Supplemental Material: Improving the Accuracy of Energy Predictive Models Using the Utilization Variables and Performance Events for Multicore CPUs

Arsalan Shahid, Muhammad Fahad, Ravi Reddy Manumachu, and Alexey Lastovetsky

School of Computer Science, University College Dublin, Belfield, Dublin 4, Ireland

The supporting materials for the main manuscript, "Improving the Accuracy of Energy Predictive Models Using the Utilization Variables and Performance Events for Multicore CPUs," are:

- Techniques for selection of PMCs.
- Rationale behind using dynamic energy consumption instead of total energy consumption.
- Steps to ensure reliable experiments.
- Application Programming Interface (API) for measurements using external power meter interfaces (HCLWattsUp).
- Calibration of WattsUp Pro power-meters.
- Methodology to determine the component-Level energy consumption using HCLWattsUp.
- Methodology to determine the component-Level energy consumption using Intel RAPL.
- Experimental methodology to determine the sample mean.
- Brief overview of *SLOPE-PMC* and *AdditivityChecker*.
- List of PMC groups provided by Likwid.

Email address: `{arsalan.shahid,muhammad.fahad}@ucdconnect.ie`, `{ravi.manumachu,alexey.lastovetsky}@ucd.ie` (Arsalan Shahid, Muhammad Fahad, Ravi Reddy Manumachu, and Alexey Lastovetsky)

1. Techniques for Selection of PMCs

Modern computing platforms such as multicore CPUs provide a large set of PMCs. The most popular tools that can be used to gather the values of the PMCs for a platform include Likwid [1], PAPI [2], Intel PCM [3], and Linux *perf* [4]. The programmers, however, can obtain only a small number of PMCs (typically 3-4) during an application run due to the limited number of hardware registers dedicated to storing them. Consider, for example, the Intel Haswell server with processor E5-2670 v3. *Likwid* tool provides 167 PMCs for this platform. To obtain the values of the PMCs for an application, the application must be executed about 53 times since only a limited number of PMCs can be obtained in a single application run.

Since only 3-4 PMCs can be collected in a single application run, selecting such a reliable subset as predictor variables is crucial to the prediction accuracy of online energy models.

We classify techniques for selecting the PMCs into following four categories:

- Techniques that consider all the PMCs offered by a tool for a platform 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.
- Techniques that are based on a statistical methodology such as correlation, principal component analysis (PCA) etc. [5], [6].
- 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 [7].
- Techniques that select parameters with physical significance based on fundamental laws such as energy conservation of computing [8].

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

We consider only the dynamic energy consumption in our work for reasons below:

1. Static energy consumption, a major concern in embedded systems, is becoming less compared to the dynamic energy consumption due to advancements in hardware architecture design in HPC systems.
2. We target applications and platforms where dynamic energy consumption is the dominating energy dissipator.
3. 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 the total energy consumption of a system to be 16500J and 18000J. It would report the prediction error to be 8.3%. If it is known that the static energy consumption of the system is 9000J, then the actual prediction error (based on dynamic energy consumption only) would be 16.6% instead.
- In our second example, consider two different energy prediction models (M_A and M_B) with the same prediction errors of 5% for 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 consumption only) for the models would be 16.6% and 8.3%. Therefore, the second model M_B should be considered more accurate than the first.

3. Steps to Ensure Reliable Experiments

To ensure that our results are reliable, we follow a statistical methodology that is summarized in the following steps:

- For an application execution, a response variable is represented by the mean value obtained after its multiple experimental runs.
- We calculate the sample mean by executing the application repeatedly until the response variable 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. 5 presents our experimental methodology to determine the sample mean.

Both servers are fully dedicated to the experiments. To ensure reliable dynamic energy measurements, we take the following precautions:

1. *HCLWattsUp* API [9] gives the total energy consumption of the server during the execution of an application using system-level physical measurements from the external power meters (in our case WattsUp Pro). This includes the contribution of components such as NIC, SSDs, fans, etc. To make sure that the value of dynamic energy consumption is a pure representation of an application's activity on CPUs and main memory, we verify that all the components other than CPUs and DRAM are idle using the following steps:

- Monitoring the disk utilization before and during the application run. We make certain that there is no I/O performed by the application using standard Linux tools such as *sar* and *iotop*.
- We calculate the theoretical estimation of main memory consumption for an application using its problem size. We make sure that the problem size used in the execution of an application does not exceed the capacity of main memory, and that swapping (paging) does not occur.
- We ensure that the network is not used by the application using Linux monitoring tools such as *sar* and *atop*.
- Each application is bind during its execution to resources using cores-pinning and memory-pinning tools such as *numactl*.

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 experiments. We ensure that 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 the server and the speed of fans before and during an application run. We obtain this information by using the Intel Intelligent Platform Management Interface (IPMI) sensors.
    - We observe that both the temperature of the servers and the speeds of the fans remain the same whether an application is running or not.
    - We set the fans at *full* speed before starting the application run. The results in terms of applications' performance and energy consumption from this experiment were the same as when the fans were run at *optimal* speed.
    - We experimentally observe that the time it takes for the memory utilization, core frequencies, and operating voltages to revert to base values after the execution of an application is approximately 100 seconds. To minimize the pipelining hazards and cache effects, the experiments are not executed in a loop and sufficient time (120 seconds) is allowed to elapse between successive application runs.
3. We obtain the dynamic energy consumption from Intel RAPL [10] using DE-Meter [11]. The measurements obtained using DE-Meter are also sample means obtained after several experimental runs of an application.

4. Application Programming Interface (API) for Measurements Using External Power Meter Interfaces (HCLWattsUp)

HCLServer1 and HCLServer2 have a dedicated power meter installed between their input power sockets and wall A/C outlets. The power meter captures the total power consumption of the node. It has a data cable connected to the USB port of the node. A Perl script collects the data from the power meter using the serial USB interface. The execution of this script is non-intrusive and consumes insignificant power.

We use HCLWattsUp API function, which gathers the readings from the power meters to determine the average power and energy consumption during the execution of an application on a given platform. HCLWattsUp API can provide the following four types of measures during the execution of an application:

- *TIME*—The execution time (seconds).
- *DPOWER*—The average dynamic power (watts).
- *TENERGY*—The total energy consumption (joules).
- *DENERGY*—The dynamic energy consumption (joules).

We confirm that the overhead due to the API is very minimal and does not have any noticeable influence on the main measurements. It is important to note that the power meter readings are only processed if the measure is not *hcl::TIME*. Therefore, for each measurement, we have two runs. One run for measuring the execution time. And the other for energy consumption. The following example illustrates the use of statistical methods to measure the dynamic energy consumption during the execution of an application.

The API is confined in the *hcl* namespace. Lines 10–12 construct the Wattsup object. The inputs to the constructor are the paths to the scripts and their arguments that read the USB serial devices containing the readings of the power meters.

The principal method of *WattsUp* class is *execute*. The inputs to this method are the type of measure, the path to the executable *executablePath*, the arguments to the executable *executableArgs* and the statistical thresholds (*pIn*) The outputs are the achieved statistical confidence *pOut*, the estimators, the sample mean (*sampleMean*) and the standard deviation (*sd*) calculated during the execution of the executable.

The *execute* method repeatedly invokes the executable until one of the following conditions is satisfied:

- The maximum number of repetitions specified in $maxRepeats$ is exceeded.
- The sample mean is within $maxStdError$ percent of the confidence interval cl. The confidence interval of the mean is estimated using the Student's t-distribution.
- The maximum allowed time *maxElapsedTime* specified in seconds has elapsed.

If any of the conditions are not satisfied, then a return code of 0 is output suggesting that statistical confidence has not been achieved. If statistical confidence has been achieved, then the number of repetitions performed, the time elapsed and the final relative standard error is returned in the output argument

```
#include <wattsup.hpp>
int main(int argc, char** argv)
{
    std::string pathsToMeters[2] = {
       "/opt/powertools/bin/wattsup1",
       "/opt/powertools/bin/wattsup2"};
    std::string argsToMeters[2] = {
       "--interval=1",
       "--interval=1"};
    hcl::Wattsup wattsup(
        2, pathsToMeters, argsToMeters
    );
    hcl::Precision pIn = {
        maxRepeats, cl, maxElapsedTime, maxStdError
    };
    hcl::Precision pOut;
    double sampleMean, sd;
    int rc = wattsup.execute(
               hcl::DENERGY, executablePath,
               executableArgs, &pIn, &pOut,
               &sampleMean, &sd
    );
    if (rc == 0)
       std::cerr << "Precision NOT achieved.\n";
    else
       std::cout << "Precision achieved.\n";
    std::cout << "Max repetitions "
              << pOut.reps_max
              << ", Elasped time "
              << pOut.time_max_rep
              << ", Relative error "
              << pOut.eps
              << ", Mean energy "
              << sampleMean
              << ", Standard Deviation "
              << sd
              << std::endl;
    exit(EXIT_SUCCESS);
}
```

Figure 1: Example illustrating the use of HCLWattsUp API for measuring the dynamic energy consumption

pOut. At the same time, the sample mean and standard deviation are returned. For our experiments, we use values of (1000, 95%, 2.5%, 3600) for the parameters $(maxRepeats, cl, maxStdError, maxElapsedTime)$ respectively. Since we use Student's t-distribution for the calculation of the confidence interval of the mean, we confirm specifically that the observations follow normal distribution by plotting the density of the observations using the R tool.

5. Experimental Methodology to Determine the Sample Mean

We followed the methodology described below to make sure the experimental results are reliable:

- The server is fully reserved and dedicated to these experiments during their execution. We also made certain that there are no drastic fluctuations in the load due to abnormal events in the server by monitoring its load continuously for a week using the tool *sar*. Insignificant variation in the load was observed during this monitoring period suggesting normal and clean behavior of the server.

- An application during its execution is bound to the physical cores using the *numactl* tool.

- To obtain a data point, the application is repeatedly executed until the sample mean lies in the 95% confidence interval with a precision of 0.025 (2.5%). For this purpose, we use Student's t-test assuming that the individual observations are independent and their population follows the normal distribution. We verify the validity of these assumptions using Pearson's chi-squared test. When we mention a single number such as execution time (seconds) or floating-point performance (in MFLOPs or GFLOPs), we imply the sample mean determined using the Student's t-test.

  The function $MeanUsingTtest$, shown in Algorithm 1, determines the sample mean for a data point. For each data point, the function repeatedly executes the application *app* until one of the following three conditions is satisfied:

  1. The maximum number of repetitions (*maxReps*) is exceeded (Line 3).
  2. The sample mean falls in the confidence interval (or the precision of measurement *eps* is achieved) (Lines 13-15).
  3. The elapsed time of the repetitions of application execution has exceeded the maximum time allowed (*maxT* in seconds) (Lines 16-18).

  So, for each data point, the function $MeanUsingTtest$ returns the sample mean *mean*. The function $Measure$ measures the execution time using *gettimeofday* function.

- In our experiments, we set the minimum and the maximum number of repetitions, *minReps* and *maxReps*, to 15 and 100000. The values of *maxT*, *cl*, and *eps* are 3600, 0.95, and 0.025. If the precision of measurement is not achieved before the completion of the maximum number of repeats, we increase the number of repetitions and also the allowed maximum elapsed time. Therefore, we make sure that statistical confidence is achieved for all the data points that we use in our experiments.

6. Calibration of WattsUp Pro power-meters

The dynamic energy consumption during the application execution is measured using a *WattsUp Pro* power meter on both servers (HCLServer1 and HCLServer2) and obtained programmatically via the HCLWattsUp interface [9]. The power meter is periodically calibrated using an ANSI C12.20 revenue-grade power meter, Yokogawa WT210. In this chapter, we explain our methodology and some results to calibrate our power-meters.

We compare the WattsUp Pro power-meter power measurements with Yokogawa using three methods that are explained as follows:

1. **Naked-eye visual monitoring**
    - We first attach the WattsUp Pro power-meters to both servers.
    - Once the servers are switched on and are in stable condition, we monitor the WattsUp pro LCDs and note the power readings in watts for both servers.
    - we carefully plugged off the WattsUp Pro power meters and plug the servers via Yokogawa power meter.
    - Once the servers are switched on and are in stable condition, we monitor the Yokogawa LCDs and note the power readings in watts for both servers.
    - On comparison, we find a difference of 2 watts and 3 watts for HCLServer1 and HCLServer2, respectively.
2. **Monitoring server base power**
    - We connect both power meters to both servers one by one and once the servers are stable, we programmatically obtain the power readings from both power meters.
    - For WattsUp Pro, a Perl script provides the power readings with a granularity of 1 second. Similarly, Yokogawa comes with its own software and allows us to read power readings at the granularity of 1 second.
    - We measure and record the base powers of both servers for 3.5 hours using both power meters. Figure 2 compare the idle power profiles of HCLServer1 and HCLserver2 using both power meters, respectively.

Algorithm 1 Function determining the mean of an experimental run using Student's t-test.

```
1: procedure MeanUsingTtest($app, minReps, maxReps,$
        $maxT, cl, accuracy,$
        $repsOut, clOut, etimeOut, epsOut, mean$)
Input:
    The application to execute, $app$
    The minimum number of repetitions, $minReps \in \mathbb{Z}_{>0}$
    The maximum number of repetitions, $maxReps \in \mathbb{Z}_{>0}$
    The maximum time allowed for the application to run, $maxT \in \mathbb{R}_{>0}$
    The required confidence level, $cl \in \mathbb{R}_{>0}$
    The required accuracy, $eps \in \mathbb{R}_{>0}$
Output:
    The number of experimental runs actually made, $repsOut \in \mathbb{Z}_{>0}$
    The confidence level achieved, $clOut \in \mathbb{R}_{>0}$
    The accuracy achieved, $epsOut \in \mathbb{R}_{>0}$
    The elapsed time, $etimeOut \in \mathbb{R}_{>0}$
    The mean, $mean \in \mathbb{R}_{>0}$

2:     $reps \leftarrow 0$; $stop \leftarrow 0$; $sum \leftarrow 0$; $etime \leftarrow 0$
3:     while ($reps < maxReps$) and ($!stop$) do
4:         $st \leftarrow$ MEASURE($TIME$)
5:         Execute($app$)
6:         $et \leftarrow$ MEASURE($TIME$)
7:         $reps \leftarrow reps + 1$
8:         $etime \leftarrow etime + et - st$
9:         $ObjArray[reps] \leftarrow et - st$
10:        $sum \leftarrow sum + ObjArray[reps]$
11:        if $reps > minReps$ then
12:            $clOut \leftarrow$ fabs(gsl_cdf_tdist_Pinv($cl$, $reps - 1$))
                   $\times$ gsl_stats_sd($ObjArray$, 1, $reps$)
                   / sqrt($reps$)
13:            if $clOut \times \frac{reps}{sum} < eps$ then
14:                $stop \leftarrow 1$
15:            end if
16:            if $etime > maxT$ then
17:                $stop \leftarrow 1$
18:            end if
19:        end if
20:    end while
21:    $repsOut \leftarrow reps$; $epsOut \leftarrow clOut \times \frac{reps}{sum}$
22:    $etimeOut \leftarrow etime$; $mean \leftarrow \frac{sum}{reps}$
23: end procedure
```

It can be seen that both profiles are almost the same. However, HCLServer1 has more power variations and HCLServer2 is considerably stable in terms of base power. For HCLServer2, there are power spikes after every half an hour for a couple of seconds. This is because of a daemon service being triggered by the OS.

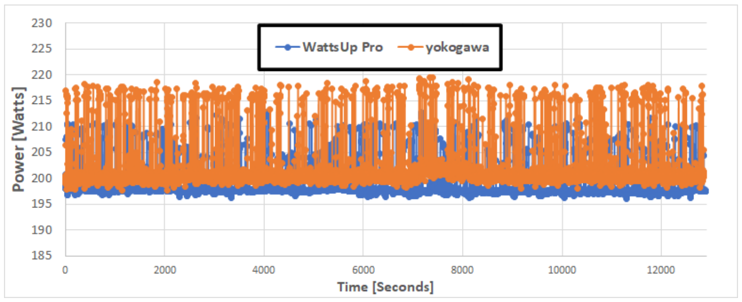
- Table 1 show the minimum, average, and maximum power consumption for both servers and power meters. If we compare the average, WattsUp Pro gives 1-2 watts less power consumption than Yokogawa.

3. **Measurement of total energy consumption for two scientific applications**

   - We choose two scientific applications: 1) DGEMM and 2) FFT from Intel MKL.
   - We execute DGEMM for problem sizes 4096×4096 to 30720×30720 with a constant step size of 1024 on HCLServer1. We then execute DGEMM for problem sizes 15360×15360 to 30720×30720 with a constant step size of 1024 on HCLServer2. We build the total energy consumption profiles using both power meters for these application executions.
   - We execute FFT for problem sizes 4096×4096 to 30720×30720 with a constant step size of 1024 on HCLServer1. We then execute FFT for problem sizes 8384×8384 to 62880×62880 with a constant step size of 2096 on HCLServer2. We build the total energy consumption profiles using both power meters for these application executions.
   - Figure 3 and 4 show the total energy consumption profiles for DGEMM and FFT on both servers, respectively.
   - Table 2 show the relative error in percentage for total energy consumptions obtained using HCLServer1 and HCLServer2. It can be seen that the average measurement error for DGEMM is 4.95% and 9.25% on HCLServer2 and HCLServer1, respectively. For FFT, the average measurement error is 6% and 7.4% on HCLServer2 and HCLServer1, respectively.

Table 1: Minimum, maximum and average of idle power using WattsUp Pro and Yokogawa PowerMeter on HCLServer1 and HCLServer2

	HCLServer1		HCLServer2	
	WattsUp Pro	Yokogawa	WattsUp Pro	Yokogawa
Min	196	197.72	99.1	99.93
Max	212.8	219.47	123.3	125.6
Average	198.2	201.0	100.03	102.06



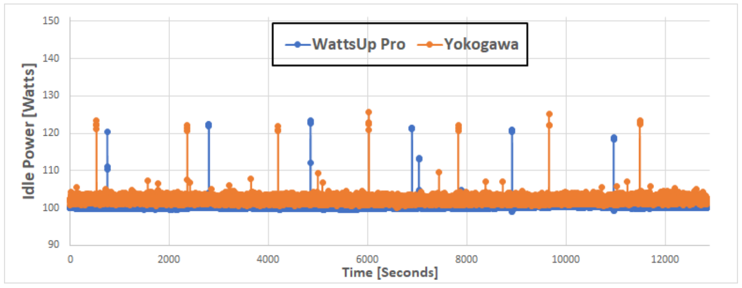
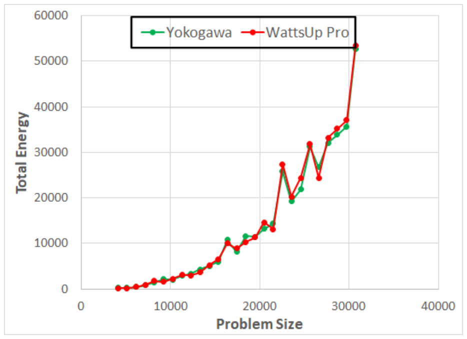


Figure 2: Calibration test for idle power using WattsUp Pro and Yokogawa PowerMeter on (a) HCLServer1 and (b) HCLServer2

Table 2: Comparison of minimum, average, and maximum measurement errors for DGEMM and FFT on HCLServer1 and HCLServer2 using WattsUp Pro and Yokogawa

	HCLServer2 Errors [%] (Min, Avg, Max)	HCLServer1 Errors [%] (Min, Avg, Max)
DGEMM	(2.01, 4.95, 8.16)	(0.58, 9.25, 33.18)
FFT	(0.11, 6.02, 15.84)	(0.20, 7.41, 16.90)



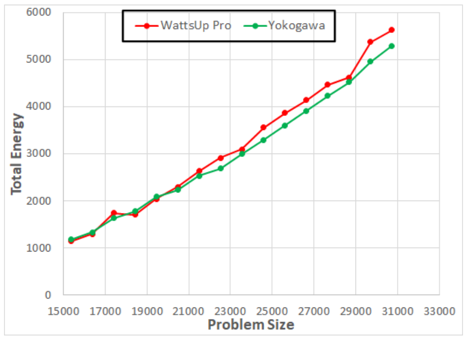
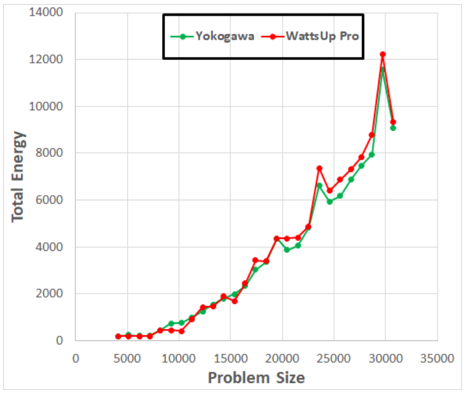


Figure 3: Comparison of total power for Intel MKL DGEMM using WattsUp Pro and Yokogawa PowerMeter on (a) HCLServer1 and (b) HCLServer2



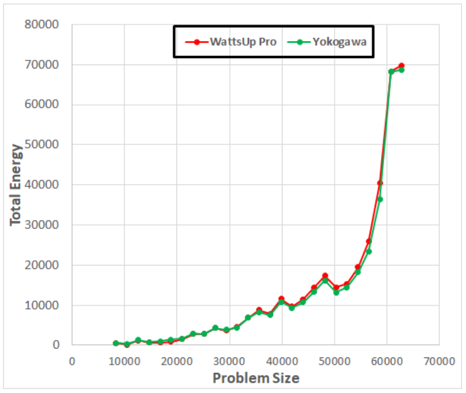


Figure 4: Comparison of total power for Intel MKL FFT using WattsUp Pro and Yokogawa PowerMeter on (a) HCLServer1 and (b) HCLServer2

7. Methodology to Determine the Component-Level Energy Consumption Using HCLWattsUp

We provide here the details of how system-level physical measurements using HCLWattsUp can be used to determine the energy consumption by a component (such as a CPU) during application execution.

We define the group of components running a given application kernel as an *abstract processor*. For example, consider a matrix multiplication application running on a multicore CPU. The abstract processor for this application, which we call *AbsCPU*, comprises of the multicore CPU processor consisting of a certain number of physical cores and DRAM. In this work, we use only such configurations of the application which execute on *AbsCPU* and do not use any other system resources such as solid-state drives (SSDs), network interface cards (NIC) and so forth. Therefore, the change in energy consumption of the system reported by HCLWattsUp reflects solely the contributions from CPU and DRAM. We take several precautions in computing energy measurements to eliminate any potential interference of the computing elements that are not part of the abstract processor *AbsCPU*. To achieve this, we take the following precautions:

1. We ensure the platform is reserved exclusively and fully dedicated to our experiments.
2. We monitor the disk consumption before and during the application run and ensure that there is no I/O performed by the application using tools such as *sar*, *iotop*, and so forth.
3. We ensure that the problem size used in the execution of an application does not exceed the main memory and that swapping (paging) does not occur.
4. We ensure that the network is not used by the application by monitoring using tools such as *sar*, *atop*, etc.
5. We set the application kernel's CPU affinity mask using SCHED API's system call SCHED_SETAFFINITY. Consider for example MKL DGEMM application kernel running on only abstract processor A. To bind this application kernel, we set its CPU affinity mask to 12 physical CPU cores of Socket 1 and 12 physical CPU cores of Socket 2.
6. Fans are also a great contributor to energy consumption. On our platform fans are controlled in two zones: (a) zone 0: CPU or System fans, (b) zone 1: Peripheral zone fans. There are 4 levels to control the speed of fans:
   - Standard: BMC control of both fan zones, with CPU zone based on CPU temp (target speed 50%) and Peripheral zone based on PCH temp (target speed 50%)
   - Optimal: BMC control of the CPU zone (target speed 30%), with Peripheral zone fixed at low speed (fixed 30%)
   - Heavy IO: BMC control of CPU zone (target speed 50%), Peripheral zone fixed at 75%

- Full: all fans running at 100%

   In all speed levels except the full, the speed is subject to be changed with temperature and consequently, their energy consumption also changes with the change of their speed. Higher the temperature of CPU, for example, higher the fans' speed of zone 0 and higher the energy consumption to cool down. This energy consumption to cool the server down, therefore, is not consistent and is dependent on the fans' speed and consequently can affect the dynamic energy consumption of the given application kernel. Hence, to rule out the fans' contribution to dynamic energy consumption, we set the fans at full speed before launching the experiments. When set at full speed, the fans run consistently at a fixed speed until we do so to another speed level. Hence, fans consume the same amount of power which is included in the static power of the platform.

7. We monitor the temperature of the platform and speed of the fans (after setting it at full) with help of Intelligent Platform Management Interface (IPMI) sensors, both with and without the application run. We find no considerable difference in temperature and find the speed of fans the same in both scenarios.

Thus, we ensure that the dynamic energy consumption obtained using HCLWattsUp reflects the contribution solely by the *abstract processor* executing the given application kernel.

8. Methodology to Obtain Dynamic Energy Consumption Using Intel RAPL

We first present a brief on RAPL before introducing our methodology to compare the measurements of dynamic energy consumption by RAPL and HCLWattsUp.

RAPL (Running Average Power Limit) [10] provides a way to monitor and -dynamically-set the power limits on processor and DRAM. So, by controlling the maximum average power, it matches the expected power and cooling budget. RAPL exposes its energy counters through model-specific registers (MSRs) It updates these counters once in every 1 ms. The energy is calculated as a multiple of model-specific energy units. It divides a platform into four domains, which are presented below:

1. *PP0* (Core Devices): Power plane zero includes the energy consumption by all the CPU cores in the socket(s).
2. *PP1* (Uncore Devices): Power plane one includes the power consumption of integrated graphics processing unit – which is not available on server platforms– uncore components.
3. *DRAM*: Refers to the energy consumption of the main memory.
4. *Package*: Refers to the energy consumption of entire socket including core and uncore: `Package = PP0 + PP1`.

PP0 is removed in the the Haswell E5 generation [12]. For our experiments, we use *Package* and *DRAM* domains to obtain the energy consumption by CPU and DRAM when executing a given application.

To obtain the energy consumption provided by RAPL, we use a well-known package, Intel PCM [3]. We ensure that the RAPL values output by this package is correct by comparing with values given by another well known package, PAPI [2].

To compare the RAPL and HCLWattsUp energy measurements, we use the following workflows of the experiments. The workflow to determine the dynamic energy consumption by the given application using RAPL follows:

1. Using Intel PCM and DE-Meter [11], we obtain the *base power* of CPUs (core and un-core) and DRAM (when the given application is not running).
2. Using HCLWattsUp API, we obtain the *execution time* of the given application.
3. Using Intel PCM/PAPI, we obtain the *total energy* consumption of the CPUs and DRAM, during the execution of the given application.
4. Finally, we calculate the *dynamic energy* consumption (of CPUs and DRAM) by subtracting the *base energy* from *total energy* consumed during the execution of the given application.

The workflow to determine the dynamic energy consumption using HCLWattsUp follows:

1. Using HCLWattsUp API, we obtain the *base power* of the server (when the given application is not running).
2. Using HCLWattsUp API, we obtain the *execution time* of the application.
3. Using HCLWattsUp API, we obtain the *total energy* consumption of the server, during the execution of the given application.
4. Finally, we calculate the *dynamic energy* consumption by subtracting the *base power* from *total energy* consumed during the execution of the given application.

We make sure that the execution time of the application kernel is the same for dynamic energy calculations by both tools. So, any difference between the energy readings of the tools comes solely from their power readings.

We analyzed 51 energy profiles of different application configurations of the aforementioned applications, using RAPL and HCLWattsUp. Our configuration parameters are: (a) Problem size ($M \times N$) where $M \leq N$, (b) Number of CPU threads or number of CPU cores.

The cost in terms of a number of measurements to determine the dynamic energy consumption of the application using sensors is the same for both tools as we need three (*Base power, Execution Time and Total Energy*) measurements to obtain a single data point of the application dynamic energy profile.

9. 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 5 summarizes the work-flow of *SLOPE-PMC.*

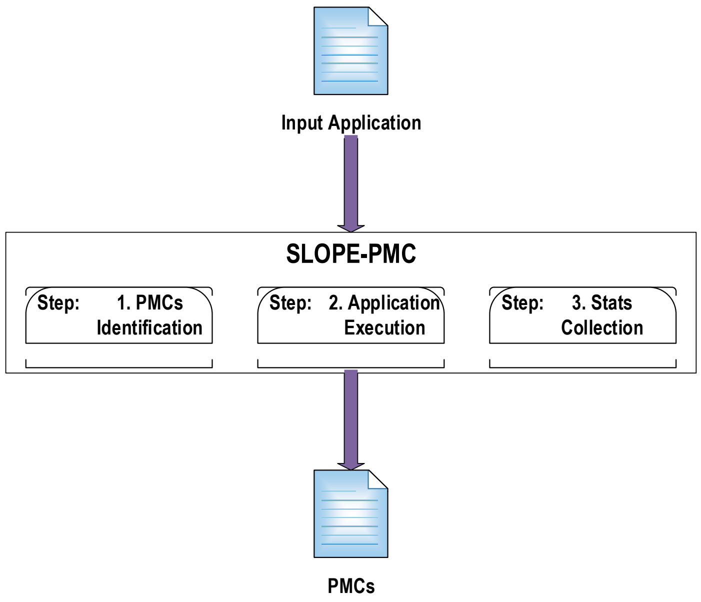

Input Application
SLOPE-PMC
Step: 1. PMCs Identification
Step: 2. Application Execution
Step: 3. Stats Collection
PMCs

Figure 5: *SLOPE-PMC*: Towards the automation of PMC collection on Modern Computing Platforms

Figure 6 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.

10. List of PMC groups Provided by Likwid

The list of PMC groups provided by Likwid tool [1] on HCLServer2 is shown in the Figure 7.

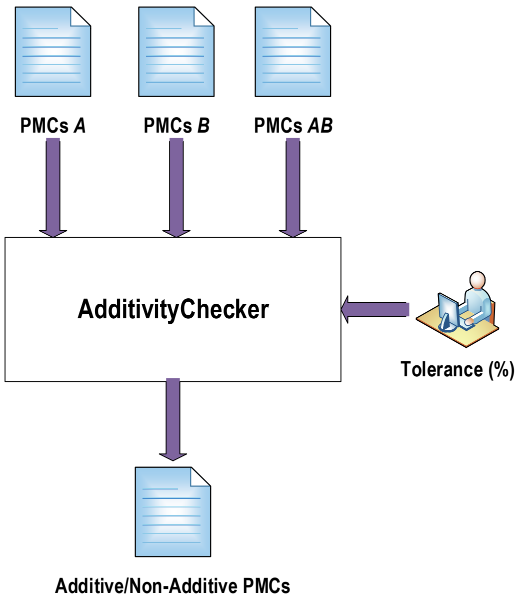


Figure 6: *AdditivityChecker*: Test PMCs for *Additivity*

References

References

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

[2] PAPI, Performance application programming interface 5.4.1 (2015).
URL `http://icl.cs.utk.edu/papi/`

[3] IntelPCM, Intel performance counter monitor - a better way to measure CPU utilization. (2012).
URL `https://software.intel.com/en-us/articles/intel-performance-counter-monitor`

[4] P. Wiki, perf: Linux profiling with performance counters (2017).
URL `https://perf.wiki.kernel.org/index.php/Main\_Page`

[5] J. Mair, Z. Huang, D. Eyers, Manila: Using a densely populated pmc-space for power modelling within large-scale systems, Parallel Computing 82 (2019) 37–56.

```
$ likwid-perfctr -a

 Group name       Description
-------------     ----------------------------------------------
       BRANCH     Branch prediction miss rate/ratio
       CACHES     Cache bandwidth in MBytes/s
         CBOX     CBOX related data and metrics
        CLOCK     Power and Energy consumption
         DATA     Load to store ratio
       ENERGY     Power and Energy consumption
  FALSE_SHARE     False sharing
    FLOPS_AVX     Packed AVX MFLOP/s
           HA     Main memory bandwidth in MBytes/s
                  seen from Home agent
       ICACHE     Instruction cache miss rate/ratio
           L2     L2 cache bandwidth in MBytes/s
      L2CACHE     L2 cache miss rate/ratio
           L3     L3 cache bandwidth in MBytes/s
      L3CACHE     L3 cache miss rate/ratio
          MEM     Main memory bandwidth in MBytes/s
         NUMA     Local and remote memory accesses
          QPI     QPI Link Layer data
     RECOVERY     Recovery duration
         SBOX     Ring Transfer bandwidth
     TLB_DATA     L2 data TLB miss rate/ratio
    TLB_INSTR     L1 Instruction TLB miss rate/ratio
         UOPS     UOPs execution info
    UOPS_EXEC     UOPs execution
   UOPS_ISSUE     UOPs issueing
  UOPS_RETIRE     UOPs retirement
CYCLE_ACTIVITY    Cycle Activities
```

Figure 7: List of PMC groups provided by Likwid tool on HCLServer2

[6] Z. Zhou, J. H. Abawajy, F. Li, Z. Hu, M. U. Chowdhury, A. Alelaiwi, K. Li, Fine-grained energy consumption model of servers based on task characteristics in cloud data center, IEEE access 6 (2018) 27080–27090.

[7] J. Haj-Yihia, A. Yasin, Y. B. Asher, A. Mendelson, Fine-grain power breakdown of modern out-of-order cores and its implications on skylake-based systems, ACM Transactions on Architecture and Code Optimization (TACO) 13 (4) (2016) 56.

[8] A. Shahid, M. Fahad, R. Reddy, A. Lastovetsky, Additivity: A selection criterion for performance events for reliable energy predictive modeling, Supercomput. Front. Innov.: Int. J. 4 (4) (2017) 50–65.

[9] HCL, HCLWattsUp: API for power and energy measurements using WattsUp Pro Meter (2020).
URL `https://csgitlab.ucd.ie/manumachu/hclwattsup`

[10] E. Rotem, A. Naveh, A. Ananthakrishnan, E. Weissmann, D. Rajwan, Power-Management architecture of the intel microarchitecture Code-Named sandy bridge, IEEE Micro 32 (2) (2012) 20–27.

[11] HCL, DE-Meter: Calculate dynamic energy consumption using rapl meter (2019).
URL `https://github.com/ArsalanShahid116/DE-METER`

[12] C. Gough, I. Steiner, W. Saunders, Energy Efficient Servers Blueprints for Data Center Optimization, Apress, 2015.