MERIC and RADAR generator: tools for energy evaluation and runtime tuning of HPC applications

Ondrej Vysocky[a], Martin Beseda[a], Lubomir Riha[a], Jan Zapletal[a], Michael Lysaght[b], and Venkatesh Kannan[b]

[a]IT4Innovations National Supercomputing Center,
VŠB-Technical University of Ostrava, Czech Republic
[b]Irish Centre for High End Computing, Ireland
{ondrej.vysocky,martin.beseda,lubomir.riha,jan.zapletal}@vsb.cz
{michael.lysaght,venkatesh.kannan}@ichec.ie
http://www.it4i.cz
https://www.ichec.ie

Abstract. This paper introduces two tools for manual energy evaluation and runtime tuning developed at IT4Innovations in the READEX project. The MERIC library can be used for manual instrumentation and analysis of any application from the energy and time consumption point of view. Besides tracing, MERIC can also change environment and hardware parameters during the application runtime, which leads to energy savings.
MERIC stores large amounts of data, which are difficult to read by a human. The RADAR generator analyses the MERIC output files to find the best settings of evaluated parameters for each instrumented region. It generates a LaTeX report and a MERIC configuration file for application production runs.

Keywords: READEX, MERIC, RADAR, energy efficient computing, HDEEM, RAPL

1 Introduction

The Horizon 2020 project READEX (Runtime Exploitation of Application Dynamism for Energy-efficient eXascale computing) [18] deals with manual and also automatic tools that analyze High Performance Computing (HPC) applications, and searches for the best combination of tuned parameter settings to use them optimally for application needs. This paper presents tools developed in the READEX project for manual evaluation of the dynamic behavior of the HPC applications - the MERIC and RADAR generator.

The MERIC library evaluates application behavior in terms of resource consumption, and controls hardware and runtime parameters such as the Dynamic Voltage and Frequency Scaling (DVFS), Uncore Frequency Scaling (UFS), and number of OpenMP threads through external libraries. User applications can be

instrumented using the MERIC manual instrumentation to analyze each part of the code separately. The energy measurements are provided by the High Definition Energy Efficiency Monitoring (HDEEM) system [8], or by Running Average Power Limit (RAPL) counters [10].

The MERIC measurement outputs are analyzed using the RADAR generator, which produces detailed reports, and also a MERIC configuration file, which can be used to set the best parameter values for all evaluated regions in the application.

There are several research activities in HPC application energy saving due to applying power capping [11][6] to the whole application run instead of parsing the application into regions and applying dynamic tuning. Other research is dealing with scheduling system using dynamic power capping with negligible time penalty based on previous application runs [16]. Dynamic application tuning is the goal of the READEX project, which should deliver a tool-suite for fully automatic application instrumentation, dynamism detection and analysis. The analysis should find the configuration that provide the maximum energy savings and can be used for the future production runs. The READEX tools are very complex and may not be easy to apply. Our tools present the same approach with focus on usage friendliness, albeit providing manual tuning only. Furthermore, the READEX tools are focused on x86 platforms only, which is not the case for MERIC.

2 Applications Dynamism

The READEX project expects that HPC applications have different needs in separate parts of the code. To find these parts inside a user application, three dynamism metrics are presently measured and used in the READEX project. They include:

1. Execution time
2. Energy consumed
3. Computational intensity

Among these three metrics, the semantics of execution time and energy consumed are straightforward. Variation in the execution time and energy consumed by regions in an application during its execution is an indication of different resource requirements. The computational intensity is a metric that is used to model the behaviour of an application based on the workload imposed by it on the CPU and the memory. Presently, computational intensity is calculated using the following formula 1 and is analogous to the operational intensity used in the roofline model [22].

$$\textit{Computational intensity} = \frac{\textit{Total number of instructions executed}}{\textit{Total number of L3 cache misses}} \tag{1}$$

Selected regions in the user application are called significant. To detect the significant regions manually, profiling tools such as Allinea MAP [1] are used.

The dynamism observed in an application can be due to variation of the following factors:

- Floating point computations (for example, this may occur due to variation in the density of matrices in dense linear algebra).
- Memory read/write access patterns (for example, this may occur due to variation in the sparsity of matrices in sparse linear algebra).
- Inter-process communication patterns (for example, this may occur due to irregularity in a data structure leading to irregular exchange of messages for operations such as global reductions).
- I/O operations performed during the application's execution.
- Different inputs to regions in the application.

To address these factors, a set of tuning parameters has been identified in the READEX project to gain possible savings due to static and dynamic tuning. The list of the parameters contains the following:

- hardware parameters of the CPU
  - Core Frequency (CF)
  - Uncore frequency (UCF) [1]
- system software parameters
  - number of OpenMP threads, thread placement
- application-level parameters
  - depends on the specific application

All parameters can be set before an application is executed (this is called static tuning), in addition some of them can be tuned dynamically during the application runtime. For instance core and uncore frequencies can be switched without additional overhead, but switching the number of threads can affect performance due to NUMA effects and data placement and must be handled carefully. Static and dynamic tuning leads to static and dynamic savings, respectively.

Presently the MERIC tool (Section 3) is being developed and used in the READEX project to measure the above-mentioned dynamism metrics and evaluate applications. When using MERIC it is possible to dynamically switch CPU core and uncore frequencies and the number of used OpenMP threads. The measurements collected by these tools for an application are logged into a READEX Application Dynamism Analysis Report (RADAR) as described in Section 4.

3 Manual Dynamism Evaluation with MERIC

MERIC [2] is a C++ dynamic library (with an interface for Fortran applications) that measures energy consumption and runtime of annotated regions inside a user application. By running the code with different settings of the tuning

[1] Uncore frequency refers to frequency of subsystems in the physical processor package that are shared by multiple processor cores, e.g., L3 cache and on-chip ring interconnect.

[2] MERIC repository: https://code.it4i.cz/vys0053/meric

parameters, we analyze possibilities for energy savings. Subsequently, the optimal configurations are applied by changing the tuning parameters (list of parameters mentioned in the previous Section 2) during the application runtime, which can be also done by using MERIC. MERIC wraps a list of libraries that provide access to different hardware knobs and registers, operating system and runtime system variables, i.e. tuning parameters, in order to read or modify their values. The main motivation for the development of this tool was to simplify the evaluation of various applications dynamic behavior from the energy consumption point of view, which includes a large number of measurements.

The library is easy to use. After inserting the MERIC initialization function, it is possible to instrument the application through the so-called probes, which wrap potentially significant regions of the analysed code. Besides storing the measurement results, the user should not notice any changes in the behavior of the application.

3.1 MERIC features

MERIC has minimal influence on the application's runtime despite providing several analysis and tuning features. Its overhead depends on the energy measurement mode as described in this section, the amount of hardware performance counters read, as well as the number of instrumented regions.

Environment settings
During the MERIC initialization and at each region start and end, the CPU frequency, uncore frequency and number of OpenMP threads are set. To do so, MERIC uses the OpenMP runtime API and the cpufreq [3] and x86_adapt [17] libraries.

Energy measurement
The key MERIC feature is energy measurement using the High Definition Energy Efficiency Monitoring (HDEEM) system located directly on computational nodes that records 100 power samples per second of the CPUs and memories, and 1000 samples of the node itself via the BMC (Baseboard Management Controller) and an FPGA (Field Programmable Gate Array). Figure 1 shows the system diagram and a picture a node with the HDEEM.

HDEEM provides energy consumption measurement in two different ways, and in MERIC it is possible to choose which one the user wants to use by setting the MERIC_CONTINUAL parameter.

In one mode, the energy consumed from the point that HDEEM was initialized is taken from the HDEEM Stats structure (a data structure used by the HDEEM library to provide measurement information to the user application). In this mode we read the structure at each region start and end. This solution is straightforward, however, there is a delay of approximately 4 ms associated with every read from the HDEEM API. To avoid the delay, we take advantage of the fact that during measurement HDEEM stores power samples in its internal memory. In the second mode MERIC only needs to record timestamps at the beginning and the end of each region instead of calling the HDEEM API. This

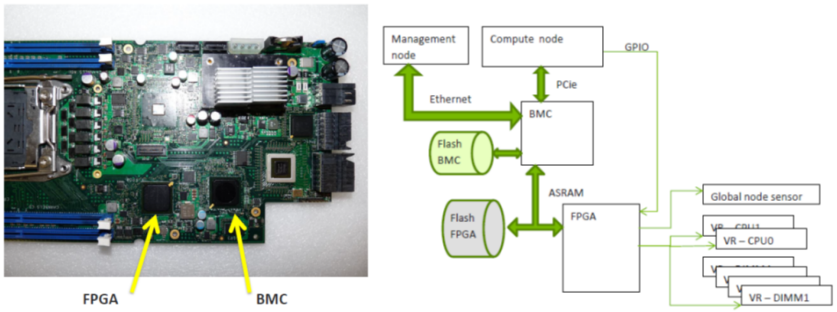

Fig. 1. A HDEEM system located on a node and the system diagram [2].

results in a very small overhead for MERIC instrumentation during the application runtime because all samples are transferred from the HDEEM memory at the end of the application runtime. The energy consumption is subsequently calculated from the power samples based on the recorded timestamps.

Contemporary Intel processors support energy consumption measurements via the Running Average Power Limit (RAPL) interface. MERIC uses the RAPL counters with 1 kHz sampling frequency to allow energy measurements on machines without the HDEEM infrastructure as well as to compare them with the HDEEM measurements.

The main disadvantage of using RAPL is that it measures CPUs and memories power consumption only, without providing information about the power consumption of the blade itself. In the case of nodes with two Intel(R) Xeon(R) CPU E5-E5-2680 v3 (2×12 cores) processors the power baseline is approximately 70 W. To overcome this handicap we statically add this 70 W to our measurements when using RAPL counters. MERIC uses the x86_adapt library to read the RAPL counters.

The minimum runtime of each evaluated region has been set in the READEX project to 100 ms when using HDEEM or RAPL, to have enough samples per region to evaluate the region optimum configuration correctly.

Hardware performance counters

To provide more information about the instrumented regions of the application, we use the perf_event and PAPI libraries, which provide access to hardware performance counters. Values from the counters are transferred into cache-miss rates, FLOPs/s [3] and also the computational intensity that is a key metric for dynamism detection as described in Section 2.

[3] The Intel Haswell processors do not support floating-point instructions counters. MERIC approximates FLOPs/s based on the counter of Advanced Vector Extensions (AVX) calculation operations. For more information visit https://github.com/RRZE-HPC/likwid/wiki/FlopsHaswell.

Shared interface for Score-P
The Score-P software system, as well as the MERIC library, allows users to manually (and also automatically) instrument an application for tracing analysis. Score-P instrumentation is also used in the READEX tool suite [13].

A user that has already instrumented an application using Score-P instrumentation or would want to use it in the future may use the readex.h header file that is provided in the MERIC repository. This allows the user to only insert the user instrumentation once, but for both MERIC and Score-P simultaneously. When a user application is compiled, one has to define the preprocessor variables USE_MERIC, USE_SCOREP (Score-P phase region only) or alternatively USE_SCOREP_MANUAL to select which instrumentation should be used.

Table 1 shows the list of functions defined in the header file, with their MERIC and Score-P equivalents. Brief description of the mentioned MERIC functions is provided in Section 3.2, description of the Score-P functions can be found in its user manual [20].

Shared interface	**MERIC function**	**Score-P function**
READEX_INIT	MERIC_INIT	–
READEX_CLOSE	MERIC_CLOSE	–
READEX_REGION_DEFINE	–	SCOREP_USER_REGION_DEFINE
READEX_REGION_START	MERIC_MeasureStart	SCOREP_USER_REGION_BEGIN
READEX_REGION_STOP	MERIC_MeasureStop	SCOREP_USER_REGION_END
READEX_PHASE_START	MERIC_MeasureStart	SCOREP_USER_OA_PHASE_BEGIN
READEX_PHASE_STOP	MERIC_MeasureStop	SCOREP_USER_OA_PHASE_END

Table 1. Function names defined in the readex.h header file, that can be used for MERIC and Score-P instrumentation.

MERIC requirements
MERIC currently adds synchronization MPI and OpenMP barriers into the application code to ensure that all processes/threads under one node are synchronized in a single region when measuring consumed resources or changing hardware or runtime parameters. We realize that this approach inserts extra overhead into application runtime and may discriminate a group of asynchronous applications. In future the library will allow the user to turn these barriers off.

Beyond the inserted synchronization the MERIC library requires several libraries to provide all previously mentioned features:

- Machine with HDEEM or x86_adapt library for accessing RAPL counters
- Cpufreq or x86_adapt library to change CPU frequencies
- PAPI and perf_event for accessing hardware counters

ARM Jetson TX1
The MERIC library was originally developed to support resource consumption measurement and DVFS on Intel Haswell processors [9], however it has been

extended to also provide support for the Jestson/TX1 ARM system [12] located at the Barcelona Supercomputing Center [14] (ARM Cortex-A57, 4 cores, 1.3 GHz) which supports energy measurements.

ARM systems are an interesting platform because they allow the setting of much lower frequencies [7] and save energy accordingly. In the case that system CPU uncore frequency is not possible to set, however, one can change the frequency of the RAM. Minimum CPU core frequency is 0.5 GHz and the maximum is 1.3 GHz. The minimum and maximum RAM frequency is 40 MHz and 1.6 GHz, respectively. To change frequencies on Jetson, no third-party libraries are necessary.

To gather power data, the Texas Instrument INA3221 chip is featured on the board [4]. It measures the per-node energy consumption and stores samples values in a file. It is possible to gather hundreds of samples per second, however the measurement effects the CPU. The following Table 2 shows the impact of sampling frequency on the CPU workload evaluated using htop [4].

Sampling Frequency [Hz]	**CPU workload**
10	2 %
50	4 %
100	8 %
200	14 %
500	23 %
1000	30 %

Table 2. The Jetson/TX1 energy measurement interface and its effect on the CPU workload when reading 10 up to 1000 power samples per second. The load was evaluated using htop when running the power sampling only.

3.2 Workflow

First, the user has to analyze their application using a profiler tool (such as Allinea MAP) and find the significant regions in order to cover the most consuming functions in terms of time, MPI communication, and I/O, and insert MERIC instrumentation into code to wrap the selected sections of the code. A region start function takes a parameter with the name of the region, but the stop function does not have any input parameters, because it ends the region that has been started most recently (last in, first out).

The instrumented application should be run as usual. To control MERIC behaviour it is possible to export appropriate environment variables or define a MERIC configuration file that allows the user to specify the settings not only for the whole application run (as in the case of environment variables), but also

[4] htop repository: https://github.com/hishamhm/htop

control the behavior for separate regions, computation nodes, or their sockets. The user can define hardware and runtime settings (CPU frequencies and number of threads) as well as select energy measurement mode, hardware counters to read and more.

4 RADAR: Measurement data analysis

RADAR presents a brief summary of the measurement results obtained with MERIC. This is a merged form of automatically generated dynamism report by both the RADAR generator (by IT4Innovations), described in detail in Section 4.1 and the `readex-dyn-detect` (by the Technical University of Munich), described in [19]. The report depicts diagrams of energy consumption with respect to a set of tuning parameters. It also contains different sets of graphical comparisons of static and dynamic significant energy savings across phases for different hardware tuning parameter configurations. In each perspective, the measured dynamism metrics are presented for the default configurations that are used for the tuning parameters.

4.1 The RADAR generator

The RADAR generator [5] allows users to evaluate the data measured by the MERIC tool automatically, and to get an uncluttered summary of the results in the form of a LaTeX file. Moreover, it is possible to include the report generated by the `readex-dyn-detect` tool, as mentioned above.

Uncore freq [GHz (uncore)] / Frequency [GHz (core)]	1.2	1.4	1.6	1.8	2.0	2.2	2.4	2.6	2.8	3.0
1.2	12.256	11.071	10.633	10.084	9.407	8.937	9.284	8.581	8.513	8.296
1.4	11.829	10.57	10.152	9.178	8.682	8.684	8.094	8.192	7.966	7.666
1.6	11.723	10.178	9.438	8.706	8.373	8.008	7.821	7.471	7.552	7.212
1.8	10.996	9.969	8.952	8.57	7.929	7.779	7.477	7.138	7.085	6.93
2	10.607	9.516	8.925	8.203	7.79	7.356	7.096	6.908	6.802	6.744
2.2	10.23	9.734	9.02	7.977	7.5	7.23	7.129	6.778	6.827	6.361
2.4	10.775	9.438	8.416	7.919	7.367	7.208	6.772	6.577	6.436	6.356
2.5	10.798	9.086	8.366	7.856	7.555	7.072	6.66	6.605	6.257	6.107

Table 3. Heat map generated by the RADAR generator comparing impact of using different CPU core and uncore frequencies at application runtime in seconds.

The report itself contains information about both static and dynamic savings, represented not only by tables, but also plots and heat-maps. Examples can be seen in Figure 3 and Table 3.

The generator is able to evaluate all chosen quantities at once, i.e. users do not have to generate reports for energy consumption, and compute intensity and

[5] RADAR generator repository: https://code.it4i.cz/bes0030/readex-radar

execution time separately, because they can be contained in one report together. This provides the advantage of direct visual comparison of all optimal settings, so users can achieve a greater understanding of the application behavior quickly. The execution time change for energy-optimal settings is also included in the report, as can be seen in Table 4.

Overall application evaluation

	Default settings	**Default values**	**Best static config.**	**Static Savings**	**Dynamic Savings**
Energy consumption [J] (Samples), Blade summary	24 threads, 3.0 GHz UCF, 2.5 GHz CF	2473.63 J	12 threads, 3.0 GHz UCF, 2.5 GHz CF	371.80 J (15.03%)	4.87 J of 2101.83 J (0.23 %)
Runtime of function [s], Job info - hdeem	24 threads, 3.0 GHz UCF, 2.5 GHz CF	6.37 s	18 threads, 3.0 GHz UCF, 2.5 GHz CF	0.26 s (4.10%)	0.0073 s of 6.11 s (0.12 %)
Run-time change with the energy optimal settings: -0.01s (98.19% of default time)					

Table 4. Summary table generated by the RADAR generator presenting possible energy or runtime saving that can be reached if the best static and also best dynamic settings for each region would be set.

This evaluation is performed not only for the main region (usually the whole application), but for its nested regions too. Users can also specify an iterative region which contains all the nested ones and which is called directly in the main region. In this way certain iterative schemes (e.g., iterative solvers of linear systems) are understood in detail, because every iteration (or phase) is evaluated separately.

With this feature users have information about the best static optima just for the main region (which serves as the best starting settings), information about optimal settings of nested regions in an average phase, and the above-mentioned information about optimal settings of nested regions in every individual phase. If we wanted to process multiple regions like one, we can group them under one *role*, as can be seen in Figure 2, where *Projector_l* and *Projector_l_2* are different regions comprising the region *Projector*. If multiple runs of the program are measured, then both the average run and separate runs are evaluated.

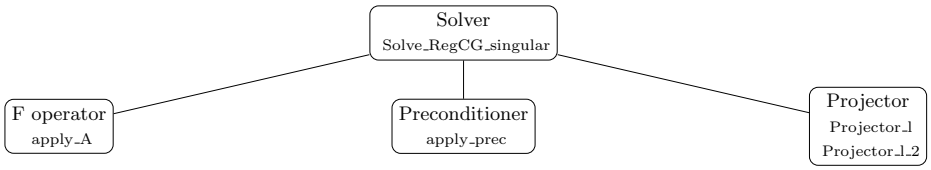

Fig. 2. Example of multiple regions on one role

For some programs such a report could be impractically long and so the generator offers the possibility to create a shorter version containing only the overall summary and the average phase evaluation.

The generator also supports evaluation in multiples of the original unit used in the measurement. Both the static and dynamic baseline for the energy consumption, i.e. the constant baseline and the baseline dependent on settings, are supported too.

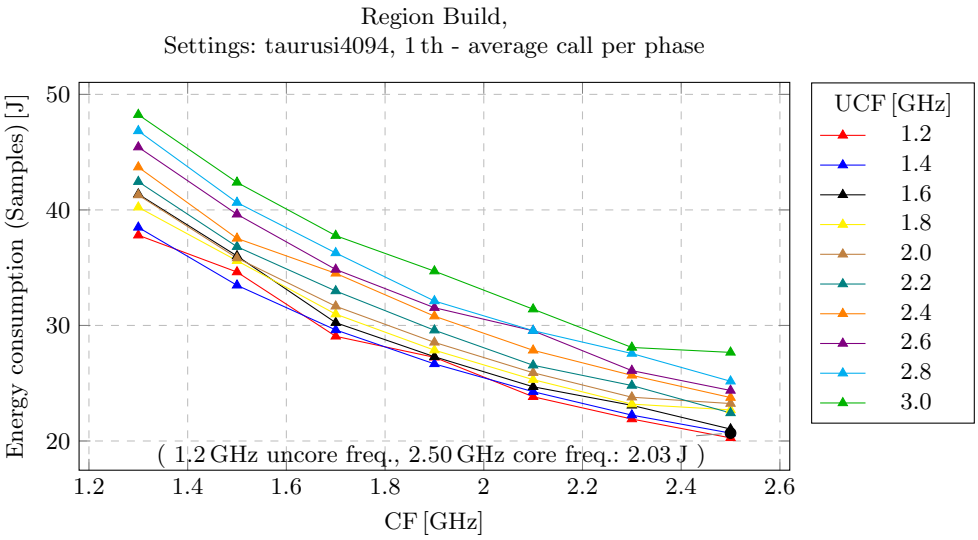

Fig. 3. Plot example generated by the RADAR generator showing the effect of using different CPU core and uncore frequencies from the energy consumption point of view.

Finally, the optimal settings for all regions and every measured quantity can be exported into the separated files, which can be used as an input for the MERIC tool, as described in Section 3.2.

All the above-mentioned settings are listed in the external configuration file, which is set by the generator's flag, so users can easily change several different settings for their reports.

5 Test case

The ESPRESO library [6] was selected to present MERIC and RADAR generator usage. The library is a combination of Finite Element (FEM) and Boundary Element (BEM) tools and TFETI/HTFETI [5][15] domain decomposition solvers.

[6] ESPRESO library website: http://espreso.it4i.cz/

The ESPRESO solver is a parallel linear solver, which includes a highly efficient MPI communication layer designed for massively parallel machines with thousands of compute nodes. The parallelization inside a node is done using OpenMP.

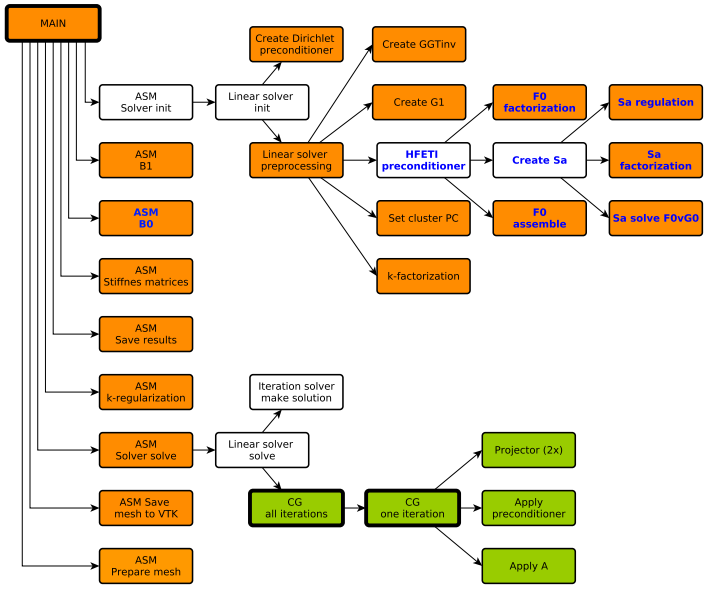

Fig. 4. Graph of significant regions in the ESPRESO library. The green boxes depict multiply called regions in an iterative solver, the orange ones are only called once during the application runtime.

Overall application evaluation

	Default settings	**Default values**	**Best static configuration**	**Static Savings**	**Dynamic Savings**
Energy [J] RAPL counters	12 threads, 3.0 GHz UCF, 2.5 GHz CF	4549.13 J	12 threads, 2.2 GHz UCF, 2.4 GHz CF	181.76 J (4.00%)	325.89 J of 4367.37 J (7.46 %)
Runtime [s]	12 threads, 3.0 GHz UCF, 2.5 GHz CF	15.90 s	12 threads, 3.0 GHz UCF, 2.5 GHz CF	0.00 s (0.00%)	0.39 s of 15.90 s (2.43 %)

Table 5. Table of resultant static and dynamic savings of the ESPRESO library test. Rows respectively focus on possible savings from the energy and runtime points of view.

The following test was performed on the IT4Innovations Salomon cluster powered by two Intel Xeon E5-2680v3 (Haswell-EP) processors per node using a RAPL counter with a 70 W baseline for the energy consumption measurement. The processor is equipped with 12 cores and allows for CPU core and uncore frequency scaling within the range of 1.2–2.5 GHz and 1.2–3.0 GHz, respectively. We evaluated ESPRESO on a heat transfer problem with 2.7 million unknowns using one MPI process per socket.

Table 5 shows the possible savings made by using different numbers of OpenMP threads during the runtime, and by switching CPU core and uncore frequencies. This table shows that it is possible to save 4 % of the overall energy just by statically setting different CPU core and uncore frequencies that can be applied even without instrumenting the application at all. Table 6 shows the impact of using different CPU frequencies in this test case, from the energy consumption point of view.

Another 7.46 % of energy can be saved through dynamic switching of the tuned parameters to apply the best configuration for each significant region. Overall energy savings in this test case were 11.16 %. Table 7 in the appendix of this paper contains the regions' best settings.

UnCF [GHz] / CF [GHz]	1.2	1.4	1.6	1.8	2.0	2.2	2.4	2.6	2.8	3.0
1.2	5,971	5,825	5,764	5,725	5,698	5,783	5,859	5,962	6,127	6,232
1.4	5,519	5,350	5,238	5,220	5,208	5,219	5,357	5,432	5,513	5,639
1.6	5,226	5,029	4,902	4,840	4,829	4,819	4,890	4,986	5,093	5,185
1.8	5,080	4,897	4,739	4,711	4,649	4,656	4,720	4,760	4,859	4,956
2	5,054	4,852	4,707	4,587	4,565	4,528	4,585	4,636	4,736	4,817
2.2	4,985	4,774	4,605	4,520	4,464	4,442	4,469	4,540	4,632	4,653
2.4	4,984	4,783	4,593	4,442	4,391	4,367	4,408	4,438	4,503	4,578
2.5	5,211	4,858	4,675	4,547	4,479	4,422	4,445	4,439	4,482	4,549

Table 6. An ESPRESO library energy consumption heat-map showing the impact of different CPU core and uncore frequencies when using 12 OpenMP threads.

6 Conclusion

The paper presented two tools that allow easy analysis of HPC applications' behavior, with the goal to tune hardware and runtime parameters to minimize the given objective (e.g., the energy consumption and runtime).

Resource consumption measurement and dynamic parameter changes are provided by the MERIC library. The currently supported parameters that can be switched dynamically include the CPU core and uncore frequencies, as well as the number of active OpenMP threads.

The RADAR generator analyses the MERIC measurement outputs and provides detailed LaTeX reports describing the behavior of the instrumented regions.

These reports also contain information about the settings that should be applied for each region to reach maximum savings. The RADAR generator produces the MERIC configuration files that should be used for production runs of the user application to apply the best settings dynamically during the runtime.

Possible savings that can be reached when using MERIC and the RADAR generator are presented in [21], where we show that the energy savings can reach up to 10-30 %.

Acknowledgement

This work was supported by The Ministry of Education, Youth and Sports from the National Programme of Sustainability (NPU II) project "IT4Innovations excellence in science - LQ1602" and by the IT4Innovations infrastructure which is supported from the Large Infrastructures for Research, Experimental Development and Innovations project "IT4Innovations National Supercomputing Center LM2015070".

The research leading to these results has received funding from the European Union's Horizon 2020 Programme under grant agreement number 671657.

The work was additionally supported by VŠB – Technical University of Ostrava under the grant SP2017/165 and by the Barcelona Supercomputing Center under the grants 288777, 610402 and 671697.

References

1. Allinea MAP - C/C++ profiler and Fortran profiler for high performance Linux code, `https://www.allinea.com/products/map`
2. High definition energy efficiency monitoring, `http://www.ena-hpc.org/2014/pdf/bull.pdf`
3. Brodowski, D.: Linux CPUFreq, `https://www.kernel.org/doc/Documentation/cpu-freq/index.txt`
4. BSC: Power monitoring on mini-clusters, `https://wiki.hca.bsc.es/dokuwiki/wiki:prototype:power_monitor#jetson-tx1`
5. Dostal, Z., Horak, D., Kucera, R.: Total FETI-an easier implementable variant of the FETI method for numerical solution of elliptic PDE. Communications in Numerical Methods in Engineering 22(12), 1155–1162 (jun 2006), `http://dx.doi.org/10.1002/cnm.881`
6. Eastep, J., Sylvester, S., Cantalupo, C., Geltz, B., Ardanaz, F., Al-Rawi, A., Livingston, K., Keceli, F., Maiterth, M., Jana, S.: Global extensible open power manager: A vehicle for hpc community collaboration on co-designed energy management solutions. In: ISC (2017)
7. eLinux.org: Jetson/TX1 controlling performance, `http://elinux.org/Jetson/TX1_Controlling_Performance`
8. Hackenberg, D., Ilsche, T., Schuchart, J., Schöne, R., Nagel, W., Simon, M., Georgiou, Y.: Hdeem: High definition energy efficiency monitoring. In: Energy Efficient Supercomputing Workshop (E2SC) (Nov 2014)

9. Hackenberg, D., Schöne, R., Ilsche, T., Molka, D., Schuchart, J., Geyer, R.: An energy efficiency feature survey of the Intel Haswell processor. In: Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International (May 2015)
10. Hähnel, M., Döbel, B., Völp, M., Härtig, H.: Measuring energy consumption for short code paths using rapl. SIGMETRICS Perform. Eval. Rev. 40(3), 13–17 (Jan 2012), http://doi.acm.org/10.1145/2425248.2425252
11. Haidar, A., Jagode, H., Vaccaro, P., YarKhan, A., Tomov, S., Dongarra, J.: Investigating power capping toward energyefficient scientific applications. Concurrency and Computation: Practice and Experience 0(0), e4485, https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4485
12. NVIDIA: NVIDIA Jetson, http://www.nvidia.com/object/embedded-systems-dev-kits-modules.html
13. Oleynik, Y., Gerndt, M., Schuchart, J., Kjeldsberg, P.G., Nagel, W.E.: Runtime exploitation of application dynamism for energy-efficient exascale computing (READEX). In: Plessl, C., El Baz, D., Cong, G., Cardoso, J.M.P., Veiga, L., Rauber, T. (eds.) Computational Science and Engineering (CSE), 2015 IEEE 18th International Conference on. pp. 347–350. IEEE, Piscataway (Oct 2015)
14. Rajovic, N., Rico, A., Mantovani, F., Ruiz, D., Vilarrubi, J.O., Gomez, C., Backes, L., Nieto, D., Servat, H., Martorell, X., Labarta, J., Ayguade, E., Adeniyi-Jones, C., Derradji, S., Gloaguen, H., Lanucara, P., Sanna, N., Mehaut, J.F., Pouget, K., Videau, B., Boyer, E., Allalen, M., Auweter, A., Brayford, D., Tafani, D., Weinberg, V., Brömmel, D., Halver, R., Meinke, J.H., Beivide, R., Benito, M., Vallejo, E., Valero, M., Ramirez, A.: The Mont-blanc prototype: An alternative approach for HPC systems. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 38:1–38:12. SC '16, IEEE Press, Piscataway, NJ, USA (2016), http://dl.acm.org/citation.cfm?id=3014904.3014955
15. Riha, L., Brzobohaty, T., Markopoulos, A., Jarosova, M., Kozubek, T., Horak, D., Hapla, V.: Implementation of the efficient communication layer for the highly parallel total feti and hybrid total feti solvers. Parallel Computing (2016)
16. Rountree, B., Lowenthal, D.K., de Supinski, B.R., Schulz, M., Freeh, V.W., Bletsch, T.K.: Adagio: making dvs practical for complex hpc applications. In: ICS (2009)
17. Schoene, R.: x86_adapt, https://doc.zih.tu-dresden.de/hpc-wiki/bin/view/Compendium/X86Adapt
18. Schuchart, J., Gerndt, M., Kjeldsberg, P.G., Lysaght, M., Horák, D., Říha, L., Gocht, A., Sourouri, M., Kumaraswamy, M., Chowdhury, A., Jahre, M., Diethelm, K., Bouizi, O., Mian, U.S., Kružík, J., Sojka, R., Beseda, M., Kannan, V., Bendifallah, Z., Hackenberg, D., Nagel, W.E.: The READEX formalism for automatic tuning for energy efficiency. Computing pp. 1–19 (2017), http://dx.doi.org/10.1007/s00607-016-0532-7
19. Venkatesh, K., Lubomir, R., Michael, G., Anamika, C., Ondrej, V., Martin, B., David, H., Radim, S., Jakub, K., Michael, L.: Prace whitepaper: Investigating and exploiting application dynamism for energy-efficient exascale computing (2017), www.prace-ri.eu
20. VI-HPS: Score-p user manual 3.1 (2017)
21. Vysocky, O., Beseda, M., Riha, L., Zapletal, J., Nikl, V., Lysaght, M., Kannan, V.: Evaluation of the hpc applications dynamic behavior in terms of energy consumption. In: Proceedings of the Fifth International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering. Civil-Comp Press, Stirlingshire, UK, Paper 3 (2017)

22. Williams, S., Waterman, A., Patterson, D.: Roofline: An insightful visual performance model for multicore architectures. Commun. ACM 52(4), 65–76 (Apr 2009), http://doi.acm.org/10.1145/1498765.1498785

Appendix

Significant regions energy summary and its best dynamic configuration

Region	% of 1 phase	Best dynamic configuration	Dynamic savings
Assemble Stiffness Matrices	16.77	12 threads, 2.0 GHz UCF, 2.4 GHz CF	6.51 J from 685.54 J (0.95%)
Assembler–AssembleB0	0.2	12 threads, 2.0 GHz UCF, 2.5 GHz CF	0.10 J from 8.07 J (1.24%)
Assembler–AssembleB1	4.12	12 threads, 2.0 GHz UCF, 2.5 GHz CF	6.61 J from 168.36 J (3.93%)
Assembler-K_Regularization	5.00	2 threads, 2.2 GHz UCF, 2.5 GHz CF	47.64 J from 204.24 J (23.32%)
Assembler–PrepareMesh	12.66	2 threads, 1.8 GHz UCF, 2.5 GHz CF	77.70 J from 517.68 J (15.01%)
Assembler–SaveMeshtoVTK	6.89	2 threads, 1.2 GHz UCF, 2.5 GHz CF	39.80 J from 281.63 J (14.13%)
Assembler–SaveResults	3.38	2 threads, 1.2 GHz UCF, 2.5 GHz CF	24.67 J from 138.34 J (17.83%)
Assembler–SolverSolve	27.92	12 threads, 2.2 GHz UCF, 1.6 GHz CF	114.50 J from 1141.58 J (10.03%)
Cluster–CreateF0-AssembleF0	5.67	12 threads, 2.2 GHz UCF, 2.4 GHz CF	0.00 J from 231.68 J (0.00%)
Cluster–CreateG1-perCluster	0.43	12 threads, 2.2 GHz UCF, 2.0 GHz CF	0.64 J from 17.47 J (3.69%)
Create_GGT_Inv	0.21	2 threads, 2.2 GHz UCF, 2.5 GHz CF	2.01 J from 8.56 J (23.46%)

Cluster–CreateF0-FactF0	0.08	12 threads, 2.8 GHz UCF, 2.5 GHz CF	0.21 J from 3.26 J (6.36%)
Cluster–Kfactorization	14.47	12 threads, 2.2 GHz UCF, 2.4 GHz CF	0.00 J from 591.46 J (0.00%)
Cluster–CreateSa-SaFactorization	0.51	6 threads, 2.8 GHz UCF, 2.5 GHz CF	2.31 J from 20.70 J (11.30%)
Cluster–CreateSa-SolveF0vG0	0.86	6 threads, 2.8 GHz UCF, 2.5 GHz CF	3.02 J from 35.20 J (8.58%)
Cluster–SetClusterPC	0.85	12 threads, 2.4 GHz UCF, 2.4 GHz CF	0.18 J from 34.95 J (0.52%)

Table 7: Table of the regions analysis from the energy point of view for the test case presented in the Section 5. For every region, this table contains the percentage of energy the region consumed compared to the entire application, and each regions' best configuration and energy savings if the configuration were applied during the application runtime in its the best static configuration.