. Increasing the tolerance to about 30% results in other 3 non-additive Likwid PMCs and 5 non-additive PAPI PMCs becoming potentially additive.

Thus, one can see that there are still a large number of PMCs that are non-additive even after increasing the tolerance to as high as 30%. Some of these PMCs have been used as key predictor variables in energy predictive models [2, 4–7, 10, 17, 20, 23, 27, 28].

To summarize, the non-additive PMCs that exceed a specified tolerance must be excluded from the list of PMCs to be considered as predictor variables for energy predictive modeling, because they can potentially damage the prediction accuracy of these models due to their highly non-deterministic nature. Also the list of potentially additive PMCs must be further tested exhaustively for more diverse applications and platforms to secure more confidence in their additivity.

In our future work, we would study how much the prediction error is affected due to the presence of non-additive PMCs in all the linear predictive energy models that we surveyed.

**Conclusion**

Performance events (PMCs) are now dominant predictor variables for modeling energy consumption. Considering the large set of PMCs offered by modern processors, several techniques have been devised to select the best subset of PMCs to be used for energy predictive modeling. However, the existing techniques have not considered one fundamental property of predictor variables that should have been taken into account in the first place to remove PMCs unsuitable for modeling energy. We have addressed this oversight in this paper.

We proposed a novel selection criterion for PMCs called additivity, which can be used to determine the subset of PMCs that can potentially be considered for reliable energy predictive modeling. It is based on the experimental observation that the energy consumption of a serial execution of two applications is the sum of energy consumptions observed for the individual execution of each application. A linear predictive energy model is consistent if and only if its predictor variables are additive in the sense that the vector of predictor variables for a serial execution of two applications is the sum of vectors for the individual execution of each application.

We studied the additivity of PMCs offered by two popular tools, Likwid and PAPI, using a detailed statistical experimental methodology on a modern Intel Haswell multicore server CPU. We showed that many PMCs in Likwid and PAPI are non-additive and that some of these PMCs are key predictor variables in energy predictive models thereby bringing into question the reliability and reported prediction accuracy of these models.
In our future work, we would classify the non-additivity of a PMC into application-specific and platform-specific categories. We will also look at additivity of PMCs offered by accelerators such as Graphical Processing Units (GPUs). For instance, Nvidia GPUs provide CUDA Profiling Tools Interface (CUPTI) that provides functions to determine around 140 PMCs. However, implementing a compound application (or kernel) from two or more base applications (kernels) is not straightforward. While CUPTI allows a continuous event collection mode, we found it is not widely supported and hence unusable presently for implementation of compound applications.

Acknowledgement

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

This paper is distributed under the terms of the Creative Commons Attribution-Non Commercial 3.0 License which permits non-commercial use, reproduction and distribution of the work without further permission provided the original work is properly cited.

References


