CHALMERS
UNIVERSITY OF TECHNOLOGY

ERASE: Energy Efficient Task Mapping and Resource Management for Work Stealing Runtimes

ERASE: Energy Efficient Task Mapping and Resource Management for Work Stealing Runtimes

JING CHEN, MADHAVAN MANIVANNAN, MUSTAFA ABDULJABBAR, and MIQUEL PERICÀS, Chalmers University of Technology, Sweden

Parallel applications often rely on work stealing schedulers in combination with fine-grained tasking to achieve high performance and scalability. However, reducing the total energy consumption in the context of work stealing runtimes is still challenging, particularly when using asymmetric architectures with different types of CPU cores. A common approach for energy savings involves dynamic voltage and frequency scaling (DVFS) wherein throttling is carried out based on factors like task parallelism, stealing relations, and task criticality. This article makes the following observations: (i) leveraging DVFS on a per-task basis is impractical when using fine-grained tasking and in environments with cluster/chip-level DVFS; (ii) task moldability, wherein a single task can execute on multiple threads/cores via work-sharing, can help to reduce energy consumption; and (iii) mismatch between tasks and assigned resources (i.e., core type and number of cores) can detrimentally impact energy consumption. In this article, we propose EneRgy Aware SchedulEr (ERASE), an intra-application task scheduler on top of work stealing runtimes that aims to reduce the total energy consumption of parallel applications. It achieves energy savings by guiding scheduling decisions based on per-task energy consumption predictions of different resource configurations. In addition, ERASE is capable of adapting to both given static frequency settings and externally controlled DVFS. Overall, ERASE achieves up to 31% energy savings and improves performance by 44% on average, compared to the state-of-the-art DVFS-based schedulers.

CCS Concepts: • **Computing methodologies → Parallel computing methodologies**; • **Computer systems organization → Embedded systems;**

Additional Key Words and Phrases: Energy, task scheduling, resource management, work stealing, runtimes

ACM Reference format:
Jing Chen, Madhavan Manivannan, Mustafa Abduljabbar, and Miquel Pericàs. 2022. ERASE: Energy Efficient Task Mapping and Resource Management for Work Stealing Runtimes. *ACM Trans. Arch. Code Optim.* 19, 2, Article 27 (March 2022), 29 pages.
https://doi.org/10.1145/3510422

This work has received funding from the European Union Horizon 2020 research and innovation programme under grant agreement No. 780681 (https://legato-project.eu/). This work has also 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. The computations were enabled by resources provided by the Swedish National Infrastructure for Computing (SNIC), partially funded by the Swedish Research Council through grant agreement No. 2018-05973 (https://www.vr.se/).
Authors' address: J. Chen, M. Manivannan, M. Abduljabbar, and M. Pericás, Chalmers University of Technology, Gothenburg, Sweden; emails: chjing@chalmers.se, madhavan@chalmers.se, musabdu@chalmers.se, miquelp@chalmers.se.

1544-3566/2022/03-ART27 $15.00
https://doi.org/10.1145/3510422

1 INTRODUCTION

Reducing energy consumption of multiprocessor systems is a major requirement to enhance battery life, reduce the cost of cooling, and overall improve system's scalability and reliability.

Asymmetric multiprocessor systems featuring different types of cores with different performance and power characteristics have been introduced to reduce the total energy consumption while still enabling high performance. Examples of such systems include ARM's big.LITTLE architecture [23], Intel's hybrid Lakefield CPU [40], and Apple's A12-A14 bionic processors [42]. However, effectively scheduling parallel applications on asymmetric multiprocessors remains an open challenge. Random work stealing is a well-known approach for scheduling task-parallel applications targeting load balancing and scalability [7, 12] that has been implemented in several production runtimes, such as Cilk [20], TBB [15], and OpenMP's explicit tasks [34]. In this article, we show that random work stealing does not produce energy efficient schedules, particularly on asymmetric architectures. The two principles on which random work stealing is based, namely the work-first principle and random victim selection, are neither aware of task characteristics (such as the task's arithmetic intensity) nor of the cores' performance and energy profiles. As a result, random work stealing could detrimentally impact energy consumption and performance of parallel applications on asymmetric platforms.

A few recent proposals have targeted energy efficient task scheduling by introducing extensions on top of work stealing [10, 38, 39, 44]. The main idea is to throttle **dynamic voltage and frequency scaling (DVFS)** depending on factors like task parallelism, stealing relations and/or task criticality. The reliance on DVFS limits their applicability due to several reasons. First, DVFS switching overheads limit the potential of schemes that leverage per-task throttling. Studies have shown that DVFS transition delay is around 100 microseconds [10, 35, 41, 44]. Exposing fine-grained task parallelism, with task execution times being in the order of microseconds, enables scalability and load balancing in multicore and manycore architectures [29, 46]. However, applying DVFS on a per-task basis for energy savings would incur unacceptably large overheads for fine-grained tasking.

Second, per-core DVFS control entails significant hardware cost and leads to low conversion efficiency due to the overheads introduced by the on-chip voltage regulators [22, 28]. Many systems turn to cluster-level DVFS where voltage settings can only be controlled for a subset of cores (referred to as core-cluster). This applies both to embedded platforms such as the ODROID XU-3 [2] and NVIDIA Jetson products [1] and to specific server processors from Intel and AMD [17, 43]. With cluster-level DVFS, however, multiple tasks attempting frequency changes within the same cluster will result in destructive interference, since the decision taken to reduce energy consumption of a task mapped to a specific core can affect concurrently running tasks on the same cluster. A similar problem occurs in processors with multi-threaded cores, where one thread's DVFS actions impact the performance of other threads running on the same core.

A third concern with application-level DVFS techniques is that DVFS is most often not under the control of the application. In a multiuser OS, many processes share a subset of cores, and DVFS control is commonly restricted to the kernel [26], the system administrator [8], or power management frameworks such as GEOPM [19]. Consequently, an energy efficient runtime designed to be reactive to both given static frequency settings and externally controlled DVFS, instead of relying on actively changing DVFS settings, has the potential to be a more general solution to the problem of energy-aware scheduling.

In this article, we address the problem of reducing the total energy consumption of task-based applications running on symmetric and asymmetric multicore platforms for given core frequencies. We focus on designing an intra-application task scheduler based on the most popular choices for task-based runtimes and multicore organizations, namely work-stealing [7] and core-clusters with

common frequency [17, 43]. Our proposal, **EneRgy Aware SchedulEr (ERASE)**, leverages the following novel insights:

(1) *Task moldability* [30, 36, 47, 48], i.e., executing a single task on multiple threads/cores via work-sharing can help to reduce the energy consumption by decreasing resource oversubscription or making use of otherwise idle resources. Moldability helps to exploit the internal parallelism within a single task by supporting 1:M mapping (i.e., a single task to multiple threads/cores) in addition to the traditional 1:1 mapping (i.e., a single task to a single thread/core). To enable moldable execution, the runtime partitions the task's workload and dynamically maps them to M ($\geq$1) resources.
(2) *Task-type aware execution* exploits information about different task characteristics (such as arithmetic intensity) to reduce energy consumption. We explore a combination of task moldability and task-type awareness to figure out the optimal resource assignment for minimizing the total energy consumption. Specifically, we identify the optimal execution place (core type, number of cores) for each task, since mismatch between task properties and assigned resources on asymmetric architectures can negatively impact the total energy consumption and performance.

The design goals of ERASE are (i) simplicity, to enable low overhead implementation, and (ii) adaptability, to enable its effective application across given static frequency settings and externally controlled DVFS on multicore platforms. In a nutshell, ERASE reduces energy consumption of an application by attempting to execute each task in the application with the lowest possible energy consumption. ERASE comprises four essential components: (i) *online performance modeling*, (ii) *power profiling*, (iii) *core activity tracing*, and (iv) *a task mapping algorithm*. Online performance modeling monitors task execution and continuously updates a performance model that provides performance predictions for incoming tasks. Power profiling provides the runtime with estimates of CPU power consumption with respect to different resource configurations (i.e., number/type of cores) for given core frequencies. Core activity tracing continuously tracks the activities (i.e., work stealing attempts) and status (i.e., active or sleep) of each core and infers the instantaneous task parallelism, which gives the task mapping algorithm a hint for attributing power consumption to concurrently running tasks accurately. Finally, the task mapping algorithm integrates the aforementioned information and guides scheduling decision for each task based on the energy estimates of running the task on different resource configurations.

The results obtained on two different platforms (i.e., NVIDIA Jetson TX2 and Tetralith) show that our proposal is a general approach for reducing the total energy consumption of executing task-based applications and the proposed models in ERASE achieve reasonable accuracy. In summary, the main contributions of this article are as follows:

- We propose ERASE: an energy efficient task scheduler that combines power profiling, performance modeling and core activity tracing for energy efficient mapping (i.e., choosing the cluster) and resource management (i.e., selecting the number of cores per task). The proposal exploits the insights of task moldability, task-type awareness, and instantaneous task parallelism detection for guiding scheduling decisions to reduce the total energy consumption of parallel applications.
- We describe how to integrate ERASE on top of work stealing runtimes, using the XiTAO [5] runtime for the prototype implementation.
- We compare ERASE to state-of-the-art scheduling techniques on top of the runtime and the evaluation shows that ERASE achieves up to 31% energy savings and outperforms the state-of-the-art by 44% on average.

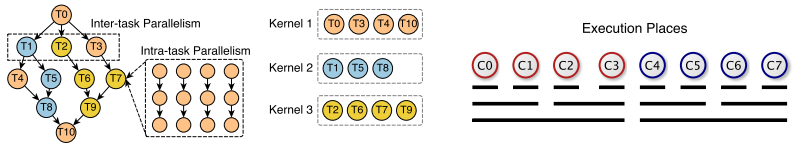


Fig. 1. The illustration of a task DAG, kernels, and possible execution places.

The remainder of this article is organized as follows. Section 2 presents the background and motivation for the proposed approach. Section 3 formulates the problem of energy efficient task scheduling and then discusses the design of ERASE in detail. Section 4 describes how ERASE is integrated on top of a work stealing runtime. Section 5 introduces the experimental setups, which are used for evaluation in Section 6. Section 7 presents related work, and Section 8 concludes the work.

2 BACKGROUND AND MOTIVATION

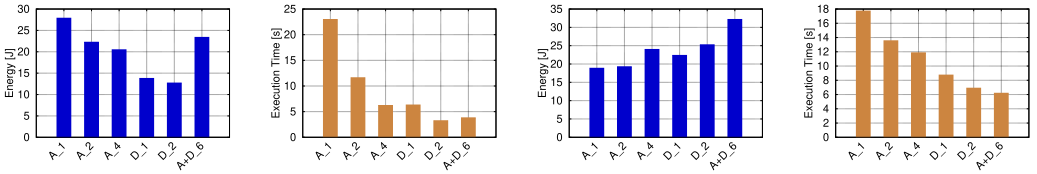
We first provide the necessary background in Section 2.1. In Section 2.2, we study the impact of task moldability and task-type aware scheduling on energy consumption to motivate the approach adopted by ERASE. Next, we show that schedulers that leverage per-core DVFS to reduce energy consumption have limited effectiveness on architectures with cluster-level DVFS, in Section 2.3.

2.1 Background

In this article, we focus on parallel applications that are expressed as task **directed acyclic graphs (DAGs)**, where the nodes denote tasks and the edges denote the dependencies between tasks. Figure 1 shows an example task DAG. A DAG implicitly expresses the inter-task parallelism (i.e., the independent tasks that can be executed in parallel). We allow each task to furthermore contain internal parallelism; in such a case, moldability can be used to execute a single task over multiple resources. Given our focus on HPC applications, tasks are commonly used to represent numerical kernels. Each kernel consists of one or multiple tasks, and the routines executed by different tasks of the same kernel are identical. In Figure 1, the task DAG contains three different kernels. Given a platform, an *execution place* refers to execution resources that are assigned to each task. They commonly share tightly coupled resources, such as cache levels, memory channels, NUMA nodes, and so on, to reduce interference. Figure 1 shows an example of supported execution places on an asymmetric platform with two clusters and four cores in each cluster.

2.2 Impact of Task-type Awareness and Task Moldability on Asymmetric Platforms

To showcase the impact of task-type aware execution and task moldability on energy consumption and performance, we select two representative benchmarks: Matrix Multiplication (MatMul) (compute-bound) and *Memory Copy* (Copy) (memory-bound) and run on the NVIDIA Jetson TX2 board. It consists of two CPU clusters: Denver and A57. The former comprises a high-performance dual-core NVIDIA Denver CPU, while the latter comprises a comparatively lower-performance quad-core ARM CPU. Additional details about the benchmarks and the platform are provided in Section 5. Figure 2 presents energy consumption and execution time when running the benchmarks with different configurations. The labels in the x-axis follow the format `x_y`, where `x` denotes the cluster name and `y` denotes the number of cores in the cluster for executing each task in the benchmark. When running with a specific configuration, the rest of the cores are disabled to obtain an accurate measure of the energy consumption. For instance, A_1 represents the case when tasks execute only on a single A57 core and all the other cores are disabled, while A_4



(a) MatMul - Energy (b) MatMul - Execution Time (c) Copy - Energy (d) Copy - Execution Time

Fig. 2. Energy consumption and execution time of MatMul and Copy when running with different configurations on Jetson TX2.

represents the case when a benchmark executes only on four A57 cores (with both the Denver cores disabled) and each task executes on all four cores to exploit task moldability. The label A+D_6 represents the configuration where each task executes on all six cores.

Figure 2(a) shows that executing a matrix multiplication task on high-performance Denver core(s) consumes less energy compared to that on lower-performance A57 core(s). This can be attributed to the 3.5× increase in Instruction Per Cycle in a Denver core compared to an A57 core. It can be seen that energy consumption is lowest when running on two Denver cores, i.e., D_2, which achieves 8% energy savings in comparison to using a single Denver core (D_1). This is because running with D_2 achieves linear speedup of 2, while the power consumption of D_2 only increases by 87% comparing with D_1. Additionally, it is clear in Figure 2(b) that the most energy efficient configuration (D_2) is also the fastest, which illustrates that task moldability can help to reduce energy consumption and improve performance simultaneously. The results also show that D_2 achieves 46% energy savings and improves performance by 15% in comparison to A+D_6, which indicates the importance of selecting the appropriate core type and number of resources over simply attempting to use all available resources.

Figure 2(c) shows that, for memory copy, running on a single A57 core consumes less energy than running on a Denver core. This can be attributed to the fact that running on a fast out-of-order core, i.e., Denver, does not provide sufficient performance benefit to compensate the additional power costs. However, unlike matrix multiplication, the energy consumption of A_2 and A_4 is higher than A_1. This is because the power cost of using additional A57 cores cannot be offset by performance benefits. Hence, the configuration that consumes the least energy is the slowest. Additionally, the results in Figure 2(d) show that attempting to use all available resources achieves the lowest execution time but leads to the highest energy consumption.

In summary, these results indicate that the best configuration for minimizing energy consumption is task-type dependent and that exploiting task moldability can help to further reduce energy consumption (e.g., for matrix multiplication in this case), which motivate us to consider task-type awareness and task moldability together when designing energy efficient work stealing runtimes.

2.3 Impact of Per-core DVFS Scheduling on Cluster-level DVFS

In this section, we study (1) the impact of task granularity on energy when using DVFS and (2) how much energy efficiency is lost when a runtime assuming per-core DVFS scheduler is deployed on a platform that features only cluster-level DVFS. The results motivate us to consider the negative impact of DVFS on fine-grained task DAGs and to devise scheduling strategies that take the platform DVFS into account. To showcase the effects, we evaluate the **Criticality Aware Task Acceleration scheduler (CATA)** [10] on Jetson TX2. CATA dynamically tunes the frequency of each core based on incoming task criticality and the available power budget at the moment to improve performance and energy efficiency, i.e., **Energy-Delay Product (EDP)**. We reproduce

Table 1. EDP and Execution Time Comparisons among CATA, RWS, and ERASE on Jetson TX2

Application		MatMul				Copy				Stencil			
Parallelism		dop = 2	dop = 4	dop = 6	dop = 8	dop = 2	dop = 4	dop = 6	dop = 8	dop = 2	dop = 4	dop = 6	dop = 8
EDP	CATA	27138.5	16832.6	11270.4	7693.9	5362.2	2486.8	2158.7	2089.2	2275.0	1990.7	1062.0	736.5
	RWS	15723.0	6898.1	4267.4	3315.4	7266.2	4230.4	3932.3	2937.9	5648.8	2475.2	1828.5	1377.1
	ERASE	1979.9	1463.4	1433.4	1338.6	2140.5	1886.2	2096.6	2020.7	310.0	260.8	280.5	262.2
Execution Time[s]	CATA	178.59	114.50	85.57	66.15	69.83	36.83	31.10	29.71	34.19	28.52	20.43	17.56
	RWS	102.23	56.84	37.97	28.93	108.16	61.16	53.06	40.94	57.96	30.74	24.54	19.57
	ERASE	25.85	19.76	19.41	19.31	36.74	31.83	32.37	31.48	9.87	8.41	9.00	8.46

CATA on the Jetson TX2 with two frequency levels (2.04 GHz, 0.35 GHz) and set a power cap whereby only one of the clusters on TX2 can operate in maximum frequency.

Table 1 compares the EDP and execution time of CATA with **Random Work Stealing (RWS)**, a random work stealing scheduler that we use as the baseline and the scheduler proposed in this work (ERASE). The latter is given to provide an idea of the potential energy reductions that can be achieved when a scheduler is aware of the platform DVFS characteristics. All three schedulers are evaluated with three synthetic benchmarks featuring a **degree configurable parallelism (*dop*)** that ranges from 2 to 8. Additional details about the schedulers, the benchmarks and the platform are provided in Section 5.

The results show that when CATA is applied for fine-grained task DAGs on a platform that features only cluster-level DVFS, it does not consistently achieve the best performance or lowest EDP, even when compared to a simple RWS strategy. The comparison with ERASE furthermore shows that large improvements are possible. The limited benefit from CATA can be attributed to the following reasons: (1) DVFS control, triggered from user space, leads to reconfiguration serialization, and becomes a performance bottleneck; (2) when per-core DVFS is employed on platforms that only support cluster-level DVFS, it leads to interference in DVFS settings among cores in the same cluster; and (3) fine-grained tasks further exacerbate the DVFS overheads due to the reconfiguration serialization issue. Overall, these results indicate that DVFS-aware task schedulers need to be carefully designed to avoid DVFS reconfiguration overheads and the interference that results from multiple cores requesting different frequencies.

3 ENERGY EFFICIENT TASK SCHEDULER

We first discuss the problem of minimizing the total energy consumption when executing a task DAG. Next, we propose our energy efficient task scheduler (ERASE) and provide an overview of the essential modules that constitute ERASE, followed by the description of all modules in detail and an explanation of how they interact with each other to enable energy efficient task scheduling. Finally, we present the complexity and overheads analysis for ERASE.

3.1 Problem Formulation

We assume a platform that includes m core-clusters $\{\beta_1, \ldots, \beta_m\}$, where each cluster comprises p cores of the same type. With task moldable execution, each task can be executed with variable number of cores of the same type $\in \{\gamma_1, \gamma_2, \ldots, \gamma_p\}$. Given a task DAG that includes τ tasks $\{T_0, \ldots, T_{\tau-1}\}$, the problem is to minimize the total energy consumed for executing the entire DAG on this platform.

3.2 ERASE Overview

ERASE assumes that the total energy consumption (TE) of running the DAG on such a platform consists of two components: the energy consumed for running each task and the energy consumed

in the idle periods due to the dependencies between tasks. Thus, the problem of minimizing the total energy consumption of executing a task DAG can be partitioned into two parts: (1) minimizing the energy consumed by each task and (2) minimizing the energy consumed in idle periods, as shown in Equation (1),

$$min\ TE = \sum_{i=0}^{\tau-1} min\ E_{T_i} + \sum_{j=1}^{m}\sum_{k=1}^{p} min\ E_{idle}. \tag{1}$$

Minimizing the energy consumed by each task. We show in Section 2.2 that with different execution places, the energy consumption varies greatly. Thus, to minimize the energy consumption when running a task, it is necessary to figure out the best execution place for the task. There are two knobs at runtime for the configuration selection: (i) the choice of cluster and (ii) the number of cores. Therefore, for a specific task T_i, energy minimization can be expressed as

$$\begin{aligned} min\ E_{T_i} = min(&\underbrace{E_{T_i}(\beta_1,\gamma_1), E_{T_i}(\beta_1,\gamma_2), \ldots, E_{T_i}(\beta_1,\gamma_p)}_{\beta_1}, \underbrace{E_{T_i}(\beta_2,\gamma_1), \ldots, E_{T_i}(\beta_2,\gamma_p)}_{\beta_2}, \\ &\underbrace{E_{T_i}(\beta_3,\gamma_1), \ldots, E_{T_i}(\beta_m,\gamma_p)}_{\beta_3\ to\ \beta_m}). \end{aligned} \tag{2}$$

According to Equation (2), the runtime needs to evaluate all different execution places to obtain the configuration that consumes the least energy for each task. For each possible configuration, energy consumed by a task is the product of execution time and power consumed during the task execution. It is crucial that the runtime can predict the execution time and the power consumption before execution. Thus, designing a scheduling system that achieves energy minimization goal requires a performance model for execution time prediction and a power model for power prediction, respectively.

Minimizing the energy consumed in idle periods. In a work stealing runtime, cores that become idle (i.e., because there are no tasks in their work queues) attempt to steal tasks from other queues for load balancing. However, when not enough ready tasks are available, the idle cores continuously attempt work stealing without success, which can lead to energy waste. The problem of minimizing energy consumption during idle periods requires the runtime to be able to detect the cores' instantaneous utilization and dynamically put idle cores to sleep. Meanwhile, during the sleeping periods, it is possible that ready tasks are assigned to the idle cores or that there are tasks available for work stealing. If the idle cores cannot be woken up in time to deal with the tasks, then it could cause performance degradation and energy increase. The challenge is to determine the sleep duration such that energy consumption during the period is minimized with minimal performance impact. This is described in Section 4.

To address the aforementioned challenges, we propose an energy efficient task scheduler called ERASE. Figure 3 highlights the four essential modules in ERASE. To enable per-task energy estimation, the scheduler predicts the performance and power consumption when a task is mapped onto an execution place. Performance prediction is provided by the online performance modeling module, as discussed in Section 3.3. The information about power consumption estimation is obtained through power profiling as discussed in Section 3.4. The core activity tracing module, discussed in Section 3.5, dynamically tracks how many cores are effectively utilized and how many are idle at any given moment, which provides the instantaneous task parallelism. The task mapping algorithm, presented in Section 3.6, leverages information provided by the aforementioned modules to estimate the energy consumption on different execution places and map each task on the execution place that consumes the least energy. These modules cooperate with each other and enable the runtime to determine the best execution place for each task for the total energy savings.

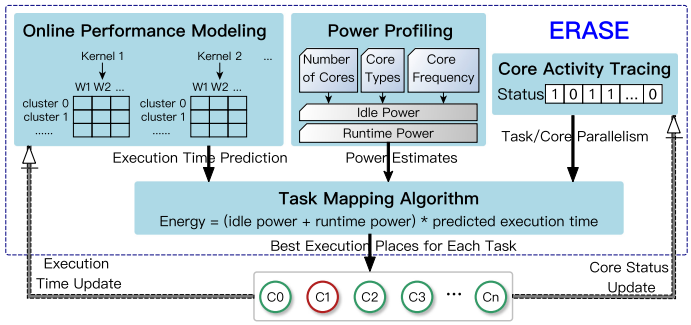


Fig. 3. An overview of ERASE comprising four modules. C_x denotes the core id, W_y denotes the possible resource widths. In status, "1" denotes that C_x is in active state while "0" denotes the core is in sleep state.

ALGORITHM 1: Online Performance Modeling Algorithm

```
1:  Initialize all entries to zeros
2:  for i in cluster indexes do
3:      for j in resource widths of cluster i do
4:          temp_time = task->access_performance_table(i, j)
5:          if temp_time == 0 then
6:              task->leader = starting core id of cluster i + (rand() % (number of cores / j)) × j
7:              task->width = j
8:              return
9:          end if
10:     end for
11: end for                                                        } Phase 1: Training
12: Time_prediction = task->access_performance_table(i, j)         ▷ Phase 2: Prediction given cluster i, resource width j
13: if lc is leader core && lc ∈ cluster i then
14:     task->update_performance_table(i, j, execution_time)       } Phase 3: Update corresponding entry after execution
15: end if
```

In ERASE, each possible execution place is expressed as a tuple (leader core, resource width). The *leader core* denotes the starting core identifier (id) of the allocated resources. The *resource width* denotes the number of cores assigned to a task to enable moldable execution.

3.3 Online Performance Modeling

The online performance modeling module enables ERASE to predict the execution time of a task when mapped on an execution place. In ERASE, we adopt a history-based model since it is a simple yet effective method for estimating performance and detecting performance heterogeneity and system interference for fine-grained tasking applications [11, 13, 24, 37]. The online performance module is implemented as a look-up table for each kernel. The size of each look-up table equals the product of the number of clusters and the number of available resource widths on the platform. For the example shown in Figure 1, ERASE utilizes a 2×3 performance look-up table for each of the kernels (3 in total) to predict the performance on the Jetson TX2 platform. This module only requires information about the number of clusters and their organization into core-clusters with shared caches, which can readily be obtained using tools like *hwloc*.

Algorithm 1 shows the three phases in the module: training, prediction, and update. In training phase, all the entries of a look-up table are initialized to zeros to guarantee that all execution places are visited at least once (lines 1–11). The execution time prediction for a task is obtained by

accessing the corresponding value of the specific execution place in the table as shown in line 12. After a task completes the execution, the corresponding entry in the table is updated by the leader core of a task to reduce cache migrations (lines 13–15). Each table is organized to fit into cache lines where each cluster only accesses one cache line indexed with the cluster id, which helps avoid false sharing. To avoid large fluctuations in the values in the table, which could result from short isolated events, entries are updated by computing a weighted average of the previous and the new value.

The online performance module is capable of reacting to the externally controlled DVFS changes. The performance look-up tables record the CPU cycles and the execution time consumed during task execution, then speculate the frequency from cycles and execution time. By comparing to the previous frequency record, the online performance module is able to detect the frequency change instantly. Once the frequency change is detected, the module will reset the look-up table entries to zeros and retrain the model. We evaluate the overhead of retraining the performance modeling tables and the result shows that with fine-grained tasking retraining introduces little overhead (see Section 6.4 for details). Nevertheless, for coarse-grained tasking, the scheme would have limited effectiveness due to larger training overhead, since the execution time spent on training phase occupies higher percentage of the total execution time.

In comparison to other complex models based on an offline linear regression or a trained neural network, the history-based model can be implemented without any offline training and with low overhead, but one has to take into account that its performance depends on a fine-grained partitioning of the application into tasks. The scheme has the following advantages. First, the table size is relatively small. For instance, the training phase on the TX2 platform only requires five tasks' execution times to fill in all the entries of a table. Second, in a parallel application, each kernel generally contains a large number of tasks and each task of the kernel contributes to the update of the table. Thus, the number of tasks for training is negligible in comparison to the total amount of tasks in an application. We show that the module provides reasonably accurate predictions for different applications and across multiple platforms (refer to Section 6.5 for details).

3.4 Power Profiling

The goal of this module is to estimate power consumption when scheduling a task on an execution place. A simple approach to estimate power consumption is to run a single task (one from each kernel) on an execution place and profile the power consumption values obtained from the power sensors as reference. Since the number of tasks is usually much higher than the number of possible execution places (as is the case with long-running applications and/or with fine-grained tasking), the online profiling approach would have a relatively low overhead. However, there are limitations with power measurement support on existing systems that preclude the use of online profiling approach described above.

First, the power measurement supports, that are available on the platforms we present in Section 5.1, provide composite power consumption values for the entire chip/socket instead of values at the per-core level. This makes it difficult to isolate the power consumption for a single task that executes on a subset of cores, especially when there are concurrently running tasks. Second, our analysis of the power measurement overheads on Jetson TX2 shows that it takes tens of milliseconds before the power sensor begins to accurately capture the change in power consumption since the values reported are averaged over a time window. With the execution time of fine-grained tasks being in the order of microseconds, which is much lower in comparison, the online power profiling approach becomes ineffective. Due to these limitations, we adopt a power characterization approach based on profiling representative microbenchmarks to help estimate the

power consumption. This power profiling step just needs to be done once for a specific platform, for example at install-time or boot-time, and has no impact on execution time.

Task Type Detection. We consider the tasks of an application to be broadly grouped into one of the three representative types in ERASE: compute-bound, memory-bound, and cache-intensive. Most modern architectures provide **Performance Monitoring Counters (PMC)** to monitor the activity of cores, such as cpu-cycles, cache-misses, cache-references, and number of retired instructions per cycle. The availability of PMCs differs between different platforms. In this work, we utilize the **arithmetic intensity (AI)**, i.e., the ratio of floating point operations over memory traffic [33], to determine online the task type category for a specific kernel. We record the number of cycles and the last level cache misses during the task execution by using the Linux system call *perf_event_open()* [31]. The two performance counters required to implement this calculation are widely available on the majority of platforms. AI is calculated as follows:

$$AI = \frac{number\ of\ cycles \times FLOPs/cycle}{number\ of\ cache\_misses \times 64\ bytes}. \tag{3}$$

We profile a set of configurable synthetic microbenchmarks (refer to Section 5.3 for more details) offline and record the corresponding PMCs and the average power consumption during the execution when running them with different number/type of cores and frequencies. After calculating the AI values for all microbenchmarks, we employ a k-Nearest Neighbor algorithm to cluster them into three groups by computing the euclidean distances of the AIs and finally choose an AI threshold belonging to the frontier region between the clusters. The averaged power values from each group are used as power estimates once a task is classified as belonging to a certain task type. Further analysis about the accuracy of this approach is presented in Section 6.5.

The inputs of the power profiling step are the number of cores, core types, and available frequency levels on the platform, as shown in Figure 3. During power profiling, two different components are measured for different frequencies: *idle power* and *runtime power*. Idle power of a platform denotes the power consumed even when no useful work is being performed. It can be simply measured when all the cores are online but do not perform any useful activity. Runtime power is defined as the additional power consumed when performing useful work. Note that runtime power only represents the additional power consumed to do useful work and does not include idle power. Thus, the power consumption for a task is estimated by summing up the runtime power and the idle power consumed by this task. The runtime power estimate for a task executing on an execution place can be simply obtained from the power profiling results. However, idle power obtained with power profiling is the total idle power of the entire chip/cluster. It cannot be attributed to the task directly because this is shared between concurrently running tasks. In other words, each concurrently running task shares a portion of idle power at any given moment, and attributing the entire idle power from power profiling to a single task may lead to inaccurate energy estimation. This observation motivates the need for core activity tracing module, described in Section 3.5, which addresses the problem by continuously tracking the number of active cores and giving the task mapping algorithm a hint for attributing the idle power consumption to concurrently running tasks accurately.

3.5 Core Activity Tracing

In this section, we introduce the core activity tracing module, which solves the problem of idle power sharing described in Section 3.4. The module maintains an integer array *status[]* that stores the instantaneous core states, as shown in Figure 3. The module continuously tracks the number of active cores doing useful work and dynamically updates the *status* entries if cores become idle. To correctly attribute the idle power among concurrently executing tasks, it is essential to detect

ALGORITHM 2: Task Mapping Algorithm

```
1: Input: ready tasks that are woken up by the core; Output: best execution place for each task
2: for each ready task do
3:     for each execution place do        ▷ Randomly select an execution place from each (cluster, width) configuration
4:         NumActCores_A = status[0]+ status[1] + · · · + status[a-1]        ▷ Cluster A includes a cores
5:         NumActCores_B = status[a]+ status[a+1] + · · · + status[a+b-1]        ▷ Cluster B includes b cores
6:         idleP_temp_A = (NumActCores_B > 0)? idleP_A : idleP_chip
7:         for core i in current evaluated execution place do        ▷ Check if all cores in the execution place are all active
8:             if status[i] is 0 then
9:                 NumActCores_A++        ▷ Selected cores are currently sleeping but could later be used for execution
10:            end if
11:        end for
12:        resource_occupation = resource width / NumActCores_A        ▷ Resource occupation
13:        idleP = idleP_temp_A × resource_occupation        ▷ Idle power prediction
14:        runtP = task->power_profile (task type, core type, frequency, resource width)        ▷ Runtime power prediction
15:        execution time = task->visit_performance_table(cluster A, resource width)        ▷ Execution time prediction
16:        energy = ( idleP + runtP ) × execution time        ▷ Energy prediction
17:        if energy <minimum then
18:            minimum = energy; update best placement with new execution place
19:        end if
20:    end for
21: end for
```

the instantaneous task parallelism during execution. Since each task can execute on a variable number of cores, the instantaneous task parallelism detection can be transformed into detecting the *resource occupation* (i.e., the percentage of the task resource width over the number of all active cores in the same cluster) of each parallel task. In other words, the idle power consumed by a task is computed by splitting the total idle power obtained from profiling proportionally to each task's resource occupation among the running tasks in parallel.

3.6 Task Mapping Algorithm

The task mapping module leverages the other modules described previously to determine the most energy efficient execution place for each task. The pseudo-code for the module is shown in Algorithm 2. The inputs to the algorithm are the ready tasks that are woken up by a core and the outputs are the predicted execution places that will consume the least energy consumption for these tasks. In Algorithm 2, we show a case that is tailored for a platform with two clusters (A with a cores and B with b cores). The algorithm can be easily extended to other platforms.

The algorithm iterates over all possible (cluster, resource width) configurations to find the one that consumes the least energy for each task (line 3). We randomly select one execution place from each possible (cluster, resource width) configuration for the energy prediction. We assume in this example for the purposes of illustration that the current evaluated configuration is in cluster A. The first step is to obtain information about the number of active cores on each cluster (lines 4 and 5). This information is provided by the core activity tracing module discussed in Section 3.5. The idle power value (*idleP_temp_A*) attributed to tasks running on cluster A depends on the number of active cores on cluster B (line 6). If there are active cores on cluster B, then the parallel tasks running on cluster A share the idle power of cluster A alone. Otherwise, the whole cluster B is in sleep and the tasks running on cluster A share the idle power of the entire chip. The information about idle power consumption, i.e., *idleP_chip* and *idleP_A*, is obtained from the power profiling module. As this is an estimation before task execution, it is possible that the selected cores are in sleep state. Therefore, if the ready task is going to execute on a configuration that contains cores

that are currently in sleep state, then their status will be updated after waking up and the number of active cores increases in this case, which is shown in lines 7–11. Line 12 computes the resource occupation for the task relative to other tasks on cluster A, and line 13 estimates the idle power consumed by the task if it is scheduled in the execution place. Next, line 14 estimates the runtime power for the task by accessing power profiling results based on the task type, core type, current detected cluster frequency and the resource width. Execution time prediction is obtained from the online performance modeling module, as shown in line 15. Finally, the algorithm estimates the total energy consumption if the ready task is scheduled to this execution place (line 16). The algorithm iterates through all possible configurations to determine the one that consumes least energy for each ready task (lines 17–19).

Algorithm 2 illustrates the steps to determine the most energy efficient execution place for a single ready task. The algorithm is run every time ready tasks are released. As multiple cores may release tasks in parallel, the algorithm may execute concurrently. It should be noted that the online performance model does not explicitly model the effect of inter-task concurrency for predicting the execution time. However, the impact of concurrency is implicitly captured in the execution time history, which reflects on the performance table and influences future predictions. While the power profiling module explicitly models concurrency for accurate idle power prediction, it does not take this into account for runtime power prediction. When sufficient bandwidth is available, the runtime power consumed by a task is dominated by the core's and private cache's performance, while the impact of interference from other co-running tasks is negligible. Our evaluation shows that the models are able to predict performance, power and energy consumption across a range of benchmarks and concurrency settings with reasonable accuracy (see Section 6.5 for details).

3.7 ERASE Complexity and Overhead Analysis

In this section, we analyze the complexity and the execution overheads of ERASE.

Time Complexity. For each task, Algorithm 2 includes two nested loops. First, ERASE traverses all possible execution configurations to find the one that consumes the least energy (line 3). Considering a platform includes n cores in total, m core-clusters such that each cluster comprises $\frac{n}{m}$ cores, the number of available resource widths of each cluster is $\log_2 \frac{n}{m}$. Therefore, the total number of possible configurations on the platform equals $m \times \log_2 \frac{n}{m}$. The *for* loop in line 7 traverses the selected core(s) in each execution place, which could be $2^0, 2^1, \ldots, \frac{n}{m}$. The time complexity of the loop is $O(\frac{n}{m})$. Therefore, the time complexity of Algorithm 2 for each task becomes $O(n \times \log_2 \frac{n}{m})$. The outermost loop in line 2 handles all the ready tasks in a DAG. When taking number of tasks (nr_task) into account, the time complexity becomes $O(nr\_task \times n \times \log_2 \frac{n}{m})$. Regarding the performance model, the number of tasks utilized for model training equals the product of the number of kernels in an application and the total number of execution places. Hence, the time complexity of building up the performance model per kernel is $O(m \times \log_2 \frac{n}{m})$. The power model is built offline. Accessing power model entries for a given execution place takes constant time. Consequently, the time complexity of accessing the power model is $O(1)$.

Space Complexity. In ERASE, the space taken by the performance model equals the product of the number of kernels in an application and the performance look-up table size. Since the number of kernels remains unchanged regarding the number of cores n, just the look-up table size (i.e., $m \times \log_2 \frac{n}{m}$) increases with larger platforms. Therefore, the space complexity of the performance model is $O(m \times \log_2 \frac{n}{m})$. The power profiling module uses three offline power models and each power model profiles the idle power and the runtime power of all possible execution places. Thus, the space complexity of the power model is also $O(m \times \log_2 \frac{n}{m})$.

Overheads. We evaluate the task scheduling overheads by calculating the time consumed in running the ERASE framework. The total execution time of an application can be partitioned into

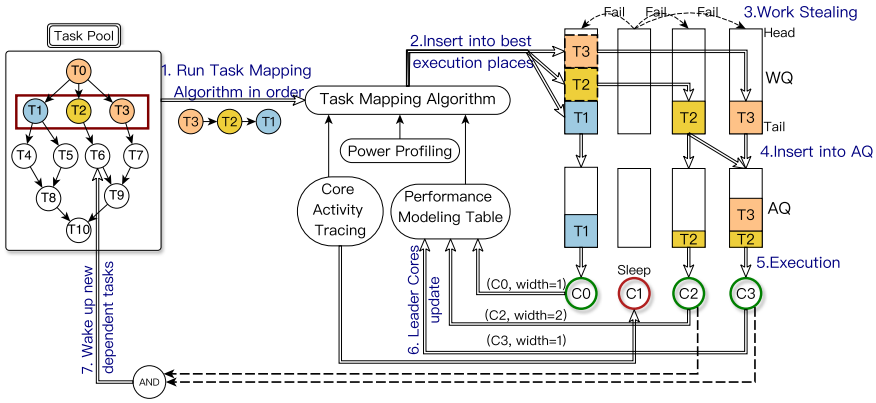


Fig. 4. An illustration of task execution flow with ERASE in XiTAO runtime, when core 0 finishes the execution of T0 and releases three dependent tasks.

three portions on each core: the accumulation of task execution time on the core, sleeping time and the time portion that runs the ERASE framework. We compute the average overhead by averaging the total overheads of all cores, which can be expressed by

$$Overhead = \frac{\sum_{i=1}^{n} 1 - \frac{\sum TaskExecTime_i + SleepTime_i}{Total\ Execution\ Time}}{n}. \quad (4)$$

The evaluation on both platforms shows that the overheads are rather small. For instance, with MatMul on TX2, the overhead is 0.23% on average across multiple degrees of parallelism.

4 INTEGRATION OF ERASE WITH A WORK STEALING RUNTIME

In this section, we describe how ERASE can be integrated into a work stealing environment. We use the open source XiTAO[1] runtime to exemplify the integration. The overall architecture of ERASE within XiTAO is shown in Figure 4. In XiTAO, each logical core manages two queues: a **work queue (WQ)**, subject to work stealing, and a FIFO **assembly queue (AQ)**, used to implement embedded work-sharing regions. The WQs hold the tasks for which the best execution place has been computed. When a task is fetched from the tail of WQ, pointers to the task are then inserted into all AQs representing the execution places for the task. The FIFO nature of the AQs guarantees that at some later point each core will fetch these pointers and execute the task. Note that in an asymmetric platform with multiple clusters, work stealing is only allowed among cores within the same cluster. Two cases are to be considered in the event that two ready tasks select the same execution place: (1) If resource width < number of cores in the cluster, then other available core(s) in the cluster can steal the second ready task, insert it into the AQs and start execution immediately to avoid delay, and (2) if resource width = number of cores in the cluster, then no available resources for the second ready task exist. Hence, the task is inserted into the AQs and it must wait for the completion of the first task to begin execution.

Figure 4 illustrates the execution flow of ERASE implemented in XiTAO using the DAG example. Here we assume that the four cores are from the same cluster for the purpose of simplicity. Thus, work stealing between the four cores does not lead to task resource width change due to the identical characteristics. Task pool collectively represents all the tasks in the sample application. We capture the state when T0 has finished execution on C0 and the core has released three dependent

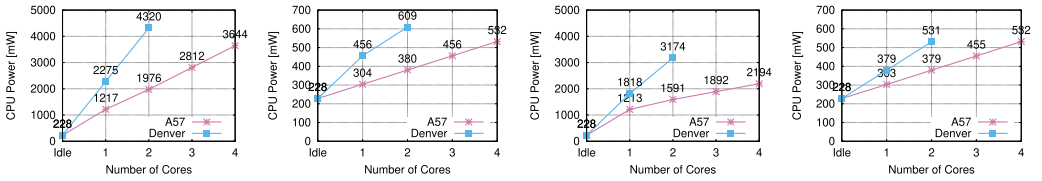
[1] https://github.com/CHART-Team/xitao.git.

ALGORITHM 3: Exponential Back-off Sleep Approach

```
1:  idle_tries = 0, backoff_param = 0
2:  while true do
3:      Check local WQ or try to steal from other WQs
4:      if no available task then
5:          idle_tries++
6:          if idle_tries == N then
7:              status[core_id] = 0
8:              sleep (1 ≪ backoff_param)
9:              idle_tries = 0; backoff_param++
10:         end if
11:     else
12:         status[core_id] = 1
13:         idle_tries = 0; backoff_param = 0
14:         execute(task)
15:     end if
16: end while
```

ready tasks T1, T2, and T3. T4 to T10 are pending tasks that will only be released after T1, T2, and T3 finish execution. When T1, T2, and T3 become ready, the runtime determines the most energy efficient execution place for each task by running Algorithm 2 (step #1). When a core releases multiple ready tasks, finding the best task placement for these tasks follows the task creation order. In this example, the most energy efficient execution places for the tasks T1, T2, and T3 are (C0,1), (C0,2), and (C0,1), respectively. The information about the best configuration obtained from the task mapping algorithm is embedded in the task. Consequently, all the three tasks are enqueued in the WQ of C0 (step #2). Each core first looks for tasks in its own WQ before attempting to steal from other cores. In the case of C0, it finds T1 in the tail of its WQ and therefore inserts the task in its own AQ. C2 and C3 do not find tasks in their local WQs and consequently end up stealing tasks T2 and T3 from C0 and inserting them in their respective queues (step #3). After C2 picks up T2 from the local WQ, it inserts T2 in the AQ of C2 and C3 since the best resource width is two (step #4). Similarly, C3 picks up T3 from its local WQ and inserts it in the AQ of C3. Tasks in the AQ are consequently picked up and executed by the respective cores (step #5). If the work stealing attempts are continuously unsuccessful and the number of attempts exceeds a threshold, then the core status tracked by the runtime is transitioned to "sleep" state, as shown for C1. After the execution, each leader core is responsible for updating the corresponding performance model with its execution time (step #6). Note that the execution flow is asynchronous, i.e., cores that execute a moldable task do not need to wait for the completion of each other. The last core that finishes the execution is responsible for waking up the dependent tasks. In this example, we assume that T2 finishes execution of C2 after C3 and therefore wakes up the dependent task T6 (step #7). If T1, T2, and T3 complete execution on different cores at the same time, then the task mapping algorithm is executed concurrently on each of these cores, which in turn ends up determining the most energy efficient execution place for T4, T5, T6, and T7.

To minimize energy waste from continuous work stealing attempts by idle cores, we adopt an *exponential back-off sleep* strategy [9], as shown in Algorithm 3. The core activity tracing module keeps track of the number of unsuccessful steal attempts (idle_tries) for each core and compares it with a runtime threshold parameter **N**. When the idle_tries matches **N**, the core is transitioned into sleep status (lines 6–10). Upon wake up, if the core finds any ready task from its own work queue or makes a successful steal from other work queues, then it resets the number of unsuccessful attempts (idle_tries) and the back-off parameter (backoff_param) to zeros (lines 12 and 13). Otherwise, it will sleep for an exponentially increasing time (ranging from 1 ms to 64 ms in this work) if it still cannot find any ready task. The evaluation in XiTAO shows that in comparison to the case where idle cores continuously attempt work stealing, exploiting the exponential back-off sleep strategy reduces energy by up to 67% during under-utilization periods with minimal impact on performance. Sleep optimizations are also used in popular work stealing runtimes such as cilk [3] and TBB [15].



(a) Compute-bound:MAX (b) Compute-bound:MIN (c) Memory-bound:MAX (d) Memory-bound:MIN

Fig. 5. Power profiling results of a compute-bound microbenchmark and a memory-bound microbenchmark on Jetson TX2 with two frequency levels.

5 EXPERIMENTAL METHODOLOGY

5.1 Evaluation Platforms

We evaluate the ERASE scheduler on two different platforms.

Asymmetric platform: The NVIDIA Jetson TX2 platform features a dual-core NVIDIA Denver 64-bit CPU and a quad-core ARM A57 CPU (each with 2 MB L2 cache). The board is set to `MAX-N` nvpmodel mode [25]. The Denver and the A57 cores implement the ARMv8 64-bit instruction set and are cache coherent. The platform only supports cluster-level DVFS and the two clusters support the same range of frequencies, i.e., between MAX (2.04 GHz) and MIN (0.35 GHz).

Symmetric platform: We evaluate the scheduler on a dual-socket 16-core Intel Xeon Gold 6130 node (32 cores in total, code-named "Tetralith") that runs Linux version 3.10.0. This machine is part of a larger compute cluster called Tetralith of National Supercomputer Centre at Linköping University [45]. As common with such infrastructure, frequency settings are controlled by the computing center's administrators and beyond our control. Therefore we do not carry out evaluations that require frequency adaptation on this platform. Intel's **Running Average Power Limit (RAPL)** [27] interface is used for energy measurement and modeling.

5.2 Power Profiling Using INA3221 and RAPL

INA3221: The NVIDIA Jetson TX2 platform, that we use in our evaluation, features an on-board `INA3221` power sensor. We read the power samples every 5 ms (Δt = 5 ms) and accumulate these samples obtained throughout the duration of application execution to compute the total energy consumption: Energy $= \int_{t_0}^{t_s} P(f,t)dt \approx \sum_{t_1}^{t_s}[P(f,t_i) - P(f,t_{i-1})].\Delta t$, where f denotes frequency. Figure 5 presents the power profiling results for a compute-bound microbenchmark and a memory-bound microbenchmark with different number/type of cores running on TX2, respectively. Here, we monitor power consumption for two different frequencies: MAX and MIN as an example. It can be observed that the total idle power of the entire chip is 228 mW, independent of the frequencies and the task type. Note that the Denver cluster and the A57 cluster differ in their idle power consumption. The idle power of the A57 cluster is obtained by powering off all the cores in the Denver cluster, which is evaluated to be ∼152 mW. Since one of A57 cores cannot be powered off on the platform, the idle power of the Denver cluster is obtained by subtracting the A57 cluster idle power from the total idle power of the entire chip, i.e., 228 mW−152 mW = 76 mW.

Taking the compute-bound microbenchmark as an example, we observe from Figure 5(a) and (b) that the runtime power consumption increases linearly with the number of cores used within a cluster. This linearity is not affected by the underlying frequency configuration in this case. With MAX frequency, the runtime power consumption is much higher than idle power consumption. For example, the runtime power consumption of using one A57 core is 989 mW (computed by subtracting 228 from 1217), which accounts for 81.3% of the total power consumption and dominates

the total power consumption in this case. When using all the four A57 cores, the runtime power occupies an even larger fraction (around 95%) of the total power consumption. However, at MIN frequency, the idle power makes up a larger fraction of the total power consumption. For example, the idle power constitutes 75% of the total power consumption when using one A57 core and 55% of the total when using one Denver core, as shown in Figure 5(b). This illustrates the importance of accurate power consumption attribution to each concurrently running task.

RAPL: RAPL interface is used to measure the energy and build the power profiles for the scheduler when running on an Intel platform. Since RAPL provides the energy readings for each CPU socket/package instead of power, it is necessary to convert the sampled energy values to the average power consumption during a fixed time window (5 milliseconds). We obtain the average power consumption using the equation: $Power_i = (Energy_i - Energy_{i-1})/\Delta t$.

5.3 Benchmarks

We evaluate the proposal using three synthetic benchmarks, two commonly used algebra kernels: dot product and Sparse LU factorization and four applications from the Edge and HPC domains.[2] Table 2 lists the configurations of the benchmarks. The magnitude of task execution time in these benchmarks ranges from 10 microseconds to 100 milliseconds (at 2.04 GHz on a Denver core). We introduce the notion of task criticality, since several schedulers we compare against (discussed in Section 5.4) rely on task criticality to improve energy efficiency. We define *critical path* as the longest path in a DAG. The tasks on the critical path are denoted as *critical tasks*. We define DAG parallelism (*dop*) as the total amount of tasks divided by the length of the critical path.

Synthetic Benchmarks are constructed with a configurable DAG parallelism and a notion of criticality. The DAG is constructed in such a manner that the root node releases *dop* tasks and one of the released nodes further releases *dop* tasks. This process recursively continues until the total number of tasks spawned reaches the user specified limit. We mark tasks that spawn the maximum number of tasks as critical tasks. The DAG comprises tasks that are of one of three types: MatMul, Copy, or Stencil. MatMul represents the *compute-bound* class, where matrices are pre-allocated and partitioned in N × N tiles. The matrix size is set to fit into L1 cache, which aims to minimize the cache-misses as the analysis requires focusing on the CPU power dissipation arising from the arithmetic units. Copy represents the *memory-bound* class. Each task reads and writes a large matrix to memory, effectively creating streaming behavior to access the main memory continuously. Stencil represents the *cache-sensitive* class as we set the task size greater than the last level cache but far less than DRAM. It involves repeated updates of values associated with points on a multi-dimensional grid using the values at a set of neighboring points.

Dot Product computes the sum of the products of two equal-length vectors. In XiTAO, we partition the vectors into blocks and mark each block as a single task. The DAG is iteratively executed for a predefined number of iterations (100 in this work).

Sparse LU Factorization (SLU) computes LU matrix factorization over sparse matrices [18] It includes four kernels: LU0, FWD, BDIV, and BMOD. Among them, all LU0, FWD, BDIV, are marked as critical tasks and some BMOD tasks are also marked as critical. The matrix is composed of N × N blocks, each of which has a pointer to a sub-matrix of size M × M.

Image Classification Darknet-VGG-16 CNN (VGG-16) is an inference deep learning algorithm that is typical of mobile and edge devices. It is a 16-layered deep neural network implemented as a fork-join DAG that spans all layers.

Alya is high-performance computational mechanics code developed at Barcelona Supercomputing Center [21]. The application involves solving partial differential equations, and its

[2]All benchmarks are available in XiTAO github repository, refer to https://github.com/CHART-Team/xitao.git.

Table 2. The Configurations of Evaluated Benchmarks

Application	MatMul		Copy		Stencil		VGG-16	Alya
Platform	TX2	Tetralith	TX2	Tetralith	TX2	Tetralith	TX2	TX2
Input Size	$2^6 \times 2^6$	$2^7 \times 2^7$	$2^{12} \times 2^{12}$	$2^{13} \times 2^{13}$	$2^8 \times 2^8$	$2^{10} \times 2^{10}$	BlockSize = 64	50k/100k/200k CSR non-zeros
Nr_Tasks	50000	20000	20000	10000	10000	10000	509	7789/21360 /47840
Criticality	√	√	√	√	√	√	×	×

Application	Heat	Sparse LU	Dot Product	Biomarker Infection
Platform	Tetralith	Tetralith	TX2	TX2
Input Size	Resolution = 40960	32 blocks, blocksize = 1024	200 blocks, blocksize = 32k	SampleSize = 2, NumBiomarkers 6217
Nr_Tasks	32032	1512	20000	6217
Criticality	×	√	√	×

parallelization strategy is based on mesh partitioning. We test with three different input sizes, i.e., 50K, 100K, 200K non-zeros expressed in compressed sparse row format.

Heat Diffusion (Heat) is implemented on a 2D grid by using one of the iterative numerical methods: two-dimensional Jacobi stencil. The DAG is iteratively executed for a predefined number of iterations (1000 in this work). We also experiment with other decompositions, e.g., pencil (2D), without change in conclusions. The Heat DAG is symmetrical, and thus no task criticality assignment is performed.

Biomarker Infection Research (BioInfection) is a medical use case developed using XiTAO runtime in the LEGaTO project [4]. The application serves as an indicator of a biological condition, identify risk factors, examine diseases, predict diagnoses, determine the state of the disease, or measure the effectiveness of treatment. The dataset tested belongs to a pilot study for differentiating between periprosthetic hip infection and aseptic hip prosthesis loosening.

5.4 Evaluated Schedulers

We evaluate the effectiveness of ERASE by comparing it to different scheduling techniques. The schedulers we evaluate can be broadly categorized into two groups. All schedulers are implemented on top of the XiTAO runtime with the exponential back-off sleep strategy described in Section 4.

The first group comprises schedulers that do not rely on DVFS and are described below:

(i). RWS is a widely used baseline scheduler, which tries to keep all cores busy through work stealing and also works well in asymmetric environments. Note that RWS schedules a task on a single core and does not leverage task moldability.

(ii). **Criticality-Aware Task Scheduler (CATS)** is a performance-oriented scheduler that leverages task criticality initially proposed by Chronaki et al. [14]. Tasks marked as critical are scheduled on the faster cores, while non-critical tasks are scheduled on the slower cores. CATS permits work stealing in specific cases, i.e., the faster cores can steal among themselves and also from the slower cores. Note that CATS does not exploit task moldability.

(iii). **Criticality-Aware Low Cost (CALC)** is a scheduler designed to evaluate the benefit of combining task moldable execution with criticality-based performance scheduler like CATS. For each critical task, the runtime globally compares the parallel execution cost, i.e., execution time × resource width, of all execution places and then selects the best one that minimizes the cost and it cannot be stolen. For non-critical tasks, the runtime only locally determines the best resource width that minimizes the cost while keeping the leader core fixed.

In the second group, we compare to a scheduler that exploits DVFS on a cluster-level DVFS platform and it is described below:

(i). **Aequitas** [38] is a round-robin time-slicing scheduler that leverages HERMES [39] to achieve energy efficiency. HERMES utilizes workpath sensitive and workload sensitive algorithms to tune

Table 3. Energy Consumption and Execution Time Comparison between Aequitas and ERASE on TX2

Application		MatMul				Copy				Stencil			
Parallelism		dop = 2	dop = 4	dop = 6	dop = 8	dop = 2	dop = 4	dop = 6	dop = 8	dop = 2	dop = 4	dop = 6	dop = 8
Energy[J]	Aequitas	92.85	87.93	84.06	80.36	71.01	53.31	52.66	57.83	45.22	41.51	39.62	35.15
	ERASE	76.59	74.06	73.85	69.32	58.26	59.26	64.77	64.19	31.41	31.01	31.17	30.99
Execution Time[s]	Aequitas	80.70	43.72	34.11	32.60	39.21	28.03	28.51	25.31	25.38	20.27	16.73	12.44
	ERASE	25.85	19.76	19.41	19.31	25.85	19.76	19.41	19.31	9.87	8.41	9.00	8.46

the frequency of individual cores based on thief-victim relations and the number of ready tasks available in the WQ. Additionally, for architectures with cluster-level DVFS, Aequitas lets each active core within a cluster control DVFS for a short interval in a round-robin manner. We reproduce Aequitas on TX2 using five frequency levels (i.e., 2.04 GHz, 1.57 GHz, 1.11 GHz, 0.65 GHz, and 0.35 GHz) as in the original proposal, four thresholds for workload sensitive algorithm ($\lceil \frac{dop+1}{5} \rceil$, $2\lceil \frac{(dop+1)}{5} \rceil$, $3\lceil \frac{(dop+1)}{5} \rceil$, and $4\lceil \frac{(dop+1)}{5} \rceil$) and a slicing time interval of 1s. We conducted a sensitivity analysis with time slicing intervals of 2 seconds, 1 seconds, 0.5 seconds, and 0.25 seconds. The evaluation shows that using 1-second interval gives the best results. Hence, in this work we present the Aequitas results with 1-seconds interval.

6 EVALUATION

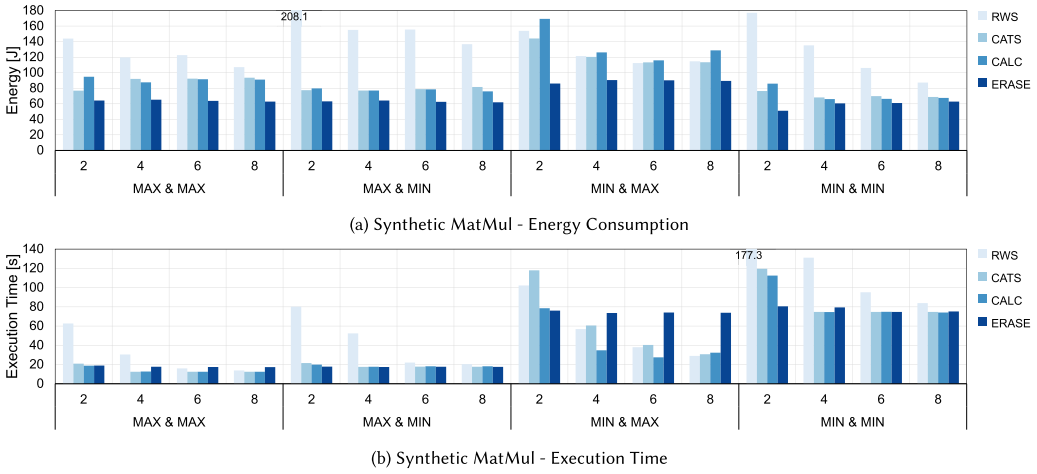
We evaluate the effectiveness and adaptability of ERASE by comparing it to several state-of-the-art scheduling techniques on two different platforms. We first compare ERASE to a state-of-the-art scheduler that relies on DVFS throttling in Section 6.1. Then, we compare ERASE to other schedulers that do not rely on DVFS on both TX2 and Tetralith platforms in Sections 6.2 and 6.3. We also present how fast ERASE can react to a dynamic DVFS environment in Section 6.4. Last, we evaluate the proposed models' accuracy in ERASE, which is shown in Section 6.5. We repeat each experiment 100 times and report the average energy and execution time numbers. The observed coefficient of variation in the measurements is below 1.3%.

6.1 Comparison to the State-of-the-art

Table 3 compares the energy consumed by ERASE and Aequitas [38], when running three synthetic benchmarks (*dop* ranging from 2 to 8) on the Jetson TX2 platform. Since the scheduling strategies have significant impact on performance, we also provide execution time comparisons to better understand the tradeoff between energy and performance. A general observation is that ERASE consumes less energy and is faster than resource-agnostic and non-moldable counterparts Aequitas across most of the benchmarks and different degrees of parallelism.

Overall, ERASE achieves 15% energy reduction on average and up to 68% performance improvement for MatMul, compared to Aequitas, which results in 59% EDP reduction. For Stencil, ERASE achieves 22% energy reduction and 49% performance improvement and therefore 60% EDP reduction on average than Aequitas. In the case of Copy, ERASE is more energy efficient than Aequitas with DAG parallelism of 2, and it reduces energy consumption and EDP by 18% and 46%, respectively.

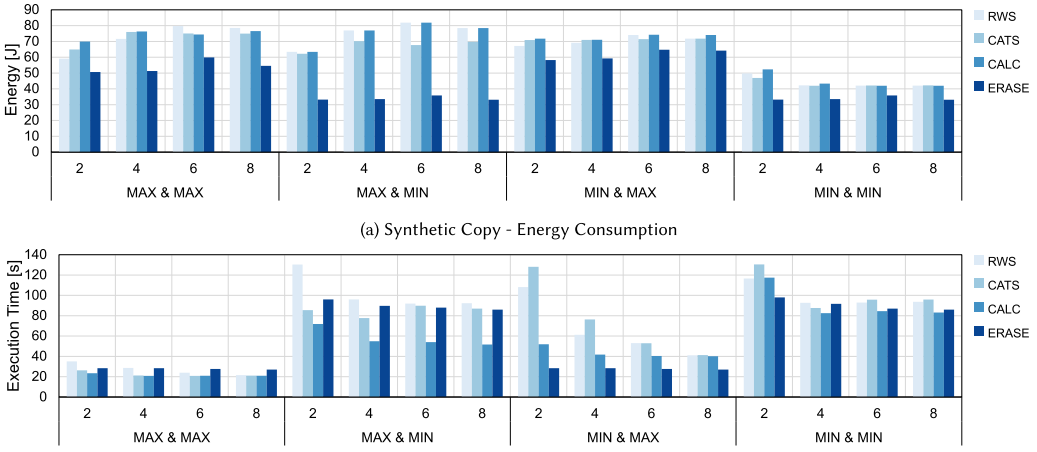
Analysis: The inefficiency of Aequitas can be attributed to two reasons: (1) cores that acquire domination control (the ability for changing the cluster frequency for a specific timeslice) can decrease the frequency of the whole cluster resulting in delaying execution of critical tasks thereby increasing idleness, and (2) Aequitas does not take core asymmetry into account. If critical tasks are scheduled on low-performance cores and the cluster's frequency is decreased, then the amount of idleness can increase considerably especially when there are fewer ready tasks in work queues overall. Aequitas consumes less energy than ERASE with Copy of *dop* = 4 to 8. Because



(a) Synthetic MatMul - Energy Consumption

(b) Synthetic MatMul - Execution Time

Fig. 6. Energy consumption and execution time comparison among RWS, CATS, CALC, and ERASE with Synthetic MatMul on Jetson TX2.



(a) Synthetic Copy - Energy Consumption

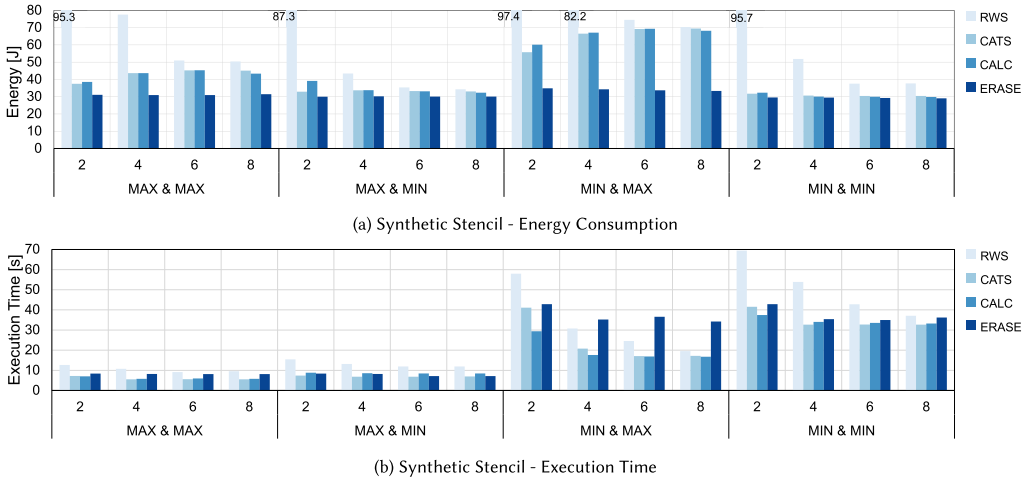
(b) Synthetic Copy - Execution Time

Fig. 7. Energy consumption and execution time comparison among RWS, CATS, CALC, and ERASE with Synthetic copy on Jetson TX2.

Copy is memory-bound and frequency reduction helps to save more energy without impacting performance.

6.2 Evaluation on Asymmetric Platform

Next, we compare ERASE to RWS, CATS, and CALC on TX2. These schedulers do not rely on DVFS but use other approaches for improving performance and energy efficiency. To compare the efficacy of the schedulers at different static frequency settings, we perform the evaluation using the four MIN and MAX frequency permutations on the two clusters, as shown in the x-axis labels of Figures 6, 7, 8, and 11. For example, MAX & MIN means that the Denver cluster is set to maximum frequency and the A57 cluster is set to minimum frequency.



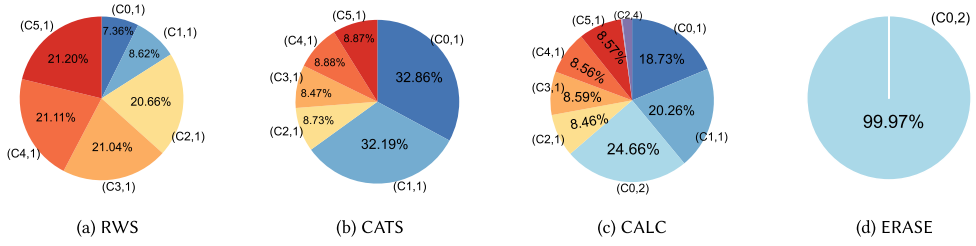
(a) Synthetic Stencil - Energy Consumption

(b) Synthetic Stencil - Execution Time

Fig. 8. Energy consumption and execution time comparison among RWS, CATS, CALC, and ERASE with Synthetic Stencil on Jetson TX2.

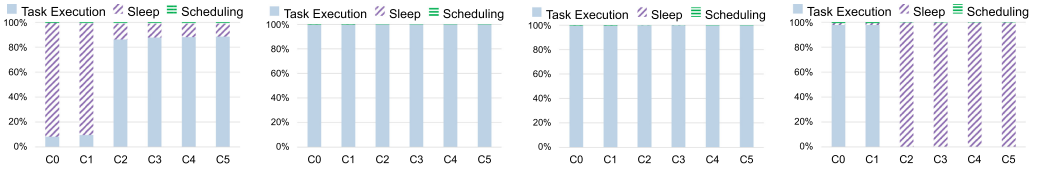
6.2.1 Synthetic Benchmarks. Figures 6, 7, and 8 compares the energy and execution time of the four schedulers with MatMul, Copy, and Stencil over different DAG parallelism, respectively. The results show that ERASE generally consumes the least amount of energy compared to other three scheduling policies. Another important observation is that ERASE does consistently well independent of DAG parallelisms and frequency settings. While the energy consumption of other three schedulers varies considerably with different frequency settings and is usually much higher with low parallel slackness (i.e., fewer tasks in all the work queues). If we consider MatMul in Figure 6 as an example, then ERASE achieves 47% energy reduction and improves performance by 8% on average compared to RWS across different frequency and parallelism settings. It is clear that in most cases, the energy reduction with ERASE far exceeds the performance losses, which results in 43% EDP reduction on average. In comparison to CATS and CALC, ERASE achieves an average energy reduction of 30% and an average reduction in EDP of 10% across different DAG parallelisms.

Analysis: We choose MatMul with *dop* = 4 in MAX&MAX as an example to illustrate why ERASE ends up consuming less energy in comparison to the other schedulers we evaluate. Figure 9 presents the task distribution on each execution place chosen by the four schedulers, where C0 and C1 denote two Denver cores with a maximum possible resource width of two and C2 to C5 denote the four A57 cores with a maximum possible resource width of four. Figure 10 shows the time distributed between scheduling activity, sleep, and actual task execution when using these schedulers. It can be observed that with RWS the majority of tasks are executed on the four A57 cores. Consequently, the two Denver cores are idle and are put to sleep for energy savings. CATS permits work stealing if the Denver cores attempt to steal tasks from the A57 cores mainly to permit migration of critical tasks to high-performance cores. Consequently, more than 65% of tasks execute on the high-performance Denver cores, leading to 60% performance improvement compared to RWS. The result of CALC shows that 27% ((C0,2) and (C2,4)) of tasks exploit the task moldability. As discussed earlier in Section 2.2, executing a single MatMul task with two Denver cores is the most energy efficient configuration on this platform and this results in an additional energy reduction of 7% in comparison to CATS. For CATS and CALC, almost 100% of the time is spent in executing tasks as shown in Figure 9(b) and (c). ERASE correctly predicts and executes most of the tasks on Denver cores with a resource width of two since it is the most energy efficient



(a) RWS (b) CATS (c) CALC (d) ERASE

Fig. 9. Task distribution of RWS, CATS, CALC, and ERASE with Synthetic MatMul on Jetson TX2.



(a) RWS (b) CATS (c) CALC (d) ERASE

Fig. 10. Time distribution of RWS, CATS, CALC, and ERASE with Synthetic MatMul on Jetson TX2.

option thereby consuming the least energy when compared to the other schedulers. In MIN&MAX, executing MatMul tasks with two Denver cores also consumes the least energy. However, the performance is worse than other schedulers with high *dop* since all tasks are executed with the lowest frequency on Denver cores while the A57 cores are idle.

6.2.2 Applications. Figure 11 shows the four schedulers when running VGG-16, Alya, Dot product, and Biomarker Infection on TX2. The results indicate that ERASE is the most energy efficient scheduler across the majority of environment settings. In comparison to RWS, CATS, and CALC, ERASE achieves between 8% and 59% energy reduction and outperforms them by 19% on average across four different environment settings on VGG-16. For Alya with three different input sizes, ERASE reduces energy consumption by 18% and improves performance by 17% on average, compared to the other three schedulers. Additionally, Dot Product and Biomarker Infection respectively achieve 34% and 25% energy savings on average compared to the other three schedulers. We observe that the energy reduction is significant when the performance gap of two clusters is large (e.g., 52% on average with Dot Product and configuration MAX(D)&MIN(A)). In contrast, less energy savings are observed when the performance gap is small (e.g., 7% on average with Dot Product and configuration MIN(D)&MAX(A)). Similar trends can also be observed in other tested cases.

6.3 Evaluation on Symmetric Platform

On a symmetric platform, the scope of energy efficient task scheduling is dominated by task moldability, i.e., the selection of an appropriate resource width for executing a task. Note that CATS is only designed for asymmetric platforms, thus it is not included in the evaluation on Tetralith.

6.3.1 Synthetic Benchmarks. Figure 12(a) to (f) present the energy and execution time comparison of RWS, CALC, and ERASE with *dop* of 8 to 64. The results show that ERASE consumes 15% less energy and achieves 18% performance improvement on average when compared to the other two schedulers across different *dop*. The energy reduction is significant specially when *dop* is low.

Analysis: The baseline scheduler RWS only utilizes a single core for each task, thus with low task parallelism, idle cores consistently attempt work stealing. With our exponential back-off sleep

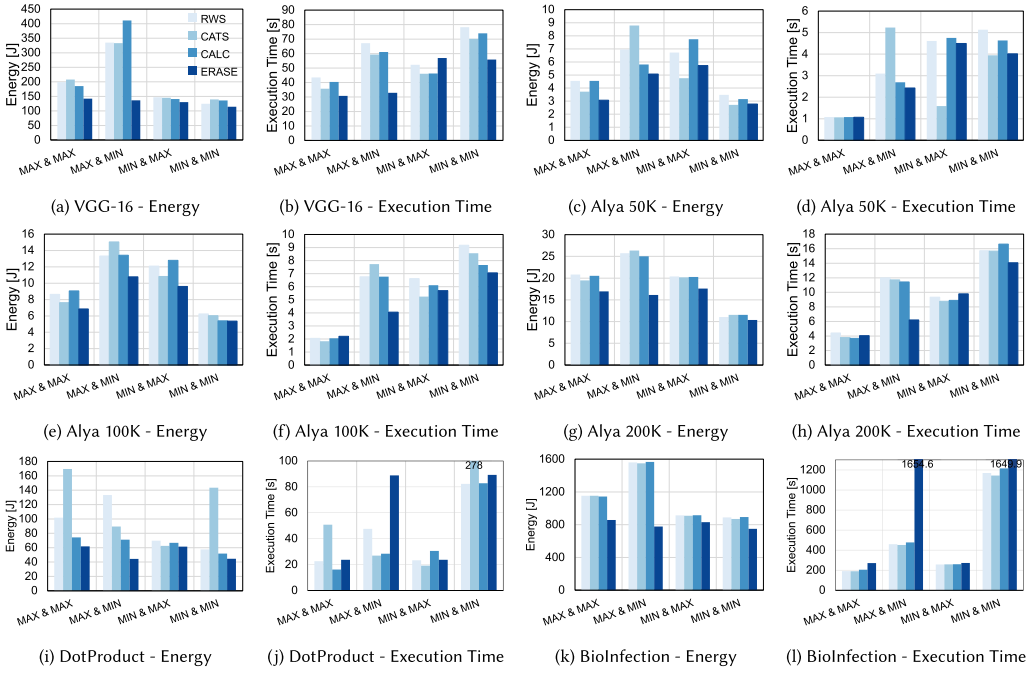


Fig. 11. Energy consumption and execution time comparison of VGG-16, Alya, Dot Product, and Biomarker Infection Research between RWS, CATS, CALC, and ERASE on Jetson TX2.

strategy, RWS reduces energy consumption by 14% (11%) on average than baseline with *dop* = 8 (16). Even so, RWS still performs worse than the other two schedulers with low parallelism due to resource under-utilization. In contrast, CALC permits task moldability and the wider width selection effectively reduces the under-utilization periods. Therefore, fewer cores are idle in CALC and the back-off sleep has limited effects on energy consumption.

Figure 12(g) and (h) present the energy consumption and execution time comparison, where each MatMul and Copy task is executed with preset specific width (i.e., 1, 2, 4, 8, 16) at *dop* of 8. For MatMul, the majority of tasks with CALC execute with resource widths of 8. This minimizes the parallel execution cost, improving performance by efficiently using available resources and reduces the energy waste from under-utilization compared to RWS. In comparison, ERASE executes 99% of MatMul tasks with a resource width of 16, since execution with width 16 gains more performance benefits with less power costs compared to the execution with width 8. This results in the resource width selection of 16 thereby achieving energy minimization and also faster execution (as shown in Figure 12(g)). In the case of Copy, more than 97% of tasks in CALC execute with width 1 for parallel execution cost minimization, since moldable execution does not bring linear performance improvement. Therefore, the energy and execution time are almost the same as RWS. We observe that ERASE is able to figure out the best resource width for different DAG parallelism, specifically width 4 (2) for *dop* = 8 (16). As shown in Figure 12(h), execution with lower widths (i.e., 1 and 2) does not effectively utilize all resources and sleeping does not bring enough energy benefits to compensate for the performance loss. In contrast, execution with higher width (i.e., 8 and 16) leads to high power costs with limited performance improvement.

6.3.2 Applications. The evaluation of Heat and Sparse LU showed in Table 4 indicates again that ERASE consumes less energy than other scheduling policies.

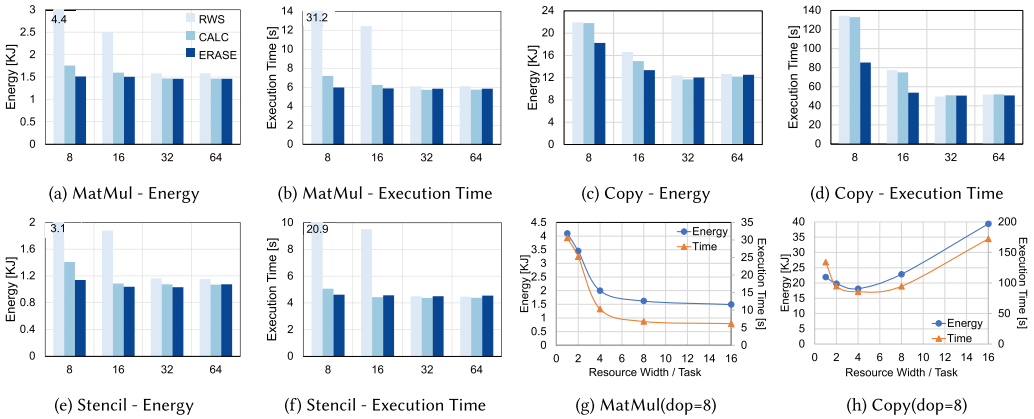


Fig. 12. Energy and execution Time comparison of RWS, CALC, and ERASE on Tetralith.

Table 4. Energy and Execution Time of RWS, CALC, and ERASE with Sparse LU and Heat on Tetralith

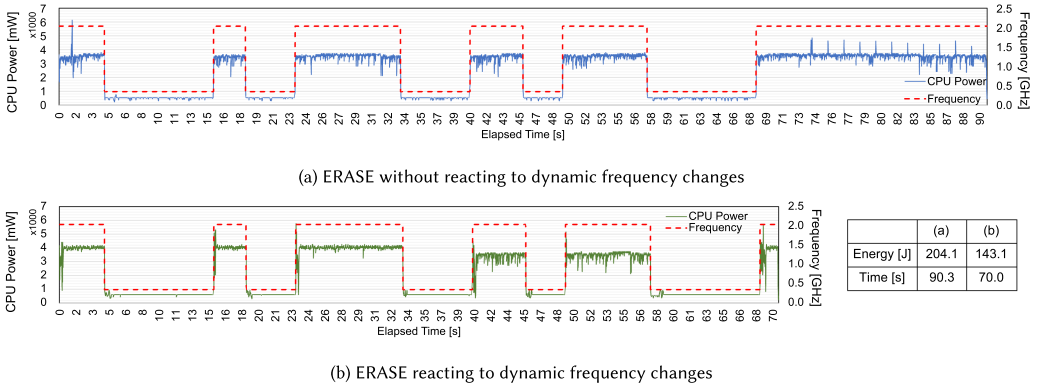
Application	Sparse LU			Heat		
	RWS	CALC	ERASE	RWS	CALC	ERASE
Energy [J]	15446	11597	10489	10853	10729	8972
Execution Time [s]	63.2	55.1	41.9	54.2	55.7	62.6

Analysis: We compare the distribution of resource widths for tasks with CALC and ERASE when running Heat to understand why ERASE performs better in comparison to the other schedulers. In the case of CALC, 89% of tasks execute with a resource width of 1 due to reduced parallel execution cost associated with the execution. In the case of ERASE, the majority of tasks execute with wider resource widths and the numbers of tasks that execute with a resource width of 4 and 16 are almost equal. This happens because in Heat there are two different types of kernels with different scalability characteristics: *jacobi* and *copy*. The *jacobi* kernel is compute-bound and while *copy* is memory bound, ERASE is able to determine the best width for each kernel. During the run, 99% of *jacobi* tasks execute with width 16, while 98% of the *copy* tasks execute with width 4.

6.4 DVFS Awareness of ERASE

To analyze the adaptability of ERASE to externally controlled DVFS changes, we execute MatMul on Jetson TX2 with randomly inducing frequency changes (shown as red dotted curves in Figure 13). We generate the DVFS changes externally in a different running process and alternate the frequencies for both Denver and A57 clusters between the highest and the lowest (2.04 and 0.35 GHz) by writing the specific frequency into system files and then sleeping a random time between 3 seconds to 12 seconds.

Figure 13(a) shows the case where ERASE models are unaware of the frequency change and keep using the same model settings detected at the beginning of execution. While Figure 13(b) shows the case where ERASE models detect the dynamical frequency changes and automatically adapt to the new frequency instantly. The results show that without DVFS awareness, the majority of tasks (>98%) are executed with sub-optimal execution place settings. In comparison to the DVFS unaware case, ERASE achieves 30% of energy savings and 23% performance improvement when the models are capable of reacting and adapting to DVFS and recalculate the best execution places accordingly. As shown in Figure 13(b), at the beginning of the execution, CPU power consumption



(a) ERASE without reacting to dynamic frequency changes

	(a)	(b)
Energy [J]	204.1	143.1
Time [s]	90.3	70.0

(b) ERASE reacting to dynamic frequency changes

Fig. 13. The comparison of ERASE execution with MatMul between DVFS awareness and non-awareness.

shows a short period of variation ranging from 1 W to 5 W, indicating that ERASE performance model is in the training phase. After that, when the frequency changes (either from lowest to highest or from highest to lowest), performance models are retrained instantly, resulting in CPU power variations that can only be observed over a short interval immediately following the change before stabling. After training the performance model, the corresponding power model for the newly detected frequency is used for new execution place predictions.

To evaluate the overhead associated with adapting to externally controlled DVFS and to determine how quickly ERASE can react to DVFS, we compare energy and execution time between ERASE and the ideal scenario where the best execution place for each task is determined by offline analysis thereby precluding the use of power and performance models. Our evaluation shows that the differences in energy and execution time between ERASE and the ideal case are 0.6%.

6.5 Model Accuracy

We evaluate the accuracy of the performance model and the power model used in ERASE using benchmarks listed in Section 5.3. Moreover, to explore the accuracy of combining the two models, we evaluate the accuracy of predicted energy consumption. We adopt the statistical metric **Mean Absolute Percentage Error (MAPE)** as a measure of accuracy, which is computed as follows:

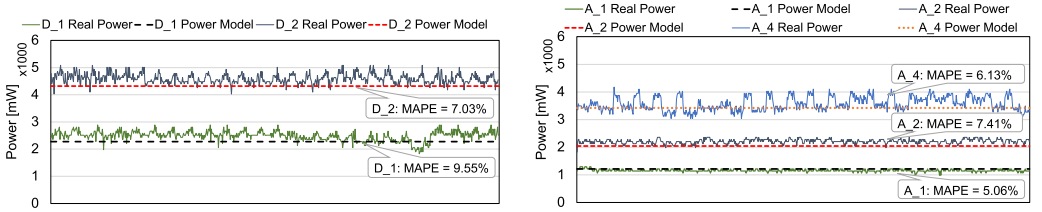
$$MAPE = \frac{\sum_{i=0}^{\tau-1} \left| \frac{real_i - predicted_i}{real_i} \right|}{\tau} \times 100, i = task\ 0, \ldots, \tau - 1. \quad (5)$$

Performance Model Accuracy: We evaluate the prediction accuracy by comparing the predicted execution time obtained from the performance model to the actual execution time of each task. Our evaluation on all seven benchmarks shows that the average MAPE for execution time prediction is 2.2%. In other words, the model achieves 97.8% prediction accuracy on average.

Power Model Accuracy: We evaluate the prediction accuracy by comparing the power consumption estimates obtained from the power profile to each power sample collected from the embedded INA3221 power sensor during the benchmark execution. Table 5 shows the prediction deviations of four evaluated benchmarks from three categories, including the results measured in maximum frequency, minimum frequency and by averaging the deviations measured with all 12 possible frequency degrees on the Jetson TX2 platform. For task type categorization, we apply the methodology described in Section 3.4 to empirically determine the following thresholds for Jetson TX2: $AI < 6.25$ for memory-boundness, $6.25 < AI < 18.75$ for cache-intensive and $AI > 18.75$ for compute-boundness. We present the AI values of each benchmark on the two frequency levels

Table 5. Power Prediction Deviations Evaluated with Four Benchmarks on Jetson TX2

	MatMul			Copy			Stencil			Sparse LU		
	2.04 GHz	0.35 GHz	Average	2.04 GHz	0.35 GHz	Average	2.04 GHz	0.35 GHz	Average	2.04 GHz	0.35 GHz	Average
D_1	1.88%	4.33%	1.25%	6.64%	6.24%	3.10%	1.14%	5.07%	1.47%	9.55%	3.87%	2.82%
D_2	5.62%	0.13%	3.54%	22.87%	17.90%	17.18%	6.19%	3.74%	3.81%	7.03%	0.92%	2.61%
A_1	6.46%	1.13%	3.88%	7.34%	2.62%	6.03%	4.16%	2.05%	4.22%	5.06%	0.13%	2.58%
A_2	3.96%	3.16%	2.94%	7.71%	8.56%	7.16%	7.92%	6.11%	5.28%	7.41%	9.30%	2.86%
A_4	0.46%	1.50%	4.54%	5.02%	2.28%	5.31%	7.32%	4.55%	5.10%	6.13%	9.68%	5.92%
AI	423.8	400.6	—	2.07	1.03	—	17.23	16.88	—	31.5	23.8	—
	compute-bound			memory-bound			cache-intensive			compute-bound		



(a) on Denver cluster (b) on A57 cluster

Fig. 14. Power model accuracy evaluation by comparing real power and modeled power when running Sparse LU on different execution places of Jetson TX2 (frequency = 2.04 GHz).

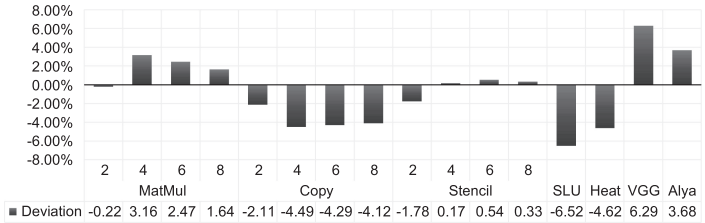


Fig. 15. ERASE energy prediction deviations with all evaluated benchmarks on Jetson TX2.

in the table. The results indicate that, despite its simplicity, this low overhead power modeling approach achieves reasonable and reliable accuracy for estimating power consumption of tasks. Specifically, we take a real application–Sparse LU as an example to show the comparison between the modeled power and the real power trace when running with different execution configurations. The corresponding results are shown in Figure 14. All tasks from the four different kernels in Sparse LU (see Section 5.3) with 2.04 GHz are categorized as compute-bound, since the average AI of all Sparse LU tasks is 31.5. Thus, at runtime, the corresponding compute-bound power profile values are used as power estimates. The power comparison results on Sparse LU show that the power prediction deviations are within 10% (7% on average) across different execution places.

Energy Prediction Accuracy: Figure 15 presents the predicted deviations of energy consumption measured in MAPE across all evaluated benchmarks. The MAPE value of each benchmark is computed by comparing the predicted energy consumption of each task to the real energy consumption. The evaluation shows that the differences between predicted energy predictions and the real energy consumption are within 7% (3.5% on average).

7 RELATED WORK

Energy efficient task scheduling has received some attention in recent years. Existing proposals rely on DVFS reconfiguration, scheduling strategies, or combinations thereof. There are some prior

works applying to the environments with a centralized scheduler for all cores. Costero et al. [16] developed policies for frequency scaling and task scheduling for asymmetric platforms. First, they tune DVFS as a function of the ratio between running critical tasks and non-critical tasks. Next, they schedule tasks either to the big or LITTLE cluster based on the current number of ready tasks (parallel slackness). Sanjeev et al. [6] proposed EADAGS, a two-phase task scheduler to minimize the total execution time and energy consumption using dynamic voltage scaling and decisive path scheduling. In the first phase, EADAGS creates a single queue and puts critical tasks into the queue and then schedules critical tasks one by one to a processor that minimizes the execution time of the task. Then the energy consumption is calculated and it tunes down the voltage of processors if the decision does not bring the performance slowdown. These methods using a centralized queue are more suitable for small systems and they present the issues of poor scalability and increasing communication cost between cores and the centralized queue on larger shared memory systems.

In contrast, work stealing allows each core to have its own work queue and idle cores greedily steal tasks from others to keep workload balancing, which achieves better scalability and reduces the contention [7, 12]. Haris et al. [39] proposed HERMES, which exploited DVFS per core by checking the task stealing sequences between cores and workloads on two AMD machines. However, their works [38, 39] target only symmetric platforms and do not consider the opportunities on asymmetric architectures. Torng et al. [44] proposed AAWS that targets for improving the performance of CPU-bound applications in work stealing runtime on asymmetric platforms. It used work-pacing, work-sprinting and work-mugging strategies along with the per-core DVFS capability to map the critical tasks to the most appropriate resources and balance the workloads among cores. For moldable streaming tasks, Melot et al. [32] proposed a heuristic for "crown" frequency scaling using a simulated annealing technique to achieve energy efficient mapping on symmetric manycore systems. In contrast, ERASE achieves energy reduction by task type awareness, exploiting moldability and task parallelism detection for both static and dynamic frequency settings.

8 CONCLUSION

We propose ERASE, an intra-application task scheduler for reducing energy consumption of executing fine-grained task-based parallel applications and discuss the design and implementation on top of a work stealing runtime. The design of the scheduler is motivated by the following observations: (1) leveraging DVFS in per task basis is ineffective when using fine-grained tasking and in environments with cluster/chip-level DVFS; (2) task moldability, wherein a single task can execute on multiple threads/cores via work-sharing, can help to reduce the energy consumption; and (3) combining task type-aware execution and task moldability is crucial for energy savings, particularly on asymmetric platforms. The scheduler works by estimating the energy consumption of different execution places at a per-task level and performs task mapping decisions so as to minimize the energy consumption of each task. It also adaptively puts cores that repeatedly execute work stealing loop without success to sleep state by applying an exponential back-off sleeping strategy. In addition, ERASE is capable of adapting to both given static frequency settings and externally controlled DVFS. We evaluate ERASE by comparing it to several state-of-the-art scheduling techniques on two different platforms. The results indicate that ERASE can provide significant energy savings in comparison, across a range of benchmarks and system environment settings.

REFERENCES

[1] 2014. NVIDIA Jetson. Retrieved from https://developer.nvidia.com/EMBEDDED/Jetson-modules.
[2] 2014. ODROID XU3. Retrieved from https://www.hardkernel.com/shop/odroid-xu3/.
[3] 2018. Cilk scheduler. Retrieved from https://github.com/OpenCilk/cilkrts/tree/main/runtime.

[4] 2020. Biomarker Discovery. Retrieved from https://legato-project.eu/use-cases/healthcare.
[5] 2021. XiTAO. Retrieved February 2, 2021 from https://github.com/CHART-Team/xitao.
[6] Sanjeev Baskiyar and Rabab Abdel-Kader. 2010. Energy aware DAG scheduling on heterogeneous systems. *Cluster Comput.* 13, 4 (2010), 373–383.
[7] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling multithreaded computations by work stealing. *J. ACM* 46, 5 (Sept. 1999), 720–748.
[8] Dominik Brodowski. [n.d.]. CPU Frequency and Voltage Scaling Code in the Linux(TM) Kernel. Retrieved from https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt.
[9] Byung-Jae Kwak, Nah-Oak Song, and L. E. Miller. 2005. Performance analysis of exponential backoff. *IEEE/ACM Trans. Netw.* 13, 2 (Apr. 2005), 343–355. https://doi.org/10.1109/TNET.2005.845533
[10] Emilio Castillo, Miquel Moretó, Marc Casas, Lluc Alvarez, Enrique Vallejo, Kallia Chronaki, Rosa M. Badia, Jose Bosque, Ramon Beivide, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. 2016. CATA: Criticality aware task acceleration for multicore processors. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS'16)*. IEEE, 413–422. DOI: 10.1109/IPDPS.2016.49
[11] Jing Chen, Pirah Noor Soomro, Mustafa Abduljabbar, Madhavan Manivannan, and Miquel Pericas. 2020. Scheduling task-parallel applications in dynamically asymmetric environments. In *Proceedings of the 49th International Conference on Parallel Processing (ICPP: Workshops'20)*. Association for Computing Machinery, New York, NY, Article 18, 10 pages. https://doi.org/10.1145/3409390.3409408
[12] Quan Chen, Yawen Chen, Zhiyi Huang, and Minyi Guo. 2012. WATS: Workload-Aware task scheduling in asymmetric multi-core architectures. In *Proceedings of the IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS'12)*. https://doi.org/10.1109/IPDPS.2012.32
[13] H. Choi, Dong Oh Son, S. G. Kang, J. Kim, Hsien-Hsin Lee, and C. Kim. 2013. An efficient scheduling scheme using estimated execution time for heterogeneous computing systems. *J. Supercomput.* 65, 2 (2013), 886–902. DOI: 10.1007/s11227-013-0870-6
[14] Kallia Chronaki, Alejandro Rico, Rosa M. Badia, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. 2015. Criticality-Aware dynamic task scheduling for heterogeneous architectures. In *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS'15)*. Association for Computing Machinery, New York, NY, 329–338. https://doi.org/10.1145/2751205.2751235
[15] Gilberto Contreras and Margaret Martonosi. 2008. Characterizing and improving the performance of intel threading building blocks. In *Proceedings of the IEEE International Symposium on Workload Characterization*. IEEE, 57–66.
[16] Luis Costero, Francisco Igual, Katzalin Olcoz, and Francisco Tirado. 2017. Energy efficiency optimization of task-parallel codes on asymmetric architectures. In *International Conference on High Performance Computing & Simulation (HPCS'17)*. IEEE, 402–409. https://doi.org/10.1109/HPCS.2017.67
[17] J. Doweck, W. Kao, A. K. Lu, J. Mandelblat, A. Rahatekar, L. Rappoport, E. Rotem, A. Yasin, and A. Yoaz. 2017. Inside 6th-generation intel core: New microarchitecture code-named skylake. *IEEE Micro* 37, 2 (2017), 52–62. https://doi.org/10.1109/MM.2017.38
[18] Alejandro Durán, 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. In *Proceedings of the International Conference on Parallel Processing*. https://doi.org/10.1109/ICPP.2009.64
[19] Jonathan Eastep, Steve Sylvester, Christopher Cantalupo, Brad Geltz, Federico Ardanaz, Asma Al-Rawi, Kelly Livingston, Fuat Keceli, Matthias Maiterth, and Siddhartha Jana. 2017. Global extensible open power manager: A vehicle for HPC community collaboration on co-designed energy management solutions. In *High Performance Computing*.
[20] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*. 212–223. DOI: 10.1145/277650.277725
[21] Houzeaux Guillaume and Vazquez Mariano. [n.d.]. Alya Application. Retrieved from https://www.bsc.es/research-development/research-areas/engineering-simulations/alya-high-performance-computational.
[22] 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)*. 38–43. https://doi.org/10.1145/1283780.1283790
[23] Brian Jeff. 2012. Advances in big.LITTLE technology for power and energy savings improving energy efficiency in high-performance mobile platforms. *White Paper released by ARM*.
[24] Víctor Jiménez, Lluís Vilanova, Isaac Gelado, Marisa Gil, Grigori Fursin, and Nacho Navarro. 2009. Predictive runtime code scheduling for heterogeneous architectures. In *International Conference on High-Performance Embedded Architectures and Compilers*. Springer, 19–33. https://doi.org/10.1007/978-3-540-92990-1_4

[25] kangalow. 2017. NVPModel—NVIDIA Jetson TX2 Development Kit. Retrieved March 25, 2017 from https://www.jetsonhacks.com/2017/03/25/nvpmodel-nvidia-jetson-tx2-development-kit/.
[26] The kernel development community. [n.d.]. Energy Aware Scheduling. Retrieved from https://www.kernel.org/doc/html/latest/scheduler/sched-energy.html.
[27] Kashif Khan, Mikael Hirki, Tapio Niemi, Jukka Nurminen, and Zhonghong Ou. 2018. RAPL in action: Experiences in using RAPL for power measurements. *ACM Trans. Model. Perf. Eval. Comput. Syst.* 3, 2 (01 2018), 26 pages. https://doi.org/10.1145/3177754
[28] T. Kolpe, Antonia Zhai, and S. S. Sapatnekar. 2011. Enabling improved power management in multicore processors through clustered DVFS. In *Proceedings of the Design, Automation and Test in Europe Conference | The European Event for Electronic System Design & Test (DATE'11)*, 1–6. https://doi.org/10.1109/DATE.2011.5763052
[29] Sanjeev Kumar, Christopher J. Hughes, and Anthony Nguyen. 2007. Carbon: Architectural support for fine-grained parallelism on chip multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA'07)*. Association for Computing Machinery, New York, NY, 162–173. https://doi.org/10.1145/1250662.1250683
[30] Ching-Chi Lin, Jian-Jia Chen, Pangfeng Liu, and Jan-Jan Wu. 2018. Energy-Efficient core allocation and deployment for container-based virtualization. In *IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS'18)*. IEEE, 93–101. https://doi.org/10.1109/PADSW.2018.8644537
[31] Linux Programmer's Manual. [n.d.]. perf_event_open()—Set up Performance Monitoring. Retrieved from https://man7.org/linux/man-pages/man2/perf_event_open.2.html.
[32] Nicolas Melot, Christoph Kessler, Jörg Keller, and Patrick Eitschberger. 2015. Fast crown scheduling heuristics for energy-efficient mapping and scaling of moldable streaming tasks on manycore systems. *ACM Trans. Archit. Code Optim.* 11, 4, Article 62 (Jan. 2015), 24 pages. https://doi.org/10.1145/2687653
[33] Georg Ofenbeck, Ruedi Steinmann, Victoria Caparros, Daniele Spampinato, and Markus Puschel. 2014. Applying the roofline model. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'14)*, 76–85. https://doi.org/10.1109/ISPASS.2014.6844463
[34] OpenMP Architecture Review Board. 2018. OpenMP Application Program Interface. Version 5.0.
[35] Sangyoung Park, Jaehyun Park, Donghwa Shin, Yanzhi Wang, Qing Xie, Massoud Pedram, and Naehyuck Chang. 2013. Accurate modeling of the delay and energy overhead of dynamic voltage and frequency scaling in modern microprocessors. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 32, 5 (2013), 695–708. https://doi.org/10.1109/TCAD.2012.2235126
[36] Miquel Pericàs. 2018. Elastic places: An adaptive resource manager for scalable and portable performance. *ACM Trans. Archit. Code Optim.* 15, 2, Article 19 (May 2018), 26 pages. https://doi.org/10.1145/3185458
[37] Jacques A. Pienaar, Anand Raghunathan, and Srimat Chakradhar. 2011. MDR: Performance model driven runtime for heterogeneous parallel platforms. In *Proceedings of the International Conference on Supercomputing (ICS'11)*. Association for Computing Machinery, New York, NY, 225–234. https://doi.org/10.1145/1995896.1995933
[38] Haris Ribic and Yu Liu. 2016. AEQUITAS: Coordinated energy management across parallel applications. In *Proceedings of the 2016 International Conference on Supercomputing*. 1–12. https://doi.org/10.1145/2925426.2926260
[39] 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)*. Association for Computing Machinery, New York, NY, 513–528. https://doi.org/10.1145/2541940.2541971
[40] Wilfred Gomes Sanjeev Khushu. August 2019. Lakefield: Hybrid Cores in 3D Package. Retrieved from https://newsroom.intel.com/wp-content/uploads/sites/11/2019/08/Intel-Lakefield-HotChips-presentation.pdf.
[41] Robert Schöne, Thomas Ilsche, Mario Bielert, Markus Velten, Markus Schmidl, and Daniel Hackenberg. 2021. Energy efficiency aspects of the AMD Zen 2 architecture. arXiv:2108.00808. Retrieved from https://arxiv.org/abs/2108.00808.
[42] Stephen Shankland. September 2018. iPhone XS A12 Bionic Chip Is Industry-first 7nm CPU. Retrieved from https://www.cnet.com/news/iphone-xs-a12-bionic-chip-is-industry-first-7nm-cpu/.
[43] T. Singh, S. Rangarajan, D. John, C. Henrion, S. Southard, H. McIntyre, A. Novak, S. Kosonocky, R. Jotwani, A. Schaefer, E. Chang, J. Bell, and M. Co. 2017. 3.2 Zen: A next-generation high-performance ×86 core. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC'17)*. 52–53. https://doi.org/10.1109/ISSCC.2017.7870256
[44] Christopher Torng, Moyang Wang, and Christopher Batten. 2016. Asymmetry-Aware work-stealing runtimes. In *Proceedings of the ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA'16)*. 40–52.
[45] Linköping University. [n.d.]. Tetralith. Retrieved from https://www.nsc.liu.se/systems/tetralith/.
[46] Kyle B. Wheeler, Richard C. Murphy, and Douglas Thain. 2008. Qthreads: An API for programming with millions of lightweight threads. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing*. 1–8. https://doi.org/10.1109/IPDPS.2008.4536359

[47] Song Wu, Qiong Tuo, Hai Jin, Chuxiong Yan, and Qizheng Weng. 2015. HRF: A resource allocation scheme for moldable jobs. In *Proceedings of the 12th ACM International Conference on Computing Frontiers (CF'15)*. Article 17, 8 pages. https://doi.org/10.1145/2742854.2742870

[48] Wei Wu, Aurelien Bouteiller, George Bosilca, Mathieu Faverge, and Jack Dongarra. 2015. Hierarchical DAG scheduling for hybrid distributed systems. In *IEEE International Parallel and Distributed Processing Symposium*. IEEE, 156–165. https://doi.org/10.1109/IPDPS.2015.56

Received June 2021; revised December 2021; accepted January 2022