Building the Table of Energy and Power Leverages for Energy Efficient Large Scale Systems

Issam Raïs*, Mathilde Boutigny*, Laurent Lefèvre*, Anne-Cécile Orgerie†, Anne Benoit*
*University of Lyon, Inria, CNRS, ENS de Lyon, Univ. Claude-Bernard Lyon 1, LIP
Email: issam.rais@inria.fr, laurent.lefevre@inria.fr, anne.benoit@ens-lyon.fr
†Univ Rennes, Inria, CNRS, IRISA, Rennes, France
Email: anne-cecile.orgerie@irisa.fr

***Abstract*—Large scale distributed systems and supercomputers consume huge amounts of energy. To address this issue, an heterogeneous set of capabilities and techniques that we call leverages exist to modify power and energy consumption in large scale systems. This includes hardware related leverages (such as Dynamic Voltage and Frequency Scaling), middleware (such as scheduling policies) and application (such as the precision of computation) energy leverages. Discovering such leverages, benchmarking and orchestrating them, remains a real challenge for most of the users. In this paper, we formally define energy leverages, and we propose a solution to automatically build the table of leverages associated with a large set of independent computing resources. We show that the construction of the table can be parallelized at very large scale with a set of independent nodes in order to reduce its execution time while maintaining precision of observed knowledge.**

***Index Terms*—energy leverages, high performance computing, energy efficiency.**

I. Introduction

Large scale distributed systems consume huge amounts of energy. This consumption has direct financial and environmental consequences for Cloud and supercomputer infrastructures.

To increase the energy efficiency of data centers and to lower their consumption, several techniques have been developed. These techniques, named energy leverages, can act at three levels: hardware, middleware, and application. At the hardware level, Dynamic Voltage and Frequency Scaling (DVFS) [1], [2] and shutdown techniques [3], [4] constitute the two most studied leverages. DVFS reduces the voltage and frequency of processors when they do not require all their computational power. As for shutdown techniques, they consist in switching off entire servers or putting them in sleep modes when they are idle.

At the middleware level, data center managers can employ energy-efficient resource allocation policies to schedule the jobs by respecting energy budget or power cap [5], consolidating the workload on fewer servers [6], benefiting from intermittent renewable energy sources [7], or modifying the number of used OpenMP threads [8].

Finally, leverages at the application level include green programming [9], vectorization techniques [10], and computation precision [11].

While studies have been conducted on each of these leverages, only few work considers combining them. For instance, in [8], and [12], the authors combine the number of OpenMP threads and DVFS, and in [13], the authors combine shutdown and DVFS leverages. In the case of shutdown, this leverage has obvious impacts on other leverages: in the off state, no other leverage can be employed at the application level, for instance. Indeed, the utilization of a given energy leverage can impact both the utilization and the efficiency of another leverage. Moreover, the variety of leverages and the complexity of modern hardware architectures, in terms of size and heterogeneity, makes the energy efficiency more complex to reach for users. The authors of [14] and [15] propose energy-aware runtimes that exploit the dynamic behavior of HPC applications in order to improve performance and energy-efficiency. Yet, energy leverages' characterization is a prerequisite for building such runtimes.

In this paper, we propose a first approach toward a completely automated process to characterize the energy leverages available on data center servers. The key idea of our contribution consists in building a score table with a value for each leverage combination and each studied metric. These scores are obtained through the execution of a representative benchmark. Based on this score table, we can provide hints to users about the most suitable solution for their application.

This paper makes the following contributions:

1) We propose a generic framework formalizing the combination of leverages through the definition of a table of energy leverages;
2) We present a comprehensive experimental method based on benchmarks and a detailed overview of its concrete implementation to build the table of energy leverages;
3) We analyze experimental results on several servers demonstrating how to parallelize the building of the table.

The remaining of this paper is structured as follows. Section II presents our definition and formalism of a leverage and details the leverages that are used as examples in this study. Section III shows our process to build the table of leverages, and Section IV explains how this formalism is implemented. Section V presents the experimental setup and a first full example of table of leverages. Section VI demonstrates the parallelization of the creation process of the table of leverages. Finally, Section VII concludes this work and gives perspectives.

II. Leverage definition

A *leverage* L is composed of $S = \{s_0, s_1, \ldots, s_n\}$, the set of available valid states of L, and s_c, the current state of L.

An energy or power leverage is a leverage that has an impact on the power or energy consumption of a device: the energy consumption may differ depending on the current state at which the application is executed. Of course, switching from one state to another can have a cost in terms of time and energy, but we focus on studying the impacts of leverage combinations over a single intensive application phase, and thus we do not study the switching costs between states in this work.

We focus on three leverages that are available on current hardware: multi-thread, computation precision and vectorization. While multi-thread is a middleware level leverage, the two last are at the application level, as detailed below.

1) Multi-thread leverage: Multi-core architectures are nowadays the de facto standard in modern supercomputers. The first studied leverage is a middleware-level leverage that permits the usage of multiple cores during computation. OpenMP [16], a well-known application programming interface abstraction for multi-threading, can be used to exploit this intra-node parallelism of multi-cores. In particular, it is well known for its simplicity and portability. It consists of a set of directives that modifies the behavior of the executed code, where a master thread forks a specific number of slave threads that run concurrently. This multi-thread leverage increases the CPU utilization of the node. Consequently, because of the non-power proportionality of current hardware architectures [3], this leverage can improve the energy efficiency of the node.

This leverage is denoted by *nbThreads* or *#Threads*, and the set of states is $\{1, \ldots, n_{max}\}$, where 1 means that one OpenMP thread is used, and n_{max} corresponds to the maximum number of threads that could be launched simultaneously on the node without hyperthreading.

2) Computation precision leverage: The second leverage belongs to the application level and exploits the various computation precision options available on actual hardware (i.e., int, float, double). Such a leverage alters the precision of the results computed by the application, but lower precision translates into shorter data representation and so, less computation and less energy consumption. At the application level, the user can specify a desired Quality-of-Service that can be expressed as accessible computation precision states.

This precision leverage is denoted by *Precision* or *Prec.*, and the set of states is {int, float, double}, corresponding to the data format for the application. For each of these states, a different code version is provided.

3) Vectorization leverage: Finally, the last studied leverage concerns the application level. Current CPUs allow the usage of vectorization capabilities to exploit intra-core parallelism. On Intel architectures, it started with MMX instruction in Pentium P5 architectures in 1997 [17]. It was then extended to SSE [18]. SSE was then extended to SSE2, SSE3, SSSE3 and finally SSE4. AVX [19] then introduces new instructions, followed by AVX2 and finally AVX512 available in XeonPhi architecture. In this paper, we focus on SSE3 and AVX2, which are representative of the SSE and AVX families. These instruction sets permit single instruction on multiple data (SIMD) at application level.

This vectorization leverage is denoted by *Vectorization* or *Vect.*. The set of states is {none, SSE3, AVX2}, where *none* means that no vectorization is used. For each of these states, a different code version is provided using the adequate compilation flags for each version.

Note that the methodology proposed in this paper can be applied to any number and any type of energy leverages, even though we focus here on three leverages, chosen to be representative examples of available energy leverages on modern architectures.

III. Formalism of table of leverages

In this section, we describe the methodology applied to build a table of energy leverages, which relies on metrics and benchmarks to characterize the performance and energy impact of each leverage combination on a given node. For each metric and each benchmark, a score is attributed to a given leverage combination. First, we describe the basic concepts used to build the table: the metrics and benchmarks. Then, we present the formal definition of the table of leverages, and finally, the methodology for building it.

A. Metrics

Leverages may influence the quality of service or performance of an application. For instance, shutdown techniques may induce latency in waking up the required nodes. Consequently, for these leverages, users need to determine their acceptable trade-off between energy-related metrics and performance metrics.

Here, three different metrics that represent both energy and performance constraints are explored. These metrics are measured for a given period of time corresponding to the time spent during the execution of the benchmark.

The two first metrics are energy and power related metrics. To define them, we introduce the following notations:

- $T = \{t_0, \ldots, t_N\}$ is the set of time stamps of energy consumption measurements of a given run; t_0 and t_N represent the starting and ending timestamps, respectively;
- p_j, $j \in [0, N]$, represents the power consumption (in Watt), of the considered node for the timestamp t_j.

a) Average Watt: denoted $avrgWatt$, it represents the average power consumption of a chosen run. It is defined as follows:

$$avrgWatt = \frac{\sum_{j \in [0,N]} p_j}{N+1}. \tag{1}$$

b) Joules: denoted $Joules$, it represents the energy consumption of the run. It contains the energy consumption of the complete node used between t_0 and t_N. It is defined as follows:

$$Joules = \sum_{j \in [0,N-1]} (t_{j+1} - t_j) \times p_j. \tag{2}$$

c) Execution time: Finally, the execution time, denoted $Time$, is the whole execution time of a run, including initialization time.

B. Benchmarks

A benchmark corresponds to a self-contained application that is representative of typical applications or portions of applications. The benchmark is compiled before the run, and once launched, the metrics previously defined are collected during its execution.

Here, for the sake of clarity, we evaluate only one benchmark for a set of embedded leverages. We chose to focus on a well-known CPU intensive code: the line per line matrix multiplication (LpL MM) of dense random matrices. The same algorithm is implemented for the various leverage combinations. As detailed in Section II, the considered leverages are multi-thread, computation precision and vectorization. For the last two leverages, a different state means a different version of code, here generated by hand. Automatic generation is possible but it is not the focus of this paper.

C. Formalization of the table of leverages

1) Format of the table of leverages: Here, we describe how to compute the score associated to each metric for each leverage. Let X, Y, Z be the sets of available states of three leverages χ, ψ, ω: $X = \{x_0, \ldots, x_{n_x}\}$, $Y = \{y_0, \ldots, y_{n_y}\}$, and $Z = \{z_0, \ldots, z_{n_z}\}$.

Let $g_1, \ldots, g_m$ be the measured metric functions, as for instance $avrgWatt$, $Joules$, and $Time$. For all u ($1 \leq u \leq m$), $g_u(x_i, y_j, z_k)$ is the value of metric g_u for the states x_i, y_j, z_k respectively for the leverages χ, ψ, ω.

In the table of leverages, each line corresponds to a combination of states for each leverage and the columns correspond to the measured metrics. In order to ease the comparison, we normalize each value on the minimum value for each metric. These normalized values constitute the scores indicated in the table of leverages. Let $h_1, \ldots, h_m$ be the normalized versions of $g_1, \ldots, g_m$. So, we have, for $1 \leq u \leq m$:

$$h_u(x_i, y_j, z_k) = \frac{g_u(x_i, y_j, z_k)}{\min\limits_{x_{i'} \in X, y_{j'} \in Y, z_{k'} \in Z} g_u(x_{i'}, y_{j'}, z_{k'})},$$

with $h_u(x_i, y_j, z_k)$ being the value in the table of leverages in column of metric u and corresponding to the line for the states x_i, y_j, z_k respectively for the leverages χ, ψ, ω.

2) Methodology to build the table: Building the table of leverages requires to run the benchmark in its adequate version for each leverage combination. Hereafter, we describe our methodology for running all the required executions.

Algorithm 1 shows the generic pseudo-code to execute the adequate benchmark version on the correct leverage combination for a given set of metrics. This algorithm has two inputs: $LeverageTree$ is a tree representing the set of selected states on the studied leverages, and $SelectedStates$ keeps trace of every current state of leverage involved so far. The functions $root(X)$ and $unseen\_children(X)$ return respectively the root of tree X, and the first unseen children node of tree X. mM corresponds to an entity gathering metric values (as defined before in our case: $avrgWatt$, $Joules$, and $Time$). $Benchmark$ corresponds to the entity that matches the current state of every leverage $SelectedStates$ and the corresponding binary file to execute, $exec$ corresponds to the execution of the benchmark. Thus, for all the considered leverages ($LeverageTree$), the algorithm is executed recursively over their respective states (S) and collects the metrics (mM) before moving to the next leverage combination. The metrics gathered during the executions are saved in the $TableOfLeverages$ entity.

Algorithm 1: Building the table of leverages: benchmark execution for each leverage combination for a given set of metrics.

```
Input: LeverageTree: leverages to benchmark
Input: SelectedStates: name of states of leverages being
       currently benchmarked
mM: metric measurements;
for s_c in root(LeverageTree).S do
    if root(LeverageTree) is leaf then
        Add s_c to SelectedStates;
        mM.start();
        Benchmark(SelectedStates).exec();
        mM.end();
        tableOfLeverages[SelectedStates] ← mM;
    else
        Add s_c to SelectedStates;
        Algorithm1(unseen_children(LeverageTree),
        SelectedStates);
    end
end
```

Figure 1 shows an example of input used for Algorithm 1 in the following table of leverages of this paper. Rounded bullets represent states of the three considered leverages. The benchmark chooses the corresponding binary, for leverages having different binaries in set of states S, here *Precision* and *Vectorization*. Leverage *nbThreads* changes its state through environment variable.

When the execution of Algorithm 1 with Figure 1 as input is finished, the table of leverages is complete for the considered benchmark.

IV. Implementation of formalism

In order to be able to build the table of leverages, we created a framework able to identify available known leverages on a given hardware, benchmark the leverages combination and collect the associated metrics. This tool can run on a single node or on an entire cluster. It is designed to be as flexible as possible on three basic concepts: leverage, metric and benchmark. This framework fits the needs of a wide type of users, going from basic users without specific knowledge to more experts ones capable of implementing new leverages, benchmarks and metrics collection methods.

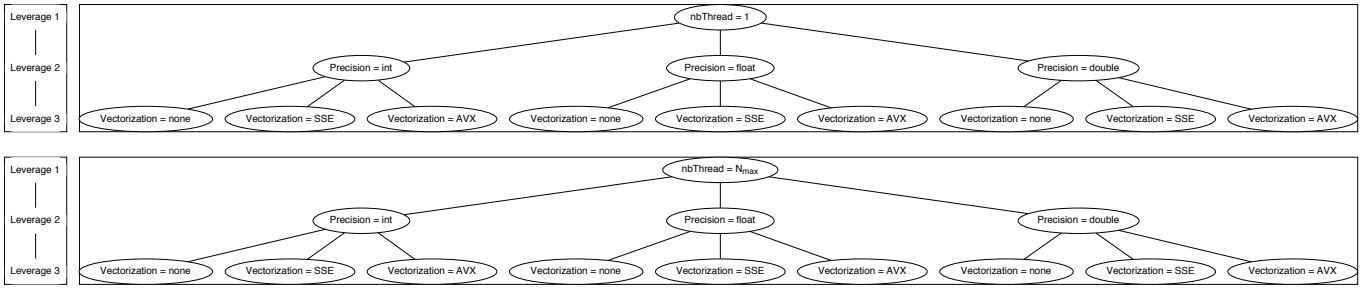


Fig. 1: Example of $LeverageTree$ input for Algorithm 1.

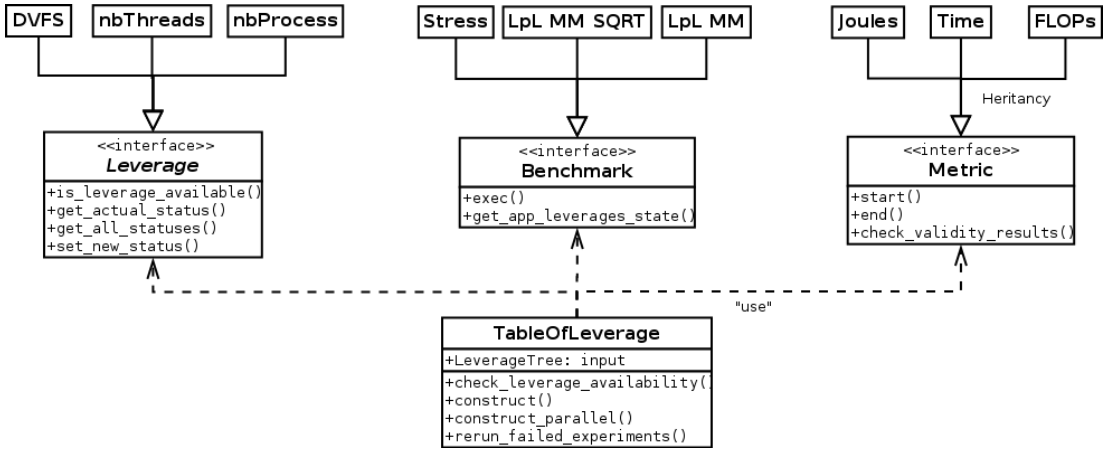


Fig. 2: Framework UML diagram

A. Leverages

As shown in Figure 2, the framework provides multiple interfaces, or contracts, to fully describe our basic concepts. These contracts are implemented through a fully abstract class, forcing a class inheriting from it to implement the needed functions. Thus, every leverage class must be able to detect it's availability (*is_leverage_available()*), to retrieve it's current status (*get_actual_status()*), to retrieve it's list of available statuses (*get_all_statuses()*) and to change it's actual status with a valid one (*set_new_status()*). For example, the availability of the DVFS can be validated if the file */sys/devices/system/cpu/cpu0/cpufreq* exists. The actual status of the DVFS leverage for a specific core, here 0, can be found in a configuration file */sys/devices/system/cpu/cpu0/cpufreq/cpuinfo_cur_freq*. Reading the *scaling_available_frequencies*[1] file permits to extract all statuses. Finally, the current state can be changed using the command *cpufreq-set*.

B. Metrics

The framework provides an interface for the metrics. It imposes to be able to start and end the monitoring of a metric. It also imposes a metric to check the validity of obtained results (*check_validity_results()*). The actual implementations of the metric contract allows various focus for various metrics.

Grid'5000 provides the Kwapi API [20], to get the collected data from the wattmeters for a given time period. Once a benchmark has been executed on a node, the contract asks Kwapi for the node's consumption during this time period.

Another possibility given by Grid'5000 is to use the live metric webpage. This webpage returns the consumption metrics of the nodes of Grid'5000 every second. We created a script that collects them every second. The framework then gets the metrics from the script by giving the starting and ending timestamps of the benchmark execution.

Such method could also be exported to platforms without wattmeters. For example, we implemented the contract for captors such as RAPL or IPMI, also to get energy related metrics.

The framework also implements a contract to retrieve FLOPs (FLoating point OPeration per seconds). In order to retrieve the FLOPs, a script that collects the flops metrics during a benchmark execution using the PAPI framework is deployed on used nodes. This method is not specific to Grid'5000 and could be used on a different architecture.

C. Benchmark

The final contract is relative to the benchmark execution. The first function executes the given binary. The

[1]Path file is */sys/devices/system/cpu/cpu0/cpufreq/*

second gives the current states of application leverages. A family of leverage is relative to the application. Thus, the state of application leverages changes for every binary (*get_app_leverages_state()*).

The framework copies and executes the given binary on chosen nodes. We assume that compiled and ready to used binary files are passed to the framework.

D. Construction of the table of leverages

The table of leverage class uses previously presented contracts to implement algorithm 1. Various mode of construction are provided.

a) Default method: construct() method runs the same experiment on every node. Thus, every node will make the same leverage exploration. This method can be time consuming. For example, using our testbed, combining the first and last status of *nbThreads* leverage, *Precision* (int, float and double) and *Vectorization* (SSE3 and None) leverages, takes approximately 1 hour and 30 minutes.

b) Automatic node spread work method: In the *construct_parallel()* method, the framework runs one scenario by dividing and assigning automatically work on nodes. The execution time of the framework would be divided by the number of used nodes. If we take the previous example combining the *nbThreads*, *Precision* and *Vectorization*, with 5 nodes, the execution time of the framework for this scenario is around 18 minutes. However, the user will need to have a good knowledge of the nodes and asked metrics in order to ensure coherent results.

V. Experimental setup and first table of leverages

In this section, we present the table of leverages built on a node from our experimental testbed.

A. Experimental setup

To evaluate our methodology in various computing environments, Grid'5000, a large-scale and versatile testbed for experiment-driven research in all areas of computer science, is used as a testbed [21]. Grid'5000 deploys clusters linked with dedicated high performance networks in several cities in France (Lille, Nancy, Sophia, Lyon, Nantes, Rennes, Grenoble).

TABLE I: Server Node characteristics

Features	Taurus	Nova
Server model	Dell PowerEdge R720	Dell PowerEdge R430
CPU model	Intel Xeon E5-2630	CPU E5-2620 v4
# of CPU	2	2
Cores per CPU	6	8
Memory (GB)	32	32
Storage (GB)	2 x 300 (HDD)	2 x 300 (HDD)
Date of arrival	11.2012	03.2017

As our focus is on energy and performance related metrics, we used the Grid'5000 Lyon site, where the energy consumption of every node from all available clusters, as shown in Table I, is monitored through a dedicated wattmeter, exposing one power measurement per second with a 0.125 Watts accuracy.

TABLE II: Table of energy leverage states for LpL MM benchmark on a Nova node

Leverage states			avrgWatt(W)	Joules(J)	Time(sec)
#Threads	Prec.	Vect.			
1	int	none	1.05	65.09	61.89
1	int	SSE3	1.06	28.26	26.56
1	int	AVX2	1.06	29.32	27.67
1	float	none	1.05	72.97	69.67
1	float	SSE3	1.06	33.8	31.89
1	float	AVX2	1.05	36.8	34.89
1	double	none	1.06	81.59	76.89
1	double	SSE3	1.07	58.52	54.89
1	double	AVX2	1.06	57.72	54.22
32	int	none	1.43	13.48	9.44
32	int	SSE3	1.4	4.68	3.33
32	int	AVX2	1.0	1.0	1.0
32	float	none	1.45	7.4	5.11
32	float	SSE3	1.41	3.76	2.67
32	float	AVX2	1.56	3.11	2.0
32	double	none	1.53	8.34	5.44
32	double	SSE3	1.53	8.52	5.56
32	double	AVX2	1.54	7.0	4.56

B. Table of leverages for three leverages

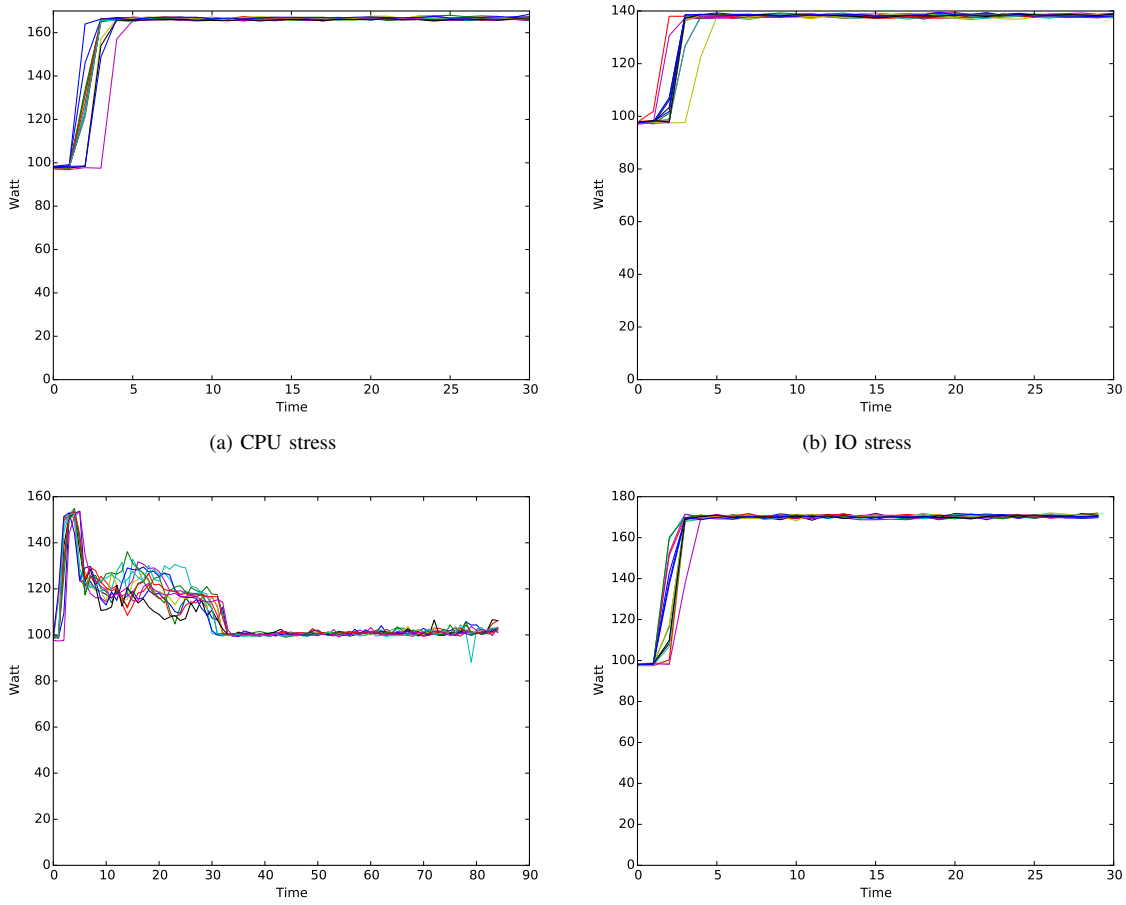
We applied our previous methodology for the three chosen leverages to the CPU intensive benchmark. This allows us to explore all possible states of chosen leverages, and thus to build a complete table of leverages. In this paper, it has the following format: the first three columns present the states of the $nbThreads$, $Precision$, and $Vectorization$ leverages respectively, while the last three columns show the normalized results of the three metrics $avrgWatt$, $Joules$, and $Time$, respectively, for every combination of leverage.

As described in Table II, a line of the table of leverage represents results of all gathered metrics for the execution of a representative load for a chosen combination of leverages. The results are normalized as shown in Section III-C. The table of leverages gathers the knowledge of a node, here Nova (Table I), for a given workload done for multiple states of leverages combined.

Note that the best combination for all metrics used here is always the $\{32$, int, AVX2$\}$ combination. This result is the best combination to choose only if we have no constraints about leverage choices. For example, this table could help a user to choose a combination taking into account a fixed leverage state. Or to answer the following question: is there a leverage or a state of leverage that is always better for a given metric?

VI. Parallelization of the table construction

The construction of the table of leverages may take a long time if many different leverage combinations are considered. In this section, we explain how the construction could be parallelized, so that the time to generate a complete table of leverages could be significantly reduced.



(a) CPU stress

(b) IO stress

(c) HDD stress

(d) RAM stress

Fig. 3: Nova-1, 30 runs of various stresses for Time (seconds) and Power (Watts) metrics

A. *Re-usability of energy and performance metric, one node*

The first hypothesis to be considered, is the fact that a node, exposed to the same load, gives the same metric results, with very low variation.

Previous work on HPC applications indicates that phases could be recognized thanks to an analysis of existing registers [22]. We run various intensive workloads using stress[2], a tool that applies a specific phase to the used node. We execute 30 times every stress on a unique "Nova" node. Figure 3 shows results of such a protocol. For HDD stress (hard drive usage), three types of patterns could be recognized. The first one from $t = 0s$ to $t = 10s$ and last one from $t = 35s$ to $t = 80s$ are the same for every run. The variation occurs between $t = 10s$ and $t = 35s$. When the disk is active and writing in various regions, the energy cost could differ depending on the chosen region for writing. We observe that for CPU, IO and RAM stress, all runs have approximately the same behavior, meaning that one run on the same node always has the same energy consumption, for this kind of stress.

[2]https://people.seas.harvard.edu/~apw/stress/

B. *Re-usability of energy and performance metric, one family hardware*

The second hypothesis that we have to evaluate here is the fact that extracted metrics could also be used by different nodes with the same hardware.

We stress several nodes of the same hardware (Taurus or Nova clusters) to observe how the standard variation concerning various energy-efficiency and performance-related metrics evolve. We chose these two families of hardware to evaluate a newly acquired node family ("Nova", 2017) and an old one ("Taurus", 2012). This evaluation is done with various intensive workloads that stress differently the energy consumption of a node (CPU, IO, hard drive usage, RAM).

Metrics and standard deviation averages of families of nodes are exposed in Table III for CPU, HDD, IO, and RAM workloads. For this experiment, every node is doing the same work at the same time. We get interesting metrics for every run (10 run averages on every node). We then average interesting values for families of nodes. 10 Taurus nodes were used, while 5 Nova nodes were used. The standard variation for the three chosen metrics are negligible for all stress benchmarks,

TABLE III: Average (Av.) and standard deviation (StD.) of various workloads for various energy and performance related metrics for various hardware architectures

Hardware family	Joules (J) Av. - StD.	AvrgWatt(W) Av. - StD.	Time(t) Av. - StD.
CPU			
Taurus	6807.0 - 68.8	205.84 - 1.37	32.81 - 0.39
Nova	4998.86 - 49.3	154.91 - 1.09	32.06 - 0.43
HDD			
Taurus	5055.98 - 365.33	140.58 - 2.98	35.85 - 2.4
Nova	9381.94 - 251.5	107.8 - 0.57	87.01 - 2.47
IO			
Taurus	3957.52 - 34.98	123.46 - 0.21	32.0 - 0.3
Nova	4194.53 - 68.06	130.3 - 0.67	32.04 - 0.66
RAM			
Taurus	5097.83 - 55.81	222.14 - 2.2	32.5 - 0.52
Nova	7282.26 - 115.89	158.53 - 0.8	31.93 - 0.44

except HDD (as already explained in the previous subsection). Even for the $Joules$ metric, we note that the variation is under the second of idle consumption of both nodes for every benchmark, meaning that differences between metric results from the same family of hardware are negligible.

These experiments show that for the same workload, same energy and performance behavior could be witnessed for various nodes having the same configuration.

C. Table of leverages, variability between nodes

In this section, we evaluate the same hypothesis for our previously defined table of leverage (Table II) with 5 nova nodes. For every leverage combination, we evaluate the standard deviation of obtained metrics on all nodes. For every leverage combination, we extract the average and the standard deviation of all used nodes. To understand these numbers easily, we also extract the percentage that represents the standard deviation to the average.

Table IV presents the same exploration that the previously presented Table of leverages II for 5 nova nodes with average, standard variation and percentage represented by the standard deviation to the average, respectively, for every extracted metrics. Table V presents the same exploration but for Taurus nodes. The same matrix dimension is used for both exploration (8192). Note that Taurus nodes don't have *AVX* as available state for the *vectorization* leverage.

Table IV underlines the fact that the most stable metric is undeniably time, with only three combinations of leverage above or equal to 1% and only one combination, $\{1$, int, none$\}$, with a standard deviation above 1 second. Same goes for Table V, where only two combinations of leverages are above 1%.

For Joules metric, every percentage is under 3.33% for Table IV, except for the less consuming combination of leverages, $\{32$, int, AVX$\}$ at 7.07%, which is still reasonable. This high value for this combination could be explained by the fact that our wattmeters are giving a power value every second, thus if a run is a bit longer than the 9 seconds, it will get an extra power value that others won't have. Because there are not a lot of values (one per second), an extra value on such a short run has high repercussions on the standard deviation. Because runs are longer on Table V, percentage are more stable, between 2.34% and 1.32%. Finally, for the *AvrWatt* metric, every percentage is under 3.27% for Table IV and under 3.55 for Table V.

These previous results (Table III, Table IV and Table V) analysis underlines the fact that for the same workload, same energy and performance behavior could be witnessed for various nodes having the same configuration, under a low percentage of difference. Thus, for large scale computing systems with large amount of computing nodes with the same configuration, the table of leverage could be done on only one node, or derived from a segmented construction on multiple nodes, and used as knowledge for other nodes.

VII. Conclusion

Energy efficiency is a growing concern. In the context of HPC and datacenters where the size of infrastructures grows drastically, energy consumption has to be taken into account as a high expense.

There is a wide range of hardware and software techniques, that we formally define as leverages, that permits to modulate the computing capabilities and/or the energy/power used by a device. We propose a generic solution to create a score table about multiple metrics for a given set of leverages, called the table of leverages. We propose a fully implemented and highly modular framework which allows an easy discovery, combination and exploration of leverages. Finally, we underline the fact that parallelization of the building of the table of energy and power leverages permits high gain of time while maintaining high precision.

Possible future work concerns reducing the completion time for building such a table. In fact, the time to solution here could be greatly reduced, for example by predicting which run is not necessary to know values of relevant metric using learning or prediction techniques.

Acknowledgments

This work is supported by the ELCI project, a French FSN ("Fonds pour la Société Numérique") project that associates academic and industrial partners to design and provide software environment for very high performance computing. Experiments were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (https://www.grid5000.fr).

References

[1] Suleiman and al., "Dynamic voltage frequency scaling (DVFS) for microprocessors power and energy reduction," in *International Conference on Electrical and Electronics Engineering*, 2005.

[2] X. Mei, Q. Wang, and X. Chu, "A survey and measurement study of GPU DVFS on energy conservation," *Digital Communications and Networks*, vol. 3, no. 2, 2017.

[3] Rais and al., "Impact of Shutdown Techniques for Energy-Efficient Cloud Data Centers," in *ICA3PP*, Dec. 2016.

TABLE IV: Table of energy leverage states for LpL MM benchmark on a 5 Nova nodes, average, standard deviation and percentage of standard deviation for average

Leverage states			avrgWatt (W)			Joules (J)			Time (sec)		
#Threads	Prec.	Vect.	avrg	std dev	% std dev	avrg	std dev	% std dev	avrg	std dev	% std dev
1	int	none	118.99	3.53	2.96	66041.42	2016.32	3.05	555.0	1.1	0.2
1	int	SSE3	120.48	3.44	2.86	28844.54	865.88	3.0	239.4	0.49	0.2
1	int	AVX2	120.02	3.38	2.81	29862.48	881.6	2.95	248.8	0.4	0.16
1	float	none	118.55	3.32	2.8	74258.92	2085.38	2.81	626.4	0.49	0.08
1	float	SSE3	120.08	3.35	2.79	34416.58	1005.18	2.92	286.6	0.49	0.17
1	float	AVX2	119.48	3.44	2.88	37445.13	1079.41	2.88	313.4	0.49	0.16
1	double	none	120.06	3.43	2.86	83060.58	2379.14	2.86	691.8	0.4	0.06
1	double	SSE3	120.75	3.48	2.88	59578.0	1735.06	2.91	493.4	0.49	0.1
1	double	AVX2	120.45	3.44	2.86	58661.37	1740.92	2.97	487.0	0.89	0.18
32	int	none	161.0	4.41	2.74	13684.79	374.68	2.74	85.0	0.0	0.0
32	int	SSE3	158.72	4.67	2.94	4761.68	140.14	2.94	30.0	0.0	0.0
32	int	AVX2	113.39	3.62	3.19	1044.21	73.82	7.07	9.2	0.4	4.35
32	float	none	163.04	4.46	2.73	7499.91	205.06	2.73	46.0	0.0	0.0
32	float	SSE3	158.58	4.13	2.6	3805.9	99.07	2.6	24.0	0.0	0.0
32	float	AVX2	174.68	5.7	3.26	3211.55	48.17	1.5	18.4	0.49	2.66
32	double	none	171.79	5.14	2.99	8450.54	198.45	2.35	49.2	0.4	0.81
32	double	SSE3	172.62	4.97	2.88	8594.98	205.08	2.39	49.8	0.4	0.8
32	double	AVX2	173.6	5.27	3.03	6978.92	231.53	3.32	40.2	0.4	1.0

TABLE V: Table of energy leverage states for LpL MM benchmark on a 5 Taurus nodes, average, standard deviation and percentage of standard deviation for average

Leverage states			avrgWatt (W)			Joules (J)			Time (sec)		
#Threads	Prec.	Vect.	avrg	std dev	% std dev	avrg	std dev	% std dev	avrg	std dev	% std dev
1	int	none	143.3	3.34	2.33	101226.43	2371.0	2.34	706.4	0.49	0.07
1	int	SSE3	146.49	2.57	1.76	39406.17	693.39	1.76	269.0	0.0	0.0
1	float	none	143.74	2.63	1.83	109872.73	2050.37	1.87	764.4	0.49	0.06
1	float	SSE3	146.56	2.7	1.84	46926.95	842.17	1.79	320.2	0.4	0.12
1	double	none	145.88	2.89	1.98	121376.55	2457.85	2.02	832.0	0.63	0.08
1	double	SSE3	147.59	2.71	1.84	81676.02	1498.01	1.83	553.4	0.49	0.09
24	int	none	215.39	2.9	1.35	24684.46	420.55	1.7	114.6	0.49	0.43
24	int	SSE3	211.95	2.99	1.41	8308.44	146.41	1.76	39.2	0.4	1.02
24	float	none	218.57	3.3	1.51	14906.06	197.2	1.32	68.2	0.4	0.59
24	float	SSE3	210.8	3.15	1.5	6619.05	147.45	2.23	31.4	0.49	1.56
24	double	none	228.68	3.38	1.48	16739.08	252.47	1.51	73.2	0.4	0.55
24	double	SSE3	225.99	3.55	1.57	16044.3	221.76	1.38	71.0	0.63	0.89

[4] A. Benoit, L. Lefèvre, A.-C. Orgerie, and I. Rais, "Reducing the energy consumption of large scale computing systems through combined shutdown policies with multiple constraints," *Int. J. of High Performance Computing Applications*, 2017.

[5] Georgiou and al., "A scheduler-level incentive mechanism for energy efficiency in hpc," in *CCGrid*, 2015, pp. 617–626.

[6] B. Speitkamp and M. Bichler, "A mathematical programming approach for server consolidation problems in virtualized data centers," *IEEE Transactions on services computing*, vol. 3, no. 4, pp. 266–278, 2010.

[7] Y. Li, A.-C. Orgerie, and J.-M. Menaud, "Balancing the use of batteries and opportunistic scheduling policies for maximizing renewable energy consumption in a Cloud data center," in *Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, 2017.

[8] Li and al., "Hybrid MPI/OpenMP power-aware computing," in *IPDPS*, 2010, pp. 1–12.

[9] Acar and al., "Towards a Green and Sustainable Software," in *Concurrent Engineering*, 2015, pp. 471–480.

[10] Cebrian and al., "Improving energy efficiency through parallelization and vectorization on intel core i5 and i7 processors," in *SCC*, 2012.

[11] J. Han and M. Orshansky, "Approximate computing: An emerging paradigm for energy-efficient design," in *Test Symposium (ETS), 2013 18th IEEE European*. IEEE, 2013.

[12] Marathe and al., "A run-time system for power-constrained HPC applications," in *HiPC*, 2015, pp. 394–408.

[13] Berthier and al., "Designing autonomic management systems by using reactive control techniques," *IEEE Transactions on Software Engineering*, vol. 42, no. 7, pp. 640–657, 2016.

[14] J. e. a. Eastep, "Global Extensible Open Power Manager: A Vehicle for HPC Community Collaboration Toward Co-Designed Energy Management Solutions," in *High Performance Computing*, 2017.

[15] Oleynik and al., "Run-time exploitation of application dynamism for energy-efficient exascale computing (readex)," in *Computational Science and Engineering (CSE), 2015 IEEE 18th International Conference on*. IEEE, 2015.

[16] Dagum and al., "OpenMP: an industry standard API for shared-memory programming," *IEEE computational science and engineering*, pp. 46–55, 1998.

[17] Peleg and al., "MMX technology extension to the Intel architecture," *IEEE micro*, vol. 16, no. 4, pp. 42–50, 1996.

[18] Gallas and al., "Embedded Pentium (R) processor system design for Windows CE," in *Wescon/98*. IEEE, 1998, pp. 114–123.

[19] C. Lomont, "Introduction to intel advanced vector extensions," *Intel White Paper*, pp. 1–21, 2011.

[20] F. Rossigneux, L. Lefevre, J.-P. Gelas, and M. Dias de Assuncao, "A Generic and Extensible Framework for Monitoring Energy Consumption of OpenStack Clouds," in *The 4th IEEE International Conference on Sustainable Computing and Communications*, Dec. 2014.

[21] D. Balouek and al., "Adding virtualization capabilities to the Grid'5000 testbed," in *Cloud Computing and Services Science*. Springer, 2013, vol. 367, pp. 3–20.

[22] G. L. T. Chetsa, L. Lefevre, J.-M. Pierson, P. Stolf, and G. Da Costa, "A user friendly phase detection methodology for hpc systems' analysis," in *Green Computing and Communications (GreenCom), 2013 IEEE and Internet of Things (iThings/CPSCom), IEEE International Conference on and IEEE Cyber, Physical and Social Computing*. IEEE, 2013, pp. 118–125.