Energy Consumption Analysis: on a Multicore Architecture

Cioffi Daniele, Consalvo Mario, Delli Paoli Michele
Department of Computer Science
University of Salerno
Italy

***Abstract*—In order to assess how much energy is consumed during a computation, careful measurements are needed. Intel's Running Average Power Limit (RAPL) interface is a powerful tool for this purpose. RAPL provides power limiting functions and accurate energy readings for CPU and DRAM that are easily accessible through various interfaces. The project focuses on measurements and benchmarking during computation aimed at solving arithmetic intensity problems such as matrix multiplication. Our results are shown in different forms for a comprehensive analysis.**

I. Introduction

Energy efficiency in data centers has become one of the biggest possibilities in the last decade not only because of the monetary cost but also because of environmental sustainability [1]. Data center power consumption is steadily increasing and there is a need to apply optimizations in hardware and software to achieve the best performance per watt. Many forecasting models or techniques require an accurate measurement of data center energy consumption. Until recently, measuring the energy consumption of a computer system required a separate metering system. In addition to the difficulties of purchasing, deploying and using external power meters, their measurement accuracy and granularity are usually inadequate for detailed analysis. Also, it is not possible to divide the power into different parts of the computing system inside the chip. It is possible to use model-based power estimation, which uses a set of performance counters and a computational model to transform performance readings into estimates of energy consumption. The accuracy of this approach is highly dependent on the quality of the model and is generally unable to give good results especially with highly fluctuating workloads. There are few research works regarding this problem and for this reason our study is focused to find a correlation between energy consumption and multi-threading applications on a data-center like server. This work will utilize Intel's Running Average Power Limit (RAPL) interfaces for data collection. While this interface provides the ability to set power limits for various chip domains, we will only read energy state registers, in order to better understand how the energy consumption varies from different optimizations of the same application.

II. State of art: RAPL

Intel's RAPL interface is a feature introduced in the Intel Sandy Bridge architecture. RAPL allows energy consumption to be measured at very fine granularity and a high sampling rate, on different system domains. To validate the accuracy of these energy measurements many researchers [2] have compared RAPL evaluation with external devices measurements. The results [3] are overall comparable and Intel's interface becomes more accurate with newer CPUs generations.

Domains. RAPL supports multiple power domains. Each power domain informs the energy consumption of the domain, energy measurement units, minimum or maximum power supported by the domain. The supported domains [1] (on some architecture some domains data are not available through RAPL) are:

- Package (PKG): measures the energy consumption of the entire socket. It includes the consumption of all the cores, integrated graphics, last level caches memory and controller.
- Power Plane 0 (PP0) : measures the energy consumption of all processor cores on the socket.
- DRAM: measures the energy consumption of random access memory (RAM) attached to the integrated memory controller.
- PSys: (introduced with Intel Skylake) monitors and controls the thermal and power specifications of the entire SoC and it is useful especially when the source of the power consumption is neither the CPU nor the GPU. For multi-socket server systems, each socket reports its own RAPL values.

How to read data. There are currently three ways to read RAPL results using the Linux kernel:

- Reading the files under *"/sys/class/powercap/intel-rapl/intel-rapl:0"* using the powercap interface. This requires no special permissions, and was introduced in Linux 3.13.
- Using the perf_event interface with Linux 3.14 or newer. This requires root (or a paranoid less than 1) *sudo perf stat -a -e power/energy-cores/ /bin/ls* . Available events can be found via perf list or under *"/sys/bus/event_source/devices/power/events/"*.
- Using raw-access to the underlying MSRs under *"/dev/msr"*.

This requires root.

To read the energy programmatically there are some C libraries developed around the three options mentioned before. For this work, we've chosen to start from a code base [4] developed by a researcher from the University of Maine, that uses the sysfs method to read energy from the powercap interface.

III. Methodology

Energy Consumption values. To get the energy consumption of a running application, a library based on RAPL powercap interface has been developed. The library methods return an energy value, in Joule, which states the energy consumption of the computation made by the application. Because of the System's architecture of the environment on which we executed our tests, we've only focused on reading the energy consumption of the Package Domain (PKG), which means that we've measured the energy consumption of the entire socket. More details about the Experimental Environment will be discussed in the following relative paragraph.

Power values. Because the energy consumption values are dependent from the execution time of the running application, we needed to also get the power (in Watt), which is an instant value. So, we took the power value multiple times during our programs's execution tests, with a sampling rate of 0.2 seconds.

Test methodology. We have decided to use the median as a reference parameter. The median is used as an alternative to the arithmetic mean because it is not affected by the presence of anomalous data and it is a value actually obtained during the measurement. Every execution has produced *[execution time / sampling rate]* number of power measurements and from these results, we choose the median value. Then, this median value has been integrated with the execution time and with the energy consumption value of each single execution.

Experimental Environment. The experimental environment in which we've tested our application's energy consumption is based on a NUMA Architecture. This machine has 2 main sockets, each one composed of 6 Intel Xeon CPU E5-2430 Cores. Each Core allows Hyperthreading, so each socket can schedule up to 12 threads, 24 in total for both two sockets. Each socket is identified as NUMA Node, so that we have Node 0 and Node 1. The distribution of the Cores among the NUMA Nodes is the following:

- all the Cores with an even identifier are located on NUMA Node 0.
- all the Cores with an odd identifier are located on NUMA Node 1.

To compile our applications, we've used GCC Compiler (version 7.5). To run in parallel our applications we've used OpenMP (version 4.5).

OpenMP Configuration. Dealing with NUMA Architecture, we've faced three main problems: Thread Affinity, Loop Scheduling and Thread Migration. To solve these problems, we've applied some OpenMP configurations. Knowing the Core's distribution among the Nodes is very useful to solve the Thread Affinity and the First Touch Placement Policy problems on a NUMA Architecture.

In our application codes, we initialize data in a sequential way, and we only execute the computational part in parallel. Because of the First Touch Placement Policy, the thread running in sequential initializes data on the memory of the NUMA Node which contains the Core on which the current thread is scheduled. To take advantage of locality, we wanted to force threads to run on a specific subset of Cores, because it impacts memory performance. In other words, we wanted to run our applications by scheduling the threads only on the Cores of a single socket, in order to exploit the best performance in terms of memory access.

To force threads to be scheduled on a specific subset of Cores, we've set an OpenMP environment variable called "OMP_PLACES" to the following value:

OMP_PLACES={0},{12},{2},{14},{4},{16},{6},{18},
{8},{20},{10},{22},{1},{13},{3},{15},{5},{17},
{7},{19},{9},{21},{11},{23}.

By doing this, the thread which initializes data will be scheduled on one of the Cores on NUMA Node 0, so that it will allocate Memory of the NUMA Node 0. When we're going to execute the computation in a parallel way, all the other threads will be scheduled on all the Cores with an even identifier first, which are located on NUMA Node 0, in order to exploit locality.

To solve the Loop Scheduling problem, we've set the OpenMP environment variable named "OMP_SCHEDULE" to static, to guarantee that all the chucks, in which the iteration space is divided, are assigned to threads statically, in a Round-Robin fashion, and not dynamically in a First to Finish order. Finally, to avoid that a thread, started on a specific Core, may migrate to another one, we've set the OpenMP environment variable named "OMP_PROC_BIND " to true.

IV. Implementation

After having carried out a preliminary study of the environmental problems (on NUMA) and of the measurement values to be obtained, our focus was on a possible implementation that would allow us to carry out measurements of energy consumption for our case.

Computational problems. In order to make our measurements, it was necessary to have implementations of solutions to some arithmetic problems. Our choice fell on two common problems:

- *Matrix multiplication*: it is a mathematical problem where a multiplication is made between two matrices. This problem is classified as a problem with a high arithmetic intensity.
- *Dot product*: an algebraic operation that takes two equal-length sequences (in our case two matrices) of numbers, and returns a single number. This problem is classified as a low arithmetic intensity problem.

We've implemented, for each problem, several versions with specific optimizations:

- A first version, which is sequential, and uses no optimization at all.

- A multi-threaded version, which uses OpenMP to run compiler directives, such as *#pragma omp parallel*, with two, four, and eight threads.
- A Vectorization based version, which uses AVX intrinsics vectorization for matrix-multiplication with two, four, and eight threads; and uses OpenMP auto-vectorization for dot-product (by using all the available threads).

For the Vectorization based version, we've obtained the generated code by the compiler by using an external tool called "Compiler Explorer", to verify that vectorial instructions were actually used.

Code library. Once we wrote the code for matrix-multiplication and dot-product, we've developed an ad hoc library that contains the methods necessary for the detection of our findings data during the computation. Two structs hold the power and energy data, which contains information about total cores, total packages, cpu model, energy units and domains availability. To proper detect the information mentioned before we use the following methods:

- *detect_cpu(Rapl_info rapl)*, it detects the cpu model by reading the information under "*/proc/cpuinfo*" path;
- *detect_packages(Rapl_info rapl)*, it detects the number of packages under */sys/devices/system/cpu/cpu*/topology/physical_package_id* path.

Total energy consumption is obtained by invoking the following methods in the next order:

- *rapl_sysfs(Rapl_info rapl)* first, to map the "*/sys/class/powercap/intel-rapl/intel-rapl:**" folders;
- *rapl_sysfs_start(Rapl_info rapl)*, to start energy consumption measurement;
- *rapl_sysfs_stop(Rapl_info rapl)* to stop energy consumption measurement;
- *rapl_get_energy(Rapl_info rapl)* to retrieve energy consumption value.

To read the power consumption two methods need to be called:

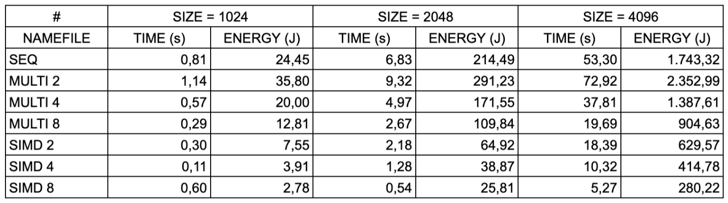
- *rapl_power_sysfs(Rapl_info rapl, Rapl_power_info rapl_power)* maps the "*/sys/class/powercap/intel-rapl/intel-rapl:**" folders.
- *read_power(Rapl_info rapl, Rapl_power_info rapl_power, int delay, int total_time, char *source_file_name)* reads the power consumption of the available domains every delay milliseconds.

We added the previous methods in our codes to obtain results.

V. Discussion

It was certainly not easy to get consistent results.

Our measurements could not be based on a single execution of the code as possible anomalies due to the environment could affect the correct execution of the code itself. For this reason, we tested each different version of our applications 10 times, in order to get enough data to analyze and exclude any outlier. As mentioned before, we've chosen to use the median value for the power value. Moreover, in order to work always on the same data, we've chosen to initialize the Matrices used in our applications by using the random function with a fixed seed. This way, there won't be any differences between multiple executions of the same code, so that the obtained results can be reproduced. Moreover, because the use of a NUMA architecture certainly does not facilitate or speed up the detection operations, on the contrary, a lot of time has been dedicated to the configuration of the environment variables as mentioned in the previous chapters.



#	SIZE = 1024		SIZE = 2048		SIZE = 4096	
NAMEFILE	TIME (s)	ENERGY (J)	TIME (s)	ENERGY (J)	TIME (s)	ENERGY (J)
SEQ	0,81	24,45	6,83	214,49	53,30	1.743,32
MULTI 2	1,14	35,80	9,32	291,23	72,92	2.352,99
MULTI 4	0,57	20,00	4,97	171,55	37,81	1.387,61
MULTI 8	0,29	12,81	2,67	109,84	19,69	904,63
SIMD 2	0,30	7,55	2,18	64,92	18,39	629,57
SIMD 4	0,11	3,91	1,28	38,87	10,32	414,78
SIMD 8	0,60	2,78	0,54	25,81	5,27	280,22

Fig. 1. Values obtained from the execution of three different optimizations with three different input sizes - matrix-multiplication.

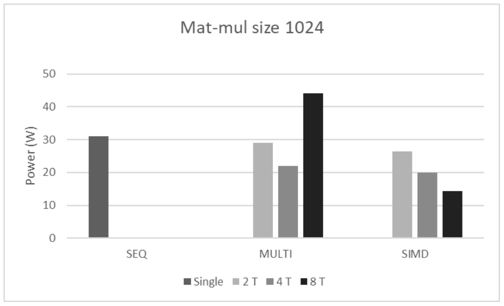


Fig. 2. Power value chart - matrix-multiplication - input size 1024

Matrix-multiplication results.

Figure 1. shows the results obtained from the energy measurements of the different optimizations applied to the Matrix Multiplication algorithm. The baseline of our measurements is the sequential code, which has been executed with an input size parameter of 1024, 2048 and 4096. As we can see from table, the energy consumption of the application is directly proportional to the execution time:

- the sequential version consumes 24 Joule for an execution of 0,81 seconds, which has a matrix size of 1024x1024;
- it consumes 214 Joule for an execution of 6,83 seconds, which has a matrix size of 2048x2048;
- it consumes 1743 Joule for an execution of 53,3 seconds, which has a matrix size of 2048x2048.

This trend is also found in the other executions with different applied optimizations. Another observation we can make from these results is that, the more optimizations we apply, the more the execution time decreases, and consequently the more the energy consumption decreases. So, the energy consumption is inversely proportional to the number of optimizations made.

The histogram graph in Figure 2. depicts the results obtained from the power measurements of the Matrix Multiplication algorithm's executions with an input size of 1024. These

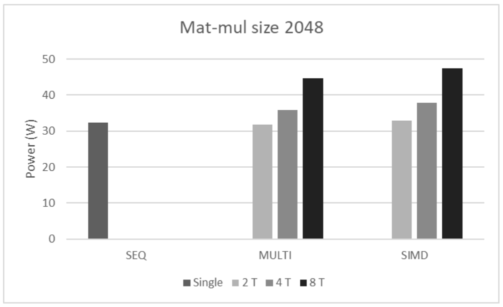


Fig. 3. Power value chart - matrix-multiplication - input size 2048

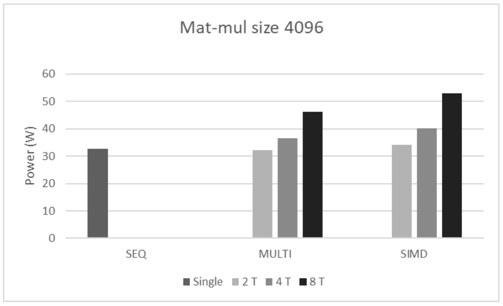
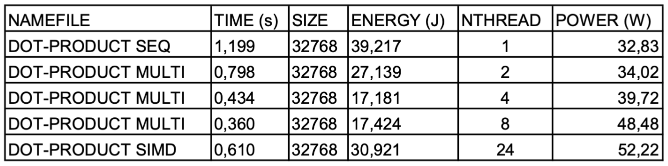


Fig. 4. Power value chart - matrix-multiplication - input size 4096



NAMEFILE	TIME (s)	SIZE	ENERGY (J)	NTHREAD	POWER (W)
DOT-PRODUCT SEQ	1,199	32768	39,217	1	32,83
DOT-PRODUCT MULTI	0,798	32768	27,139	2	34,02
DOT-PRODUCT MULTI	0,434	32768	17,181	4	39,72
DOT-PRODUCT MULTI	0,360	32768	17,424	8	48,48
DOT-PRODUCT SIMD	0,610	32768	30,921	24	52,22

Fig. 5. Values obtained from the execution of three different optimizations with only one input size

results show an unexpected trend because we can't state any correlation between the number of threads used during the execution and the power consumption, maybe because of the relatively small input size. More specifically:

- the sequential's power consumption is 30,9 W;
- the multithreaded's power consumption with 2 threads is 29 W;
- the multithreaded's power consumption with 4 threads is 21,89 W;
- the multithreaded's power consumption with 8 threads is 45 W;
- the simd's power consumption with 2 threads is 26,5 W;
- the simd's power consumption with 4 threads is 20 W;
- the simd's power consumption with 8 threads is 14,4 W.

In Figure 3. is possible to see how the less optimized version of the code turns out to have a lower power value. This trend should be confirmed by tests carried out with an even greater input size. With reference to the execution time and energy consumption of Figure 1, we can observe that as these values decrease, the value of the power increases. This increase in terms of power can also be seen in reference to the number of threads used during execution. In other words, we can state a direct correlation between the number of threads used by the different optimizations and the power consumption.

Figure 4. shows the trend of the power value, as the code optimizations vary. The lower power value is the one of the sequential encoding. The two optimizations, multithreaded and simd with only two threads, appear to have a greater power value than the one of the sequential version. Comparing this graph with the table in Figure 1, we can observe that the optimizations that report a shorter execution time are, on the other hand, the ones that obtain a higher power value.

Dot-product results.

Figure 5. shows the results obtained from the energy measurements of the different optimizations applied to the dot-product algorithm. The baseline of our measurements is the sequential code, which has been executed with an unique input size parameter: 32768.

Because the OpenMP SIMD directive doesn't allow to set the number of threads to execute, this version executes on all the available threads of the NUMA Architecture previously described.

This graph shows the power values for running the different dot-product optimizations. It is possible to note that the execution times are very small, and for this reason the values obtained are not very consistent. However, it is possible to notice the same previous trend as in matrix-multiplication results: as the parallelism increases, the value of power increases too.

VI. Conclusions

This project has been very interesting under different aspects. The approach to such a current problem as that of energy consumption and the approach to using a NUMA architecture has contributed to increasing our knowledge about them. In practical terms, this work has expanded our understanding about how to make performance benchmarking and how to interpret the results.

Depending on the point of view, the consideration of which version code is the best, in terms of energy consumption, may change: the results show that a shorter execution time consumes less energy, but at the same time, a shorter execution time is obtained by increasing the parallelism. From our results, we observe that the more we increase the parallelism, the higher the power consumption is, which means that the cores work more intensively but in a smaller range of time. The choice of the best version code must take into account all these considerations to make a right tradeoff for the prefixed objectives.

VII. SOURCE CODE

All source code (including library code, implementation code) and the results are available at the following link: https://github.com/co5m0/HPC-Energy

References

[1] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K., Nurmien, *RAPL in Action: Experiences in Using RAPL for Power Measurements*, Beijing University of Posts and Telecommunications, China, 2018.

[2] Spencer Desrochers, Chad Paradis, Vince Weaver, *A Validation of DRAM RAPL Power Measurements*, USA, 2016.

[3] Vince Weaver, https://github.com/deater/rapl_validation

[4] Vince Weaver, https://www.github.com/deater/uarch-configure/tree/master/rapl-read