Automatic Energy-Efficiency Monitoring of OpenMP Workloads

Maxime Mirka, Guillaume Devic, Florent Bruguier, Gilles Sassatelli, and Abdoulaye Gamatié
LIRMM (CNRS and University of Montpellier)
Montpellier, France
name.surname@lirmm.fr

***Abstract*—Energy-efficiency has been a major challenge in compute systems over the last decade. Both embedded and high-performance computing domains are concerned. Many efforts have been currently spent to devise solutions that are capable of providing systems with the best compromises in terms of performance and power consumption. In this paper, we propose an approach for on-line energy-efficiency analysis when executing OpenMP workloads on multicore systems. The novelty of our approach lies in the ability to monitor energy efficiency at run-time without prior knowledge of the application profile or code annotation. The solution relies on two new metrics: the Chunks per Second (CpS) and Chunks per Joule (CpJ). The former captures the quantity of work achieved by threads per unit time (i.e. a performance indicator). The latter indicates the quantity of work achieved by threads per unit energy, also corresponding to the performance per watt (i.e. an energy efficiency indicator). As most programs are made of several phases performing different computations for which CpS and CpJ cannot be related, it is crucial to be capable of detecting phase changes such as to perform intra-phase energy efficiency optimizations. For that purpose we devise a specific neural network model derived from the popular auto-encoder largely explored in the machine learning community, that is capable of understanding application profile and track phase changes at run-time. We show that these new metrics allow to perform energy efficiency optimization, and illustrate our approach on the analysis of the SRAD application from the Rodinia benchmark. The energy-efficiency profile analysis of the application is conducted on both an Intel and ARM platforms, showing its flexibility.**

***Index Terms*—Energy-efficiency, real-time metric monitoring, multicore systems, program execution phase detection, OpenMP**

I. Introduction

Energy-efficiency has been a major challenge in compute systems over the last decade in both embedded and high-performance computing domains. By energy-efficiency, we mean the best compromise between execution performance and power consumption. In high-performance computing domain (HPC) [2], it is often referred to by using the floating-point operation per second per watt (FLOPS/W) metric. In embedded computing domain, the usual metrics are rather millions of instructions per second per watt (or MIPS/W in short). In both units the first components refer to the performance (i.e., FLOPS and MIPS) while the second reflects the power consumption. Optimizing the energy-efficiency can be achieved in different manners: either increasing the performance level while preserving the power consumption, or reducing the power consumption while preserving the performance level, or ideally increasing the performance level while decreasing in the same time the power consumption. Existing approaches that aim at optimizing energy-efficiency rely on various design techniques as already surveyed in literature [14], [17]. These techniques include dynamic voltage and frequency scaling, power mode management and microarchitectural techniques focusing on both memory system and cores.

While the above techniques have been proven useful, the ability to exploit them in an adaptive way is central for optimized energy-efficiency. In particular, it is important to dynamically deal with application-specific energy consumption profile so as to take adequate decisions as early as possible. For this purpose, one usually relies on typical performance of power measurements that are informative-enough. Existing profiling tools such as PAPI [4] or scorep [12] can be used to gather the performance and power consumption numbers of a system. Such information are often obtained *a posteriori* i.e. after executing a large portion (if not the whole) of a given program, therefore, leaving only a little room for early optimizations. Analytic approaches, based on high-level estimation models for performance and power consumption, could be used for an earlier dynamic optimization. Unfortunately, such models are not necessarily reliable approximations for any execution platforms.

Contribution of this work. In this paper, we investigate a framework that enables real-time energy-efficiency measurement, correlated with system execution phases when they exist. Contrary to conventional approaches that either perform indirect optimization (i.e. on metrics that do not relate application performance directly) or require prior profiling (deriving energy efficiency from execution time), this approach taps into the OpenMP runtime and tracks application progress thereby enabling to derive energy-efficiency values. Our approach therefore applies to application programs written in the OpenMP programming model [8] which is by far the most popular shared-memory parallel API. It has been implemented by modifying the *libgomp* runtime library, that is, the GNU OpenMP implementation, though similar modifications can be devised for any OpenMP implementation. It relies on two new metrics: *Chunks per Second* (CpS) and *Chunks per Joule* (CpJ).

We show that these new metrics are suitable information allowing one to easily distinguish different programs' execution phases. This opens opportunities for phase-dependent optimizations. A deep learning autoencoder model [11] is used to enable a systematic on-line detection of programs' execution phases. We illustrate our approach on the analysis of the SRAD application available from the Rodinia benchmark-suite [7]. The energy-efficiency profile analysis of the application is conducted on both an Intel and ARM platforms, showing its flexibility: an Intel Xeon server with two Xeon E5-2630 v4 microprocessors (i.e. bi-socket system with 20 cores (10 cores per socket)), and an Odroid platform with an heterogeneous big.LITTLE architecture (4 "big" and 4 "LITTLE" ARM Cortex-A cores).

Outline. The remainder of this paper is organized as follows: Section II discusses a few related work; then Section III introduces the basic idea behind the new energy-efficiency characterization proposed in the current work; Section IV illustrates the exploitation of the metrics presented in the previous section to analyze the energy-efficiency profile of an OpenMP program running on multicore systems; finally, Section V gives some concluding remarks and draws the perspective of this paper.

II. Related work

As mentioned in the previous section, energy-efficiency has been characterized in literature via different metrics (e.g., FLOPS/W and MIPS/W) according to computing domains. In all cases, these metrics follow the general definition:

$$\text{Energy Efficiency} = \frac{\text{Quantity of Work per Unit Time}}{\text{Power Consumption}}$$

The characterization proposed in the present paper relies on the same definition. It mainly differs in the way the quantity of work is considered: we use an OpenMP-related concept, known as *chunks*. OpenMP divides loop iterations into chunks that are distributed to threads for execution.

Several metric measurement techniques have been proposed over the last decades for characterizing energy-efficiency [1], [5], [16]. They can be distinguished according to the level of the compute system where they operate.

Authors of these papers reported a number of energy consumption data acquisition techniques through hardware sensors, found in different physical locations within systems. These techniques have their pros and cons in terms precision of measurement, temporal resolution, cost of deployment and intrusiveness. They include measurement circuitry integrated in hardware components such as a GPU and CPU; instrumentation devices inserted within compute nodes that are capable of probing at either component or power lane level; and power meters that can gather the total load outside the nodes power supply.

Beyond the above hardware-level techniques, software-based approaches are also used. The Performance API (PAPI) [4] is a well-known library interface (i.e. a software layer) for hardware performance counters that facilitates the extraction of various execution statistics. On the other hand, further profiling tools include the Tuning and Analysis Utilities (TAU) [18], Score-P [12], Scalasca [10] and PowMon [19].

Software-level power profiling tools comprise pTop [9], PowerAPI [13] and Jalen [15]. The former is similar to the top program of GNU/Linux and provides energy consumption data from running processes. The PowerAPI application programming interface and Jalen software-level profiling architecture are enable energy monitoring in real-time. However, they do not provide any energy-efficiency measurement.

The approach proposed in this paper operates at software level, via the OpenMP *libgomp* runtime. Energy-efficiency data are collected in real-time based on the chunk distribution between parallel OpenMP threads. Beyond the collection of the data, an automatic analysis is applied to detect potential system execution phases, characterized by specific energy-efficiency levels. Such a detection allows one to identify typical regions of interest for possible optimizations, e.g., by selecting some architecture configurations. Phase detection is automatically achieved here via a deep learning method relying on an autoencoder.

III. Energy-efficiency characterization

OpenMP is the most popular parallel programming model for shared-memory systems. Its API supports C/C++ and Fortran programming languages among others on almost all architectures and operating systems, which makes it widely used.

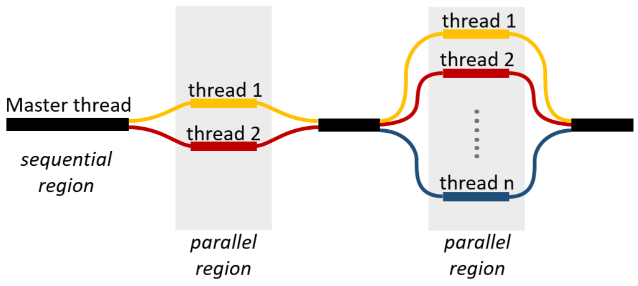


Fig. 1. OpenMP Fork and join mechanism.

OpenMP programs manage parallelism mostly by using threads. OpenMP tasking (i.e. using communicating tasks) has been introduced in OpenMP 3.1 and further developed in OpenMP 4.0 and beyond but remains marginally used as of this date, thereby left aside in this work. Threads are the smallest unit of processing that can be scheduled by an OS. A set of tools is provided by OpenMP to control parallelization and synchronizations. It is all based on two fundamental concepts: Fork and Join. As illustrated in Figure 1, the fork mechanism is the transition from a sequential region (master thread) to a parallel region where the program instructions are executed in parallel among the available team threads. Then, the opposite transition, from a parallel to a sequential region, corresponds to the Join mechanism. This is where the team threads reach a barrier and synchronize, yielding execution to the master thread.

Its features include parallel loops, in which jobs are partitioned in blocks of instructions, also called chunks. Chunks are assigned to the parallel threads previously defined during execution. The scheduling strategy can be either static (i.e. each thread is assigned same work) or dynamic, in which case chunks are assigned to threads dynamically depending on their progress. This latter strategy, though incurring a scheduling overhead, proves efficient and is further desired in systems that are asymmetric such as big.Little architectures.

A. *Light-weight energy-efficiency metrics*

In the following, we rely on this specific feature of OpenMP to define new light-weight metrics for characterizing both the performance and energy-efficiency of an application executing on multicore architectures.

Definition 1 (Chunks per Second - CpS)**.** *The number of chunks executed in one second, where a chunk is a block of instruction assigned to a thread for execution. It defines a speed of work, i.e., it is a performance metric.*

Definition 2 (Chunks per Joule - CpJ)**.** *The number of chunks executed per Joule, where the Joules designate the quantity of energy used by the compute system. It is also defined as CpS per Watt or CpS/W, the number of chunks per second executed per Watt. It defines a quantity of work per quantity of energy, i.e., it is an energy efficiency metric.*

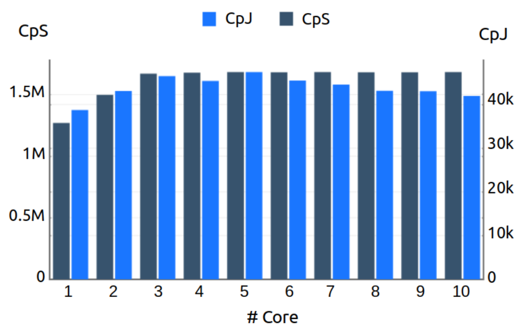
Example 1. *Let us consider the simple OpenMP code depicted in Figure 2. It consists in changing the value of the ith element in the the array B, by the addition of the ith element of both A and B arrays. It is a simple code, going through a memory space, and doing simple computation on each memory cell.*

```
00    #include <omp.h>
01  // Initialization
02    n = 1e9; nthreads = 10;
03    double *A, *B;
04    posix_memalign((void**)&A, 64,
                        n*sizeof(double));
05    posix_memalign((void**)&B, 64,
                        n*sizeof(double));
06    for (i = 0; i < n; ++i) {
07        A[i] = 0.1;
08        B[i] = 0.0;
09    }
10  // Parallelization
11    omp_set_num_threads(nthreads);
12    #pragma omp parallel for
            schedule(dynamic, 1) // directive OpenMP
13    for (i=0; i<n; i++){
14       B[i] = A[i] + B[i];
15    }
16    return 0;
```

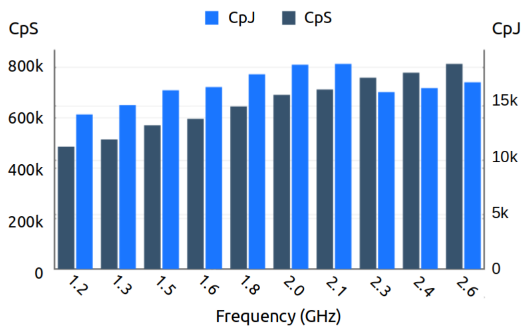
Fig. 2. Simple OpenMP C code example.

The calculation is done in a for loop which corresponds to the workload to be parallelized. To understand how the parallelization is made possible, let us refer to Figure 1. The OpenMP directive line 12 initiate a parallel region, where the chunks are dynamically distributed to the team threads one by one (option "schedule(dynamic, 1)"). The team threads are set up with the command line 11, here 10 threads are created.

When executing this code on various architecture configurations in terms of core count or operating frequencies, we obtain variable performance and energy-efficiency outcomes, through the obtained CpS and CpJ metrics, as shown in Figure 3. In Figure 3a, performance (i.e. CpS) increases up to 3 cores and reaches a plateau due to an obvious saturation of the memory subsystem given the rather memory intensive nature of the application. Having more than 3 cores further results in a decrease in energy efficiency as most cores are idling waiting for data to process whilst consuming power. When varying the frequency on a single core a monotonic increase in performance is observed in Figure 3b. Energy efficiency, however, drops past a certain frequency, possibly to rising contentions to the main memory.



(a) Configurations with different core counts. Frequency = 1.2GHz.



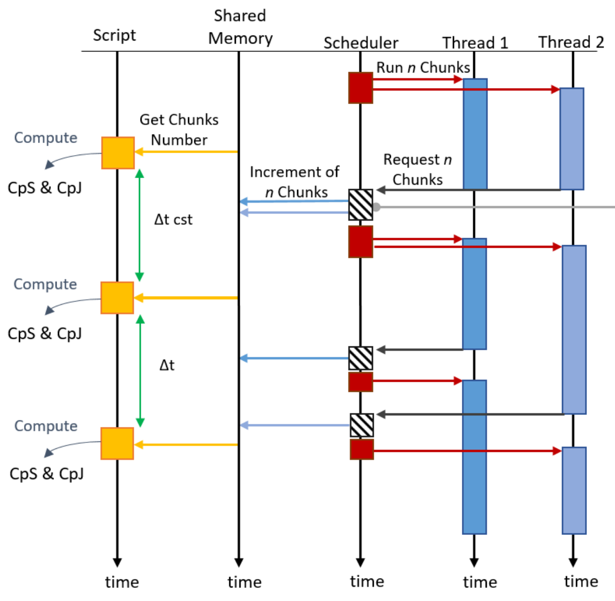
(b) Configurations with different core frequencies. #core = 1.

Fig. 3. CpS and CpJ for different system configurations.

Note: The chunk metric is defined as a relative metric. According to the OpenMP terminology, one chunk corresponds to one iteration of a parallel loop. Hence the size of a chunk varies according to the iteration's workload. We will cover this aspect in Section IV with the CpS and CpJ analysis.

B. *Metrics extraction*

The GNU OpenMP API implementation is embedded in the *libgomp* library which is part of GCC. In particular, the scheduling mechanisms responsible to dispatch chunks to threads are described in this library which is linked against the runtime after compilation. Chunk tracing is made at this



```
Update the counter's value, saved in a shared memory

Initialization:
Create a shared memory with the identification number "shmid".
Initialize nb_chunk_done[n]: table of long with n cells, where n
is the number of thread.
Initialize nb_chunk_updated: total number of chunks done

Get number of chunk done from the current thread
int tid = omp_get_thread_num();// OpenMP directive
nb_chunk_done[tid] = Number of chunk done by the tidth thread.

Get time of the capture:
act_time = get_time();

Update the chunk-counter value:
for i in NumberOfThread do
 nb_chunk_updated += nb_chunk_done[i];

Prepare data to write in the Shared Memory:
StringToWrite = "nb_chunk_updated" & "act_time";

Attached shared memory and write data (ipc/shm
directives):
shared_memory = shmat(shmid, NULL, 0))
memcpy(shared_memory, &StringToWrite,
                          strlen(StringToWrite)+1);
Housekeeping...
```

Fig. 4. Chunk collection sequence diagram (left) and chunk counter updating methodology (right, pseudo-code).

level. The libgomp library is modified in order to include the automatic chunk collection mechanism described in the left part of Figure 4. Thus, at any allocation of a chunk to a thread, a counter is updated. The chunk counter is saved in a shared memory region. This shared memory is created by using the inter-process communication (IPC) API made available by the operating system. Application binaries linked with this modified version of GCC/libgomp therefore perform this automatic instrumentation. It then makes it possible to monitor CpS and CpJ values during application execution, by reading out from the shared memory region in another process.

We implemented a library that makes it possible to create scripts that access CpS and CpJ values and perform arbitrary processing on those data. By periodically collecting the chunk number, the CpS rate is directly calculated and the CpJ derived from accessing the power consumption. The power consumption information of the Odroid platform is collected by reading directly the current sensors, while the Intel server requires the use of the Intel Performance Counter Monitor (PCM) API and its power utility to get those data. These scripts can further dynamically control various parameters among which the frequency and thread assignment of threads to cores of the system.

The overall concept is illustrated on Figure 4. In particular, it depicts the periodic reading of the shared memory by the script, while the scheduler manages, in an asynchronous fashion, both chunks allocations to threads and shared memory updates. Only the scheduler can write in the shared memory. Thus, there is no concurrency between threads to be managed. No memory security mechanisms, like ”mutexes”, are implemented as the probability of the script and the scheduler accessing the shared memory at the same time is very low. Indeed, we observed that such a conflict error is negligible and canceled out during the analysis process, e.g. neural network training.

These metrics are easily implemented and since it operates at OpenMP level, it is suitable for any architecture for which *libgomp* is available, i.e. most systems. As described previously, they can also be used at runtime, opening perspectives for online analysis of compute systems' efficiency. These perspectives will be developed in the following sections.

IV. Progressive Energy-Efficiency Analysis

The ability to track application execution and its efficiency can be leveraged to select the most beneficial system configurations from the energy-efficiency point of view.

Application codes often show several phases that differ in their behaviors from the perspective of hardware resource usage, e.g., compute-intensive versus memory-intensive. Let us take the trendy example of a running application on the cloud to illustrate this thought. As greatly depicted by A.Bhattacharyya et al. [3], a cloud application usually go through many types of workloads, from a memory storage and load period to a computing period. It can also deal with communication and latency issues. In fact, there are as many types of phases as there are available features on the application. Following in this idea, another common example is a smartphone's application software. This family of application does the interface between the user and the Operating System. Thus, it goes through various workload period, like making a call or access pictures, or simply idling (sleep mode).

Therefore, phase shift means change in workload nature. More particularly, in the case of a parallel workload, this change means modification of chunk characteristics. Indeed, the chunk metric is relative to the workload of an iteration. These changes lead to a variation in the energy consumption, hence the need to adapt the system configuration to the current application phase, in order to optimize the energy consumption.

A. CpS and CpJ Exploitation

We illustrate the need to adapt the system configuration to the current application's phase by illustrating the difference between a compute-intensive and a memory-intensive application. Subsequent to the previous chunk definition, the CpS and CpJ values interpretation follows the next rule: the higher the CpS and CpJ values, the better the performance and energy-efficiency.

Example 2. *Let us consider the OpenMP code depicted in Figure 5. This program is designed to have two phases, with different characteristics.*

```
   #include <omp.h>
 // Initialization
   nthreads = 10;
   double *A, *B, *C, *D;
   posix_memalign((void**)&A, 64,
                          n*sizeof(double));
   posix_memalign((void**)&B, 64,
                          n*sizeof(double));
   posix_memalign((void**)&C, 64,
                          n*sizeof(double));
   posix_memalign((void**)&D, 64,
                          n*sizeof(double));
   for (i = 0; i < n1; ++i) {
       A[i] = 0.1;
       B[i] = 0.0;
   }
   for (i = 0; i < n2; ++i) {
       C[i] = 0.1;
       D[i] = 0.0;
   }
   omp_set_num_threads(nthreads);
   for (j = 0; j < n; ++j) {
     // Parallel region
     #pragma omp parallel for
         schedule(dynamic, 1) // directive OpenMP
     for (i=0; i<n1; i++){
         B[i] = A[i] + B[i];
     }
     // Parallel region
     #pragma omp parallel for
         schedule(dynamic, 1) // directive OpenMP
     for (i=0; i<n2; i++){
         D[i] = fct(C[i], D[i]);
     }
   return 0;
```

Fig. 5. A Simple OpenMP C program alternating compute-intensive and memory-bound computing phases.

It consists in one main for-loop, executing two other consecutive for-loops. These two intricate for-loops are the two different phases (i.e. two different types of chunks) of the main program. The first is a simple addition, corresponding to a memory intensive region. The second is a compute intensive region, where fct() is a computationally demanding function.

Figure 6 is a plot of both CpJ and CpS against time, collected during the execution of this program on a fixed configuration. Energy consumption is plotted as well. Two phases can be observed on both the CpS and CpJ plots, with similar behaviours. Moreover, despite being rather noisy the power profile can still be assumed constant as no significant pattern appear. From the plots it is fair to assume the two phases visible on the plots correspond to those two loops listed above. These results show both CpJ and CpS metrics are suitable for phase analysis.

When executing this code on various architecture configurations, we obtain the energy-efficiency outcomes shown in Figure 7.

Figure 7 shows in (a) the performance (CpS) and in (b) the energy-efficiency (CpJ) of the entire application but also of each of the two phases for three specific configurations. The breakdown (in %) of the execution time between the two phases is depicted in (c). We first define the optimal configuration as that leading to the best energy-efficiency. As it can be seen in the figure each phase has a distinct optimal configuration. Indeed, according to the notation used on Figure 7 configurations 1 and 3 are optimal for respectively phase 1 and phase 2. Moreover, the global best energy efficiency is obtained for configuration 2, different from the phases optimums. Note that having the ability to switch between configurations 1 and 3 would obviously lead to a better average, that corresponding to configuration 2 being obviously a tradeoff.

In this specific example, switching between configuration 1 and 3, regarding the current execution phase, instead of running the entire application with the configuration 2 would lead to an increase of up to 15% (considering a perfect system without switching cost).

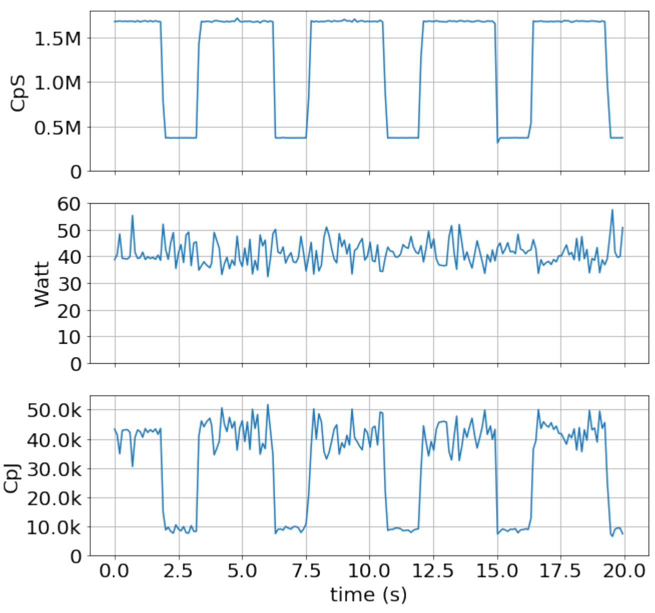
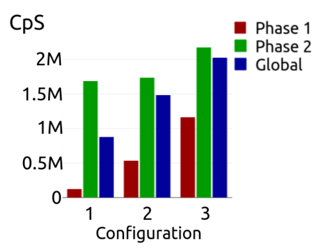
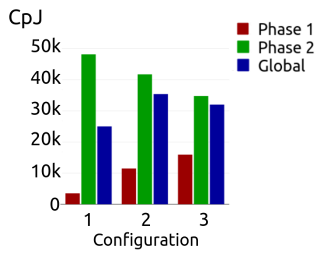


Fig. 6. Synthetic application's profile run on Intel-Server. From top to bottom: CpS, energy consumption (Watt), CpJ.

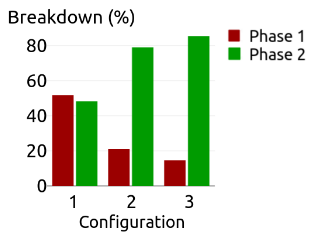
From the above example, we see that CpS and CpJ are adequate metrics that allow one to conveniently capture the energy-efficiency of a system during execution, according to



(a) Global and Phases CpS means



(b) Global and Phases CpJ means



(c) Phases execution time breakdown

Fig. 7. Metrics variation for 3 configurations on an Intel server. 1: 2 cores and f= 1.5GHz; 2: 9 cores and f= 1.5GHz; 3: 17 cores and f= 2.1GHz.(a): Describes the CpS mean values, for the global execution and for each execution phase, (b): Same as (a) regarding the CpJ, (c): Phases duration breakdown

its phases. Indeed, we observe different CpS and CpJ behaviours according to the different phases. It means that phases (i.e. chunk types) have their own CpS and CpJ characteristics, leading to the conclusion that these metrics allow to capture application phases.

While the above phases have been observed a posteriori after program execution, a runtime system is required for automating the process i.e. figuring out how many phases exist in the application, enumerating these and finally identifying which phase is presently active. This then would make it possible to identify in a given phase which configuration works best and select it.

B. Automatic Phase Detection

In the previous section, we observed that both CpS and CpJ reveals in real time the execution phases of an application. We also observed that, from an energy-efficiency perspectives, phases may require different configurations. Hence, detecting the phases and extracting the best configuration are key for efficient dynamic control.

Here, we present a solution to detect in real-time application phases. We achieved this ”phase detection” with excellent results by using a trained autoencoder [11].

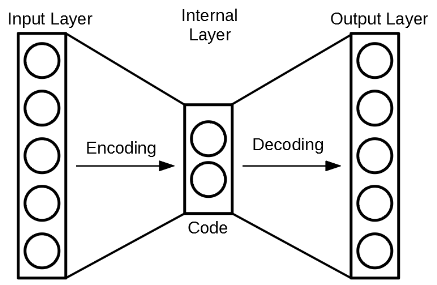


Fig. 8. Illustration of the autoencoder concept

Autoencoders are specific deep neural network topologies that are increasingly popular. They find application in a number of areas such as image denoising [20]. They aim at reducing the dimensionality of the neural network inputs, e.g. the size of images for the image denoising area mentioned before. This dimensional compression is performed by a bottleneck-shaped neural network architecture as illustrated in Figure 8. Indeed, autoencoders have a symmetric shape, where the mirror plane is the internal layer, having a smaller dimension than the input. Hence, two parts can be identified: the encoder and the decoder. The former defines the part from the input layer to the internal layer, while the latter is the part from the internal layer to the output layer. Multiple hidden layers can be implemented within both parts, making the overall neural network deeper. Autoencoders are trained such that the produced output is equal to the input. The compact representation of the input is then available at the boundary between the encoder and the decoder. This is represented by the internal layer as shown in Figure 8.

Here, an autoencoder is trained to reproduce the CpS value and the system’s configuration data by passing through the internal layer, where the information about the phase can be recovered. The autoencoder designed for this phase-detection problem is illustrated in Figure 9 and described in the next.

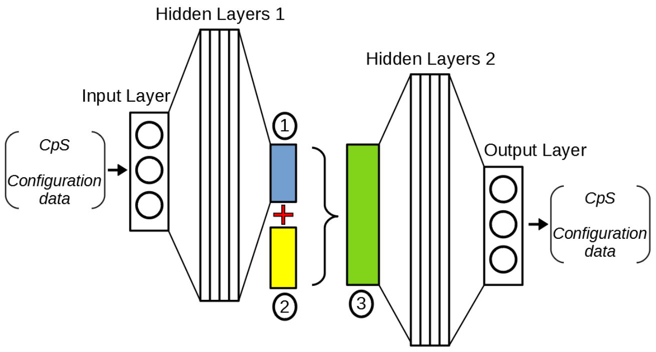


Fig. 9. Designed autoencoder. 1: internal layer, 2: configuration data (#core, frequency), 3: concatenation of 1 and 2

Designed autoencoder principle, cf. Figure 9: In order to make it extract the phase value, a discrete layer (blue box) is used as the internal layer. The information about the configuration (frequency and number of active cores), represented by the yellow box, is then given to the decoder input (green box) by concatenating it to the information contained in the internal layer. Hence, the autoencoder is constrained to build a discrete representation of the CpS, based on the configuration information. This representation matches the phase information. It actually results in training an unsupervised classifier. Indeed, the number of classes corresponding to different phases is not given a priori to the autoencoder. Each class resulting from the training of the autoencoder corresponds to an identified execution phase. The training requires a prior collection stage to sweep across the various system configurations and collect corresponding CpS and CpJ values. After training, the model only requires the CpS value and the configuration to figure out which phase the program is currently in. Thus, real-time phase detection is made possible.

C. Application to SRAD

We illustrate the application of our proposed framework to the SRAD benchmark, from the Rodinia benchmark-suite [7]. It is tested on both an Intel Xeon dual-socket multicore system and an Odroid XU3 platform based on the Armv7 big.LITTLE architecture.

For the rest of this section, the compute system's configuration will refer to a pair of features: the number of cores assigned to the program execution, and the frequency of these cores. As illustrated in Figure 10, the SRAD benchmark is a suitable program to start with to test the framework. Indeed, in this figure both CpS and CpJ of the SRAD benchmark execution are plotted, for a fixed configuration, and two execution phases can be observed.

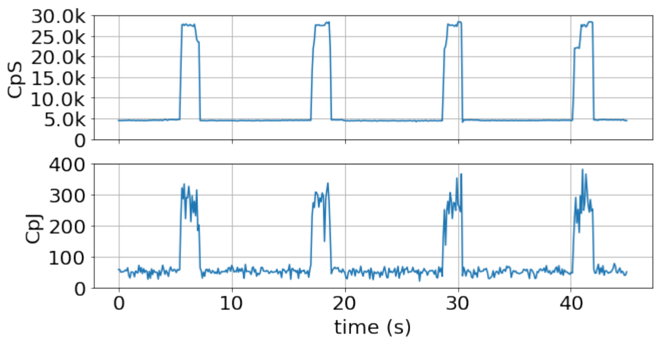
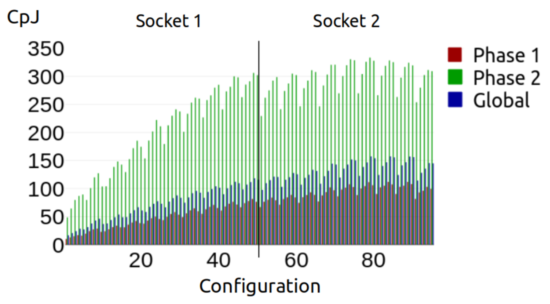
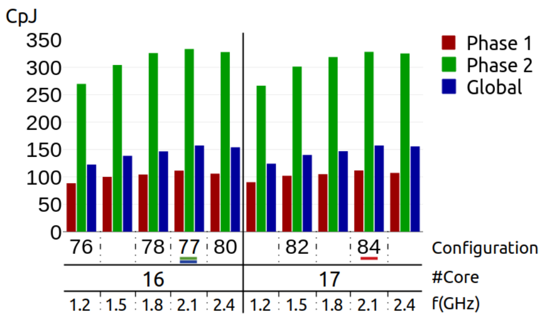


Fig. 10. SRAD-Application execution sample, CpS and CpJ profiles.

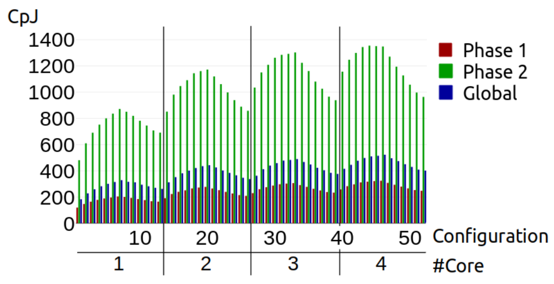
A characterization of the application is run on both compute systems by exploring all possible configurations, i.e. all possible frequencies and core counts. Figure 11a and 11b show the result of the characterization on the Intel dual-socket system. The configuration range starts with a single core up until 19 as a core dedicated to the data collection was excluded from the pool. For each number of cores, a set of frequency is explored, from 40 to 80% of the maximum frequency, with a 10% step. Higher frequencies were not used in the analysis as incurring frequency throttling due to the processor TDP. This gives a total of 95 different configurations. The same experiment has been conducted on the Odroid board, depicted in Figures 11c and 11d with similar results. Note for both systems the best performing configurations are distinct and marked as such in figures 11b and 11d. This simple example demonstrates that the proposed metrics (CpS and CpJ) are relevant information for applications running on embedded systems such as the Odroid board, as well as on high-performance compute nodes like the Intel server.



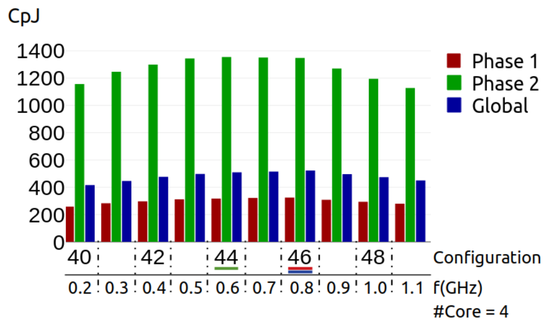
(a) CpJ on Intel-Server, 95 configurations



(b) CpJ on Intel-Server, zoom in optimal configurations (underlined in corresponding phase's color)



(c) CpJ on Odroid Board, 52 configurations



(d) CpJ on Odroid Board, zoom in optimal configurations (underlined in corresponding phase's color)

Fig. 11. SRAD-Application Characterization, on two architectures: an Intel server with 20 cores and an Arm platform with 4 cores.

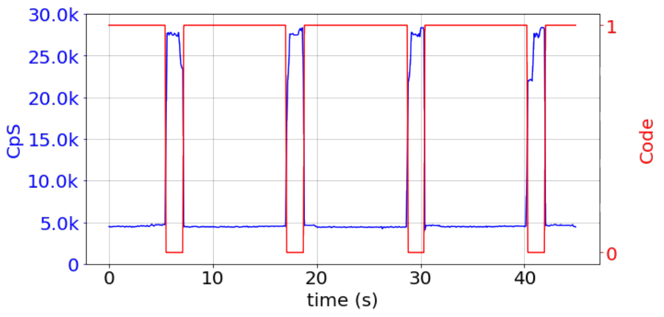


Fig. 12. SRAD-Application phase detection sample on the CpS profile.

Figure 12 shows the CpS plot with the two different phases as discovered by the autoencoder runtime. It illustrates the overall great results provided by the autoencoder, after a relative short training period. Indeed, autoencoder trainings took between 1min and 5min for both the Intel-server and Odroid-board datasets, and converged to a final loss value after few tens of seconds. They have been conducted on an Intel Xeon server featured with a Xeon E3-1225v3 CPU. As expected from the model, once trained, the encoder gives a code corresponding to each phase. The code value is arbitrary and may change from one training to another and shall be rather regarded as an enumerated type.

V. Conclusion and Perspectives

On-line energy-efficiency analysis is essential for better management of resources in multicore systems. In this paper we focus on OpenMP, a parallel programming model widely used in high-performance computing domain among others. We propose two new metrics to evaluate both performance and energy-efficiency of compute systems during run-time. Based on these metrics, we propose a deep learning autoencoder to perform on-line detection of programs' execution phases. Results show great capabilities from autoencoders to effectively extract the phase information from the execution profile. This on-line analysis method has been conducted on both an Intel and Arm architectures to show its flexibility.

This work opens new perspectives for energy consumption optimization. Indeed, the on-line analysis made possible by the introduced metrics can then be exploited by an optimization engine that identifies best configurations after an exploration phase for significant energy-efficiency gains. Further work rely on devising such an algorithm for adaptive systems, where adequate system configurations will be selected upon the feedback from the CpS and CpJ analysis. This could contribute to a more efficient execution of OpenMP programs on heterogeneous platforms [6].

References

[1] Francisco Almeida, Javier Arteaga, Vicente Blanco, and Alberto Cabrera. Energy measurement tools for ultrascale computing: A survey. *Supercomputing Frontiers and Innovations*, 2(2), 2015.

[2] Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Jon Hiller, Sherman Karp, Stephen Keckler, Dean Klein, Robert Lucas, Mark Richards, Al Scarpelli, Steven Scott, Allan Snavely, Thomas Sterling, R. Stanley Williams, Katherine Yelick, Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Jon Hiller, Stephen Keckler, Dean Klein, Peter Kogge, R. Stanley Williams, and Katherine Yelick. Exascale computing study: Technology challenges in achieving exascale systems peter kogge, editor & study lead, 2008.

[3] A. Bhattacharyya, S. Sotiriadis, and C. Amza. Online phase detection and characterization of cloud applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 98–105, Dec 2017.

[4] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *Int. J. High Perform. Comput. Appl.*, 14(3):189–204, August 2000.

[5] A. Butko, F. Bruguier, A. Gamatié, G. Sassatelli, D. Novo, L. Torres, and M. Robert. Full-system simulation of big.little multicore architecture for performance and energy exploration. In *2016 IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC)*, pages 201–208, Sep. 2016.

[6] Anastasiia Butko, Florent Bruguier, Abdoulaye Gamatié, and Gilles Sassatelli. Efficient Programming for Multicore Processor Heterogeneity: OpenMP versus OmpSs. In *OpenSuCo 1 (ISC17)*, Frankfurt, Germany, June 2017.

[7] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, Oct 2009.

[8] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, January 1998.

[9] Thanh Do, Suhib Rawshdeh, and Weisong Shi. ptop: A process-level power profiling tool. In *in Proceedings of the 2nd Workshop on Power Aware Computing and Systems (HotPower09*, 2009.

[10] Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The scalasca performance toolset architecture. *Concurr. Comput. : Pract. Exper.*, 22(6):702–719, April 2010.

[11] Geoffrey E. Hinton and Richard S. Zemel. Autoencoders, minimum description length and helmholtz free energy. In *Proceedings of the 6th International Conference on Neural Information Processing Systems*, NIPS'93, pages 3–10, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[12] Diethelm Kai. Tools for assessing and optimizing the energy requirements of high performance scientific computing software. *PAMM*, 16:837838, 2016.

[13] Sandia National Laboratories. High performance computing power application programming interface (api) specification. http://powerapi.sandia.gov, 2014.

[14] Sparsh Mittal. A survey of techniques for improving energy efficiency in embedded computing systems. *IJCAET*, 6(4):440–459, 2014.

[15] A. Noureddine, A. Bourdon, R. Rouvoy, and L. Seinturier. Runtime monitoring of software energy hotspots. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 160–169, Sep. 2012.

[16] Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. A review of energy measurement approaches. *SIGOPS Oper. Syst. Rev.*, 47(3):42–49, November 2013.

[17] Anne-Cecile Orgerie, Marcos Dias de Assuncao, and Laurent Lefevre. A survey on techniques for improving the energy efficiency of large-scale distributed systems. *ACM Comput. Surv.*, 46(4):47:1–47:31, March 2014.

[18] Sameer S. Shende and Allen D. Malony. The tau parallel performance system. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, May 2006.

[19] M. J. Walker, S. Diestelhorst, A. Hansson, A. K. Das, S. Yang, B. M. Al-Hashimi, and G. V. Merrett. Accurate and stable run-time power modeling for mobile and embedded cpus. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(1):106–119, Jan 2017.

[20] Junyuan Xie, Linli Xu, and Enhong Chen. Image denoising and inpainting with deep neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 341–349. Curran Associates, Inc., 2012.