Fine-grain energy consumption modeling of HPC task-based programs

Jules Risse, Amina Guermouche, François Trahay

Fine-grain energy consumption modeling of HPC task-based programs

Jules Risse
Samovar
Inria
Télécom SudParis
Institut Polytechnique de Paris
91120 Palaiseau, France
jules.risse@inria.fr

Amina Guermouche
University of Bordeaux, CNRS,
Bordeaux INP, Inria, LaBRI
Talence, France
amina.guermouche@inria.fr

François Trahay
Samovar
Télécom SudParis
Inria
Institut Polytechnique de Paris
91120 Palaiseau, France
francois.trahay@inria.fr

***Abstract*—The power consumption of supercomputers is and will be a major concern in the future. Therefore, reducing the power consumption of high performance computing (HPC) applications is mandatory.**

Monitoring the energy consumption of HPC programs is a good first step: using external or software power meters, one can measure the energy consumption of an entire compute node or some of its hardware components. Unfortunately, the differences in scope and time scale between power meters and code level functions prevent the identification of power hungry code blocks.

For this work, we propose leveraging the tracing mechanism of the StarPU runtime system in order to estimate task level power consumption. We trace the execution of the application while regularly measuring coarse-grain energy consumption of central processing units (CPUs) and graphics processing units (GPUs) using vendor software interfaces. After execution, we identify the executed tasks on each processing unit for every coarse-grain energy measurement interval. We then use this information to generate an overdetermined linear system linking tasks and energy measurements. Subsequently, solving the system allows us to estimate the fine-grain power consumption of each task independently of its actual duration.

We achieve mean average percentage errors (MAPE) ranging from 0.5 % to 5 % on various CPUs, and from 10 % to 28 % on GPUs. We show that a solution generated from a run can be used to predict the energy consumption of other runs with different scheduling policies.

***Index Terms*—Software engineering, High performance computing, Energy efficiency, Energy modeling, Runtime system**

I. Introduction

Supercomputers play a key role in computational science and are used to perform intensive workloads in a wide array of fields, yet they can consume a significant amount of power. As an example, El Capitan, the fastest supercomputer in the world according to the top500 list [1] at the time of writing, consumes around 30 MW of power. As a result, footing the electricity bill is one of the major operating costs of supercomputers.

Reducing the energy consumption of parallel applications running on supercomputers is crucial to limit data center power consumption. Improving the energy efficiency of GPUs is a good way of limiting the power consumption of compute nodes. However, exploiting GPUs efficiently is hard, and task-based programming models offer a convenient way to seamlessly exploit the GPU computing power [2] [3] [4]. With such programming models, the application parallelism is expressed by defining tasks and data dependencies.

Understanding the energy consumption in existing task-based programming models still remains difficult, and identifying power-hungry tasks is tedious.

As a matter of fact, tasks typically run for a few milliseconds on all cores of a CPU or on multiple GPU streams, whereas power meter capture the consumption of the entire computing unit at a scale ranging from milliseconds to seconds. Estimating the energy consumption of tasks typically requires to run every type of task repetitively in isolation while measuring overall power consumption.

In this paper, we propose a mechanism for estimating the energy consumption of tasks. We first run the task-based program once while tracing its execution and measuring its energy consumption at a coarse grain (e.g. one measurement every few hundreds of milliseconds). After the execution, we analyze the execution trace and build a linear system that describes, for each energy measurement, the energy contribution of each type of task. Solving the linear system allows us to estimate the power consumption of each task. The contributions of this paper are the following:

- we propose a new mechanism for estimating the power consumption of tasks in task-based programming models running on heterogeneous computers;
- we implement this mechanism in StarPU [5]. We modify StarPU so that it logs the energy consumption of computing resources in addition to its existing logging system for tasks execution;
- our evaluation show that our proposed approach accurately models the power consumption of tasks, and that this energy model can be reused for predicting a task energy consumption.

The remainder of this paper is structured as follows. Section II introduces key concepts about energy measurements on computing clusters and task-based runtime systems. In Section III, we then present our contributions and our proposed task-based regression model for energy consumption. We evaluate our proposal in Section IV. We present related work in Section V. Finally, Section VI concludes the paper and outlines directions for future work.

II. Motivation

In this section, we review energy measurement tools available on HPC clusters, with a focus on software-based solutions. We then introduce the StarPU runtime system and discuss the challenges of integrating energy aware scheduling policies to it.

A. Measuring the energy consumption of applications

To improve the energy efficiency of an application, one of the first important steps is to precisely measure its energy consumption. The power consumption of a server can be measured using external power meters (typically at the Power Distribution Unit (PDU) level) [6], or using software power meters that measure the power consumption of several hardware components (typically for the CPU, the GPU, or the memory) [7].

In this paper, we are mainly interested in software power meters, as they are widely available and do not require to equip servers with additional equipment. The main software power meters that we consider are presented in Table I. In this table, *granularity* refers to the refresh period of the hardware counters as exposed through their software interface.

TABLE I
Power meter interfaces and sampling granularities

HW	Interface	Granularity
Intel/AMD CPU	Running Average Power Limit (RAPL)	1 ms
NVIDIA GPU	NVIDIA Management Library (NVML)	20–100 ms
AMD GPU	Radeon Open Compute System Management Interface (rocm-smi)	1 ms

The RAPL (Running Average Power Limit) interface exposes the energy consumption of various parts of Intel and AMD CPUs [8]. Figure 1 presents a hierarchical overview of RAPL domains and their underlying measured components. On compute nodes, the *package* domain is the most widely available on both AMD and Intel CPUs. Depending on the CPU model and architecture, other domains may be available. RAPL measurements are generally considered accurate [9] and can be accessed with a low overhead [10].

The energy consumption of NVIDIA and AMD GPUs can be measured with the NVML and the ROCm libraries, respectively. NVIDIA claims that the reported values have an accuracy of $\pm 5\%$, but their sampling method may lead to inaccurate measurements [11]. On NVIDIA GPUs, the energy consumption counter update period depends on the GPU model but ranges from 20 ms to 100 ms. To our knowledge, no work has been done on validating the reported energy values on AMD GPUs.

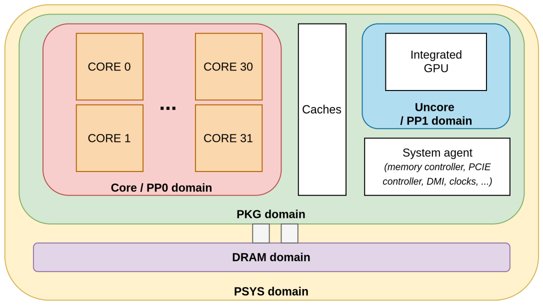

Fig. 1. RAPL domains hierarchy

B. Task-based runtime systems

Task-based runtime systems such as StarPU [5] are able to efficiently exploit heterogeneous architectures. In such programming models, developers express their application parallelism by defining tasks and data dependencies. For each task, a developer can provide StarPU with one implementation per computing resource (typically a CUDA kernel for running the task on a GPU, and a function for running the task on a CPU). The tasks and their dependencies form a Directed Acyclic Graph (DAG). For example, Figure 2 represents the DAG of a Cholesky decomposition on a 5x5 matrix.

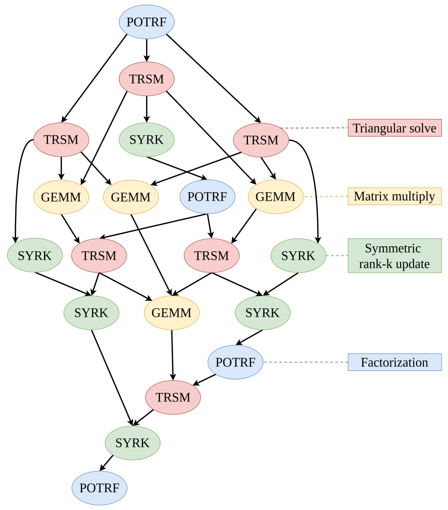

Fig. 2. DAG of A 5x5 matrix Cholesky decomposition

The StarPU runtime system implements several scheduling policies that execute tasks and transfer data on the available computing resources (through *workers*) in order to minimize the application makespan. StarPU includes a history based performance model to improve scheduling decisions and provide performance feedback [12]. In practice, the execution times of tasks on different computing units are measured as the program is running and inserted inside a performance model.

This allows schedulers to decide which task to execute on which computing resource. For example, if two worker types are available (for CPU and GPU), a compute intensive matrix multiplication will probably be scheduled on the GPU by a

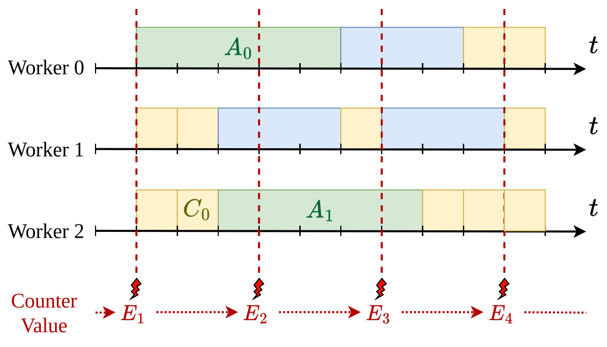

Fig. 3. Granularity mismatch between tasks and energy counters

GPU worker, while a branch-and-bound task will more likely be scheduled on the CPU and executed by a CPU worker.

C. Integrating energy aware scheduling policies to StarPU

In order to reduce the energy consumption of a StarPU application, scheduling policies can take into account the energy costs of tasks on the available computing resources.

However, a major issue and the motivation for this work stems from differences in scope and scale between software power meters and StarPU tasks:

- **spatial granularity**: on CPU, RAPL domains capture the energy consumption of the entire socket (or the sum of all cores at best), on which many tasks are executed in parallel. Similarly, GPU tasks can be executed on parallel streams while the energy measured encompasses the entire card;
- **time granularity**: The typical duration of a StarPU task is a few milliseconds (see Section IV-F) while the granularity of software power meters provided by various vendors ranges from 1 to 100 milliseconds. Even for cases with comparable durations, the measurement error can be high as the exact timestamp of the hardware counter update is unknown.

Figure 3 presents these issues in a typical StarPU program. The hardware energy counter updates are represented in red. This energy is used by all 3 workers, which perform different tasks in parallel. Since each task can have a different energy cost, partitioning global energy across workers is not trivial.

Even if we could isolate the energy of a single worker, we would still face problems:

- task C is too short to be measured, and for C_0 we would have $E_{C_0} = E_1 - E_1 = 0J$ as the counter value is the same at the beginning and the end of the task;
- the duration of task A is slightly shorter than twice the counter update period. Therefore, we would underestimate energy consumption for A_0 with $E_{A_0} = E_2 - E_1$ and overestimate it for A_1 with $E_{A_1} = E_3 - E_1$.

As a result, automatically building an energy-consumption model of StarPU tasks with direct measurements is currently impractical.

Our goal is to provide fine-grain energy values for StarPU tasks, in order to enable the development of energy-aware

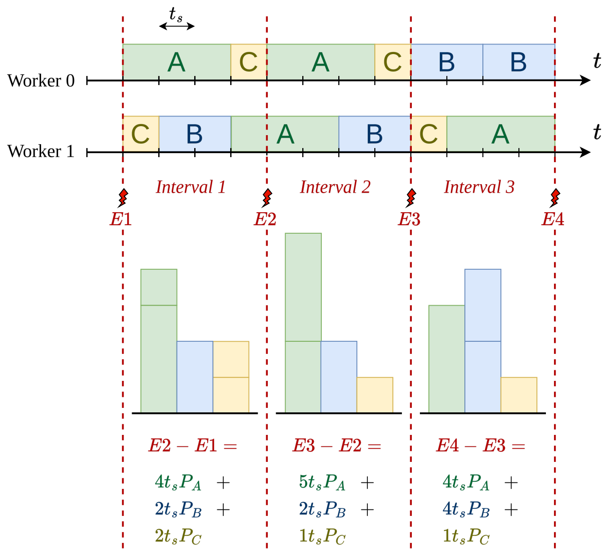

Fig. 4. Fine-grain energy consumption methodology

scheduling policies that can reduce the energy consumption of a cluster while retaining acceptable performances.

III. Modeling the fine-grain energy consumption of tasks

We propose to model the energy consumption of fine-grain StarPU tasks based on several coarse-grain energy measurements.

As illustrated in Figure 4, we propose to run the StarPU application once and generate an execution trace that captures both energy consumption and worker states.

In StarPU, workers do not only execute tasks: they may also check their task queues, wait for new tasks, perform data movements, or execute callbacks. At any given time, a worker is in a well-defined state, which is recorded by the tracing system.

During the execution of the application, the energy consumption of each processing unit (CPU or GPU) is measured at a coarse granularity (typically every 50 ms).

Let n be the number of measurement intervals on a given processing unit. The energy consumed during interval i of duration Δt_i is denoted by:

$$\delta_i = E_{i+1} - E_i \quad (1)$$

Let m be the number of workers associated with the processing unit, and let $S = \{S_1, S_2, \dots, S_l\}$ be the finite set of l possible states (including computational tasks and auxiliary states).

We log the time spent in each state across all workers for each energy measurement interval. Let $t_i^{S_k}$ denote the total time spent in state S_k during interval i, aggregated across all workers:

$$t_i^{S_k} = \sum_{j=1}^{m} t_i^{S_k j},$$

where $t_i^{S_k j}$ is the time spent by worker j in state S_k during interval i.

We model the total energy consumption in interval i as the sum of state durations weighted by their unknown per-worker power consumption P^{S_k}:

$$\delta_i = t_i^{S_1} \cdot P^{S_1} + t_i^{S_2} \cdot P^{S_2} + \cdots + t_i^{S_l} \cdot P^{S_l}$$

Since each worker is always in exactly one state at any given moment, the sum of all state durations within an interval must account for all worker time. Therefore, the following holds for interval i:

$$\sum_{k=1}^{l} t_i^{S_k} = m \cdot \Delta t$$

By collecting multiple intervals, we obtain the following overdetermined linear system, with $n \gg l$:

$$\begin{cases} \delta_0 = t_0^{S_1} \cdot P^{S_1} + t_0^{S_2} \cdot P^{S_2} + \cdots + t_0^{S_l} \cdot P^{S_l} \\ \delta_1 = t_1^{S_1} \cdot P^{S_1} + t_1^{S_2} \cdot P^{S_2} + \cdots + t_1^{S_l} \cdot P^{S_l} \\ \quad \vdots \\ \delta_n = t_n^{S_1} \cdot P^{S_1} + t_n^{S_2} \cdot P^{S_2} + \cdots + t_n^{S_l} \cdot P^{S_l} \end{cases}$$

Once the execution completes, we build this linear system from the trace and solve it independently for each processing unit. This approach allows estimating the power of short-lived tasks that may only run for a few milliseconds.

Note that both static and dynamic power components are implicitly included in the model: low-power states such as *Idle* naturally capture static power contributions.

This modeling of worker states power consumption relies on three assumptions:

1) all occurrences of a state S_k have the same power consumption. Since StarPU tasks are typically linear algebra kernels that process blocks of data, this should be mostly true whatever the block size;
2) StarPU is the only program using the available processing units. Therefore, StarPU worker states are sufficient to explain a processing unit's power consumption. In practice, StarPU pins workers on all available cores and uses all visible GPU devices, validating this assumption;
3) the linear system is not underdetermined: there should be more measurement intervals than worker states. In our experiments, the maximum numbers of states in an application was 14, and running the application for a few seconds was enough to build an overdetermined linear system.

It is important to note that the obtained value is the power consumption of a state **on one worker**. There are typically as many workers as cores on a CPU, and multiple GPU workers can handle the same accelerator.

A. Tracing tasks and energy measurements

In StarPU, tracing relies on the `FxT` library [13]. StarPU records events when reaching key steps, such as a task submission or the beginning and end of a task execution [14]. The event probes contain information about context and are automatically timestamped and saved by `FxT`. The trace is saved at the end of the program execution, and may be flushed to disk earlier during execution if it is too large.

Events are divided into categories relating to the mechanisms to trace: task scheduling, worker states, data transfers, lock states, user events, MPI communications, and more. We configure StarPU to collect information about worker states and executed tasks.

We also modified StarPU so that it regularly collects the energy consumption of available computing units. We implemented a lightweight library for monitoring the energy consumption of CPUs (using a `perf_event_open` wrapper) and GPUs (using NVML for NVIDIA GPUs, and rocm-smi for AMD GPUs) [1].

Measurements are done by CPU workers in their active waiting loop which occurs right after finishing a task. Therefore, we do not have to dedicate a thread to energy measurement, which would take away a StarPU worker.

To minimize measurement errors, we define a measurement interval for each processing unit as follows:

$$delay = \max(50,\ 5 \times granularity)$$

with $delay$ being the sampling period in milliseconds and $granularity$ the update period of the software counter associated with the processing unit. We measure $granularity$ during StarPU's initialization phase. The 50 ms value is the minimally allowed sampling interval.

For CPU, we query the RAPL *package* domain as it is the most widely available one. After each measurement, StarPU records an FxT event with the total accumulated energy since initialization.

Delays may also be set by the user through environment variables. Increasing the delay lowers overhead and measurement errors but also reduces the number of intervals taken and thus the degree of freedom of the resulting linear system.

B. Post-mortem trace analysis

After the execution of the application, we process the FxT trace to generate a linear system that allows us to model the power consumption of worker states. We seek to reconstruct, for each energy measurement interval and for each computing unit, the duration of all the tasks and auxiliary states executed by the underlying workers. We process the FxT trace and generate a CSV file that contains worker states alongside energy measurements.

The generated CSV file is subsequently processed by `EnergySolver` [2], a Python program that builds and solves overdetermined systems, linking the power consumption of

[1]Available as open-source at https://gitlab.com/JulesRisse/energy-reader.
[2]Available as open-source at https://gitlab.com/JulesRisse/energy-solver.

StarPU worker states and energy measured on the different processing units of the system. Moreover, `EnergySolver` also records all analysis results to a database and allows their visualization through a web application, as shown in Figure 5.

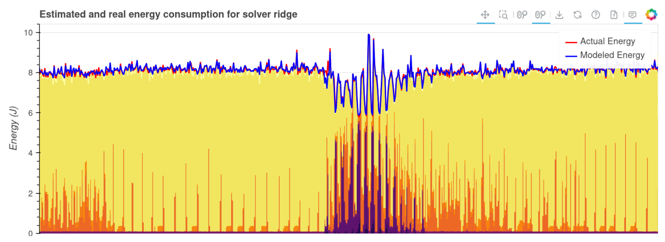

Fig. 5. Web application visualization of interval energies, where each color corresponds to a task

1) Building the linear system: `EnergySolver` first determines the time spent in each worker state for every energy measurement interval, as well as the energy consumed. Recorded states include tasks, *Sleeping* (periods when a worker is waiting without a task), data transfers, *Idle* (which occurs once at worker initialization), energy measurements, and *Overhead*, which encompasses time spent in code regions not explicitly instrumented.

Energy measurement intervals and states can be mapped with the following logic: for each interval, we use timestamps to compute the intersection of all states pertaining to the interval. Since energy measurements are either made on a CPU socket or a GPU, the state selection methodology differs:

- for a package wide measurement (CPU socket), we take into account CPU workers pinned on a core of the package and compute time spent in each state;
- for a GPU measurement, we take into account all GPU workers handling the GPU and compute time spent in each task. Multiple workers can handle the same GPU and drive their own CUDA/HIP stream which enables parallel execution of GPU tasks.

Each energy interval corresponds to a line in our linear system. We also take advantage of the power equation:

$$E = P \times t$$

with E the energy in joules (J), P the power in watts (W) and t the time in seconds (s).

Therefore, the explanatory variables and the results of `EnergySolver` correspond to the average power consumption of tasks and states.

Figure 4 shows an ideal and easily solvable system containing three equations and three variables. In practice, we end up with dozens of variables and hundreds of equations: the system is overdetermined and has no explicit solutions, but one can be approached by solving a least squares problem.

2) Solving the overdetermined linear system: Once a linear system is built with n time intervals and l distinct states, we construct:

- a matrix $\mathbf{A} \in \mathbb{R}^{n \times l}$ where rows encode the duration of each state in a given interval;
- a vector $\mathbf{b} \in \mathbb{R}^{n}$ representing the total energy consumed during those intervals;
- an unknown coefficient vector $\mathbf{x} \in \mathbb{R}^{l}$, representing the average power consumption associated with each state.

The goal is to solve the overdetermined system:

$$\mathbf{Ax} \approx \mathbf{b},$$

by minimizing the sum of squared residuals:

$$\hat{\mathbf{x}} = \arg\min_{\mathbf{x}} \|\mathbf{Ax} - \mathbf{b}\|_2^2 .$$

This ordinary least squares (OLS) problem provides a best-fit solution when $n \gg l$ and $\mathbf{A}$ is full rank, i.e. when it has linearly independent rows and/or columns.

In our model, the regularity of a HPC application can create intervals with identical state distribution across workers. It is also likely to have multiple states with similar power consumptions. As a consequence, the matrix $\mathbf{A}$ may show high multicollinearity or lack sufficient rank. Furthermore, OLS does not support the enforcement of physical constraints, and we frequently observed that its unconstrained solution produced negative or impossibly large power estimates.

To address these issues, we explored constrained least squares formulations that introduce physical bounds on the coefficients, as well as regularized methods to increase numerical stability and partially correct multicollinearity.

Among regularization techniques, the Ridge regression method modifies OLS by calculating coefficients that account for potentially correlated predictors. It solves the following penalized least squares problem:

$$\hat{\mathbf{x}} = \arg\min_{\mathbf{x}} \|\mathbf{Ax} - \mathbf{b}\|_2^2 + \alpha \|\mathbf{x}\|_2^2 ,$$

where $\alpha > 0$ is a regularization hyperparameter.

This penalty term adds bias but reduces the variance of the estimates and improves generalization. In our study, we use the Ridge regression implementation from `scikit-learn` with $\alpha = 0.5$, which we have found to achieve the best bias-variance tradeoff across tested hardware platforms.

We also implement bootstraping to estimate the standard error for each coefficient, as well as the associated t-statistics and p-values metrics. The t-statistic indicates the significance of each coefficient, while the p-value reflects the probability of observing such a result by chance. High t-statistics and p-values of 0 are better.

Once we obtain average power values of tasks, we can determine their average energy consumption by referring to StarPU's performance model, which does contain accurate execution time values.

IV. Evaluation

In this section, we evaluate our proposed approach for modeling the energy consumption of fine-grain tasks. We first measure the overhead of collecting energy and performance data. Then, we evaluate the accuracy of our energy model, and we assess whether this model can be used for predicting the energy consumption of tasks.

A. *Experimental settings*

We run experiments on four different machines from the Plafrim and Grid5000 [15] testbeds:

- **paradoxe** is a compute node equipped with two Intel Xeon Gold 5320 CPUs (26 cores each) running at 2.2 GHz and 384 GiB of RAM;
- **suroit** is a compute node equipped with two AMD Zen4 Genoa EPYC 9224 CPUs (24 cores each) running at 2.4 GHz and 256 GiB of RAM;
- **grouille** is a compute node equipped with two AMD Zen2 Rome EPYC 7452 CPUs (32 cores each), two NVIDIA A100 GPUs (40 GiB), and 128 GiB of RAM;
- **enbata** is a compute node equipped with two AMD Zen4 Genoa EPYC 9334 CPUs (32 cores each) running at 2.7 GHz, two AMD MI210 GPUs, and 384 GiB of RAM.

Table II summarizes thermal design power (TDP) values for each node's CPU and GPU if any.

TABLE II
THERMAL DESIGN POWER (TDP) OF COMPUTE NODES

Node	CPU TDP (W)	GPU TDP (W)
paradoxe	185	–
suroit	200	–
grouille	155	250
enbata	210	300

We evaluate our energy model approach using two linear algebra libraries that rely on StarPU:

- Chameleon [16] is a library that provides parallel algorithms to perform BLAS/LAPACK operation on dense matrices. It supports CPU, CUDA and HIP. We evaluate two double precision use cases:
  - **dgesv_nopiv**: general linear system solve, using LU decomposition without pivoting [17];
  - **dpoinv**: symmetric positive definite matrix inversion.

  On CPU only nodes (paradoxe and suroit), we use randomly generated dense square matrices of size 40000 with a block size of 384. On GPU powered nodes (grouille and enbata), we set matrices size to 64000 and block size to 1500.
- QR Mumps [18] is a linear algebra library for solving sparse matrices. It implements a direct solution method based on the QR factorization. We evaluate one double precision use case: **spgeqr** or sparse general QR solver, used on the TF17 matrix from the SuiteSparse collection [19] with a block size of 384. Because of StarPU version incompatibilities, the QR Mumps use case is only executed on CPU nodes.

We run each application using 3 different StarPU scheduling policies:

- the **dmdas** (deque model data aware sorted) strategy takes task execution performance models into account to schedule tasks where their termination time will be minimal. It also takes into account data transfer time and tasks priorities when they are available;
- the **random** scheduler uses one queue per worker, and distributes tasks randomly according to assumed worker overall performance;
- the **lws** (locality work stealing) scheduler uses a central queue from which workers draw tasks to work on. When a worker becomes idle, it steals one from neighbor workers.

As a result, each execution of an application on a given platform is different because the scheduling policy decides at runtime the tasks to execute on each computing resource.

As for StarPU's configuration, we set 2 GPU workers per GPU if they are present, and fill the rest of the available cores with CPU workers. Hyperthreading is disabled.

B. *Overhead of energy measurement*

We first evaluate the performance impact of tracing a StarPU application to collect performance and energy data. We run the dpoinv application 10 times on each node and collect the performance in Gflop/s computed by Chameleon after each run. Each series of run corresponds to one of the following settings:

- **vanilla** runs the application with StarPU compiled without FxT;
- **fxt_disabled** runs the application with StarPU compiled with FxT and trace recording disabled;
- **fxt** runs the application while generating a trace with FxT, with workers states but without energy measurements;
- **fxt+energy** runs the application while generating a trace with FxT, with both workers states and energy measurements.

Figure 6 reports the relative performance of the different settings. We observe no added overhead from the (**fxt+energy**) setting compared to the **fxt** one, although the enbata node shows higher tracing overhead and standard error in general.

We then analyze in detail the overhead of collecting the energy consumption of CPUs and GPUs by manually instrumenting StarPU. We measure the granularity of each computing device (how often the energy consumption counter is updated), and the cost of measuring its energy consumption. The results of this experiment are reported in Table III, which contains measured granularity and average overhead for all the experiments.

TABLE III
TRACING OVERHEAD BY NODE AND PROCESSING UNIT

Node	Computing Resource	Granularity (ms)	Overhead (μs)
paradoxe	Intel Xeon Gold 5320	1	29.32
suroit	AMD EPYC 9224	1	15.28
grouille	NVIDIA A100	100	6011.00
grouille	AMD EPYC 7452	1	82.71
enbata	AMD EPYC 9934	1	12.89
enbata	AMD MI210	1	405.22

We observe that the measurement overhead on CPU is low, with some variations depending on the vendor and generation: AMD Zen4 CPUs show the lowest overhead at 13 μs and 15 μs, followed by the Intel CPU at 29 μs and the Zen 2 AMD CPU with 83 μs.

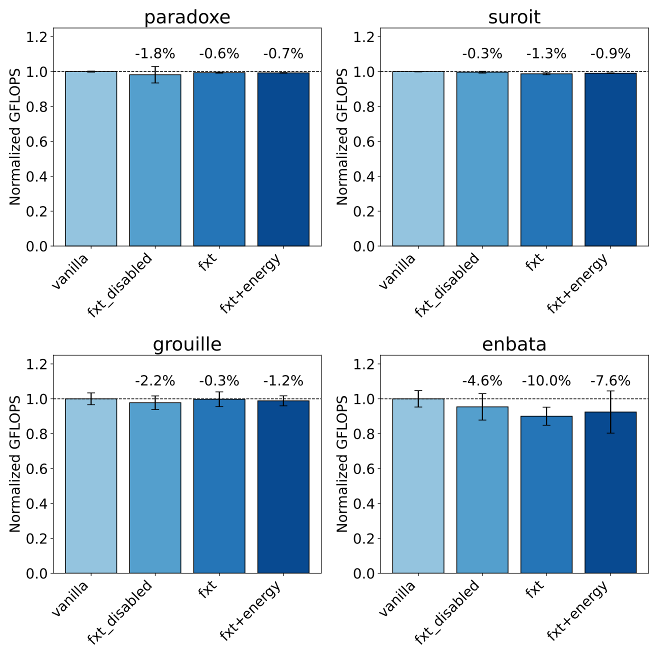

Fig. 6. Normalized tracing overhead per machine

GPUs show much higher overhead, with 405 μs for the AMD card and 6 ms for the NVIDIA one.

Further testing on the NVIDIA A100 showed lower response time when the card is idle (at around 3.2 ms) and very high tail overhead of up to 600 ms during our runs, which tend to push the GPU to its limits.

Energy counter granularity is 1 ms for all computing units but the NVIDIA A100, where the counter is updated every 100 ms.

In the remaining of the paper, we follow the formula presented in Section III-A and set the interval duration as follows: all CPU and AMD GPU intervals last for 50 ms while the NVIDIA A100 interval is set to 500 ms for the grouille node.

C. Empirical validation of the energy-consumption model

Validating the accuracy of our fine-grain energy consumption model is tedious because we cannot measure the actual energy consumption of short live tasks in isolation. This energy consumption can however be estimated by running the same small task many times and measuring the overall energy consumption of the computing unit for the entire run to precisely infer the average power consumption of said task.

In this experiment, we use a command from the Chameleon library CLI utilities (`chameleon_batch_testing`) that repetitively executes a single task on all available workers, and we measure the task average energy consumption. Tasks available for batch testing are small tasks that are often used in many use cases. We focus on the dgemm (matrix multiplication) task, which is predominant in terms of execution time on both our Chameleon use cases.

We perform 10 batch testing runs for the dgemm task on each node and divide the total power consumption by the number of workers per processing unit to be able to compare it with the model. The average worker dgemm power consumption of our model is obtained from 30 executions for each node. On nodes with GPUs, we only report results for GPU dgemm implementations since it is where the task is overwhelmingly scheduled and executed in practice.

TABLE IV
MEAN DGEMM POWER BY NODE AND PROCESSING UNIT

Node / Method	paradoxe (Intel CPU)	suroit (AMD CPU)	grouille (A100)	enbata (MI210)
Batch	7.364	8.118	107.419	87.467
Model(dgesv_nopiv)	7.677	8.25	74.91	85.94
Error (%)	+4.25 %	+1.62 %	-30.26 %	-1.75 %
Model(dpoinv)	7.543	8.36	144.83	100.56
Error (%)	+2.43 %	+2.98 %	+34.83 %	+14.96 %

The results in Table IV compare the mean dgemm power consumption measured empirically (”Batch”) and predicted by our model (”Model”) when running dgesv_nopiv or dpoinv.

On CPU-only nodes (paradoxe and suroit), the power-consumption of dgemm estimated by the two models are close to the power-consumption measured when batch testing. These low error margins validate the model’s accuracy on a highly used task on CPUs.

On GPU nodes (grouille and enbata), the two models’ estimations and the batch estimation are significantly different. This may be due to inaccurate energy consumption data reported by the GPUs. Indeed, a previous study of energy measurements on NVIDIA GPUs shows that NVML may report drastic under/overestimation of power usage [11] because only 25 % of the GPU runtime is sampled on a A100 card (25 ms out of the 100 ms between two counter updates). This may not be a problem when the GPU executes the same task over and over again, because its power consumption would probably be constant. However, when the application runs several types of tasks, the GPU power consumption may vary during a measurement interval, and the energy consumption data reported by the GPU may prove to be inaccurate. This inaccurate data could explain the model failing to accurately predict task power consumption on GPU.

D. Statistical validation of the energy-consumption model

We now evaluate how the energy-consumption model produced by `EnergySolver` fits the actual energy consumed by an application. We run the three use case applications described in Section IV-A while tracing executions of tasks and energy consumption. `EnergySolver` then computes the model for each run on all computing resources. Finally, we compare the predicted energy consumption with the actual measurements.

Figure 7 reports the average of all Mean Average Percentage Errors (MAPE) of predictions. On CPU nodes (paradoxe and suroit), the models derived from the three applications are able

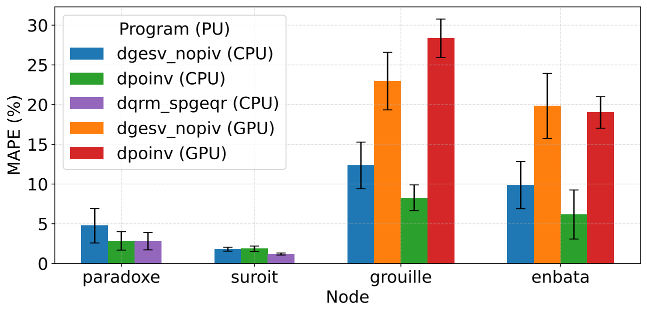

Fig. 7. Mean average percentage error by node and processing unit

to predict the application's energy consumption with a MAPE of less than 5 %.

On both GPU nodes, the estimations of the energy consumption of the GPU are inaccurate, and results are similar to those from Section IV-C. On these machines, the prediction of the CPU energy consumption is also less accurate than for the CPU-only nodes, while still achieving less than 12 % of errors. This is because in CPU+GPU setting, StarPU mostly schedules tasks on the GPU, with the CPU spending 80 % of its time inside the *Overhead* State, which is challenging to predict since it contains all unrecorded regions of code.

E. Predicting the energy consumption using a model

We now evaluate whether the energy model solution obtained from a single execution can be used to accurately predict the energy consumption of other program runs, or other scheduling policies that distribute the tasks' executions differently.

For each pair of `training` and `testing` executions on a processing unit and a for a given program, we generate a power model from `training`. We then apply it to `testing` to predict its energy consumption, and generate the resulting MAPE metric.

Figures 8, 9, 10, and 11 report the results of this experiment in the form of prediction accuracy matrices for each application. For GPU nodes (enbata and grouille), we use the dmdas scheduler because other scheduling policies tend to leave the GPU idle for a significant part of the execution. For readability, we report the Mean Average Percentage Error of prediction for 9 `training` executions on 9 `testing` executions.

On CPU nodes (paradoxe and suroit), the results show that a model can accurately predict the energy consumption of other runs of the application with different scheduling policies. In most cases, the prediction error is less than 5 %. On the paradoxe node, models created with the dmdas policy for the dgesv_nopiv use case show higher error rate. Observations show that using the dmdas scheduler is 22% slower than using lws or random, with additional time spent in the *Overhead* state (20 % of the time versus 7% for dpoinv and 11% for dqrm_spgeqr). Since this state corresponds to unrecorded regions of code, uncaptured variability in this state may degrade the accuracy of our modeling.

This shows that, on CPUs, the fine-grain energy consumption model extracted from traces can predict the power consumption of a task with a good accuracy.

On GPU nodes (enbata and grouille), prediction errors are higher (between 15 % and 22 % on the AMD GPU, and between 15 % and 36 % on the NVIDIA GPU).

Observed errors on the Nvidia GPU node are consistent with the results presented in Section IV-D and may stem from the lack of accurate energy measurement probes on GPUs used. A mitigation in this case could be to only account for traces that fall entirely within the subsampled interval of time, assuming the sampling process is deterministic.

As for AMD GPU nodes, further testing shows that the compute frequency of the MI210 varies during our applicative runs, breaking the assumption that all states consume the same power (Section III) as frequency and power consumption are tightly correlated. It also explains the difference observed in table IV, as dgesv_nopiv is less impacted by these frequency variations than dpoinv. A solution to tackle variable frequencies could be to periodically measure and trace GPU frequency and characterize worker states per frequency or frequency range.

Another general strategy to improve precision could be to rely on an external wattmeter to measure the power consumption of the GPU or the node, in order to precisely model the fine-grain power consumption of tasks running on GPUs, and to achieve the same accuracy as for CPUs.

F. Model results summary

We now report statistics on the tasks and worker states measured or modeled for all the tested applications. For each compute node, states model results are grouped per processing unit and application.

We compute the average power value and pooled standard error across experiments. This allows us to derive the corresponding t-statistic by dividing the mean estimated power value by its pooled standard error. This ratio indicates the significance of the estimation: predictions with high t-statistics are more likely to correspond to the actual power-consumption of a task. We also compute the mean duration for each state, which is only relevant for tasks. Finally, we compute occupation for each state, i.e. the percentage of time spent by workers in that state compared to the total worker execution time of the program.

Tables V, VI, VII, and VIII report these statistics for the four evaluated machines. We filter the reported power state coefficients by retaining only those which account for at least 1 % of total state measured time for the run.

We observe that the typical duration of a task ranges from a few milliseconds to a few hundred milliseconds. The estimated power consumption of most tasks ranges from 2 W to 8 W on CPUs, and between 14 W and 145 W on GPUs. It is consistent with computing resources' TDP values. This is because power consumption corresponds to a task executed on a worker.

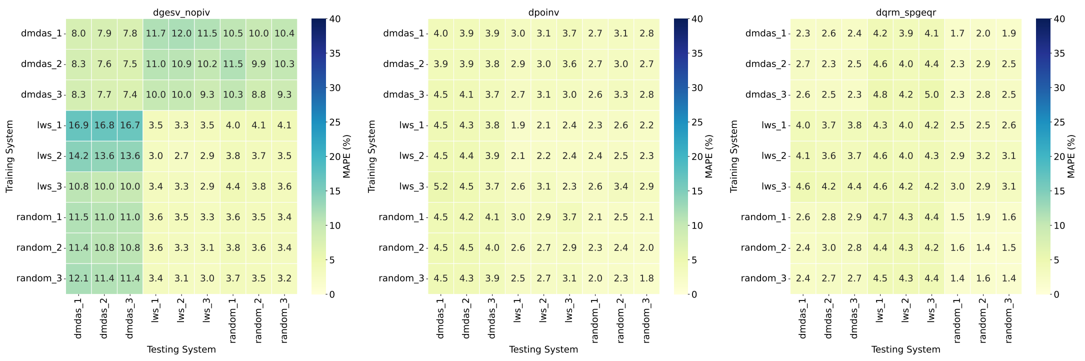

Fig. 8. Prediction accuracy matrices for paradoxe (Intel Xeon Gold 5320 CPU)

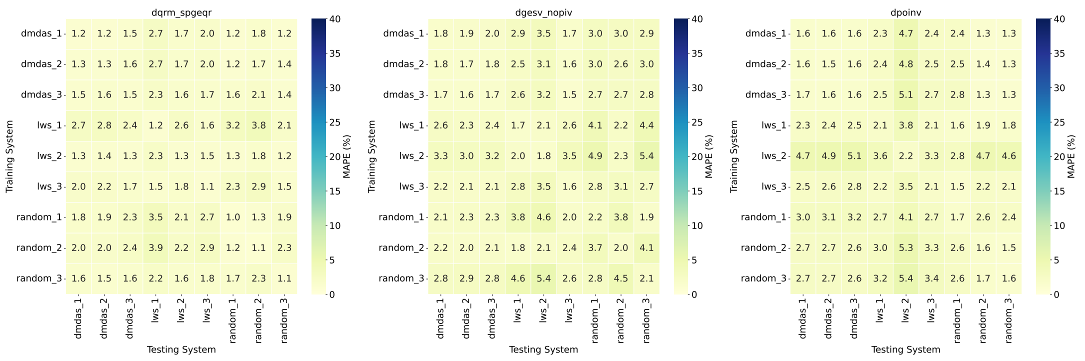

Fig. 9. Prediction accuracy matrices for suroit node (AMD Zen4 Genoa EPYC 9224 CPU)

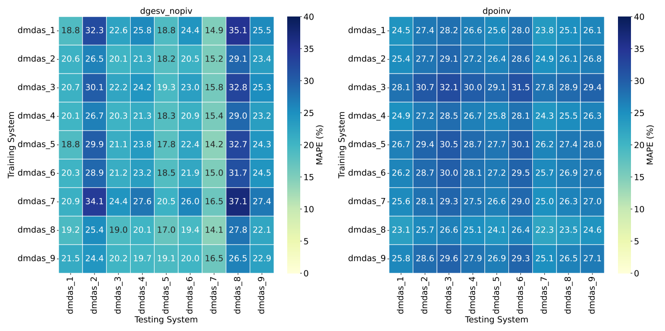

Fig. 10. Prediction accuracy matrices for grouille node (NVIDIA A100 GPU)

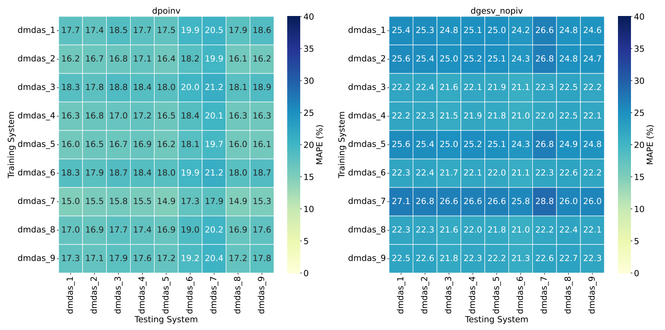

Fig. 11. Prediction accuracy matrices for enbata node (AMD MI210 GPU)

For example, if all workers were to execute the dgemm task at the same time on a CPU on the paradoxe node, the total CPU power would be

$$n_{cores} \times P_{dgemm} = 24 \times 7.5 = 180W$$

which is close to the TDP of the processor (185 W).

A few tasks or worker states are found in multiple programs. For example, `dgemm` or `dtrsm` are building blocks of Chameleon applications. Similarly, StarPU internal states (*Idle*, *Sleeping*, *Overhead*) are found in all the tested applications. We observe that the *Sleeping* state is not a low-power mode : workers in this state actively wait on condition variables, which still involves regular CPU activity and power consumption. For tasks, the results show that models obtained from different use cases predict the same power consumption on a CPU for a given task. Thus, a fine-grain power consumption model obtained from an application could be reused in other applications, as long as the underlying task is the same.

On GPUs, the different models for a given task may predict different power-consumption, as presented in Section IV-C. However, despite this inaccuracy, the models still show that the `dgemm` task is the heaviest contributor to power consumption for both GPUs.

V. Related Work

The article from Jay et al. [7] is a good overview of software-based power meters on CPU and GPU. Software

TABLE V
WORKER STATE POWER BREAKDOWN ON CLUSTER `PARADOXE`

Program	State	Power (W)	T-stat	Duration (ms)	Occup. (%)
CPU Worker					
dgesv_nopiv	Idle	2.59	4.55	-	3.721
dpoinv	Idle	2.60	5.048	-	1.651
dgesv_nopiv	Overhead	4.95	15.74	-	21.534
dpoinv	Overhead	5.16	16.77	-	7.758
dqrm_spgeqr	Overhead	5.03	69.20	-	11.522
dgesv_nopiv	Sleeping	5.53	68.23	-	4.855
dpoinv	Sleeping	5.51	30.71	-	2.613
dqrm_spgeqr	Sleeping	5.42	300.39	-	35.238
dgesv_nopiv	dgemm	7.68	182.36	2.402	66.395
dpoinv	dgemm	7.54	318.94	2.575	82.171
dgesv_nopiv	dtrsm	3.95	5.74	1.515	1.192
dpoinv	dtrsm	4.33	4.58	1.873	1.739
dqrm_spgeqr	dqrm_do_subtree	3.48	11.02	154.772	1.0912
dqrm_spgeqr	dqrm_gemqrt	7.23	47.86	16.579	3.004
dqrm_spgeqr	dqrm_init_block	3.96	15.62	6.271	1.508
dqrm_spgeqr	dqrm_tpmqrt	7.97	221.91	17.153	26.158
dqrm_spgeqr	dqrm_tpqrt	6.79	107.95	272.587	16.583

TABLE VI
WORKER STATE POWER BREAKDOWN ON CLUSTER `SUROIT`

Program	State	Power (W)	T-stat	Duration (ms)	Occup. (%)
CPU worker					
dgesv_nopiv	Idle	4.42	19.87	-	1.491
dgesv_nopiv	Overhead	5.95	53.54	-	4.425
dpoinv	Overhead	5.66	20.98	-	2.0878
dqrm_spgeqr	Overhead	5.39	76.53	-	2.868
dgesv_nopiv	Sleeping	4.93	76.05	-	5.386
dpoinv	Sleeping	5.13	47.28	-	2.320
dqrm_spgeqr	Sleeping	5.29	225.01	-	10.418
dgesv_nopiv	dgemm	8.25	866.08	2.742	84.194
dpoinv	dgemm	8.36	543.23	2.765	87.856
dgesv_nopiv	dtrsm	5.46	19.80	2.494	2.179
dpoinv	dtrsm	5.42	8.40	2.495	2.412
dpoinv	dsyrk	3.35	11.34	2.635	1.624
dqrm_spgeqr	dqrm_gemqrt	6.75	213.88	35.0677	8.367
dqrm_spgeqr	dqrm_geqrt	5.73	57.52	238.071	2.843
dqrm_spgeqr	dqrm_tpmqrt	7.30	1532.18	33.143	66.438
dqrm_spgeqr	dqrm_tpqrt	4.75	44.99	26.646	2.136
dqrm_spgeqr	dqrm_do_subtree	5.82	35.35	165.078	1.523

TABLE VII
WORKER STATE POWER BREAKDOWN ON CLUSTER `GROUILLE`

Program	State	Power (W)	T-stat	Duration (ms)	Occup. (%)
CPU worker					
dgesv_nopiv	Idle	0.55	1.97	-	5.162
dpoinv	Idle	0.00	0.00	-	8.446
dgesv_nopiv	Overhead	2.60	31.59	-	82.587
dpoinv	Overhead	3.06	21.61	-	84.341
dgesv_nopiv	Sleeping	5.34	21.76	-	4.565
dpoinv	Sleeping	4.39	6.56	-	2.778
dgesv_nopiv	dgemm	2.58	4.53	1605.12	5.0422
dgesv_nopiv	dgetrf_nopiv	12.86	6.41	853.563	1.149
dpoinv	dpotrf	11.54	7.86	657.802	2.166
dpoinv	dtrtri	7.20	8.03	871.794	2.506
GPU worker					
dgesv_nopiv	Idle	19.70	32.98	-	82.551
dpoinv	Idle	30.78	11.82	-	74.852
dgesv_nopiv	dgemm	74.91	6.16	1.593	14.991
dpoinv	dgemm	144.83	11.49	0.823	19.201
dgesv_nopiv	dtrsm	106.69	3.76	3.785	2.458
dpoinv	dtrsm	57.05	5.27	1.236	4.612

TABLE VIII
WORKER STATE POWER BREAKDOWN ON CLUSTER `ENBATA`

Program	State	Power (W)	T-stat	Duration (ms)	Occup. (%)
CPU worker					
dgesv_nopiv	Idle	3.86	4.74	-	15.136
dpoinv	Idle	3.38	3.32	-	14.018
dgesv_nopiv	Overhead	4.32	10.91	-	76.061
dpoinv	Overhead	4.58	13.85	-	80.916
dgesv_nopiv	Sleeping	5.05	3.79	-	7.120
dpoinv	Sleeping	5.17	2.10	-	5.446
GPU worker					
dgesv_nopiv	Idle	23.69	22.63	-	63.110
dpoinv	Idle	23.10	10.71	-	33.354
dgesv_nopiv	dgemm	85.94	34.12	1.204	31.812
dpoinv	dgemm	100.56	89.00	1.987	53.951
dgesv_nopiv	dtrsm	13.97	9.41	2.833	5.079
dpoinv	dtrsm	23.41	18.35	3.586	7.083
dpoinv	dsyrk	14.66	15.16	3.282	4.385
dpoinv	dtrmm	7.22	4.86	1.890	1.227

such as PowerAPI [20] [21] or Scaphandre [22] can be used to model the consumption of processes, containers or virtual machines.

At finer grain, JoularJX [23] can be used to monitor code level power on CPU in Java programs by analyzing threads stacktraces while performing measurements with its PowerJoular tool. AccelWatch [24] can create cycle-level power models for NVIDIA GPUs based on trace-driven simulation environments, hardware performance counters, or a combination of the two. ALEA [25] uses a probabilistic approach to model user defined code blocks at fine-grain on CPU. However, the approach requires numerous profiling runs to be accurate and can add overhead when profiling a large number of threads. PowerPack [26] uses external power sensors, manual code instrumentation and execution traces to model the energy consumptions of code blocks for parallel multicore applications. Tprof [27] uses online sampling of performance counters, task tracing and offline analysis to profile tasks consumption for OpenMP-like task-parallel programs. Tprof is based on the BDDT runtime system [28]. Still, their work focuses on CPU implementations and was only tested on a quad-core Intel processor running independent workloads.

To our knowledge, the work presented in this paper is the only fine-grain energy modeling method on a task-based runtime system capable of generating results for code level tasks executed on both CPU and GPU. It also does not employ any performance monitoring counters, making it portable and compatible with any system capable of running StarPU.

VI. Conclusion

Measuring the energy consumption of fine-grain tasks is tedious because of the granularity of available power meters. We proposed a methodology for modeling the power consumption of tasks at a fine-grain level using traces and coarse grain energy measurements.

Experiments show that this methodology can accurately predict the power consumption of tasks running on a CPU. In contrast, GPUs may provide imprecise energy measurements, which currently limits the accuracy of our model in those cases. In future work, we plan to explore ways to mitigate this limitation, by addressing measurement inaccuracies and improving other aspects of GPU energy modeling.

Our end goal is to provide task energy to StarPU's performance model to create energy-aware scheduling policies. Performance models can be created for every CPU and GPU of a node. Therefore, we could create low-energy CPU/GPU devices by using power cap. This would be captured by the fine-grain energy consumption model, as it is also generated per device. This can open up more scheduling decisions than just deciding wether to schedule a task on CPU or GPU.

Moreover, the fine-grain model can be used to provide insights into the impact of hyperparameters tuning on the power consumption of individual tasks and worker states.

Finally, the method is not StarPU-specific and could apply to other task-based runtimes like OpenMP, SYCL, PARSEC, or StarSS, provided function execution tracing is possible. With additional work, it could also support classical MPI/OpenMP paradigms.

Acknowledgment

Experiments presented in this paper were carried out using the following testbeds:

- The PlaFRIM experimental testbed, supported by Inria, CNRS (LABRI and IMB), Université de Bordeaux, Bordeaux INP and Conseil Régional d'Aquitaine (see https://www.plafrim.fr).
- 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 (see https://www.grid5000.fr).

As part of the "France 2030" initiative, this work has benefited from a national grant managed by the French National Research Agency (Agence Nationale de la Recherche) attributed to the Exa-Soft project of the NumPEx PEPR program, under the reference ANR-22-EXNU-0003.

References

[1] H. W. Meuer, E. Strohmaier, J. J. Dongarra, and H. D. Simon, "TOP500 Supercomputer sites 11/2000," Tech. Rep. LBNL–47461, 843058, Nov. 2000.

[2] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez, "Task-Based Multifrontal QR Solver for GPU-Accelerated Multicore Architectures," in *2015 IEEE 22nd International Conference on High Performance Computing (HiPC)*, Dec. 2015, pp. 54–63.

[3] A. Humphrey, Q. Meng, M. Berzins, and T. Harman, "Radiation modeling using the Uintah heterogeneous CPU/GPU runtime system," in *Proceedings of the 1st Conference of the Extreme Science and Engineering Discovery Environment: Bridging from the eXtreme to the Campus and Beyond*. Chicago Illinois USA: ACM, Jul. 2012, pp. 1–8.

[4] V. Martínez, D. Michéa, F. Dupros, O. Aumage, S. Thibault, H. Aochi, and P. O. Navaux, "Towards Seismic Wave Modeling on Heterogeneous Many-Core Architectures Using Task-Based Runtime System," in *2015 27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Oct. 2015, pp. 1–8.

[5] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "Starpu: a unified platform for task scheduling on heterogeneous multicore architectures," in *15th International Euro-Par Conference*, 2009, pp. 863–874.

[6] M. E. M. Diouri, M. F. Dolz, O. Glück, L. Lefèvre, P. Alonso, S. Catalán, R. Mayo, and E. S. Quintana-Ortí, "Assessing Power Monitoring Approaches for Energy and Power Analysis of Computers," *Sustainable Computing: Informatics and Systems*, vol. 4, no. 2, pp. 68–82, Jun. 2014.

[7] M. Jay, V. Ostapenco, L. Lefevre, D. Trystram, A.-C. Orgerie, and B. Fichel, "An experimental comparison of software-based power meters: Focus on CPU and GPU," in *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. Bangalore, India: IEEE, May 2023, pp. 106–118.

[8] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "RAPL: Memory power estimation and capping," in *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design*. Austin Texas USA: ACM, Aug. 2010, pp. 189–194.

[9] K. N. Khan, M. Hirki, T. Niemi, J. K. Nurminen, and Z. Ou, "RAPL in Action: Experiences in Using RAPL for Power Measurements," *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, vol. 3, no. 2, pp. 1–26, Jun. 2018.

[10] G. Raffin and D. Trystram, "Dissecting the software-based measurement of CPU energy consumption: A comparative analysis," Jul. 2024.

[11] Z. Yang, K. Adamek, and W. Armour, "Accurate and Convenient Energy Measurements for GPUs: A Detailed Study of NVIDIA GPU's Built-In Power Sensor," in *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2024, pp. 1–17.

[12] C. Augonnet, S. Thibault, and R. Namyst, "Automatic Calibration of Performance Models on Heterogeneous Multicore Architectures," in *Euro-Par 2009 – Parallel Processing Workshops*, H.-X. Lin, M. Alexander, M. Forsell, A. Knüpfer, R. Prodan, L. Sousa, and A. Streit, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, vol. 6043, pp. 56–65.

[13] V. Danjean, R. Namyst, and P.-A. Wacrenier, "An efficient multi-level trace toolkit for multi-threaded applications," in *11th International Euro-Par Conference*, 2005, pp. 166–175.

[14] A. Denis, E. Jeannot, P. Swartvagher, and S. Thibault, "Tracing task-based runtime systems: Feedbacks from the StarPU case," *Concurrency and Computation: Practice and Experience*, vol. 36, no. 3, p. e7920, Feb. 2024.

[15] D. Balouek, A. Carpen-Amarie, G. Charrier, F. Desprez, E. Jeannot, E. Jeanvoine, A. Lebre, D. Margery, N. Niclausse, L. Nussbaum, O. Richard, C. Pérez, F. Quesnel, C. Rohr, and L. Sarzyniec, "Adding Virtualization Capabilities to Grid'5000."

[16] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, and S. Tomov, "A hybridization methodology for high-performance linear algebra software for GPUs," in *GPU Computing Gems Jade Edition*. Elsevier, 2012, pp. 473–484.

[17] A. Lisito, M. Faverge, M. Kuhn, F. Pruvost, and P. Ramet, "Scalable and portable LU factorization with partial pivoting on top of runtime systems," in *IPDPS25 - 39th IEEE International Parallel and Distributed Processing Symposium*, Jun. 2025.

[18] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez, "Implementing multifrontal sparse solvers for multicore architectures with sequential task flow runtime systems," *Acm transactions on mathematical software (toms)*, vol. 43, no. 2, pp. 1–22, 2016.

[19] S. P. Kolodziej, M. Aznaveh, M. Bullock, J. David, T. A. Davis, M. Henderson, Y. Hu, and R. Sandstrom, "The suitesparse matrix collection website interface," *Journal of Open Source Software*, vol. 4, no. 35, p. 1244, 2019.

[20] G. Fieni, D. R. Acero, P. Rust, and R. Rouvoy, "PowerAPI: A Python framework for building software-defined power meters," *Journal of Open Source Software*, vol. 9, no. 98, p. 6670, Jun. 2024.

[21] G. Fieni, R. Rouvoy, and L. Seinturier, "SmartWatts: Self-Calibrating Software-Defined Power Meter for Containers," in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, May 2020, pp. 479–488.

[22] B. Petit, "scaphandre," 2023. [Online]. Available: https://github.com/hubblo-org/scaphandre

[23] A. Noureddine, "PowerJoular and JoularJX: Multi-Platform Software Power Monitoring Tools," in *2022 18th International Conference on Intelligent Environments (IE)*, Jun. 2022, pp. 1–4.

[24] V. Kandiah, S. Peverelle, M. Khairy, J. Pan, A. Manjunath, T. G. Rogers, T. M. Aamodt, and N. Hardavellas, "AccelWattch: A Power Modeling Framework for Modern GPUs," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. Virtual Event Greece: ACM, Oct. 2021, pp. 738–753.

[25] L. Mukhanov, P. Petoumenos, Z. Wang, N. Parasyris, D. S. Nikolopoulos, B. R. De Supinski, and H. Leather, "ALEA: A Fine-Grained Energy Profiling Tool," *ACM Transactions on Architecture and Code Optimization*, vol. 14, no. 1, pp. 1–25, Mar. 2017.

[26] R. Ge, X. Feng, S. Song, H.-C. Chang, D. Li, and K. W. Cameron, "PowerPack: Energy Profiling and Analysis of High-Performance Systems and Applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 5, pp. 658–671, May 2010.

[27] I. Manousakis, F. S. Zakkak, P. Pratikakis, and D. S. Nikolopoulos, "TProf: An energy profiler for task-parallel programs," *Sustainable Computing: Informatics and Systems*, vol. 5, pp. 1–13, Mar. 2015.

[28] G. Tzenakis, A. Papatriantafyllou, H. Vandierendonck, P. Pratikakis, and D. S. Nikolopoulos, "BDDT: Block-Level Dynamic Dependence Analysis for Task-Based Parallelism," in *Advanced Parallel Processing Technologies*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, C. Wu, and A. Cohen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, vol. 8299, pp. 17–31.