# SWEEP: Adaptive Task Scheduling for Exploring Energy Performance Trade-offs

Jing Chen, Madhavan Manivannan, Bhavishya Goel, Miquel Pericàs
Department of Computer Science and Engineering, Chalmers University of Technology
Email: chjing@chalmers.se, madhavan@chalmers.se, goelb@chalmers.se, miquelp@chalmers.se

***Abstract*—Energy efficiency is becoming a major concern when running parallel computing systems owing to its impact on system reliability and operating cost. Recent works, that focus on energy efficient execution of task-based parallel applications on multi-core systems, leverage a subset of architectural features (core asymmetry, CPU DVFS and memory DVFS) and application attributes (inter-task parallelism, intra-task parallelism and task characteristics) to achieve this goal. More importantly, they have a fixed target metric and do not provide the flexibility to explore energy performance trade-offs (EPTO). We propose SWEEP, a task scheduler that leverages all the aforementioned architectural knobs and application attributes to facilitate EPTO exploration. SWEEP, at a high level, uses a combination of models and heuristics and works by splitting application execution into high parallelism and low parallelism phases. It then uses an adaptive task distribution algorithm, specific to the phase type, that leverages model-based predictions to determine the best task schedule and the DVFS settings for the phase. Moreover, SWEEP is able to flexibly target various EPTO metrics, a feature that is not supported by other proposals. Our evaluation shows that SWEEP achieves 19.9%, 36.4% and 9.5% reduction on average in terms of EDP, ED$^2$P and E$^2$DP compared to the best performing state-of-the-art.**

## I. INTRODUCTION

Parallel computing systems in the past have predominantly focused on improving performance, often at the expense of neglecting energy efficiency as a secondary concern. These systems impose exorbitant power and thermal requirements that adversely impact system reliability and operating cost. Hence, there is increasing emphasis on improving energy efficiency of such systems, in addition to performance. Multi-core processors, which are the workhorses in such systems, consequently accommodate several hardware features that target energy efficient execution of applications.

Dynamic voltage and frequency scaling (DVFS) is a widely adopted hardware feature for improving energy efficiency. Prior works have extensively demonstrated the benefits of exploiting CPU DVFS for reducing energy consumption [1]–[11]. Recent studies also demonstrate the potential for using memory DVFS to further improve energy efficiency, as memory energy is becoming a major contributor to the total energy consumption of a system [10], [12]–[15]. Core asymmetry is another popular hardware feature incorporated in several designs that involves integrating multiple types of cores with distinct power and performance characteristics, albeit with the same ISA, as seen in systems based on ARM's big.LITTLE architecture [16]. Designs with asymmetric core-clusters reduce the cost and complexity of implementing DVFS by grouping cores of the same type into clusters, wherein cores in the same cluster can only operate at a common frequency setting. Such architectures have been adopted in systems ranging from mobile devices to high-performance desktops, such as Apple's A17 Pro chip [17], NVIDIA's Tegra Xavier [18], and Intel's Alder Lake [19].

The knobs available in hardware, i.e. core asymmetry, CPU DVFS and memory DVFS, provide an opportunity to execute applications in a wide range of settings and enables exploration of different trade-offs between performance and power/energy consumption. The metric used for assessing the trade-offs can target a single objective, such as reducing energy or can target a combined objective, such as Energy Delay Product (EDP) that weighs energy savings and performance equally. Alternatively, its variations $\mathrm{E^mD^nP}$ could be used to prioritize either energy savings or performance. Prior works mainly focus on leveraging these knobs and exploring energy performance trade-offs (EPTO) in the context of single-threaded applications and multi-programmed workloads (comprising multiple single-threaded applications) [12]–[15].

Besides single-threaded and multi-programmed workloads, it is also important to optimize the energy efficiency of parallel applications. The task-based programming paradigm has been adopted in many parallel programming libraries and runtimes [20]–[23]. It involves expressing parallelism in an application in the form of tasks and their dependencies that are generated and resolved dynamically during execution. Different tasks exhibit diverse characteristics, such as computational intensity and memory access patterns. An application comprising tasks can be modeled as a directed acyclic graph (DAG), wherein independent tasks (that do not have any outstanding dependency) can be executed in parallel, which we refer to as inter-task parallelism. Some models furthermore support moldable task execution. A moldable task can be dynamically partitioned into a variable number of smaller subtasks, enabling execution across multiple cores, which we refer to as intra-task parallelism. This has been shown to improve performance by employing idle resources and reduce total energy consumption through lowering system idle energy [24], [25].

Work stealing is a popular task scheduling technique employed in many task-based runtimes, such as Cilk5 [22], OpenMP [20], and TBB [23]. It has been shown to be effective when there is sufficient inter-task parallelism available and can scale to large core counts while still ensuring load balancing even on asymmetric architectures. Work stealing, applied in its original form, is however not a good fit for EPTO exploration,

since it has several limitations: (1) it does not leverage DVFS knobs, thereby limiting the scope of EPTO exploration; (2) it is unaware of the task characteristics and can therefore result in energy inefficient mapping of tasks to cores due to random work stealing; (3) it does not perform well with a low degree of inter-task parallelism due to the greedy nature of work stealing, which can result in severe performance degradation (i.e. idle little cores steal tasks from big cores); and (4) work stealing lacks the capability to make use of idle cores for accelerating individual tasks.

Recent works that focus on energy efficient runtime scheduling techniques on top of work stealing can be broadly grouped into two categories. The first category includes a group of proposals that are completely heuristic-based and aim to reduce energy consumption with little performance degradation. These heuristics range from exploiting task-stealing sequences and task queue size to throttle CPU DVFS, to decreasing voltage of big cores and increasing voltage of little cores [3], [4], [26]. The second category consists of several model-based schemes that utilize pre-trained power and performance models to incorporate task-awareness and to predict the effect of tuning architectural knobs on both energy and performance of a task [9], [10], [27], in addition to using heuristics. These proposals mainly focus on reducing energy consumption by searching for appropriate configuration of architectural knobs that consumes the least energy for each individual task. Our evaluation in Section VI shows that both categories of existing proposals have limited effectiveness. More importantly, these proposals are tuned to a specific EPTO metric and lack the flexibility to explore various EPTO metrics as required.

The goal of the paper is to design a task scheduler that leverages architectural knobs (core asymmetry, CPU DVFS and memory DVFS) and application attributes (inter-task parallelism, intra-task parallelism and task characteristics), to facilitate EPTO exploration on multi-core architectures. To achieve this goal, several challenges need to be addressed. First, there are numerous factors that influence EPTO, e.g. the choice of core type, the number of cores used to run a task, the amount of tasks to run concurrently and the choice of DVFS settings. Understanding the implications of tuning each knob individually and their interplay is complex. Assuming the impact of the aforementioned knobs can be estimated, the challenge of determining an optimal schedule for a dynamically unfolding DAG on asymmetric architecture still needs to be addressed. Lastly, the scheduler should offer flexibility to target different EPTO metrics based on specific requirements.

We propose **SWEEP** (**S**tealing **W**ork **E**ffectively for exploring **E**nergy **P**erformance trade-offs), a heuristic scheduler for task-based parallel applications that combines model-based predictions with adaptive task distribution algorithms. In a nutshell, SWEEP splits application execution into phases and classifies these phases into one of the two types, namely Low-Parallelism (LP) and High-Parallelism (HP), based on the instantaneous inter-task parallelism. SWEEP then applies different task distribution algorithms for HP and LP to decide the best task schedule and DVFS settings, respectively. The use of different task distribution algorithms for HP and LP is motivated by the observation that in LP, unlike HP, there is little inter-task parallelism and stealing inadvertently can lead to severe load imbalance and drastically impact performance. The algorithms require predicting execution time and power consumption of individual tasks. SWEEP utilizes polynomial regression models to understand the impact of tuning available architectural knobs on both energy consumption and performance of tasks. In summary, the main contributions of this work are as follows:

- We propose SWEEP, a scheduler that leverages architectural knobs (core asymmetry, CPU and memory DVFS) together with application attributes (inter-task parallelism, intra-task parallelism and task characteristics) to facilitate EPTO exploration. SWEEP splits execution into phases, classifies phases as HP and LP and determines the best task schedule and DVFS settings for each phase. SWEEP adapts task schedule and DVFS settings to various EPTO metrics.
- We extensively evaluate SWEEP by comparing it with several state-of-the-art proposals that target energy efficient execution. Our evaluation demonstrates the effectiveness of SWEEP to flexibly target different EPTO metrics, a feature that is not supported by other state-of-the-art proposals. More specifically, SWEEP achieves 19.9%, 36.4% and 9.5% reduction on average in terms of EDP, $ED^2P$ and $E^2DP$ compared to the best performing state-of-the-art on a NVIDIA Jetson TX2 platform.

## II. Overview

### A. Background

In our model, a task is defined as an instance of a kernel (i.e. task type) that cannot be preempted by the runtime scheduler and always runs to completion. A task DAG of an application comprises several tasks of one or more types. The edges in a task DAG represent the inter-task dependencies while the vertices represent the individual tasks. In work stealing schedulers, there is a local task queue associated with each core, which serves to hold the ready tasks (i.e. tasks whose dependencies are satisfied). A task that is marked as ready is scheduled to a task queue. When a core is idle, it first looks for ready tasks in its own local task queue. If it does not find any, it greedily attempts to steal tasks from the local queues of other cores. Upon completion of a task, the runtime resolves dependencies dynamically and marks its successors tasks (if any) without outstanding dependencies to be ready.

### B. SWEEP Overview

Figure 1 provides a high-level overview of SWEEP. The inputs to the scheduler are the task-based parallel application, the supported settings for the different architectural knobs (number of clusters, number of cores in each cluster and available frequency settings), and a specific EPTO target metric. SWEEP then determines the task schedule and the appropriate CPU and memory DVFS settings.

SWEEP is a heuristic scheduler that combines model-based prediction with adaptive task distribution algorithms. It works

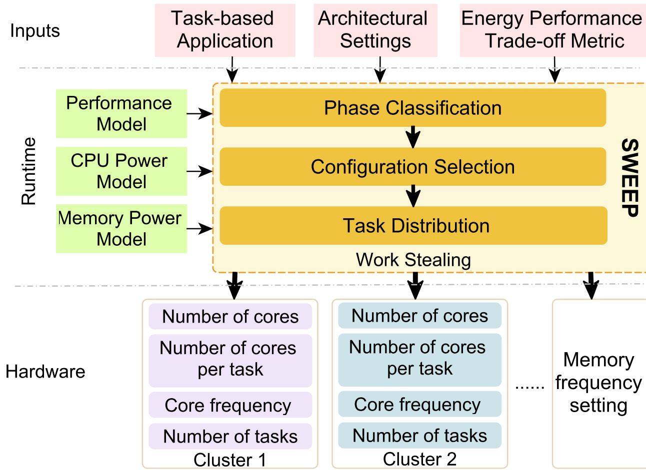



Fig. 1: A high-level overview of SWEEP.

by splitting application execution into phases, classifying each phase into HP and LP based on the instantaneous inter-task parallelism (the total number of ready tasks) and applying different task distribution algorithms on top of a work stealing scheduler for each type of phase. Details regarding phase invocation and classification process are discussed in Section III-A. The rationale behind using separate algorithms is that LP phase has less inter-task parallelism, and inadvertently allowing task stealing in LP phase can result in substantial load imbalance that drastically impacts performance.

In summary, the algorithms work by comparing different possible schedules to determine the best schedule, based on the target metric. The naive way of comparing all possible schedules for a phase comprising T independent tasks running on a system with N cores per core-cluster, C core-clusters, $\mathrm{F_C}$ CPU DVFS settings per core-cluster and $\mathrm{F_M}$ memory DVFS settings, requires evaluating $\mathrm{T!} \times \mathrm{N^T} \times (\mathrm{F_C})^\mathrm{C} \times \mathrm{F_M}$ schedules, to determine the best. This quickly becomes intractable even with small values for different parameters. To sidestep this, SWEEP simply disregards the effects of ordering and mapping of tasks (within a cluster) since its effect is expected to be small. SWEEP additionally uses heuristic algorithms to narrow the search space even further and determine the schedule that yields the best predicted EPTO with low overhead. The adaptive task distribution algorithms determine: (1) $\mathrm{NC_{T_i}}$, number of cores to be used to run a single task in a cluster (i.e. intra-task parallelism); (2) $\mathrm{NC_{C_i}}$, the number of cores to be used to run tasks concurrently; (3) $f_{\mathrm{C_i}}$ and $f_{\mathrm{M}}$, DVFS setting for core-clusters and memory when executing these tasks; and (4) $\mathrm{NT_{C_i}}$, number of tasks to be executed in a cluster, for each HP (discussed in Section III-B) and LP (in Section III-C) phase, based on the target EPTO.

Running the task distribution algorithm on phase invocation to determine the task schedule incurs a small overhead (see analysis in Section VI). Since some applications can repeatedly invoke phases on a frequent basis, the overheads can add up and can no longer be amortized. SWEEP utilizes an interval-based update approach to address this issue. SWEEP runs the algorithms and computes the configurations once per time interval. During subsequent phase invocations within the same time interval, the configurations determined earlier for the HP and LP phases are reused to reduce the computational overhead. SWEEP updates the configurations for HP and LP by running the algorithms again in the next time interval. Details regarding the SWEEP runtime execution timeline and interval-based update mechanism are discussed in Section III-D.

To facilitate task schedule determination ($\mathrm{NC_{T_i}}$, $\mathrm{NC_{C_i}}$ and $\mathrm{NT_{C_i}}$) and DVFS tuning ($f_{\mathrm{C_i}}$ and $f_{\mathrm{M}}$), the algorithm requires knowledge of per-task execution time and power consumption under different settings. SWEEP employs a set of predictive models, constructed with polynomial regression technique, similar to those proposed in related works, to predict performance and CPU and memory power consumption when tuning the hardware knobs (core-cluster type, number of cores, CPU frequency, memory frequency) for individual tasks. Details regarding the models and the scheduler implementation are discussed in Section IV.

## III. SWEEP Design

In this section, we describe how SWEEP schedules tasks and manages frequency settings. To simplify the discussion in Sections III-A to III-D, we assume the following: (1) a task-based parallel application modeled as a task DAG, where all tasks are invocations of a single kernel, running on a multi-core system comprising two asymmetric core-clusters (one featuring big cores and the other featuring little cores with different power/performance characteristics) with N cores in each cluster; (2) tasks are reasonably coarse-grained such that the overhead of frequency throttling at the core-cluster level is negligible compared to execution time of an individual task; (3) EDP is the chosen EPTO target metric; (4) prediction models for performance and power consumption of individual task types when running on different core-clusters, using different number of cores per-task and at different CPU and memory frequency settings are available (see Section IV-A). Adaptations to SWEEP that involve relaxing the first three assumptions are discussed in Section III-E.

### A. Phase Classification

SWEEP defines phase start as a point where a task releases one or more dependent tasks upon finishing its execution. A phase that has started, is deemed to conclude when all the independent sibling tasks, which were released by the same parent task, finish execution. SWEEP employs a two-step approach to classify a phase as either an HP or an LP phase based on the instantaneous inter-task parallelism, during phase invocation. In the first step, SWEEP monitors all the ready tasks, including the tasks already in the queues and the newly released ones. If the total number of ready tasks is higher than a certain threshold (details in Section V-B), the phase is classified as an HP phase. If the total number of ready tasks is below the threshold, SWEEP moves to the second step, where it checks if the number is still large enough to keep all cores busy and achieve load balancing on the system. We clarify

how this check is performed using a simple example where a task executes X times faster on one core-cluster type compared to the other. In this case, the total number of tasks required for load balancing ($T_L$) is computed using Equation 1.

$$T_L = \lfloor N \times (X + 1) \rfloor \quad (1)$$

The second step has a slightly higher overhead compared to the first step, since it involves utilizing the performance model to obtain execution time prediction for the task. A phase is only classified as LP if both these steps indicate low inter-task parallelism. Once a phase is classified during invocation, the schedule for all the newly released tasks in the phase is decided using the respective task distribution algorithm described in the following sections.

### B. High Parallelism Phase

The approach to determine the best task schedule and DVFS settings for a phase requires estimating the EDP for any task schedule at a specific DVFS setting and comparing the EDP of all possible configurations. We first describe the simplistic model used by SWEEP to estimate the EDP for a given schedule. We then discuss the heuristic algorithm to narrow the search space for different possible schedules for comparison.

**EDP estimation:** In an HP phase, the total number of ready tasks is sufficient to keep all the cores fully occupied. Hence, the time spent executing useful work on each core is nearly identical. In this scenario, the EDP for a single phase can be computed as shown in Equation 2.

$$\begin{aligned} EDP &= ((\sum_{j=0}^{2N}(Power_{cpu_j} \times max(Time_{cpu_j}))) + \\ &(Power_{mem} \times max(Time_{cpu_j}))) \times max(Time_{cpu_j}) \\ &= (\sum_{j=0}^{2N} Power_{cpu_j} + Power_{mem}) \times (max(Time_{cpu_j}))^2 \end{aligned} \quad (2)$$

SWEEP estimates execution time for a phase comprising T tasks, by using models that predict power consumption and execution time of an individual task type, in Equation 3,

$$max(Time_{cpu_j}) = \lceil \frac{T}{T_L} \rceil * \max\{Time_{little}, Time_{big}\} \quad (3)$$

where $Time_{little}$ and $Time_{big}$ are the predicted execution times for a task on a little core and a big core and $T_L$ is obtained using Equation 1. The aforementioned model is for the case where each task runs on a single core without exploiting intra-task parallelism, all cores are utilized to run tasks and the faster cluster is assigned X times more tasks compared to the slower cluster (i.e. $NC_{T_i}$=1, $NC_{C_i}$=N, $NT_{C_i}$=X:1), at a specific $f_{C_i}$ and $f_M$ setting. We discuss how the model can be easily extended to predict EDP for other values of $NC_{T_i}$, $NC_{C_i}$, $f_{C_i}$, $f_M$ and $NT_{C_i}$ later in the section.

**Heuristic algorithm:** SWEEP narrows the search space of possible configurations by independently determining the best configuration for one parameter and using it subsequently to determine the configuration for the other(s). SWEEP implements a four-step algorithm to determine the task schedule and the DVFS settings. In the first step, SWEEP determines the number of cores to be used in each cluster for executing an individual task ($NC_{T_i}$). The second step decides the number of cores ($NC_{C_i}$) to be used for concurrent task execution while fixing $NC_{T_i}$ to the value determined in step 1. The third step identifies the best frequency settings for each cluster ($f_{C_i}$) and memory ($f_M$) for executing tasks based on the settings identified in the first two steps. The final step determines the task distribution across clusters ($NT_{C_i}$) based on the settings identified in previous three steps. We discuss each of these steps in detail below.

*Step 1:* SWEEP compares the predicted execution time for running a task using different number of cores in each core-cluster and chooses to exploit intra-task parallelism ($NC_{T_i}$ > 1) only if it provides superlinear speedup. Exploiting intra-task parallelism can provide superlinear speedup due to an overall reduction in cache capacity and memory bandwidth requirements. However, using intra-task parallelism to speedup the execution of individual tasks comes at the cost of reducing task concurrency. Furthermore, if a task does not show superlinear speedup from exploiting intra-task parallelism, it will likely increase the execution time of the phase as a whole due to reduced concurrency, while consuming almost the same amount of power. The core count that yields the highest superlinear speedup is chosen as $NC_{T_i}$ for each cluster. Note that all tasks, that belong to a single phase and are scheduled to execute in a specific core-cluster, will use the same $NC_{T_i}$ determined in this step.

*Step 2:* Next, SWEEP loops through the combinations of using different number of cores ($NC_{C_i}$) to determine the best schedule for running the tasks that leads to the lowest predicted EDP. For each combination, EDP is predicted under the assumption that either all N cores or none of the cores in a cluster are utilized, i.e. $NC_{C_i}$={0, N}. Execution time for running the T tasks is predicted using Equation 3 with adaptations to determine $T_L$ for a specific <$NC_{C_{big}}$, $NC_{C_{little}}$> combination. We clarify this with an example where a single task is predicted to run $X'$ times faster when running with $NC_{T_{big}}$ on the big cluster than with $NC_{T_{little}}$ on the little cluster. In this case $T_L$ is computed as shown in Equation 4.

$$T_L = \lfloor X' \times (NC_{C_{big}}/NC_{T_{big}}) + (NC_{C_{little}}/NC_{T_{little}}) \rfloor \quad (4)$$

*Step 3:* SWEEP uses the highest $f_{C_i}$ and $f_M$ settings to determine the parameters in the first two steps. In step 3, SWEEP identifies the best frequency setting that results in the greatest decrease in power consumption with the least amount of performance degradation. The EDP prediction for fixed $NC_{T_i}$ and $NC_{C_i}$ settings (determined by the two earlier step) and varying $f_{C_i}$ and $f_M$ settings are estimated by accessing the corresponding performance and power look-up table entries. A possible approach for determining the best settings is to exhaustively loop through all possible combination of $F_C$ settings for each CPU cluster and $F_M$ for memory and compare the estimated EDP values to determine the best setting that leads to the lowest predicted EDP. However, such an approach can introduce significant computational overhead, particularly as the number of frequency settings and the number of clusters

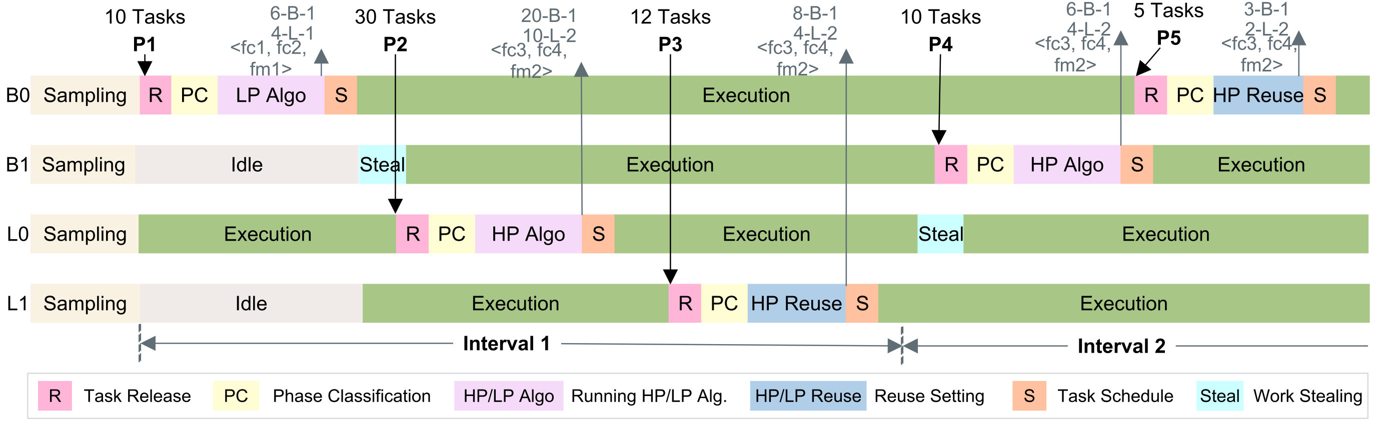



Fig. 2: SWEEP execution timeline. Note that the timeline is not to scale. The outputs of the algorithms are of the following format: $\mathrm{NT_C - B/L - NC_T}$, <$f_{\mathrm{C_B}}, f_{\mathrm{C_L}}, f_{\mathrm{M}}$>.

scale. To address this, we device another heuristic frequency selection algorithm that prunes the space and identifies the frequency setting with reduced overhead. The algorithm begins by locating the point with the highest frequency setting for both the clusters and the memory. It then compares the EDP value of the data point against all its immediate neighbours (frequency settings) and repeats this search process iteratively until it converges at a point with the lowest predicted EDP.

*Step 4:* In the last step, SWEEP computes task distribution across clusters that achieves load balancing under the settings ($\mathrm{NC_{T_i}}$, $\mathrm{NC_{C_i}}$, $f_{\mathrm{C_i}}$ and $f_{\mathrm{M}}$) determined in previous steps. We clarify this with an example where a single task is predicted to run $\mathrm{X''}$ faster on the big cluster than the little cluster under these settings. In this case $\mathrm{T_L}$ is calculated using Equation 4 and replacing $\mathrm{X'}$ with $\mathrm{X''}$. Consequently, the task distribution ratio between big and little cluster is $\mathrm{X''} \times (\mathrm{NC_{C_{big}}}/\mathrm{NC_{T_{big}}}) : (\mathrm{NC_{C_{little}}}/\mathrm{NC_{T_{little}}})$.

Finally, SWEEP assigns ready tasks to the two clusters in a weighted round-robin manner based on the task distribution ratio determined in step 4. Each task is placed into a randomly chosen local task queue within the specific cluster. The frequency of the clusters and memory are throttled to the setting determined before each task begins execution. This is achieved by storing the settings determined as part of the task data structure. SWEEP first prioritizes executing tasks from the local task queue of the core and then performs work stealing from the local queues of other cores within the same core-cluster. After the number of failed intra-cluster stealing attempts exceeds a threshold, SWEEP allows stealing attempts across core-clusters to mitigate potential load imbalances that can happen during execution of an HP phase.

### C. Low Parallelism Phase

In contrast to an HP phase, it is difficult to achieve load balancing and keep all cores utilized in an LP phase, due to the limited number of ready tasks. Varying $\mathrm{NC_{T_i}}$, $\mathrm{NC_{C_i}}$ and $\mathrm{NT_{C_i}}$ impacts resource utilization and results in large differences in power consumption and execution time across the different cores and adversely affects EDP, predicted using Equation 2. It is also important to prioritize intra-task parallelism to enhance performance and reduce system idle energy whenever it is beneficial to do so.

**Heuristic algorithm:** SWEEP implements a two-step algorithm to determine the task schedule and the DVFS settings in an LP phase. The first step determines $\mathrm{NC_{T_i}}$, $\mathrm{NC_{C_i}}$ and $\mathrm{NT_{C_i}}$ by jointly evaluating their impact on EDP to enable detailed exploration of their interplay, in contrast to HP, which evaluates each parameter independently. The second identifies the best frequency setting for CPU and memory that further reduces EDP of the phase.

*Step 1*: SWEEP loops through the possible combinations of using different <$\mathrm{NC_{T_i}}$, $\mathrm{NC_{C_i}}$, $\mathrm{NT_{C_i}}$> and selects the best combination that leads to the lowest predicted EDP. The worst case time complexity of evaluating different combinations for an LP phase with T tasks is $\mathrm{O(T \times (logN)^2)}$ assuming two clusters. EDP for each combination is predicted using Equation 2. Here the execution time for an LP phase is determined by predicting the execution time of running $\mathrm{NT_{C_i}}$ tasks with $\mathrm{NC_{T_i}}$ cores per task on different core-clusters and using the execution time of the slowest cluster. The power consumption of each core can be estimated as a weighted average of active power during periods of task execution and idle power during inactivity. The active and idle power values for cores of various types and quantities are provided by power models (discussed in Section IV-A).

*Step 2*: SWEEP uses the highest $f_{\mathrm{C_i}}$ and $f_{\mathrm{M}}$ settings to determine the parameters in the first step. Throttling $f_{\mathrm{C_i}}$ and $f_{\mathrm{M}}$ individually and in combination affects both power consumption and execution time of each core. We utilize the same frequency identification algorithm discussed in step 3 of the HP algorithm (Section III-B) to identify the $f_{\mathrm{C_i}}$ and $f_{\mathrm{M}}$ pair that yields the lowest predicted EDP.

Once SWEEP determines the task schedule and frequency settings, it distributes tasks in the same manner as in an HP phase. In an LP phase, SWEEP permits work stealing among cores in the same core-cluster, to maintain load balancing. Stealing across asymmetric clusters is disabled to avoid dis-

rupting the task distribution determined by the algorithm.

### D. SWEEP Execution Timeline

SWEEP utilizes an interval-based approach to reduce scheduling overhead whereby the algorithms identify the task distribution for HP and LP phases only once in an interval. Subsequent phase invocations during the interval reuses the configurations determined earlier in the same interval. SWEEP updates the task distributions for both HP and LP by running the algorithms again in the next interval.

Figure 2 shows an example of the SWEEP execution timeline on a four-core system comprising two big and two little cores. SWEEP performs task sampling at the beginning of application execution as it is required to enable model predictions (discussed in Section IV-A). In this example, B0 finishes executing a task, after the end of sampling period, triggering the release of 10 ready tasks and marking the start of a phase (P1). Next SWEEP (running on B0) checks the total number of ready tasks and classifies the phase as an LP phase. It consequently runs the LP task distribution algorithm to determine the task distribution and frequency settings for all the tasks in this phase. Finally it schedules the tasks for execution by placing them on different queues (6 on big cores and 4 on little cores) and begins to execute tasks from its local queue. P2 starts when L0 finishes executing a task and releases 30 ready tasks. SWEEP (running on L0) classifies P2 as an HP phase and runs the HP algorithm to determine the task distribution and the frequency settings. The ready tasks are distributed to the queues with a 2:1 ratio among big and little cores. L1 starts a new phase (P3) when it finishes executing a task and releases 12 ready tasks. P3 is classified as an HP phase after counting the total number of ready tasks. SWEEP reuses the same $\mathrm{NC_{T_i}}$, $\mathrm{NC_{C_i}}$, $\mathrm{NT_{C_i}}$, $f_{\mathrm{C_i}}$ and $f_{\mathrm{M}}$ settings computed during P2 for tasks in this phase. Next, B1 releases 10 tasks marking the beginning of P4. Note that P4 is classified as an HP phase even though it releases the same number of tasks as P1, owing to the total number of ready tasks in the different queues. As P4 is the first occurrence of an HP phase in interval 2, the HP algorithm is executed again to determine the settings for the tasks in this phase. Finally, P5 starts on B0 and is classified as HP, owing to the number of ready tasks on the different queues. P5 reuses the same settings as P4.

There are several instances in the example where the desired $f_{\mathrm{C}}$ and $f_{\mathrm{M}}$ settings for concurrently running tasks (from different phases) are not the same. Prior works have proposed to time share and run with different frequency settings in small windows, use minimum, maximum or average settings in such a context for tasks that share resources such as core-clusters and memory [3], [9]. SWEEP averages the frequency for multiple tasks running concurrently, to determine DVFS settings for both core-clusters and memory.

### E. Adaptions to SWEEP

**Applications with multiple kernels:** We adapt SWEEP for applications comprising multiple kernels by slightly modifying the task distribution algorithms. The classification mechanism, even for phases that comprise tasks of different types, remains the same. During an HP phase, SWEEP runs the task distribution algorithm several times, once for each task type in a phase, to compute the ratio of task distribution. Note that the different runs use the same value of T (total number of tasks in a phase). During an LP phase, SWEEP runs the algorithm once per task type. However, the value of T used in the different runs can differ and is equal to the number of tasks of the specific type in the phase. When different invocations of the same kernel use different input sizes, we consider them as distinct task types, since they can exhibit varying task characteristics.

**Fine-grained tasks:** SWEEP does not require any changes to the classification mechanism or the task distribution algorithms to support applications with fine grained tasks. The changes are purely limited to when and how DVFS setting, determined by the algorithm, are applied. Before doing any frequency adaptation we check whether a task is fine-grained, based on a preset execution time threshold. This threshold is determined based on the DVFS reconfiguration overhead of the platform. If the predicted execution time of a task is smaller than the threshold, we treat the task as fine-grained and use a different approach for applying frequency settings. Once SWEEP detects fine-grained tasks, it checks the tasks in the local queues of the same cluster in a round-robin manner. If there is a sufficient number of tasks with the same DVFS setting, and the accumulated sum of their predicted execution time is larger than the DVFS overhead, the DVFS setting is applied. Otherwise, SWEEP runs the fine-grained task with current frequency settings.

**EPTO metrics:** SWEEP can easily be adapted to various EPTO metrics since the algorithms used can predict execution times and power consumption values. SWEEP targets different trade-offs between energy consumption and performance, i.e. $\mathrm{E^mD^nP}$ by adapting Equation 2 as follows:

$$E^m D^n P = (\sum_{j=0}^{2N} Power_{cpu_j} + Power_{mem})^m \times (max(Time_{cpu_j}))^{m+n} \quad (5)$$

## IV. SWEEP Implementation

The task distribution algorithms require knowledge of per-task execution time and power consumption under different settings. Details regarding the models used for prediction are presented in Section IV-A. Next, changes to integrate SWEEP into a runtime system are discussed in Section IV-B.

### A. Task Sampling and Modeling

The methodology for predicting performance and power consumption for a task and the models are adopted from prior work [10]. Overall, SWEEP utilizes task sampling together with prediction models for estimating performance and CPU/memory power consumption for individual tasks when executing on different core types, different number of cores, and CPU/memory frequency settings. SWEEP samples the execution times for each task type by running on different clusters and with different number of cores. It then uses models

to predict the performance and power consumption for that task type when tuning CPU and memory frequency settings.

SWEEP uses *memory-boundness* (MB) as a metric for task characteristic to quantify the fraction of execution time that does not scale with core frequency, similar to prior works [9], [10], [28]. Execution time of a task can be estimated as the sum of computation time (1-MB) and stall time (MB) due to memory latency. When throttling the core frequency from $f_{\mathrm{C}}$ to $f'_{\mathrm{C}}$, the computation fraction will scale proportionally with frequency: $Time_{f'_C} = Time_{f_C} \times (MB + (1 - MB) \times \frac{f_C}{f'_C})$. SWEEP relies on sampling task execution times at two different core frequencies under the highest memory frequency setting to calculate MB as follows:

$$MB = (\frac{Time_{f'_C}}{Time_{f_C}} - \frac{f_C}{f'_C})/(1 - \frac{f_C}{f'_C}) \tag{6}$$

SWEEP calculates the MB values for different core types and number of cores and for each task type in a DAG during the sampling phase shown in Figure 2. The MB values are then used to predict performance and power consumption as discussed below.

SWEEP employs a set of power and performance prediction models, using second-order polynomial regression. The models investigate the interactions between task characteristics (MB) and CPU DVFS and memory DVFS. The power model specifically requires running a set of synthetic microbenchmarks designed with different amount of computations and memory accesses on all possible DVFS setting combinations during boot time, to estimate impact of MB and frequencies on power consumption. This is stored in a table and is looked up to predict power consumption during execution. The statistics from running synthetic benchmarks show that CPU power is mainly dependent on MB and $f_{\mathrm{C}}$, While memory power depends on both $f_{\mathrm{C}}$ and $f_{\mathrm{M}}$ in addition to MB. The two power models are shown in Equations 7 and 8.

$$Power_{cpu} = \sum_{0}^{1} \beta_i x_i + \sum_{0}^{1} \beta_{ii} x_i^2 + \beta_{01} x_0 x_1 + \varepsilon \tag{7}$$

where $x_i = \{MB, f_{\mathrm{C}}\}$, $0 \leq i \leq 1$.

$$Power_{mem} = \sum_{0}^{2} \beta_i x_i + \sum_{0}^{2} \beta_{ii} x_i^2 + \sum_{i=0}^{1} \sum_{k=i+1}^{2} \beta_{ik} x_i x_k + \varepsilon \tag{8}$$

where $x_i = \{MB, f_{\mathrm{C}}, f_{\mathrm{M}}\}$, $0 \leq i \leq 2$.

For the performance model, the computation time scales with the core frequency, while the stall time is dependent on $f_{\mathrm{C}}$, $f_{\mathrm{M}}$ and task characteristics (MB). Consequently, when throttling the frequencies to $f'_{\mathrm{C}}$ and $f'_{\mathrm{M}}$ for executing a task, the new execution time of the task is predicted as follows:

$$\begin{aligned} Time' =& Time \times (1 - MB) \frac{f_C}{f'_C} + Time \times (\sum_{i=0}^{2} \beta_i x_i + \sum_{i=0}^{2} \beta_{ii} x_i^2 \\ &+ \sum_{i=0}^{1} \sum_{k=i+1}^{2} \beta_{ik} x_i x_k + \varepsilon), \end{aligned} \tag{9}$$

where $x_i = \{MB, \frac{f_{\mathrm{C}}}{f'_{\mathrm{C}}}, \frac{f_{\mathrm{M}}}{f'_{\mathrm{M}}}\}$, $0 \leq i \leq 2$. Time is the sampled time at highest frequency setting. $\beta$ are the coefficients and $\varepsilon$ is the intercept, which are distinct values in Equations 7, 8 and 9.

TABLE I: Evaluated Schedulers

| | Sched. | Core Asy. | CPU DVFS | Mem. DVFS | Inter Task | Intra Task | Task Cha. | EPTO |
|---|---|---|---|---|---|---|---|---|
| Grp. 1 | **GRWS** [31] | | | | | | | |
| | **ERASE** [27] | √ | | | | √ | √ | |
| Grp. 2 | **AEQUITAS** [3] | | √ | | √ | | | |
| | **STEER** [9] | √ | √ | | | √ | √ | |
| Grp. 3 | **JOSS** [10] | √ | √ | √ | | √ | √ | √ |
| | **SWEEP** | √ | √ | √ | √ | √ | √ | √ |

### B. Runtime Implementation

We modify the baseline work stealing scheduler by implementing two levels of queues for each core [29]. Per-core task queues store ready tasks and enable load balancing via work stealing. Assembly queues are implemented per core to facilitate intra-task parallelism, where task execution is performed on multiple cores that share resources, such as caches. This is done by dynamically partitioning a task into subtasks and placing them in assembly queues among cores of the same type. Once a core finishes executing a task partition, it can continue fetching other tasks from its own task 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 releases the dependent tasks. The release marks the beginning of a phase and triggers the execution of the SWEEP algorithm. Finally, the released tasks are scheduled to the local queues based on the distribution specified by the algorithm.

## V. Experimental Setup

### A. Experimental Platform

We use NVIDIA's Jetson TX2 development board in our evaluation [30]. It is an asymmetric platform that features two CPU core-clusters (C=2): 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 from 2.04GHz to 0.35GHz ($\mathrm{F_C}$=12). The two clusters can operate at different frequencies while the cores in a cluster operate at the same frequency. The choice of using the NVIDIA Jetson TX2 is also motivated by its support for frequency scaling in the memory controller (MC) and DRAM (LPDDR4). We evaluate different frequency levels for memory ranging from 1.87GHz to 0.80GHz ($\mathrm{F_M}$=5). The integrated `INA3221` power sensor is used to sample the power consumption of CPU and memory. Power samples, obtained once every 5 milliseconds, are used to compute CPU and memory energy consumption. The CPU and memory frequencies are set to the highest before the start of a benchmark and we use the `userspace` linux governor to enable DVFS.

### B. Evaluated Schedulers

We evaluate SWEEP by comparing it to multiple state-of-the-art task-based schedulers. Table I lists all the evaluated schedulers. All schedulers, including SWEEP, are implemented on the XiTAO runtime [32]. We categorize the evaluated schedulers into three groups based on how they leverage DVFS knobs. The first group includes Greedy Random Work Stealing (GRWS) [31] and ERASE [27], which do not exploit any DVFS knobs. The schedulers in the second group, AEQUITAS [3] and STEER [9], only use CPU DVFS as a knob. The schedulers in the third group, JOSS [10] and SWEEP, employ both CPU and memory DVFS knobs.

We evaluate two variants of SWEEP to study its adaptability in the absence of DVFS knob (SWEEP_N.F) and in the presence of just the CPU DVFS knob (SWEEP_C.F) in Section VI. The default JOSS scheduler targets reducing total energy consumption. In addition, it also supports specifying performance constraints (as speedups relative to the execution time targetting energy reduction). Therefore, we also evaluate JOSS variants with manually specified performance speedups ranging from 1.2× to 1.8× and MAXP (that aims to maximise performance relative to the energy reduction scenario).

In our evaluation, we empirically determine the following values as thresholds: (1) 24 (four times the total number of cores on TX2) in the first step for phase classification discussed in Section III-A; (2) 100 attempts before stealing tasks across core-clusters in HP algorithm discussed in Section III-B; (3) 0.5ms threshold to classify tasks as fine-grained in Section III-E; (4) 4s interval length in Section III-D.

### C. Evaluated Benchmarks

We evaluate JOSS using nine benchmarks from the Edge and HPC domains that each comprise a different number of kernels. For some benchmarks, we also evaluate different input sizes and settings. This allows us to evaluate the effectiveness of the schedulers across a broad spectrum of task DAGs.

**Sparse LU Factorization** [33] computes LU matrix factorization for sparse matrices. It includes four kernels: LU0, FWD, BDIV and BMOD. We test it with four different input sizes: 32 and 64 blocks with block sizes 256 and 512, and report the geometric mean numbers in Section VI. **Biomarker Infection Research** [34] is a medical use case that acts as an indicator for differentiating between periprosthetic hip infection and aseptic hip prosthesis loosening with sample size 2. **Fibonacci Sequence** [33] computes numbers using a recursion method with term 55 and grain size 34. **Alya** [35] involves solving partial differential equations based on mesh partitioning. The input size is 200K non-zeros expressed in compressed sparse row format. **Image Classification Darknet-VGG-16 CNN** [36] is a 16-layered deep neural network algorithm that is typical of mobile and edge devices. It is implemented as a fork-join DAG that spans all layers. **Heat Diffusion** [32] is implemented on a 2D grid using one of the iterative numerical methods: 2D Jacobi stencil. We evaluate three distinct input sizes: 2048, 8192, and 16384, and we present the geometric mean values for these three inputs.

**Synthetic Benchmarks** are constructed with a configurable inter-task parallelism (*dop*). The DAG is constructed in such a manner that the root task releases *dop* tasks and the last task further releases *dop* tasks. This process recursively continues till the total number of tasks spawned reaches the user-specified limit. We test with three different kernels: Matrix Multiplication (MM, matrix size 256×256), Memory Copy (MC, matrix size 4096×4096) and Stencil (ST, size 2048×2048). We report the results for different *dop* separately, as they help demonstrate the benefits of classifying phases as HP and LP and using adaptive scheduling in SWEEP.

## VI. Evaluation

We evaluate SWEEP by comparing it with several state-of-the-art schedulers. In Section VI-A, we first evaluate the effectiveness of SWEEP when targeting EDP, which weighs energy consumption and performance equally. Next, we compare the adaptability of SWEEP when targeting other EPTO metrics that prioritizes performance ($ED^2P$) and prioritizes energy consumption ($E^2DP$) respectively in Section VI-B. Finally, we analyze overheads of SWEEP in Section VI-C. All the reported averages are geometric means.

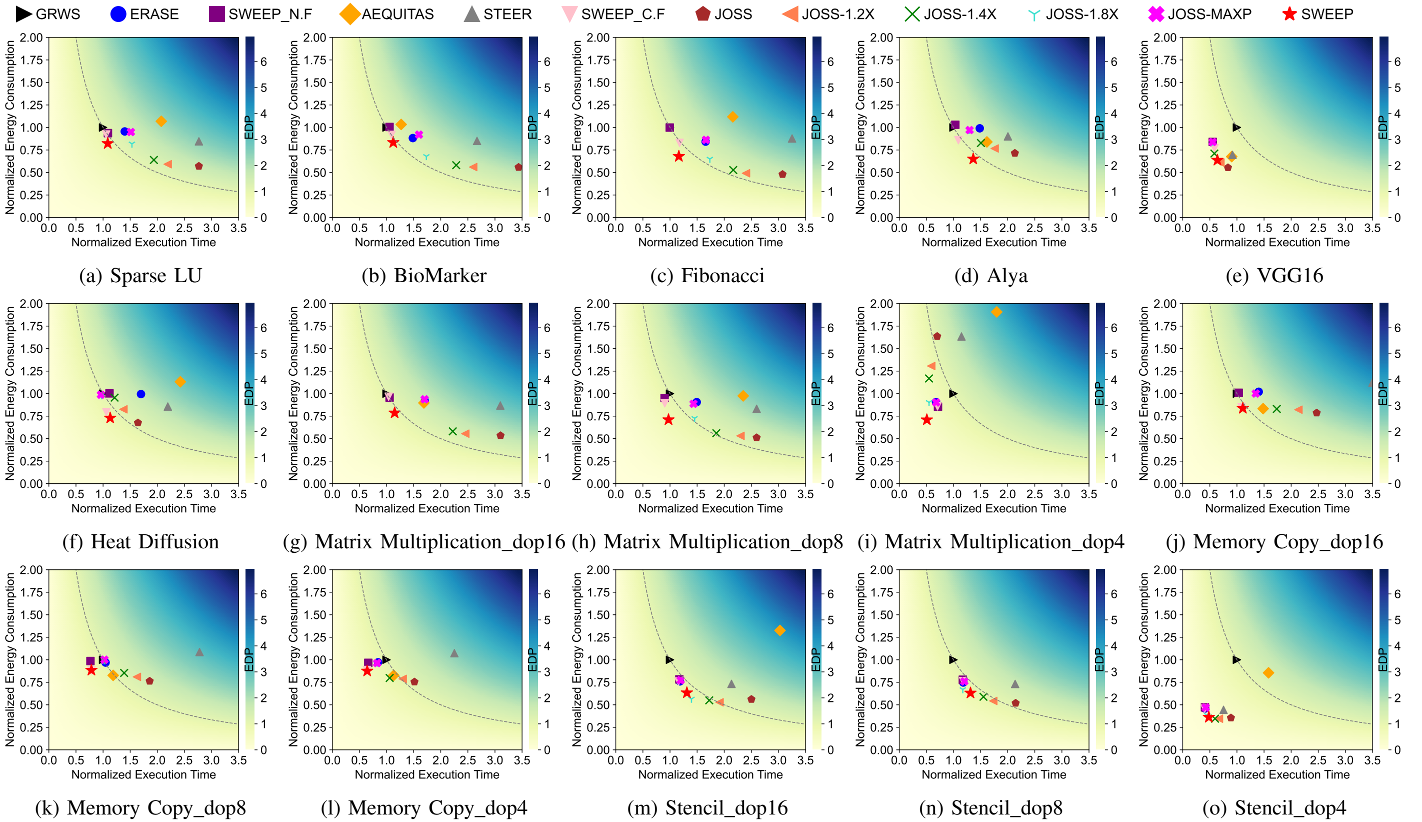
### A. EDP Comparisons to the State-of-the-art

Figure 3 shows execution time and energy consumption for ERASE, AEQUITAS, STEER and variants of JOSS and SWEEP schedulers, normalized to GRWS, for all benchmarks.

Overall, SWEEP achieves 19.9% EDP reduction on average compared to the best performing state-of-the-art scheduler (JOSS-1.8×) and 28.3% reduction than the baseline GRWS across all evaluated benchmarks. The variants of SWEEP are also quite effective. SWEEP_N.F achieves 15.5% and 23.7% EDP reduction compared GRWS and ERASE, respectively, which are from the first category. Similarly, SWEEP_C.F outperforms AEQUITAS and STEER by 110% and 108%, respectively in the second category.

We can observe that, across several applications, schedulers like ERASE, STEER and JOSS, that are effective at reducing energy consumption cause substantial slowdowns thereby impacting EDP. This can be attributed to the fact that these schedulers do not fully exploit the available inter-task parallelism since doing so can increase energy consumption. We can also observe the limited effectiveness of GRWS, especially for VGG16 and for synthetic benchmarks with low *dop*, highlighting its weaknesses. Most of the evaluated schedulers outperform GRWS both in terms of execution time and energy consumption in this context. These together emphasize the importance of having an adaptive task distribution strategy for HP and LP and for utilizing inter-task parallelism in conjunction with other application attributes. Furthermore, the results demonstrate SWEEP's effectiveness at targeting EDP even in the absence of DVFS knobs.

To understand the effectiveness of SWEEP in comparison to other schedulers, we use Matrix Multiplication (MM, *dop*=8). We start by looking at schedulers in the first category. Using




(a) Sparse LU (b) BioMarker (c) Fibonacci (d) Alya (e) VGG16

(f) Heat Diffusion (g) Matrix Multiplication_dop16 (h) Matrix Multiplication_dop8 (i) Matrix Multiplication_dop4 (j) Memory Copy_dop16

(k) Memory Copy_dop8 (l) Memory Copy_dop4 (m) Stencil_dop16 (n) Stencil_dop8 (o) Stencil_dop4

Fig. 3: Execution time and energy consumption for ERASE, AEQUITAS, STEER and variants of JOSS and SWEEP, normalized to GRWS. The color gradient indicates EDP (lower is better) and is calculated by multiplying the execution time numbers on x-axis with the energy numbers on y-axis. Points on the reference dotted line have EDP of one (GRWS at <1.0, 1.0>).

GRWS results in 52% of tasks being executed on the high-performance Denver cores and 48% of tasks being executed on the relatively low-performance A57 cores. Running a MM task on a single Denver core is 2.8× faster than an A57 core, yet a considerable amount of tasks are still executed on the A57 cores due to (four) A57 cores stealing tasks from Denver queues. Such greedy stealing leads to load imbalance, with Denver cores idling for 26% of the total execution time. ERASE estimates the CPU energy consumption of a MM task using performance and CPU power models by sampling task executions on different core types with varying number of cores at given (highest) frequency settings. ERASE results indicate that running a MM task on two Denver cores consumes the least CPU energy and 97% of the tasks in the benchmark are scheduled on the two Denver cores to minimize CPU energy consumption, while the A57 cluster remains idle for the majority of the time. Compared to GRWS, ERASE reduces the total energy by 9.5% but incurs 50% performance slowdown due to its inability to leverage the inter-task parallelism available. In comparison, SWEEP_N.F classifies most of the phases as LP, then applies the LP task distribution algorithm and determines that 5 tasks should be scheduled on Denver and three on A57 both with $\mathrm{NC}_{T_i}$=1.

We next examine the second category of schedulers. AEQUITAS tunes down the frequency of thief cores (that attempt stealing) and increases the frequency of cores with many tasks in the local queue(s). Additionally, each active core adjusts the cluster frequency for a short period in a round-robin way. A57 cores steal several tasks from the Denver cores and this leads to the scheduler reducing their frequency. This slows down the performance of the entire application especially when the (critical) task that releases *dop* tasks gets stolen and is executed on the slower core at a low frequency. This shows that stealing blindly by only relying on heuristics, without considering task characteristics, can result in severe performance slowdown without much energy reduction compared to the baseline. STEER, in contrast, identifies the best configuration for each task using models and determines that running MM tasks on two Denver cores at 1.11GHz is the most energy-efficient option for the CPU. However, STEER does not consider the memory energy and the inter-task parallelism available. STEER reduces total energy by 14% compared to AEQUITAS albeit with 15% decrease in performance. In comparison to STEER, SWEEP_C.F considers both inter-task parallelism and memory energy, leading to improved performance (2.9×) and slightly increased total energy (7.3%) from running 62% tasks on Denver and 38% on A57 with cluster frequency settings of <2.04GHz, 1.88GHz>, respectively.

Finally, we examine schedulers in the third category. JOSS, SWEEP and their variants leverage CPU and memory DVFS

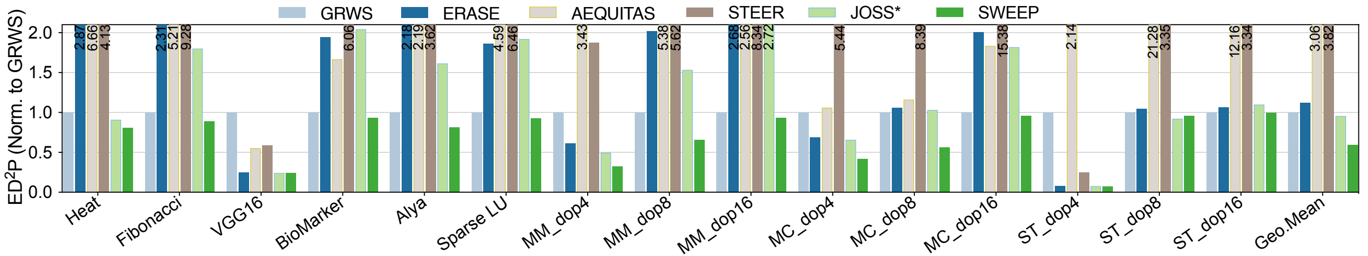



Fig. 4: $ED^2P$ comparisons between GRWS, ERASE, AEQUITAS, STEER, JOSS and SWEEP.

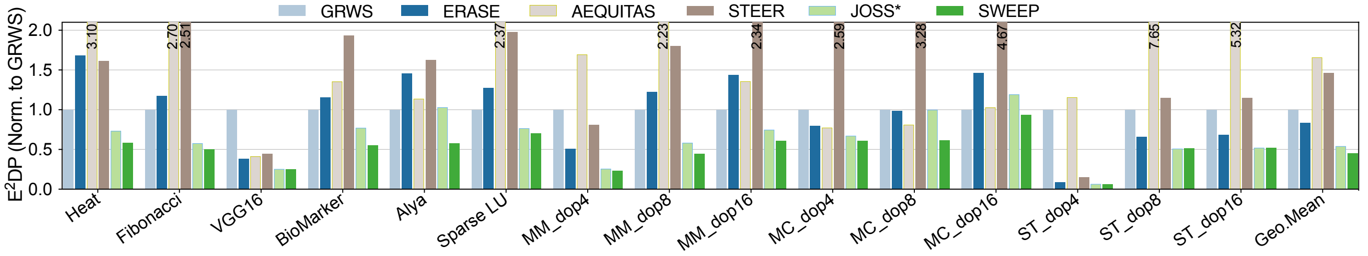



Fig. 5: $E^2DP$ comparisons between GRWS, ERASE, AEQUITAS, STEER, JOSS and SWEEP.

knobs. To mitigate the performance slowdown from targeting energy reduction, JOSS supports user-specified speedups relative to the execution time for energy minimization. The results in Figure 3h demonstrate that EDP decreases as the speedup increases up to 1.4× relative to JOSS. However, the EDP values of JOSS and JOSS with speedups remain inferior to the baseline GRWS. This is because JOSS identifies the best configuration locally for tasks, disregarding the inter-task parallelism during runtime. It ends up running all MM tasks on a single core type, leaving cores of the other type idle and resulting in poor performance. Furthermore, the EPTO exploration scheme in JOSS entails running the application first targeting energy reduction, and then using it as a baseline for specifying performance speedups, which is cumbersome. In contrast, SWEEP allows for the direct specification of a desired EPTO and it determines the most effective task distributions across clusters and the best frequency setting, i.e. with 8 ready tasks running 5 tasks on Denver at 1.88GHz and 3 tasks running on A57 cluster at 1.78GHz and the best memory frequency at 1.06GHz. This demonstrates the benefit of considering all the architectural knobs and application attributes for EDP reduction.

### B. $ED^2P$ and $E^2DP$ Comparisons to the State-of-the-art

In this section, we evaluate the ability of SWEEP to target different EPTO metrics. Figures 4 and 5 show the results from using $ED^2P$ and $E^2DP$ as metrics for SWEEP in contrast to GRWS, ERASE, AEQUITAS, STEER, JOSS. In the interest of space, the results of the best-performing JOSS scheduler among JOSS variants for each benchmark are presented as $JOSS^*$. Overall, the results in Figures 4 and 5 indicate that SWEEP achieves better trade-offs than the state-of-the-art and is able to adapt to various EPTO metrics.

The results in Figure 4 show that SWEEP outperforms the best performing state-of-the-art $JOSS^*$ by 36.4% and is 40.8% better than the second best GRWS on average in terms of $ED^2P$. It is important to note that the JOSS variant that achieves the best results differs between the benchmarks. For instance, employing JOSS-1.8× for running Sparse LU, Biomarker, Fibonacci, Alya, Stencil, MM_dop4, MM_dop8 and MC_dop8 yields the lowest $ED^2P$, while JOSS-MAXP performs the best for Heat Diffusion, VGG16, MM_dop16, MC_dop16, MC_dop4. However, this entails executing JOSS with various speedups to be able to select the best performing one, which is impractical. SWEEP can easily adapt to different target EPTO metrics. Figure 5 shows that SWEEP outperforms the best scheduler $JOSS^*$ by 9.5% and outperforms ERASE by 38.5%, on average, for $E^2DP$. In JOSS, using JOSS-1.4× achieves the lowest $E^2DP$ for the majority of benchmarks.

### C. Overhead and Sensitivity Analysis

Since SWEEP runs concurrently on all the cores, we measure the overhead by accumulating the time spent on various scheduler components, namely phase classification, execution of HP and LP algorithms, and task scheduling, on each core. This is compared to the total execution duration on all cores. Our evaluation shows that the overhead of SWEEP scheduler is 0.03%, on average. We also investigate the sensitivity of SWEEP to the interval length by comparing the default 4s interval to a 1s interval. Our results indicate that using a smaller interval window reduces the effectiveness of SWEEP by 1.5% compared to using a larger interval. This difference can be attributed mainly to frequent changes in configuration that result from recomputing task distributions.

## VII. Related Work

There has been an increasing amount of interest on performance and energy efficiency optimization for task-based parallel applications. HERMES [26] is a work-stealing runtime that slows down thief cores based on their workpath and selects appropriate frequencies based on workload sizes. CATA [5] tunes the per-core DVFS based on task criticality and the available power budget at the moment. Acun et al. [37] use per-core DVFS and adopt an online history-based approach for performance and power predictions by executing with every possible frequency. AAWS [4] targets asymmetric platforms and proposes several techniques (work-pacing, work-sprinting, and work-mugging strategies along with per-core DVFS) depending on each core's state (stealing vs executing).

Several studies have proposed different scheduling techniques targeting performance and energy efficiency of multi-core platforms featuring cluster-based DVFS. Costero et al. [38] present a group of frequency scaling policies based on task criticality and workloads, as well as task scheduling policies for idling or switching off clusters of an asymmetric cluster-based platform based on the workload. However, they evaluate these policies individually. Shafik et al. [2] search for the best concurrency and frequency in a time-interval manner on a cluster-based symmetric platform, without considering the impact of core asymmetry and task characteristics. CHRT [6] is a phase-based scheduler that predicts task placement, cluster frequency, and number of cores for each execution phase. It relies on task criticality for scheduling and phase change detection, and therefore has limited applicability for dynamically unfolding DAGs which do not include explicit criticality information. Furthermore, the performance and power models in CHRT treat all tasks equally regardless of memory-boundness. The aforementioned schedulers do not exploit memory DVFS and intra-task parallelism and cannot support EPTO exploration.

Another line of related work targets energy and/or performance for single-threaded applications or multi-programmed applications instead of task-based applications [12]–[15]. MemScale [12] targets energy consumption in the memory subsystem. They leverage dynamic profiling, performance and power modeling to guide DVFS of memory controller, memory channels and DRAM. CoScale [13] 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. Sundriyal et al [14] 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. David et al. [15] propose an intuitive algorithm that detects memory bandwidth utilization for tuning DVFS of memory subsystem.

## VIII. Conclusion

We propose SWEEP, a task scheduler that leverages architectural knobs (core asymmetry, CPU DVFS and memory DVFS) and application attributes (inter-task parallelism, intra-task parallelism and task characteristics) to facilitate EPTO exploration on multi-core architectures for task-based parallel applications. In a nutshell, SWEEP uses a combination of models and heuristics and works by splitting application execution into HP and LP phases. It uses an adaptive task distribution algorithm, specific to the phase type, that leverages model-based predictions to determine the best task schedule and the DVFS settings for that phase. Moreover, SWEEP is able to flexibly target various EPTO metrics, a feature that is not supported by other proposals. We extensively evaluate SWEEP by comparing it with several state-of-the-art proposals. Our evaluation shows that SWEEP achieves 19.9%, 36.4% and 9.5% reduction on average in terms of EDP, $ED^2P$ and $E^2DP$, compared to the best performing state-of-the-art.


## Acknowledgment

This work has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No. 956702. 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] M. Endrei, C. Jin, M. N. Dinh, D. Abramson, H. Poxon, L. DeRose, and B. R. de Supinski, "Energy efficiency modeling of parallel applications," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2018.

[2] R. A. Shafik, A. Das, S. Yang, G. Merrett, and B. M. Al-Hashimi, "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*, ser. PARMA-DITAM '15, 2015.

[3] H. Ribic and Y. Liu, "Aequitas: Coordinated energy management across parallel applications," in *2016 ACM International Conference on Supercomputing*, 06 2016, pp. 1–12.

[4] C. Torng, M. Wang, and C. Batten, "Asymmetry-aware work-stealing runtimes," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 40–52.

[5] 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, "Cata: Criticality aware task acceleration for multicore processors," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016.

[6] M. Han, J. Park, and W. Baek, "Design and implementation of a criticality- and heterogeneity-aware runtime system for task-parallel applications," *IEEE TPDS*, 2021.

[7] A. Navarro Muñoz, A. F. Lorenzon, E. Ayguadé Parra, and V. Beltran Querol, "Combining dynamic concurrency throttling with voltage and frequency scaling on task-based programming models," in *50th International Conference on Parallel Processing*, ser. ICPP 2021, 2021.

[8] A. Coutinho Demetrios, D. De Sensi, A. F. Lorenzon, K. Georgiou, J. Nunez-Yanez, K. Eder, and S. Xavier-de Souza, "Performance and energy trade-offs for parallel applications on heterogeneous multi-processing systems," *Energies*, vol. 13, no. 9, p. 2409, 2020.

[9] J. Chen, M. Manivannan, B. Goel, M. Abduljabbar, and M. Pericàs, "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)*, 2022.

[10] J. Chen, M. Manivannan, B. Goel, and M. Pericàs, "Joss: Joint exploration of cpu-memory dvfs and task scheduling for energy efficiency," in *Proceedings of the 52nd International Conference on Parallel Processing*, ser. ICPP '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 828–838. [Online]. Available: https://doi.org/10.1145/3605573.3605586

[11] M. Etinski, J. Corbalan, J. Labarta, and M. Valero, "Understanding the future of energy-performance trade-off via dvfs in hpc environments," *Journal of Parallel and Distributed Computing*, vol. 72, no. 4, pp. 579–590, 2012. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731512000172

[12] Q. Deng, D. Meisner, L. Ramos, T. F. Wenisch, and R. Bianchini, "Memscale: Active low-power modes for main memory," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[13] Q. Deng, D. Meisner, A. Bhattacharjee, T. F. Wenisch, and R. Bianchini, "Coscale: Coordinating cpu and memory system dvfs in server systems," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012.

[14] V. Sundriyal and M. Sosonkina, "Joint frequency scaling of processor and dram," *J. Supercomput.*, vol. 72, no. 4, p. 1549–1569, apr 2016.

[15] H. David, C. Fallin, E. Gorbatov, U. R. Hanebutte, and O. Mutlu, "Memory power management via dynamic voltage/frequency scaling," in *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ser. ICAC '11, 2011, p. 31–40.

[16] P. Greenhalgh, "Big.little processing with arm cortex-a15 & cortex-a7," *EE Times*, 2011.

[17] "Apple A17 Pro Chipset," https://www.gsmarena.com/apple_a17_pro_chipset_appears_on_geekbench_performance_cores_clocked_at_378ghz-news-59897.php, 2023.

[18] Wikichip, "Nvidia Tegra Xavier," https://en.wikichip.org/wiki/nvidia/tegra/xavier, 2018.

[19] E. Rotem, Y. Mandelblat, V. Basin, E. Weissmann, A. Gihon, R. Chabukswar, R. Fenger, and M. Gupta, "Alder lake architecture," in *2021 IEEE Hot Chips 33 Symposium (HCS)*, 2021.

[20] OpenMP Architecture Review Board, *OpenMP Application Program Interface. Version 5.0*, OpenMP Architecture Review Board Std., Nov 2018.

[21] "Documentation of starpu," https://files.inria.fr/starpu/doc/starpu.pdf, 2014.

[22] M. Frigo, C. E. Leiserson, and K. H. Randall, "The Implementation of the Cilk-5 Multithreaded Language," in *Proceedings of SIGPLAN 1998*, Jun. 1998.

[23] G. Contreras and M. Martonosi, "Characterizing and improving the performance of intel threading building blocks," in *2008 IEEE International Symposium on Workload Characterization*. IEEE, 2008, pp. 57–66.

[24] P.-É. Polet, R. Fantar, and T. Gautier, "Introducing moldable tasks in openmp," in *OpenMP: Advanced Task-Based, Device and Compiler Programming*, S. McIntosh-Smith, M. Klemm, B. R. de Supinski, T. Deakin, and J. Klinkenberg, Eds. Cham: Springer Nature Switzerland, 2023, pp. 51–65.

[25] M. Pericàs, "Elastic places: An adaptive resource manager for scalable and portable performance," *ACM Trans. Archit. Code Optim.*, vol. 15, no. 2, May 2018. [Online]. Available: https://doi.org/10.1145/3185458

[26] H. Ribic and Y. D. Liu, "Energy-efficient work-stealing language runtimes," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14, 2014.

[27] J. Chen, M. Manivannan, M. Abduljabbar, and M. Pericàs, "ERASE: Energy efficient task mapping and resource management for work stealing runtimes," *ACM Trans. Archit. Code Optim.*, mar 2022.

[28] B. Goel, *Measurement, Modeling, and Characterization for Energy-efficient Computing*. Chalmers University of Technology, 2016.

[29] M. Pericàs, "Elastic places: An adaptive resource manager for scalable and portable performance," *ACM Trans. Archit. Code Optim.*, vol. 15, no. 2, pp. 19:1–19:26, May 2018. [Online]. Available: http://doi.acm.org/10.1145/3185458

[30] "Jetson tx2 module," https://developer.nvidia.com/embedded/jetson-tx2, 2017.

[31] R. D. Blumofe, "Scheduling multithreaded computations by work stealing," *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pp. 356–368, 1994. [Online]. Available: https://api.semanticscholar.org/CorpusID:5428476

[32] "XiTAO runtime," https://github.com/CHART-Team/xitao.git, 2018.

[33] A. Duran, X. Teruel, R. Ferrer, X. Bofill, and E. Parra, "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.

[34] "Biomarker discovery," https://legato-project.eu/use-cases/healthcare, 2020.

[35] H. Guillaume and V. Mariano, "Alya application," https://www.bsc.es/research-development/research-areas/engineering-simulations/alya-high-performance-computational.

[36] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[37] B. Acun, K. Chandrasekar, and L. V. Kale, "Fine-grained energy efficiency using per-core dvfs with an adaptive runtime system," in *2019 Tenth International Green and Sustainable Computing Conference (IGSC)*, 2019.

[38] L. Costero, F. D. Igual, K. Olcoz, and F. Tirado, "Energy efficiency optimization of task-parallel codes on asymmetric architectures," in *2017 International Conference on High Performance Computing Simulation (HPCS)*, July 2017, pp. 402–409.