CHALMERS
UNIVERSITY OF TECHNOLOGY

# JOSS: Joint Exploration of CPU-Memory DVFS and Task Scheduling for Energy Efficiency

Downloaded from: https://research.chalmers.se, 2024-04-27 16:25 UTC

Citation for the original published paper (version of record):
Chen, J., Manivannan, M., Goel, B. et al (2023). JOSS: Joint Exploration of CPU-Memory DVFS and Task Scheduling for Energy Efficiency. 52nd International Conference on Parallel Processing (ICPP 2023): 828-838. http://dx.doi.org/10.1145/3605573.3605586

N.B. When citing this work, cite the original published paper.

research.chalmers.se offers the possibility of retrieving research publications produced at Chalmers University of Technology.
It covers all kind of research output: articles, dissertations, conference papers, reports etc. since 2004.
research.chalmers.se is administrated and maintained by Chalmers Library

(article starts on next page)

# JOSS: Joint Exploration of CPU-Memory DVFS and Task Scheduling for Energy Efficiency

Jing Chen
Chalmers University of Technology
Gothenburg, Sweden
chjing@chalmers.se

Madhavan Manivannan
Chalmers University of Technology
Gothenburg, Sweden
madhavan@chalmers.se

Bhavishya Goel
Chalmers University of Technology
Gothenburg, Sweden
goelb@chalmers.se

Miquel Pericàs
Chalmers University of Technology
Gothenburg, Sweden
miquelp@chalmers.se

## ABSTRACT

Energy-efficient execution of task-based parallel applications is crucial as tasking is a widely supported feature in many parallel programming libraries and runtimes. Currently, state-of-the-art proposals primarily rely on leveraging core asymmetry and CPU DVFS. Additionally, these proposals mostly use heuristics and lack the ability to explore the trade-offs between energy usage and performance. However, our findings demonstrate that focusing solely on CPU energy consumption for energy-efficient scheduling while neglecting memory energy consumption leaves room for further energy savings. We propose JOSS, a runtime scheduling framework that leverages both CPU DVFS and memory DVFS in conjunction with core asymmetry and task characteristics to enable energy-efficient execution of task-based applications. JOSS also enables the exploration of energy and performance trade-offs by supporting user-defined performance constraints. JOSS uses a set of models to predict task execution time, CPU and memory power consumption, and then selects the configuration for the tunable knobs to achieve the desired energy performance trade-off. Our evaluation shows that JOSS achieves 21.2% energy reduction, on average, compared to the state-of-the-art. Moreover, we demonstrate that even in the absence of a memory DVFS knob, taking energy consumption of both CPU and memory into account achieves better energy savings compared to only accounting for CPU energy. Furthermore, JOSS is able to adapt scheduling to reduce energy consumption while satisfying the desired performance constraints.

## CCS CONCEPTS

• **Computing methodologies → Parallel algorithms**.

## KEYWORDS

energy efficiency, task scheduling, performance modeling, power modeling, DVFS

This work is licensed under a Creative Commons Attribution International 4.0 License.

*ICPP 2023, August 07–10, 2023, Salt Lake City, UT, USA*
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0843-5/23/08.
https://doi.org/10.1145/3605573.3605586

**ACM Reference Format:**
Jing Chen, Madhavan Manivannan, Bhavishya Goel, and Miquel Pericàs. 2023. JOSS: Joint Exploration of CPU-Memory DVFS and Task Scheduling for Energy Efficiency. In *52nd International Conference on Parallel Processing (ICPP 2023), August 07–10, 2023, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3605573.3605586

## 1 INTRODUCTION

Energy efficiency has emerged as a crucial design constraint in various parallel computing systems ranging from battery-powered mobile devices to high performance computers. Modern chip multi - processors (CMPs) incorporate a variety of hardware features to improve energy efficiency. The integration of multiple core types on a single die, referred to as static asymmetry, in big.LITTLE architectures [2, 3, 6, 8, 39], offers the possibility to execute applications on different cores with varied performance and power characteristics. Modern CMPs also support dynamic voltage and frequency scaling (DVFS), referred to as dynamic asymmetry, which manages system power consumption and enables exploration of performance and power consumption trade-offs when executing applications. In addition, cores of the same type are typically grouped into clusters to reduce the design cost and complexity associated with enabling per-core DVFS [27]. In such core-clustered designs, the cores in a single cluster operate at the same DVFS setting [2, 3, 39].

There has been extensive research on leveraging static and/or dynamic asymmetry, as knobs to reduce CPU energy consumption since it is typically the largest contributor to the total energy consumption in a system [10, 14, 21, 26, 33, 37, 41, 44]. In order to meet the memory demand for emerging many-core architectures, main memory bandwidth and capacities have also been steadily increasing (albeit at a slower rate)[5]. This has led to the memory system also becoming a major contributor to the total energy consumption [32]. Several works have highlighted the importance and the benefit of leveraging memory DVFS, alongside static and/or dynamic asymmetry offered by the CPUs, since it opens up many opportunities for establishing trade-offs between performance and energy consumption [16–18, 43]. These proposals however focus on the potential of leveraging these knobs in the context of single-threaded and multi-programmed workloads (comprising several single-threaded applications).

Task-based parallel programming models are supported by many production parallel programming libraries and runtimes [1, 13,

22, 34], such as OpenMP, since it eases the process of expressing parallelism inherent in applications. In this model, an application is expressed as a directed acyclic graph (DAG), where the tasks (vertex) and their dependencies (edges) are generated dynamically during execution. Tasks can be of different types and exhibit diverse attributes (e.g. OPs/byte ratio) [20]. Additionally, moldable execution enabled by intra-task parallelism (i.e. using multiple cores for running a single task) has been shown to improve performance and lower the impact of system idle energy [11, 12].

Energy-efficient execution of a task DAG relies on runtime schedulers to map each task in the DAG to hardware resources (i.e. choosing the appropriate core type and number of cores for the task) and throttle the available DVFS knobs simultaneously. Several recent works have targeted energy-efficient runtime scheduling techniques for task-based parallel applications, which can be broadly grouped into three categories [10–12, 14, 26, 28, 33, 35, 37, 44]. The first category primarily exploits task characteristics (e.g. task OPs/byte, task size, task criticality) in conjunction with static asymmetry without leveraging DVFS [11, 28]. The second category employs task characteristics jointly with dynamic asymmetry, while being restricted to symmetric architectures [33, 37, 38]. The third category employs task characteristics together with static asymmetry and a subset of dynamic asymmetry knobs (CPU DVFS) for scheduling [10, 12, 14, 26, 33, 44].

Unfortunately, existing works do not leverage static and dynamic asymmetry, especially memory DVFS, in conjunction with task characteristics for scheduling. Our analysis in Section 2 shows that concurrently utilizing all the knobs, i.e. core type ($\mathbf{T_C}$), number of cores ($\mathbf{N_C}$), core frequency ($\boldsymbol{f_C}$) and memory frequency ($\boldsymbol{f_M}$), provides greater opportunities for reducing energy consumption over using a subset of the knobs. We also show that, even in the absence of a memory DVFS knob, considering CPU energy consumption alone, as done in prior works, for scheduling tasks without taking memory energy consumption into account leaves a lot of scope for improvement. Furthermore, prior art is designed for a specific target and mostly uses a set of heuristics without having the ability to explore trade-offs between performance and energy consumption.

The goal is to develop a runtime scheduling framework that leverages the aforementioned knobs and provides the ability to target various trade-offs between performance and energy consumption. To achieve this goal, several challenges need to be addressed. Firstly, the effects of tuning available knobs, individually and in conjunction, on performance and energy consumption needs to be understood. Secondly, the interplay between task scheduling decisions and the exploration of various trade-offs between energy and performance needs to be investigated. Thirdly, there is a need to accommodate applications/tasks with diverse characteristics while also ensuring low runtime overhead.

We propose **JOSS** (**JO**int **S**cheduling and **S**caling), a runtime scheduling framework that can target various energy performance trade-offs through leveraging the aforementioned knobs. Overall, JOSS achieves the goals set out by optimizing the execution of each task in the DAG. For instance, JOSS reduces the total energy consumption of a task-based application by running each task with the lowest energy possible. The operation of the framework can be summarized as follows. First, JOSS utilizes multivariate polynomial regression based models for predicting the execution time and power consumption of CPU and memory subsystem when running a task on different configurations spanning the four knobs <$T_C$, $N_C$, $f_C$, $f_M$>. Second, the JOSS task scheduler combines the model predictions together with instantaneous task concurrency during runtime to estimate the energy consumption of a task and make the scheduling decisions accordingly for various trade-off goals. To prune the large search space formed by the four knobs and reduce the overhead during runtime, JOSS uses a steepest descent approach to determine the configuration that satisfies the desired trade-off in a few steps.

We evaluate JOSS under two scenarios targeting different energy performance trade-offs: (1) reducing the total energy consumption and (2) reducing energy with user-specified performance speedup with respect to (1). The results for scenario (1) indicate that JOSS achieves 21.2% energy reduction on average compared to the state-of-the-art [12] on an NVIDIA Jetson TX2 platform. Even in the absence of a memory DVFS knob, JOSS can still provide an additional 5.2% energy reduction than the state-of-the-art. For scenario (2), we show that JOSS is able to adapt scheduling to reduce energy consumption while satisfying the desired performance constraints.

In summary, this paper makes the following contributions:

- We demonstrate that (1) leveraging static asymmetry and dynamic asymmetry, i.e. core type, number of cores, core frequency and memory frequency, together with task characteristics, enables significant reduction in energy consumption; (2) even in the absence of a memory DVFS knob, taking total energy consumption (including CPU and memory) into account for configuration selection results in lower energy consumption compared to only considering CPU energy.
- We propose the JOSS runtime scheduling framework for task-based parallel applications on multicore architectures, which provides the ability to explore various energy performance trade-offs.
- We build a set of models using multivariate polynomial regression capable of accurately predicting the execution time, average CPU power, and average memory power of each task when tuning the four available knobs, individually and simultaneously.

## 2 MOTIVATION

We motivate JOSS by demonstrating (1) the importance of taking memory energy consumption into account and its impact on selecting configurations for the different knobs; (2) the impact of concurrently leveraging the four knobs <$T_C, N_C, f_C, f_M$> on reducing energy consumption; (3) the potential for energy performance trade-off exploration (e.g. reducing energy while satisfying user-specified performance constraints). We use NVIDIA Jetson TX2 as the experimental platform since it features static asymmetry (i.e. a dual-core high-performance Denver CPU and a relatively low-performance quad-core A57 CPU) and dynamic asymmetry (CPU and memory DVFS knobs that can be tuned during execution).

For this experiment, we use two different benchmarks - Matrix Multiplication (MM, compute-intensive) and Memory Copy (MC, memory-intensive), configured with a DAG parallelism (*dop*) of one, where *dop* represents the potential task concurrency in the task DAG obtained by dividing the total number of tasks by

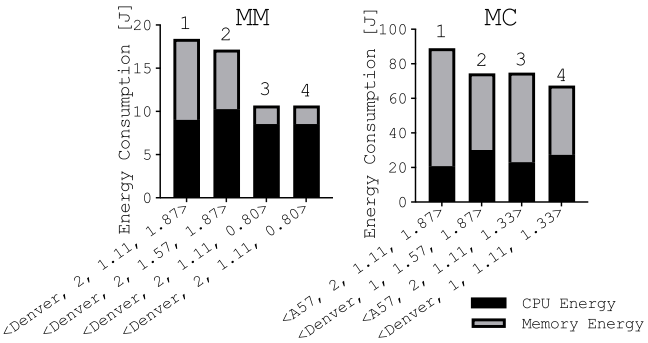



**Figure 1: Total energy consumption under the four scenarios.**

the length of the longest path. We run both benchmarks with all possible combinations of the knobs, measure CPU and memory power consumption, and execution time and use these results in our analysis. In these benchmarks, tasks represent the numerical kernels, which are typically invoked numerous times wherein the routine(s) executed by different tasks (invocations of the same kernel) are identical. Additional details about the platform and the benchmarks are provided in Section 6.

## 2.1 Importance of Including Memory Energy

To motivate the importance of including memory energy for CPU DVFS, we compare the total energy consumption (CPU+memory) under different scenarios as shown in Figure 1. Scenario 1 ($1^{st}$ bar) represents the state-of-the-art [12], where the configuration that consumes the least CPU energy is identified while tuning the three knobs <$T_C$, $N_C$, $f_C$> and fixing $f_M$ at the highest frequency 1.87GHz. In scenario 2 ($2^{nd}$ bar), we identify the configuration that consumes the least total energy while tuning <$T_C$, $N_C$, $f_C$> and fixing $f_M$ at 1.87GHz. Note that for both scenarios, we assume that memory DVFS knob is unavailable and that the memory always operates at the highest frequency. From Figure 1 we can observe that the configuration identified (see x-axis labels) when only considering CPU energy consumption is sub-optimal for both MM and MC. However, taking memory energy into account leads to a different configuration that further reduces the total energy, while still being restricted to three knobs. For instance, in the case of MC the best configuration changes from <A57, 2, 1.11> to <Denver, 1, 1.57> and leads to 16% energy reduction.

## 2.2 Leveraging Knobs in Conjunction

To motivate the importance of leveraging the four knobs in conjunction, we compare the total energy consumption under scenarios 3

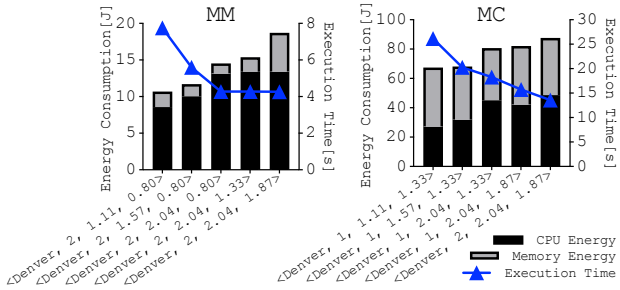



**Figure 2: Energy performance trade-offs exploration.**

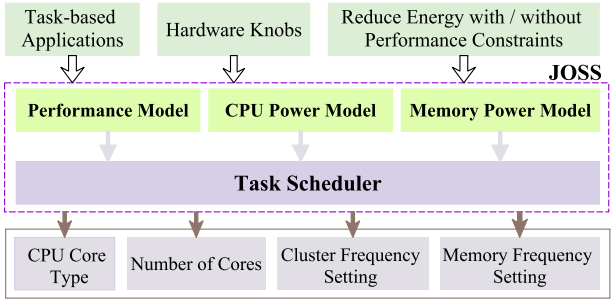



**Figure 3: High-level overview of JOSS.**

and 4 in Figure 1. Here we assume the memory DVFS knob to be tunable. Scenario 3 ($3^{rd}$ bar) enhances the state-of-the-art [12] with support for orthogonal frequency scaling where CPU and memory frequencies are throttled independently. This involves determining the best configuration of <$T_C$, $N_C$, $f_C$>, as in scenario 1 and then evaluating the total energy consumption when tuning $f_M$ while the other three knobs remain fixed. Scenario 4 ($4^{th}$ bar) represents leveraging the four knobs in conjunction (the approach adopted by JOSS) where energy consumption for the entire configuration space is searched to determine the configuration that consumes the least total energy. The results show that in the case of MC scenario 4 ends up selecting a different configuration and leads to 10% energy savings compared to scenario 3. For MM scenario 4 does not provide any additional benefit since there is no change in the configuration.

In summary, we conclude that *(i) even in the absence of a memory DVFS knob, only considering CPU energy results in sub-optimal configuration selection thereby emphasizing the importance of also taking memory energy consumption into account; (ii) in comparison to the orthogonal CPU DVFS and memory DVFS throttling, leveraging the four knobs in conjunction can lead to more energy savings.*

## 2.3 Exploring Energy Performance Trade-offs

It is crucial that the scheduler aims to reduce energy consumption while still maintaining a good level of performance. Figure 2 shows the potential trade-offs between energy consumption and performance when tuning the available knobs. Note that the first bar represents the configuration that consumes the least total energy, which we will use as the baseline for the rest of this discussion. We can observe that tuning core frequency from 1.11 to 1.57 for MM and MC provides 1.4× and 1.3× performance speedup while increasing energy consumption by 10% and 1%, respectively. The maximum speedup for MM is 1.8× which comes at the cost of 36% increase in energy consumption. MC can achieve a maximum speedup of 1.9× at the cost of 30% increase in energy consumption. *Building a runtime scheduling framework that can flexibly explore energy performance trade-offs will enable the user to customize the scheduler to their specific requirements.*

# 3 JOSS RUNTIME FRAMEWORK OVERVIEW

Figure 3 provides a high-level overview of the JOSS runtime framework. The framework takes the application, details about the architectural knobs (number of clusters, number of cores in each cluster, supported frequencies) and performance constraints (that can be specified by either user or system software) as inputs.

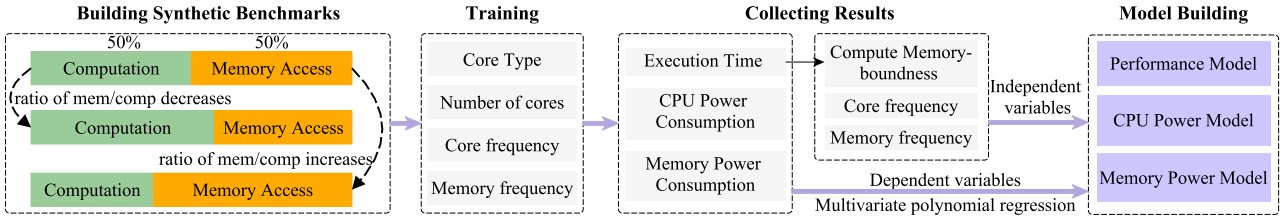



**Figure 4: Overview of the model building process.**

To enable energy performance trade-off exploration, JOSS leverages four knobs <$T_C, N_C, f_C, f_M$> whose effects on energy and performance need to be understood when tuned individually and in combination. We therefore develop performance and power models to understand the effects and guide configuration selection. JOSS specifically comprises a performance model, a CPU power model and a memory power model to provide the scheduler with the prediction of task execution time and power consumed in CPU and memory domains when executing tasks with different configurations. Energy and performance estimates are used by the task scheduler to guide configuration selection and achieve the desired trade-off goals. The scheduler maps tasks to selected CPU cores and sends the frequency throttling requests to the CPU DVFS controller and the memory DVFS controller. JOSS targets total energy reduction if a performance constraint is not specified.

In Section 4, we first introduce the development of the three prediction models in JOSS. Section 5 will then describe the task scheduler and how these models are utilized to enable energy performance trade-off exploration.

## 4 MODELS

**Challenges:** JOSS utilizes models to predict performance and power consumption of tasks and understand the energy and performance effects from tuning the four knobs <$T_C, N_C, f_C, f_M$>. Two challenges need to be addressed in this context. First, creating a model to predict the effect of four tunable knobs for varying task characteristics is complicated and expensive, especially as the number of configurations increases. To tackle this issue, JOSS combines models with runtime samples in a hybrid approach. We sample the two knobs <$T_C, N_C$> during runtime (details in Section 5.1) and use models to predict the impact of DVFS on performance and power consumption of a task from the other two knobs <$f_C, f_M$>.

Second, prediction models for execution time and power consumption proposed in prior works rely on the availability of a number of Performance Monitoring Counters (PMCs) [23, 29, 30, 35, 36, 40, 45] and/or model the impact of a subset of the four knobs [11, 12]. However, the availability of specific PMCs on different architectures limits the adoption of existing models [15, 31]. For instance, Goel et al. [23] use `CYCLE:ACTIVITY:STALLS_L2_PENDING` on Haswell to estimate the memory intensity of an application, which is unavailable on later Intel microarchitectures. Even the TX2 platform used in our evaluation does not provide PMCs related to stall cycles. To address this, the models used in JOSS do not rely on any PMCs thereby improving portability across architectures.

**Overview:** To enable performance and power predictions, we first characterize the platform by running a set of synthetic benchmarks. Since the impact of DVFS on a task depends on its use of computational and memory resources, we generate a set of synthetic benchmarks with varying levels of utilization of the two components. We execute the synthetic benchmarks while tuning all four knobs during the training stage and collect the corresponding execution time and average power values. Subsequently, we build the performance and power prediction models using multivariate polynomial regression (MPR). This technique is commonly used to model the implicit nonlinear relationships between variables [30, 40, 45]. Figure 4 depicts an overview of the model building process.

### 4.1 Synthetic Benchmarking and Profiling

The basic structure of the synthetic benchmark includes a computation loop and a memory access loop. Through controlling the number of iterations in each loop, we can generate different ratios of computations and memory access. In this work, by keeping the total execution time of synthetic benchmarks constant, we start from 50% of computation and 50% of memory access and then increase or decrease the execution time of corresponding loops by 2.5% as shown in Figure 4. In total, we generate 41 synthetic benchmarks with different ratios between computation and memory access. We profile the platform via executing these synthetic benchmarks at all possible configurations for the four knobs and measure the execution time, CPU and memory power consumption.

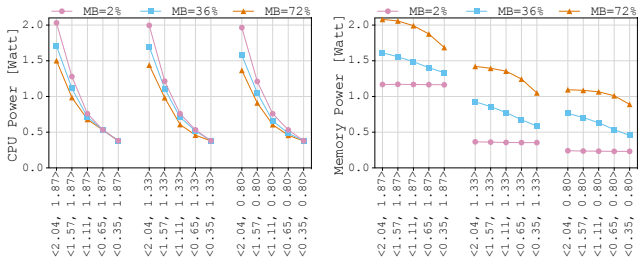
### 4.2 Performance Model

The performance model aims to predict the execution time of a task under joint CPU and memory frequency scaling. In the model, the total execution time is estimated as the sum of computation time and stall time due to memory latency: $\text{Time} = \text{Time}_{\text{comp}} + \text{Time}_{\text{stall}}$. We use *memory-boundness* (MB) to quantify the fraction of time CPU is stalled due to memory latency [12, 23]. When scaling core frequency ($f_C \rightarrow f'_C$), the computation time will scale linearly. So the computation time $\text{Time}'_{\text{comp}}$ at frequency $f'_C$ can be calculated as:

$$Time'_{comp} = Time \times (1 - MB) \times \frac{f_C}{f'_C} \quad (1)$$

$\text{Time}'_{\text{stall}}$ is dependent on core and memory frequency scaling in addition to task characteristics (MB). Memory frequency scaling directly influences the latency of memory access. Core frequency scaling, however, has an (indirect) impact on how often a core issues memory access requests. We utilize the statistics from running synthetic benchmarks presented in Section 4.1 to build the performance model using MPR as shown below:

$$Time'_{stall} = Time \times (\sum_{i=0}^{2} \beta_i x_i + \sum_{i=0}^{2} \beta_{ii} x_i^2 + \sum_{i=0}^{1} \sum_{k=i+1}^{2} \beta_{ik} x_i x_k + \varepsilon) \quad (2)$$




(a) CPU Power (b) Memory Power

**Figure 5: CPU and memory power consumption from profiling synthetic benchmarks on A57 using two cores. The labels in the x-axis are of the format <$f_C, f_M$>.**

where $x_i = \{MB, \frac{f_C}{f'_C}, \frac{f_M}{f'_M}\}$, 0≤i≤2. Here, $\beta_i, \beta_{ii}, \beta_{ik}$ are the coefficients of the linear component, the quadratic component and the interaction component, respectively, and $\varepsilon$ is the intercept. The execution time at <$f'_C, f'_M$> is $\text{Time}' = \text{Time}'_{\text{comp}} + \text{Time}'_{\text{stall}}$.

Estimating task execution time at different <$f'_C, f'_M$> settings during runtime requires knowing the MB value and a sampled execution time at a reference <$f_C, f_M$> setting. To obtain the MB value without using PMCs, we sample the execution times at two different core frequency settings during runtime (i.e. sample $Time$ and $Time'$ at $f_C$ and $f'_C$) under a fixed memory frequency. We then obtain MB as follows:

$$MB = (\frac{Time'}{Time} - \frac{f_C}{f'_C})/(1 - \frac{f_C}{f'_C}) \tag{3}$$

## 4.3 Power Models

The goal of the CPU and memory power models, used in JOSS, is to predict the impact of joint core and memory frequency scaling on CPU power and memory power consumed by a task, respectively. We leverage the statistics collected from profiling synthetic benchmarks for building the CPU and memory power models, akin to the performance model, as shown in Figure 4.

*4.3.1 CPU Power Model.* The results from running synthetic benchmarks indicate that CPU power consumption is mainly dependent on core frequency and task characteristics (MB). For instance, Figure 5a indicates that CPU power consumption, when running three synthetic benchmarks (represented by different levels of MB) with various <$f_C, f_M$> settings (represented in x-axis) on Jetson TX2, shows negligible effects from memory frequency scaling. Consequently, we build the CPU power model as shown in Equation 4. We do not use voltage explicitly since it is strongly correlated with frequency in our evaluation platform and this enables us to reduce collinearity on the regression model.

$$Power_C = \sum_{0}^{1} \beta_i x_i + \sum_{0}^{1} \beta_{ii} x_i^2 + \beta_{01} x_0 x_1 + \varepsilon \tag{4}$$

where $x_i = \{MB, f_C\}$, 0≤i≤1.

*4.3.2 Memory Power Model.* Memory power is dependent on all three influential factors, i.e. core frequency scaling, memory frequency scaling and task characteristics (MB). Figure 5b shows the impact of $f_C, f_M$ and MB on memory power on Jetson TX2. Consequently, we build the memory power model as shown in Equation 5.

$$Power_M = \sum_{0}^{2} \beta_i x_i + \sum_{0}^{2} \beta_{ii} x_i^2 + \sum_{i=0}^{1} \sum_{k=i+1}^{2} \beta_{ik} x_i x_k + \varepsilon \tag{5}$$

where $x_i = \{MB, f_C, f_M\}$, 0≤i≤2. Note that the coefficients $\beta$ and $\varepsilon$ generated for the three models in equations 2, 4 and 5 are distinct values.

*4.3.3 Idle Power.* The total predicted power consumption for a task is the sum of dynamic power and idle power. Dynamic power consumed by a task is estimated using the models discussed previously. We measure idle CPU power and idle memory power during benchmarking when cores are switched on but are not actively executing computations and use the measured values as predictions. We incorporate idle power characterization at different frequencies (voltages) in our models but do not consider temperature due to the small observed variations (<10 degrees) in operating temperature. Unlike dynamic power which is specific to each task, idle power is shared across all concurrently running tasks. We obtain information about the number of concurrently running tasks from the runtime (details in Section 5.3) and use that to attribute idle CPU and memory power proportionally among concurrently running tasks.

**Modeling for different core type and number of cores:** The aforementioned models predict the impact of tuning two knobs <$f_C, f_M$> on performance, CPU and memory dynamic power consumption. However, when tasks execute on different core types and with different number of cores <$T_C, N_C$>, MB values change due to the underlying core performance and workloads characteristics. Consequently, the coefficients in the models for different <$T_C, N_C$> are distinct and we determine them via running the synthetic benchmarks at corresponding <$T_C, N_C$>.

Our evaluation in Section 7 shows that the proposed models are accurate for determining the configuration for specified trade-off goals with low overhead during runtime. We also evaluated the effectiveness of enhancing the performance and power models with higher degree coefficients but observed that it resulted in model overfitting and increased computation overheads without further improvement in prediction accuracy. Note that the profiling and the model building steps just need to be done once for a specific platform (e.g. at install-time or boot-time), and do not impact the execution time of applications.

# 5 JOSS TASK SCHEDULER

JOSS task scheduler utilizes model predictions to take scheduling decisions and explore energy performance trade-offs. Figure 6 provides an overview of the scheduler's timeline. JOSS first samples task execution times to obtain MB values required for performance and power predictions. Details regarding the sampling process and model invocation are presented in Section 5.1. Next, in Section 5.2, we discuss how JOSS employs the predictions and identifies the best configuration for the four knobs to satisfy the energy performance trade-off goal. In Section 5.3, we discuss the task scheduling process and frequency coordination approach applied to shared resources (i.e. core-clusters and memory) where the frequency throttling by concurrently running tasks could potentially lead to interference.

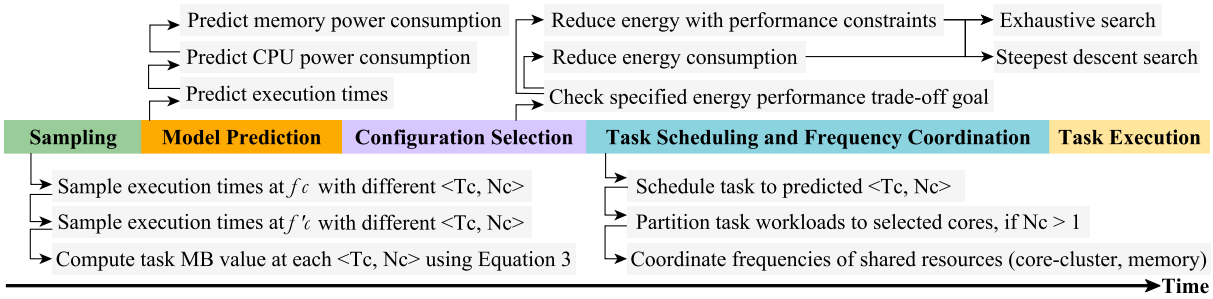



**Figure 6: JOSS task scheduler timeline.**

## 5.1 Runtime Sampling and Model Prediction

As discussed earlier, in Equation 3, power and performance models rely on MB values which are computed by sampling task execution times at two different core frequency settings $f_C$ and $f'_C$. Furthermore, it is important to sample task execution with different $<T_C, N_C>$ configurations since MB values vary with different core types and number of cores used to execute a task. Consequently, JOSS samples task execution times, for different kernels, when running with different $<T_C, N_C>$ configurations at both $f_C$ and $f'_C$ and then uses the MB values for predicting at different $<f_C, f_M>$.

JOSS performs online sampling (at the beginning of execution) for each kernel. It leverages the observation that a typical kernel is invoked several times during application execution and that it is sufficient to sample a small fraction to estimate MB without introducing prohibitive overheads from online sampling. The task scheduler initializes a separate performance look-up table, a CPU power look-up table and a memory power look-up table for storing the measured values and the predictions for each kernel.

In a nutshell, the runtime sampling and model prediction phase operates as follows: Firstly, JOSS samples the execution times of all kernels at different $<T_C, N_C>$ at $f_C$. Once all kernels are sampled at $f_C$, JOSS then switches the cluster frequency to $f'_C$ and repeats the process. Note that the frequency transitions on different clusters are asynchronous, i.e. sampling on one cluster can immediately transition from $f_C$ to $f'_C$ without waiting for the sampling completion on the other clusters. After sampling at $f'_C$, JOSS immediately computes the MB values at different $<T_C, N_C>$ configurations. It then uses it along with performance and power models to populate the per-kernel look-up tables with predicted values.

## 5.2 Configuration Selection for Different Energy Performance Trade-off Goals

Once model predictions are complete, the scheduler transitions to configuration selection for each kernel. JOSS achieves the desired energy performance trade-off goal by optimizing the execution of each individual task. For instance, JOSS reduces the total energy consumption by running each task with the lowest energy possible. JOSS utilizes predictions to determine the configuration that satisfies the desired energy performance trade-off for each kernel. The approach for selecting the best configuration for each kernel is detailed later in this section. Successive invocations of the same kernel use the identified configuration without having to incur the overhead of configuration selection repeatedly. In this section, we investigate two different scenarios: reducing total energy consumption with and without performance constraints.

*5.2.1 Reducing Total Energy Consumption.* A simple approach for configuration selection is to exhaustively loop through all possible configurations and compare the estimated energy values to determine the configuration that consumes the least energy. However, as core counts and the number of available DVFS settings scale, such an approach can result in significant computation overheads during runtime. To address this, we introduce a heuristic search algorithm based on the steepest descent method that can prune the large search space and identify the configuration with the least energy consumption with reduced overhead.

Figure 7 illustrates the pruning process. First, the algorithm computes the energy consumption of four corner configurations (representing combinations of the highest and the lowest CPU and memory frequency) for each $<T_C, N_C>$. Second, the algorithm compares the four corner values across different $<T_C, N_C>$ to identify the $<T_C, N_C>$ with the most number of lowest corner values. This step confines the search space to a specific $<T_C, N_C>$ table. In the third step, the algorithm searches for the most energy-efficient joint DVFS setting $<f_C, f_M>$ from this table. This is accomplished by starting from the corner that has the least energy consumption, comparing the energy consumption of that configuration against all its immediate neighbours and repeating this immediate neighbour search process iteratively until it converges at a configuration with the least energy consumption. The algorithm terminates once it detects that the energy value of the selected configuration is the lowest among all its immediate neighbors. We compare the overheads and the effectiveness of the two approaches in Section 7.4.

*5.2.2 Reducing Total Energy Consumption under Performance Constraints.* JOSS supports performance constraints specified as speedups relative to the execution time for energy minimization. To meet

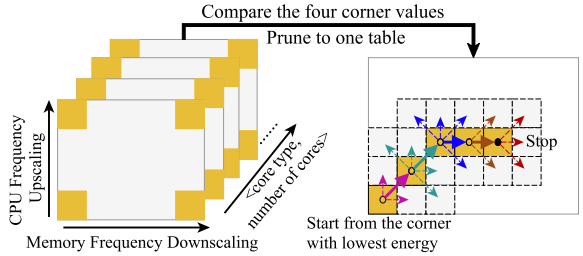



**Figure 7: The steepest descent approach used in JOSS for pruning search space and configuration selection.**

these constraints, JOSS translates the performance speedup for the whole application to individual tasks assuming that speeding up individual tasks will lead to an equivalent speedup of the entire application.

JOSS applies the steepest descent search under performance constraints as follows: It starts with the most energy efficient configuration and evaluates the performance of its three nearest data points within the same <$T_C$, $N_C$> table, which involves increasing $f_C$, $f_M$, or both. If one or more neighboring points' execution times meet the constraint, JOSS chooses the configuration with the least energy consumption. Otherwise, it incrementally raises the frequencies and repeats. If no frequency combinations within that <$T_C$, $N_C$> satisfy the constraint, JOSS first checks performance tables with more cores ($N_C$), then faster clusters ($T_C$), and repeats the search. If no configuration meets the performance constraint, the fastest configuration is selected.

## 5.3 Task Scheduling and Frequency Coordination

Once the task scheduler determines the configuration for a task whose input dependencies are satisfied and can be scheduled for execution (i.e. ready task), it places the task in a work queue of a randomly selected core of the selected core type. Note that the scheduler allows the task to be stolen by other cores of the same type, to maintain load balancing while also ensuring that the task runs on the most suitable core type. Stealing across asymmetric clusters is disabled to prevent task execution on an energy inefficient cluster and frequency setting. For instance, MM results (*dop* = 4 and 16) show that work stealing across asymmetric clusters improves performance by 41% but incurs a 87% energy increase.

Moldable task execution ($N_C > 1$) is performed on multiple cores by dynamically partitioning the task workloads among cores of the same type. Once a core finishes executing a task partition, it can continue fetching other tasks from its own work queue without waiting for partitions to finish on other cores. The core that finishes executing the partition last, declares the completion of the task and wakes up the dependent tasks.

JOSS tracks the status of each core (i.e. working or sleeping) to estimate instantaneous task concurrency. This is required to attribute the shared idle power among concurrently executing tasks. Furthermore, frequency throttling of the shared resources such as core-cluster and memory subsystem impacts concurrently executing tasks. Diverse frequency requirements for concurrent tasks can result in DVFS interference on shared resources and trigger DVFS serialization thereby introducing performance bottlenecks. JOSS therefore adopts a simple averaging heuristic to balance the demands among the concurrent tasks when it detects that there is concurrency. JOSS averages the pre-determined frequency setting for the task with the current frequency setting of the shared resources. We evaluated other heuristics such as *min*, *max*, weighted average, etc. and found arithmetic mean to perform the best.

**Fine-grained tasks:** DVFS throttling overhead for fine-grained tasks where the execution times can be as small as a few microseconds is non-negligible. Therefore, JOSS adopts the task coarsening algorithm proposed in the state-of-the-art [12], which first determines the <$T_C$, $N_C$> without any frequency throttling and then attempts to search for more tasks of the same type from the work queues of the selected core type in a round-robin manner. Once a sufficient number of fine-grained tasks of the same type is found, JOSS searches for the best joint <$f_C$, $f_M$> setting that satisfies the trade-off under the determined <$T_C$, $N_C$>.

# 6 EXPERIMENTAL METHODOLOGY

In this section, we provide details about the experimental platform, benchmarks and the state-of-the-art schedulers that we compare against.

## 6.1 Experimental Platform

We use the NVIDIA Jetson TX2 development board in our evaluation [3]. It is an asymmetric platform that features two CPU clusters: Denver and A57. The Denver cluster comprises a high-performance dual-core NVIDIA Denver CPU, while the A57 cluster comprises a comparatively lower performance quad-core ARM CPU. Both clusters support the same range of operating core frequencies. The two clusters can be operated at different frequencies but all the cores in the same cluster must operate at the same frequency. The choice of using the NVIDIA Jetson TX2 is also motivated by its support for EMC frequency scaling for the memory controller (EMC/MC) and DRAM (LPDDR4). Existing systems with support for memory DVFS typically support frequency scaling in the memory controller, the DDRIO and the DRAM device while only supporting voltage scaling in the memory controller due to design challenges associated with operating the DRAM array at multiple voltages [25]. The integrated `INA3221` power sensor is used to sample the power consumption of CPU and memory subsystem. Power samples obtained every 5 milliseconds are used to compute CPU and memory energy consumption, which is then accumulated throughout the duration of application execution. The Linux governor is set as `userspace` to enable CPU frequency scaling. The CPU and memory frequency are set at the highest, i.e. 2.04GHz for both clusters and 1.87GHz for memory, before executing a benchmark. The Linux kernel version is 4.9.253-tegra and the compiler version is g++ 7.5.0. We repeat each experiment 10 times and report the arithmetic average.

## 6.2 Evaluated Benchmarks

We evaluate JOSS using ten benchmarks from the Edge and HPC domains. Table 1 provides additional details. These benchmarks comprise a different number of kernels (i.e. task types) and exploit parallelism by invoking multiple instances of them. For our model to work, kernels need to have identical features across repetitions. Therefore, kernels that are invoked with different input sizes are treated as distinct task types by the model.

## 6.3 Evaluated Schedulers

We evaluate the effectiveness of JOSS by comparing it to multiple state-of-the-art task-based schedulers. Both JOSS and the evaluated schedulers below are implemented on top of XiTAO [4].

(1) *GRWS* (Greedy Random Work Stealing) is a widely used baseline scheduler for task-based applications [13, 22, 34], which attempts to keep idle cores busy through task stealing. GRWS does not leverage DVFS knobs and each task only runs on a single core.

**Table 1: Evaluated Benchmarks**

| Benchmark | abbr. | Description | Input Size | Num. of Tasks |
|---|---|---|---|---|
| **Heat Diffusion** [4] | HD | Heat diffusion on a 2D grid using the iterative Jacobi stencil, which includes two kernels: Copy and Jacobi. We evaluate three problem sizes of different resolutions. | 2048(small),8192(big), 16384(huge) | 320032, 32032, 16032 |
| **Dot Product** [4] | DP | Computing the sum of the products of two equal-length vectors, vectors are partitioned into blocks and computation of each block is marked as a single task, 100 iterations. | VectorSize 6400000, BlockSize 32000 | 20200 |
| **Fibonacci** [19] | FB | Fibonacci numbers computed using recursion method. | Term 55, GrainSize 34 | 57314 |
| **Darknet-VGG-16 CNN** [42] | VG | A 16-layered deep neural network that is typical of mobile and edge devices and implemented as a fork-join DAG, iteratively executed for 10 iterations. | 768×576 RGB image, blocksize 64 | 5090 |
| **Biomarker Infection** [7] | BI | A medical usecase for differentiating periprosthetic hip infection and aseptic hip prosthesis loosening. It computes the possible Biomarkers combinations to predict symptoms. | Sample Size 2 | 6217 |
| **Alya** [24] | AL | Alya is a high performance computational mechanics code to solve complex partial differential equations, and the parallelization strategy is based on mesh partitioning. | 200K CSR non-zeros | 47840 |
| **Sparse LU Factorization** [19] | SLU | Sparse matrix decomposition into the product of a lower and upper triangular matrix. It includes four kernels: LU0, FWD, BDIV and BMOD. | 64 blocks, BlockSize 512 | 11472 |
| **Matrix Multiplication** [4] | MM | A synthetic benchmark where each task computes A×B=C, A and B are partitioned in N×N tiles, N = input size. *dop* is configurable. | 256×256, 512×512 | 10000, 2000 |
| **Matrix Copy** [4] | MC | A synthetic benchmark where each task reads and writes a large matrix, creating streaming behavior to access the main memory continuously. *dop* is configurable. | 4096×4096, 8192×8192 | 20000, 10000 |
| **Stencil** [4] | ST | A synthetic benchmark where each task repeatedly updates points on a multi-dimensional grid using the values at a set of neighboring points. *dop* is configurable. | 512×512, 2048×2048 | 50000, 50000 |

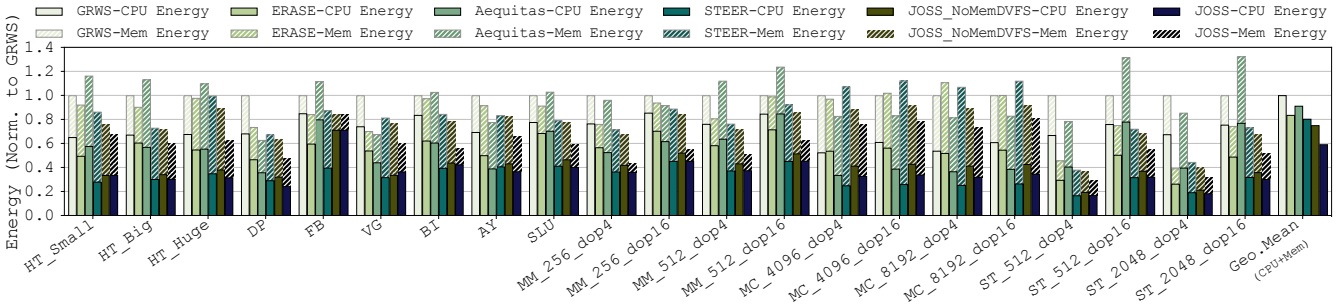



**Figure 8: Total energy consumption of evaluated benchmarks when using GRWS, Aequitas, ERASE, STEER and JOSS. All energy values are normalized to the total energy of the baseline GRWS, therefore, lower is better.**

(2) *ERASE* [11] employs an online history-based performance model and an offline categorized CPU power model to determine the configuration <$T_C$, $N_C$> that reduces CPU energy consumption without relying on explicit DVFS changes.

(3) *Aequitas* [37] is a heuristic-based scheduler that extends HERMES [38]. It first determines the core frequency for running each task based on task thief-victim relations (slow down the thief cores) and the size of the work queues. On core-clustered platforms, it lets each active core within a cluster tune the cluster frequency for a short interval (1s) in a round-robin time-slicing manner. Aequitas does not leverage the memory DVFS knob and moldable execution.

(4) *STEER* [12] is a model-based scheduler, which exploits the task characteristics and available CPU DVFS knob. It utilizes a performance model and a CPU power model to identify the configuration <$T_C$, $N_C$, $f_C$> for each task that consumes the least CPU energy. STEER does not leverage the memory DVFS knob.

## 7 EVALUATION

We evaluate JOSS under two scenarios targeting different energy performance trade-offs: (i) in Section 7.1 we evaluate the effectiveness of JOSS at reducing the total energy consumption by comparing it to several state-of-the-art schedulers; (ii) in Section 7.2 we evaluate the ability of JOSS for reducing the total energy consumption with user specified performance constraints with respect to (i). Finally, we analyze the prediction accuracy of three proposed models in Section 7.3 and present overhead analysis in Section 7.4.

### 7.1 Reducing Total Energy Consumption

Figure 8 compares the total energy consumption (incl. CPU energy and memory energy) when using GRWS, ERASE, Aequitas, STEER and JOSS across different benchmarks. We also include a new datapoint, JOSS_NoMemDVFS, where JOSS is employed for reducing the total energy consumption without leveraging the memory DVFS knob ($f_M$ is fixed at max. value). This is included to understand the impact of JOSS on asymmetric platforms, which support CPU DVFS but lack support for memory DVFS. For the supported benchmarks, we evaluate different task granularity and task DAG parallelism (*dop*) settings. This enables us to evaluate the effectiveness of the schedulers across a broad spectrum of task DAGs.

Overall, the results show that JOSS consumes the least energy across all benchmarks compared to the evaluated schedulers. Specifically, JOSS achieves 40.7% energy reduction, on average, compared to the baseline GRWS, while STEER, Aequitas and ERASE achieve 19.5%, 8.7% and 16.3% average reduction compared to the baseline respectively. These results demonstrate that JOSS achieves an additional 21.2% energy reduction compared to STEER (the best among the state-of-the-art). Even in the absence of memory DVFS knob, JOSS_NoMemDVFS achieves a 24.8% reduction in energy consumption compared to GRWS, which is still an improvement over

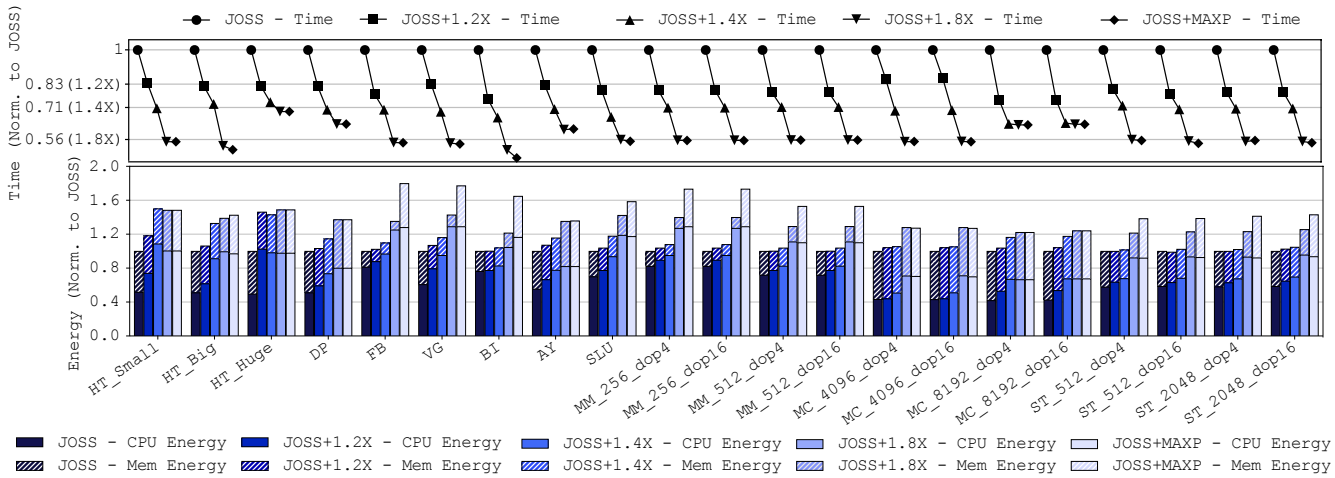



**Figure 9: Energy consumption (bottom) and execution time (top) when targeting energy reduction under performance constraints. Performance and energy values are normalized to JOSS without performance constraints.**

the state-of-the-art (e.g. 5.2% additional savings than STEER). This emphasizes the importance of taking the total energy consumption into account even when the memory DVFS knob is unavailable.

We analyze the effectiveness of JOSS using SparseLU (specifically the BMOD kernel) as an example. BMOD kernel accounts for 91% of the total number of tasks in SparseLU. With GRWS, 63% of BMOD tasks execute on the high-performance Denver cores while 37% of tasks execute on the relatively low-performance A57 cores. Although executing on a single Denver core is 3.4× faster than an A57 core, a reasonable fraction end up executing on the A57 cores, since the four A57 cores end up stealing more tasks from Denver queues. With ERASE, the CPU energy estimates obtained using performance and CPU power models indicate that running BMOD tasks on two Denver cores consumes less CPU energy since it can achieve linear speedup without doubling the CPU power consumption. Thus, ERASE reduces the CPU energy compared to GRWS. Aequitas relies on task stealing relations and the work queue size to select the core frequency and each active core tunes the cluster frequency for a short interval. A57 cores steal more BMOD tasks from the Denver cores, which make A57 become thief cores and get more workloads. Therefore, it ends up both slowing down and speeding up A57 cluster frequency for brief periods during execution (38% tasks executing on Denver and 62% executing on A57). With CPU frequency throttling, Aequitas reduces CPU energy but increases memory energy consumption in comparison to GRWS and ERASE due to the performance slowdown. STEER further reduces the CPU energy consumption by identifying the configuration <Denver, 2, 1.11GHz> for BMOD tasks. STEER however does not take memory energy consumption into account. Consequently, the performance slowdown from throttling CPU frequency setting results in higher memory energy consumption.

In contrast to STEER, JOSS_NoMemDVFS aims to reduce the total energy consumption. It utilizes three proposed models in JOSS to predict the CPU energy together with memory energy when only throttling the core frequency while memory frequency is fixed as the maximum. JOSS_NoMemDVFS selects <Denver, 2, 1.57GHz> as the configuration for reducing energy consumption. Running at 1.57GHz increases the CPU energy consumption compared to STEER. However, it also ends up reducing more memory energy consumption because of the performance improvement achieved from higher CPU frequency. JOSS leverages the memory frequency knob to further reduce the total energy consumption by identifying the configuration of <Denver, 2, 1.11GHz, 0.8GHz> for BMOD tasks. Since BMOD kernel is compute-intensive when running on two Denver cores (MB is estimated to be 1%), running with lower memory frequency does not have much impact on execution time of the tasks and leads to lower memory energy consumption.

## 7.2 Reducing Total Energy under Performance Constraints

Figure 9 shows results of JOSS reducing the total energy consumption while attempting to satisfy user specified performance constraints. In this experiment, we test three performance targets (speedups of 1.2×, 1.4× and 1.8× with respect to JOSS targeting energy reduction solely), in addition to MAXP where JOSS maximizes individual task performance without considering energy.

The top part of the figure shows the execution time of each configuration, along with the performance target. Overall, the results across benchmarks show that JOSS can achieve 1.2×, 1.4× and 1.8× speedups at the additional cost of 6%, 13% and 32% increase in energy consumption over JOSS without performance constraints. In a few cases, the ability to achieve the desired trade-off targets is impacted by the accuracy of the prediction models. For example, in the case of MC_4096, JOSS slightly misses (by 3%) the deadline of 1.2× speedup due to the inaccuracy in model predictions. The average prediction error of performance, CPU power and memory power models in this case are 9.2%, 13.8% and 18.9%, respectively. Furthermore, in benchmarks with high degree of memory intensity, JOSS does not achieve 1.8× speedup even when executing with maximum $<f_C, f_M>$, despite significant increase in total energy consumption. Ultimately, task performance is limited by processor capabilities, such as peak FLOPS and memory bandwidth, restricting JOSS' ability to reach a performance target.

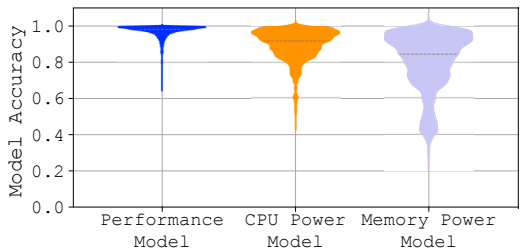



**Figure 10: Model prediction accuracy of three proposed models in JOSS. Dotted lines represent the medians.**

## 7.3 Model Accuracy

We analyze the accuracy of the performance, CPU power and memory power models proposed in JOSS. We compute accuracy using the formula: $accuracy = 1 - \frac{Absolute(real - prediction)}{real}$. We report the arithmetic average numbers across all evaluated benchmarks. The real values are collected through running each benchmark on all possible configurations for the four knobs. Figure 10 presents the prediction accuracy distribution of the three models across the evaluated benchmarks. The results show that the performance model achieves 97% accuracy on average (the median is 98.3% shown as the dotted line in Figure 10), the CPU power model achieves 90% accuracy on average (the median is 91.8%), while the memory power model achieves 80% accuracy on average (the median is 84.6%).

## 7.4 Overhead Analysis

To enable performance and power consumption prediction, JOSS implements three look-up tables per kernel for storing the measured and predicted execution times, CPU power and memory power consumption. Consider a platform with $N$ cores in total, $M$ asymmetric core-clusters such that each cluster comprises $\frac{N}{M}$ cores, the possible number of cores that can be used for each task equals $\log \frac{N}{M}$. Assume the numbers of available core frequency and memory frequency settings are $Nf_C$ and $Nf_M$. The storage overhead for three look-up tables for each kernel in JOSS is $3 \times M \times \log \frac{N}{M} \times Nf_C \times Nf_M$.

The sampling phase only requires $M \times \log_2 \frac{N}{M} \times 2$ tasks per kernel (e.g. 2 core-clusters × 3 possible numbers of cores per task × 2 CPU frequencies = 12 tasks per kernel on Jetson TX2). For the benchmarks we evaluate, our analysis shows that JOSS only spends 0.8% of the total execution time, on average, in this phase.

Next we compare the overheads of using steepest descent search and exhaustive search (details in Section 5.2.1). The results on Jetson TX2 show that using steepest descent search reduces timing overheads by 70% on average, compared to exhaustive search across all evaluated benchmarks, due to the significant reduction in the number of comparisons. Our evaluation also shows that configurations using selected steepest descent search achieves 97% energy reduction relative to the configurations selected using exhaustive search. On larger platforms, using the steepest descent search is expected to reduce timing overheads even further.

# 8 RELATED WORK

Existing works can broadly classified into three categories: those focusing on CPU energy reduction, memory energy reduction and total (CPU and memory) energy reduction.

*CPU energy on architectures with per-core DVFS:* Acun et al. [9] employ per-core DVFS and adopt an online history-based approach for performance and CPU power predictions by executing with every possible frequency. HERMES [38] proposes workpath-sensitive and workload-sensitive algorithms for per-core DVFS in a work stealing runtime. It slows down thief cores and selects appropriate frequencies based on workload sizes. CATA [10] dynamically tunes the frequency based on incoming task criticality and the available power budget at the moment. AAWS [44] targets for work stealing runtime on asymmetric platforms and proposes work-pacing, work-sprinting and work-mugging strategies (that require hardware support) by detecting the parallel slackness.

*CPU energy on architectures with cluster-based DVFS:* Besides Aequitas [37], ERASE [11] and STEER [12], discussed in Section 6.3, CHRT [26] is a phase-based scheduler that predicts task placement, cluster frequency, and number of cores for each execution phase. It uses an offline model where the online phases map to the model and takes the recorded configuration as the prediction.

*Memory energy:* MemScale [18] targets energy consumption in the memory subsystem. They leverage dynamic profiling, performance and power modeling to guide DVFS of memory controller and frequency scaling of memory channels and DRAM. David et al. [16] propose an intuitive algorithm that detects memory bandwidth utilization for tuning DVFS of memory subsystem. Both proposals are however tailored for multi-programmed workloads.

*CPU+Memory energy:* Sundriyal et al [43] target minimizing the power consumption of a system given the performance loss tolerance. They propose performance and power models using PMCs to determine the best joint frequency setting in a time window-based manner for the entire application. CoScale [17] is an epoch-based framework for multi-programmed workloads. They first collect PMCs for model prediction and then search for the best frequency pair using gradient-descent. However, their model only targets single -threaded applications and does not support task-based parallel applications.

# 9 CONCLUSION

We propose JOSS, a runtime scheduling framework that can both reduce energy consumption and explore various energy performance trade-offs for task-based parallel applications. Overall, JOSS achieves the goals set out via optimizing the execution of each task in the application. JOSS comprises a performance model, a CPU power model, a memory power model and a task scheduler. It utilizes the three models to predict the execution time and power consumption for each task when running with different configurations for the four knobs (i.e. core type, number of cores, core frequency scaling and memory frequency scaling). In contrast to existing works, JOSS manages to achieve higher energy savings by considering the impact of memory energy consumption, in addition to core asymmetry, CPU DVFS and task characteristics, and through the use of memory DVFS as a tunable knob. Our evaluation shows that JOSS achieves an additional 21.2% energy reduction on average compared to the state-of-the-art. Even in the absence of memory DVFS knob, JOSS can still save 5.2% additional energy. Furthermore, it is capable of reducing the total energy while still satisfying the performance constraints specified. We hope that

these results together with related papers that demonstrate benefits of memory scaling will encourage widespread adoption of memory DVFS knob as an additional avenue for improving energy efficiency.

## ACKNOWLEDGMENTS

This work has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No. 956702 (https://eprocessor.eu). The JU receives support from the European Union's Horizon 2020 research and innovation programme and Spain, Sweden, Greece, Italy, France, Germany. This work, in particular, has received funding from the Swedish Research Council under contract 2020-06735_3.

## REFERENCES

[1] 2014. Documentation of StarPU. https://files.inria.fr/starpu/doc/starpu.pdf.
[2] 2015. ODROID XU4. https://magazine.odroid.com/wp-content/uploads/odroid-xu4-user-manual.pdf.
[3] 2017. Jetson TX2 Module. https://developer.nvidia.com/embedded/jetson-tx2.
[4] 2018. XiTAO Runtime. https://github.com/CHART-Team/xitao.git.
[5] 2019. DDR5/4/3/2: How Memory Density and Speed Increased with each Generation of DDR. https://blogs.synopsys.com/vip-central/2019/02/27/ddr5-4-3-2-how-memory-density-and-speed-increased-with-each-generation-of-ddr/.
[6] 2020. ARM BIG.LITTLE. https://www.arm.com/why-arm/technologies/big-little.
[7] 2020. Biomarker Discovery. https://legato-project.eu/use-cases/healthcare.
[8] 2022. Apple A16 Bionic. https://en.wikipedia.org/wiki/Apple_A16.
[9] Bilge Acun, Kavitha Chandrasekar, and Laxmikant V. Kale. 2019. Fine-Grained Energy Efficiency Using Per-Core DVFS with an Adaptive Runtime System. In *2019 Tenth International Green and Sustainable Computing Conference (IGSC).*
[10] E. Castillo, M. Moreto, M. Casas, L. Alvarez, E. Vallejo, K. Chronaki, R. Badia, J. L. Bosque, R. Beivide, E. Ayguade, J. Labarta, and M. Valero. 2016. CATA: Criticality Aware Task Acceleration for Multicore Processors. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS).*
[11] Jing Chen, Madhavan Manivannan, Mustafa Abduljabbar, and Miquel Pericàs. 2022. ERASE: Energy Efficient Task Mapping and Resource Management for Work Stealing Runtimes. *ACM Trans. Archit. Code Optim.* (mar 2022).
[12] Jing Chen, Madhavan Manivannan, Bhavishya Goel, Mustafa Abduljabbar, and Miquel Pericàs. 2022. STEER: Asymmetry-aware Energy Efficient Task Scheduler for Cluster-based Multicore Architectures. In *2022 IEEE 34th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD).*
[13] Gilberto Contreras and Margaret Martonosi. 2008. Characterizing and improving the performance of intel threading building blocks. In *2008 IEEE International Symposium on Workload Characterization.* IEEE, 57–66.
[14] AM Coutinho Demetrios, Daniele De Sensi, Arthur Francisco Lorenzon, Kyriakos Georgiou, Jose Nunez-Yanez, Kerstin Eder, and Samuel Xavier-de Souza. 2020. Performance and energy trade-offs for parallel applications on heterogeneous multi-processing systems. *Energies* 13, 9 (2020), 2409.
[15] Sanjeev Das, Jan Werner, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. 2019. SoK: The Challenges, Pitfalls, and Perils of Using Hardware Performance Counters for Security. In *2019 IEEE S&P.*
[16] Howard David, Chris Fallin, Eugene Gorbatov, Ulf R. Hanebutte, and Onur Mutlu. 2011. Memory Power Management via Dynamic Voltage/Frequency Scaling. In *Proceedings of the 8th ACM International Conference on Autonomic Computing (ICAC '11).* 31–40.
[17] Qingyuan Deng, David Meisner, Abhishek Bhattacharjee, Thomas F. Wenisch, and Ricardo Bianchini. 2012. CoScale: Coordinating CPU and Memory System DVFS in Server Systems. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture.*
[18] Qingyuan Deng, David Meisner, Luiz Ramos, Thomas F. Wenisch, and Ricardo Bianchini. 2011. MemScale: Active Low-Power Modes for Main Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems.*
[19] Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Bofill, and Eduard Parra. 2009. Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. *Proceedings of the International Conference on Parallel Processing* (09 2009).
[20] Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguade. 2009. Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In *2009 international conference on parallel processing.* IEEE, 124–131.
[21] Mark Endrei, Chao Jin, Minh Ngoc Dinh, David Abramson, Heidi Poxon, Luiz DeRose, and Bronis R. de Supinski. 2018. Energy Efficiency Modeling of Parallel Applications. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis.*
[22] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language, In Proceedings of SIGPLAN 1998. *SIGPLAN.*
[23] Bhavishya Goel. 2016. *Measurement, Modeling, and Characterization for Energy-efficient Computing.* Chalmers University of Technology.
[24] Houzeaux Guillaume and Vazquez Mariano. [n.d.]. Alya Application. https://www.bsc.es/research-development/research-areas/engineering-simulations/alya-high-performance-computational.
[25] Jawad Haj-Yahya, Mohammed Alser, Jeremie Kim, A. Giray Yağlıkçı, Nandita Vijaykumar, Efraim Rotem, and Onur Mutlu. 2020. SysScale: Exploiting Multi-domain Dynamic Voltage and Frequency Scaling for Energy Efficient Mobile Processors. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA).*
[26] Myeonggyun Han, Jinsu Park, and Woongki Baek. 2021. Design and Implementation of a Criticality- and Heterogeneity-Aware Runtime System for Task-Parallel Applications. *IEEE TPDS* (2021).
[27] Sebastian Herbert and Diana Marculescu. 2007. Analysis of dynamic voltage/frequency scaling in chip-multiprocessors. In *Proceedings of the 2007 international symposium on Low power electronics and design (ISLPED '07).*
[28] Simon Holmbacka and Jörg Keller. 2017. Workload Type-Aware Scheduling on big.LITTLE Platforms. In *Algorithms and Architectures for Parallel Processing.*
[29] Canturk Isci, Gilberto Contreras, and Margaret Martonosi. 2006. Live, Runtime Phase Monitoring and Prediction on Real Systems with Application to Dynamic Power Management. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06).*
[30] Ivan Jibaja, Ting Cao, Stephen M. Blackburn, and Kathryn S. McKinley. 2016. Portable Performance on Asymmetric Multicore Processors. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO '16).*
[31] Tipp Moseley, Neil Vachharajani, and William Jalby. 2011. Hardware Performance Monitoring for the Rest of Us: A Position and Survey. In *Network and Parallel Computing.*
[32] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. 2023. A modern primer on processing in memory. In *Emerging Computing: From Devices to Systems.* Springer, 171–243.
[33] Antoni Navarro Muñoz, Arthur F. Lorenzon, Eduard Ayguadé Parra, and Vicenç Beltran Querol. 2021. Combining Dynamic Concurrency Throttling with Voltage and Frequency Scaling on Task-Based Programming Models. In *50th International Conference on Parallel Processing (ICPP 2021).*
[34] OpenMP Architecture Review Board. 2018. OpenMP Application Program Interface. Version 5.0.
[35] Thomas Rauber and Gudula Rünger. 2018. A scheduling selection process for energy-efficient task execution on DVFS processors. *Concurrency and Computation: Practice and Experience* 31 (10 2018).
[36] Basireddy Karunakar Reddy, Amit Kumar Singh, Dwaipayan Biswas, Geoff V. Merrett, and Bashir M. Al-Hashimi. 2018. Inter-Cluster Thread-to-Core Mapping and DVFS on Heterogeneous Multi-Cores. *IEEE Transactions on Multi-Scale Computing Systems* 4, 3 (2018).
[37] Haris Ribic and Yu Liu. 2016. AEQUITAS: Coordinated Energy Management Across Parallel Applications. In *2016 ACM International Conference on Supercomputing.* 1–12.
[38] Haris Ribic and Yu David Liu. 2014. Energy-Efficient Work-Stealing Language Runtimes. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14).*
[39] Efraim Rotem, Yuli Mandelblat, Vadim Basin, Eli Weissmann, Arik Gihon, Rajshree Chabukswar, Russ Fenger, and Monica Gupta. 2021. Alder Lake Architecture. In *2021 IEEE Hot Chips 33 Symposium (HCS).*
[40] Mark Sagi, Nguyen Anh Vu Doan, Martin Rapp, Thomas Wild, Jörg Henkel, and Andreas Herkersdorf. 2020. A Lightweight Nonlinear Methodology to Accurately Model Multicore Processor Power. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (2020).
[41] Rishad A. Shafik, Anup Das, Sheng Yang, Geoff Merrett, and Bashir M. Al-Hashimi. 2015. Adaptive Energy Minimization of OpenMP Parallel Applications on Many-Core Systems. In *Proceedings of the 6th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures (PARMA-DITAM '15).*
[42] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).
[43] Vaibhav Sundriyal and Masha Sosonkina. 2016. Joint Frequency Scaling of Processor and DRAM. *J. Supercomput.* 72, 4 (apr 2016), 1549–1569.
[44] Christopher Torng, Moyang Wang, and Christopher Batten. 2016. Asymmetry-Aware Work-Stealing Runtimes. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA).* 40–52.
[45] Xingfu Wu, Valerie Taylor, Jeanine Cook, and Philip J. Mucci. 2016. Using Performance-Power Modeling to Improve Energy Efficiency of HPC Applications. *Computer* 49, 10 (2016), 20–29.