Task-RM: A Resource Manager for Energy Reduction in Task-Parallel Applications under Quality of Service Constraints

Downloaded from: https://research.chalmers.se, 2024-07-17 18:26 UTC

Citation for the original published paper (version of record):
Azhar, M., Pericas, M., Stenström, P. (2022). Task-RM: A Resource Manager for Energy Reduction in Task-Parallel Applications under Quality of
Service Constraints. Transactions on Architecture and Code Optimization, 19(1).
http://dx.doi.org/10.1145/3494537

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

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

(article starts on next page)

Task-RM: A Resource Manager for Energy Reduction in Task-Parallel Applications under Quality of Service Constraints

M. WAQAR AZHAR, MIQUEL PERICÀS, and PER STENSTRÖM,
Chalmers University of Technology

11

Improving energy efficiency is an important goal of computer system design. This article focuses on a general model of task-parallel applications under quality-of-service requirements on the completion time. Our technique, called *Task-RM*, exploits the variance in task execution-times and imbalance between tasks to allocate just enough resources in terms of voltage-frequency and core-allocation so that the application completes before the deadline. Moreover, we provide a solution that can harness additional energy savings with the availability of additional processors. We observe that, for the proposed run-time resource manager to allocate resources, it requires specification of the soft deadlines to the tasks. This is accomplished by analyzing the energy-saving scenarios offline and by providing *Task-RM* with the performance requirements of the tasks. The evaluation shows an energy saving of 33% compared to race-to-idle and 22% compared to dynamic slack allocation (DSA) with an overhead of less than 1%.

CCS Concepts: • **Hardware** → **Power and energy**; • **Computer systems organization** → *Real-time systems*; Multicore architectures; *Embedded systems;*

Additional Key Words and Phrases: Energy efficiency, dynamic resource allocation, quality of service, heterogeneous multi-core architectures, DVFS, run-time systems, precedence constraint task parallel programs

ACM Reference format:
M. Waqar Azhar, Miquel Pericàs, and Per Stenström. 2022. Task-RM: A Resource Manager for Energy Reduction in Task-Parallel Applications under Quality of Service Constraints. *ACM Trans. Arch. Code Optim.* 19, 1, Article 11 (January 2022), 26 pages.
https://doi.org/10.1145/3494537

1 INTRODUCTION

Minimizing energy consumption while meeting performance goals of parallel applications on multicore systems remains an important objective of computer system design. This article considers a general model of parallel applications modeled as a **direct-acyclic graph (DAG)**, where nodes are tasks and edges are dependencies between tasks.

This research has been supported by grants from the Swedish Foundation for Strategic Research under *PRIDE* project grant no :~Dnr CHI19-0048 and the Swedish Research Council under *PRIME* project, grant no :~Dnr 2019-04929. This project has also received funding from the European Union's Horizon 2020 research and innovation programme under grant no :~826647.

Authors' address: M. W. Azhar, M. Pericàs, and P. Stenström, Chalmers University of Technology, Department of Computer Science and Engineering, SE-41296 Göteborg, Sweden; emails: {waqarm, miquelp, per.stenstrom}@chalmers.se.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2022 Association for Computing Machinery.

1544-3566/2022/01-ART11 $15.00
https://doi.org/10.1145/3494537

Our goal is to reduce energy consumption under a QoS constraint on the completion time for a task-parallel application executing on a single **Instruction Set Architecture (ISA) heterogeneous multicore platform (HMP)**, e.g., ARM *big.LITTLE* [25]. QoS specifications allow the resource manager (RM) to allocate enough resources in terms of a task-processor mapping and voltage-frequency (V-F) assignments so that the application finishes close to the deadline, thereby saving energy.

QoS constraints can be established with methods from prior art [8, 20, 22] by assuming that the execution time of each task is known or by establishing an average or upper-bound execution time through measurements. The proposed approach is agnostic to that. For simplicity, upper-bound execution time (UBET), also referred to as worst-case execution time, is assumed in this article. Using UBET and the structure of the parallel application modeled generally as a DAG, which is known at compile-time, one can derive the worst-case schedule length (WCSL) by analyzing the DAG offline for a given scheduling algorithm. Setting the deadline such that it is equal to or greater than the WCSL guarantees that the application completes before it, even if all the tasks would take their respective UBETs to finish.

Allocating enough resources to the tasks necessitates the estimation of each task's performance requirement as defined by its latest finish time (LFT) interpreted as a soft deadline. Considering these requirements, the RM can tune the resource allocation at run-time and save energy. There are three main opportunities to slow down a task's execution to save energy while meeting the programs' deadline. First, tasks often finish earlier than their respective UBETs and produce slack (i.e., Slack $= T_{\text{deadline}} - T_{\text{execution}}$) that can be used by the subsequent task. The second situation appears if there is an imbalance between a task's predecessors where one or more tasks finish earlier than others. Since a successor can only start after all of the predecessors are completed, the predecessor tasks completing earlier than required can be slowed down to save energy.

The third opportunity is enabled by the availability of additional computational resources, i.e., processors. It can happen for several reasons. For example, the user decides to port the application to a new hardware platform with additional cores (identical to the first platform). In another case, the operating system may allocate additional resources to the application as these become available. Additional resources allow the tasks to start earlier than they were originally scheduled for. This early start allows the resource manager (RM) to slowdown tasks to save further energy.

RMs to save energy for task-parallel applications are studied extensively and focus on exploiting some of the scenarios mentioned above either off-line (static allocation) or at run-time (dynamic allocation). First, off-line RM techniques are quite popular because of their low run-time overhead and ability to employ expensive optimization algorithms. In this context, Baskiyar and Abdel-Kader [8] and Alonso et al. [4] provide two algorithms to identify and slow down the tasks that are not on the critical path, by assigning lower Voltage-Frequency (V-F), yet meeting the QoS requirement. Kumar and Vidyathi [22] combine V-F scaling and processor mapping to slow down the tasks while meeting the user-specified QoS in terms of an acceptable extension of the WCSL. Lee and Zomaya [23] propose a scheduling method that minimizes the computational energy by V-F scaling and reduces the communication energy using processor mapping. All these techniques use off-line estimations of execution time, i.e., UBET of each task; thus, they neither cater to the scenario of a task finishing earlier nor do they address the situation of additional processors, leading to over-provisioning of resources and, consequently, more energy consumption.

Second, on-line or dynamic techniques make RM decisions at run-time based on the behavior of the task-parallel application and thus are better suited to save energy. Kang et al. [20] provide an on-line method based on progress tracking of the application to identify each task's early-finish time and slows down the subsequent tasks using Dynamic Voltage-Frequency (V-F) Scaling. The

method starts with an off-line schedule that assigns V-F to the tasks using the application DAG. These tasks are further slowed down at run-time in case they would finish early by re-running the scheduling algorithm on a small set of child tasks, thus incurring considerable overhead. Moreover, this technique neither uses processor mapping nor considers additional processors that limits its energy-saving potential. To fully harness the energy-saving potential, the RM must evaluate all the energy-saving prospects at run time with negligible overhead.

To this end, this article presents a new resource management approach that addresses the shortcomings of prior attempts in two ways. First, it is based on a combination of off-line analysis and on-line resource management to exploit the full potential of energy savings while keeping the overheads low. Second, along with dynamic V-F scaling, it trades V-F for more cores, when available, and also takes heterogeneity of modern multi-core platforms into account.

We propose that the process of task-scheduling must be separate from resource management decisions, such as processor mapping and DVFS. Consequently, our scheme can be used with any state-of-the-art scheduling method. The offline analysis employs the user-defined task-ordering criterion and QoS specification to compute the latest finish times (LFT) of the tasks for a range of processor counts using the UBETs of the tasks. This information is compiled in a so called *Latest Finish Time* table (i.e., LFT-table). The scenario of task imbalance is also analyzed and taken care of in the off-line analysis. Later, the RM uses the LFTs of the tasks—for a specific processor count—as the soft deadline to allocate enough resources to each task at run-time, to save energy. In case more processors are available, the RM uses the LFTs of the tasks for the available processor count from the LFT-table, thus avoiding recompilation of the schedule for the new processor count. Moreover, the RM uses simple, yet accurate, performance and energy prediction models based on architectural performance counters to make decisions.

The proposed scheme is modeled, assuming a sixteen-core platform with eight big and eight LITTLE cores arranged in four clusters employing real execution and energy statistics measured on an ARM big.LITTLE platform (odriod XU-3 board with Exonys 5422 [10]). Results show average energy savings of 33% compared to race-to-idle (for example [2]) and 22% compared to the dynamic slack allocation (DSA) [20] scheme from state of the art. In short, the contributions of this paper are as follows.

— A run-time resource manager, *Task-RM*, for general task-parallel applications to save energy under QoS constraints
— An innovative offline analysis to assist the run-time resource manager
— An evaluation of the energy-saving framework that shows average energy savings of 33% and 22% compared to the race-to-idle and DSA [20], respectively, while keeping overheads low (< 1%)

The rest of the article is organized as follows: In Section 2, we provide a motivational example for our proposal. Section 3 presents the *off-line analysis* and on-line resource manager (i.e., Task-RM). Experimental methodology and implementation-related details are presented in Section 4. Section 5 evaluates the proposed scheme. Related work is discussed in Section 6, and the article is concluded in Section 7.

2 BACKGROUND & MOTIVATION

2.1 Application Model

This article targets precedence-constrained task-parallel applications that are increasingly gaining popularity among modern task-parallel programming models, e.g., OpenMP [15, 17]. Such applications are typically represented by a DAG, where nodes represent tasks and edges represent

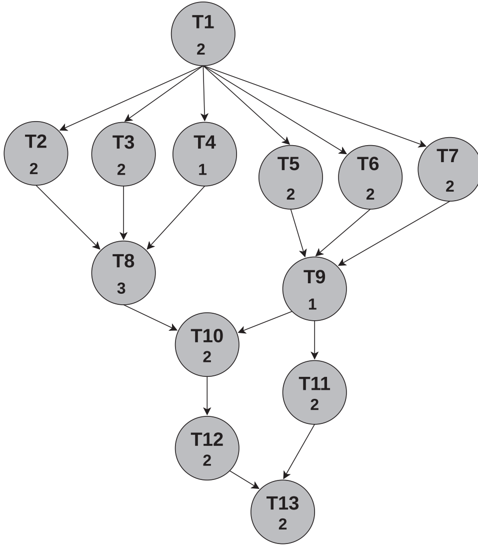


Fig. 1. A synthetic direct acyclic graph (DAG).

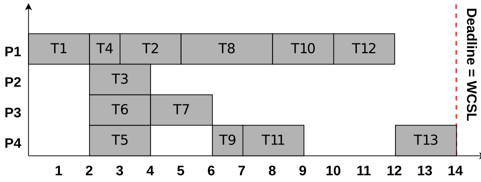
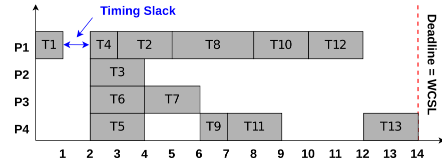
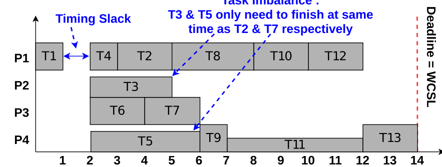


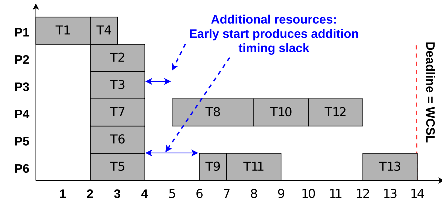
Fig. 2. An off-line schedule.



(a) A scenario depicting early task finish



(b) A scenario depicting early task imbalance



(c) A scenario depicting availability of additional resources

Fig. 3. Scenarios depicting the energy saving opportunities.

dependencies between tasks. In this article, the DAG constitutes a tuple $G = (T, E, U)$, where T is the set of t tasks, E is the set of e edges, and U is the set of t task UBETs. Moreover, the hardware platform is represented by a tuple $H = (P, C, V\text{-}F)$, where P is a set of p processors, C is a set of c core types, and V-F is a set of v legal voltage-frequency pairs. An example of a DAG is shown in Figure 1. Here, the task identities are denoted as Ti, and the UBET of a task is represented by the number at the bottom of each circle.

One can simulate the execution assuming a specific processor allocation and using the tasks' UBETs, the DAG, and a scheduling algorithm to compute an offline schedule. This simulation establishes the **worst-case schedule length (WCSL)**. However, the actual execution behavior may differ, and the RM needs to identify these variations and apply energy-saving strategies. Specification of a deadline aids the RM in applying energy-saving optimizations while meeting the deadline. This is accomplished by assigning performance requirements in terms of soft deadlines or so-called **latest finish times (LFT)** to individual tasks using the **worst-case schedule (WCS)** simulation.

A worst-case off-line schedule of a DAG is presented in Figure 1 using the **earliest finish time (EFT)** scheduling as shown in Figure 2 for a processor count of four. This is the worst-case execution under the assumed scheduling method and processor count. The schedule takes 14 **time-units (TU)** to complete, i.e., the WCSL is 14. The QoS specification must be equal to or greater than the WCSL to be feasible in all execution scenarios. Moreover, to ensure the application is completed before the deadline, all tasks must finish before the scheduled finish times in this worst-case schedule. Thus, soft deadlines, or task LFTs, are initially set equal to the finish times of the tasks as per a worst-case schedule. Such an LFT assignment is shown in Table 1, where, for example, the LFTs

Table 1. Base LFT Table

Processor	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13
4	2	5	4	3	4	4	6	8	7	10	9	12	14

Table 2. LFT Table Adjusted for Task Imbalance

Processor	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13
4	2	5	5	3	6	4	6	8	7	10	12	12	14

Table 3. Final LFT Table After Complete Off-Line Analysis

Processor	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13
4	2	5	5	3	6	4	6	8	7	10	12	12	14
5	2	5	5	5	4	6	6	8	7	10	12	12	14
6	2	5	5	5	6	6	6	8	7	10	12	12	14

of T1 and T2 are set to 2 and 5, respectively, using their finish times in the WCS simulation. There also exists some more optimization that will be discussed in the next section.

2.2 Energy Saving Opportunities

There are three main opportunities that, once identified, can be used for applying energy-saving measures. The first opportunity occurs when a task completes earlier than its UBET. The successor tasks can start early in such a case and can be slowed down to save energy. Such a scenario is depicted in Figure 3(a), where T1 completes at 1 TU; however, its scheduled finish time based on UBET is 2 TU. If we assign the LFTs to the task's successors, the RM can allocate just enough resources to the tasks to finish before their respective LFTs, thus saving energy. This applies to all the tasks in the DAG. For example, using the off-line schedule in Figure 2, we can assign the LFT for each task as shown in Table 1. In the scenario under discussion, the early finish of T1 allows its successors T4, T3, T6, and T5 to start early, but they only need to finish before their respective LFTs of 3, 4, 4, and 4 TU, respectively. The RM can use the slack produced by a task's successor to allocate just enough resources so that the tasks finish before their respective deadlines, thus saving energy.

The second opportunity is to exploit the imbalance between the predecessors of the tasks as illustrated in Figure 3(b). According to the off-line schedule in Figure 2, T3 finishes before its sibling T2. These two tasks, along with T4, are the predecessor of T8, and T8 can only start after the completion of its three predecessors. Thus, relaxing the LFT for T3, equal to the LFT of T2, allows the RM to save more energy. The same opportunity can be exploited in the case of T5 and T11, and their LFTs can also be relaxed to the LFTs of T7 and T12, respectively. The result of this optimization is shown in Table 2.

The third opportunity arises when, during execution, additional processors become available. This scenario allows tasks to start earlier than when they were originally scheduled to start. Hence, it allows more tasks to execute in parallel. Figure 3(c) depicts such a scenario, where six processors are available instead of the original assumption of four. T2 and T7, which were originally scheduled to start after T4 and T6, respectively, can now execute in parallel. This scenario does not only need less resources to T2 and T7 but also allows the allocation of less resources to T4 and T6 as they only need to be completed as late as their siblings. In traditional systems, such a change in computing resources typically requires a re-evaluation of the schedule and LFTs. However, it is not feasible in an on-line scheme because it will incur large overheads. Instead, we propose to construct an LFT

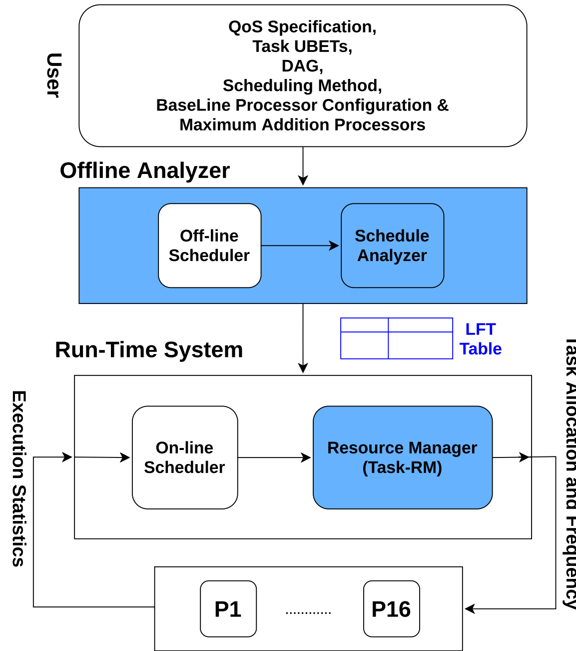


Fig. 4. System Block Diagram.

table with a set of processor counts off-line and provide it to the RM. An example of such an LFT table is shown in Table 3, where the LFT for a processor count of 4, 5, and 6 is shown. Note that, in the case of a given DAG (i.e., Figure 1), there is no advantage of increasing the processor count beyond six because the maximum parallelism available in the example DAG is six.

In summary, identification and exploitation of the aforementioned opportunities is critical for energy reduction and the next section discusses the proposed resource management scheme in detail.

3 RESOURCE MANAGEMENT

3.1 Overview

Our resource management proposal to reduce energy consumption consists of two key components: (1) Off-line analysis and (2) Run-time resource management. Our scheme requires the user to specify QoS in terms of a program deadline, task UBETs, a scheduling method, the baseline processor configuration and additional processor count. Please note that we refer to the scheduling method as the task priority criterion to pick the tasks to be executed next when the number of ready tasks is higher than the available processor count.

The off-line analysis uses this information to compute the LFT table that provides soft deadlines for the tasks. Adherence to these soft deadlines ensures that the program finishes close to its soft deadline. At run-time, the RM uses task LFTs and performance predictions to allocate just-enough resources leading to higher energy efficiency. Figure 4 depicts the system's block diagram, where the system components highlighted in blue are contributions of this article. The off-line analyzer comprises an *off-line scheduler* and a *schedule analyzer*. The RM in the run-time system is invoked whenever a task completes or processors become available and consists of an *on-line scheduler* and a *resource manager* denoted *Task-RM*.

The on-line scheduler finds the tasks that are supposed to be executed next and the *Task-RM* estimates an appropriate resource allocation in terms of processor-mapping and frequency using off-line analysis and a prediction of the task's execution time and energy consumption. Consequently, tasks execute on the estimated configuration (i.e., processor-mapping and frequency) and measured execution statistics (e.g., instruction count) are fed back to the *Task-RM*, which in turn will be used for prediction in the next scheduling epoch. The whole process repeats itself until the execution of the application is complete. Please note that both the off-line and on-line schedulers are not part of our contributions and that our framework is agnostic to the choice of scheduler.

3.2 Off-line Analysis

The off-line analyzer computes the content of the LFT table to be used by the run-time resource manager. This allows our scheme to avoid run-time overhead. The LFT table provides the latest finish times, that serve as soft deadlines, for all tasks. Finishing each task as close as possible to its respective deadline ensures that the program deadline is met and saves energy. The LFT table is computed for a range of processor counts that allow our scheme to work for any number of processors within a predefined range without re-computing the LFT table at run-time. Consequently, the RM can read LFTs of these tasks for a specific processor count at run-time.

The procedure for off-line analysis is depicted in Algorithm 1 in function OffLine_Analysis. To compute the content of the LFT table requires task UBETs, the baseline configuration, and the QoS specification. It comprises four steps. First, an off-line schedule is simulated using task UBETs and the DAG for the baseline processor count, as represented by Compute_Base_Schedule on line 13. This is a trivial step that is well established in state-of-the-art and will not be covered in detail. The baseline schedule ($\text{Schedule}_{\text{base}}$), the baseline LFT, the task ready-time (RT) and the task start-time (ST) are computed in this step. An example of such a schedule is shown in Table 4, where each entry in the table represents a scheduling instant. At each instant, a new task is assigned to any of the processors, as shown in columns P1, P2, P3, P4. The column labeled "Time" represents the scheduling time. If no task is allocated to a certain processor at a certain instant, the corresponding entry is marked by "-1". Moreover, the ready, start, and completion times of the task are also recorded as shown in Table 6, where ST and RT refer to start-time and ready-time, respectively. *Ready-time* is when a task becomes ready after all its predecessor tasks have completed execution, and *start-time* is when a task begins execution. The completion time of each task provides the base LFT table. The completion time of the last task in this schedule defines the overall completion time or makespan. The ST and RT tables will be used in the analysis of schedules.

The next step is to find the imbalance between sibling tasks and relax their LFTs. The Analyze_Schedule function carries out this operation (line 14). The function definition for Analyze_Schedule is shown on line 26. The algorithm loops over every entry in the schedule (line 28) and checks if any processor is available, i.e., is marked by -1. In case a processor is not allocated a task (lines 29–30), the task previously allocated to that processor (Previous_Task) is retrieved (lines 31–32) and checked (line 33). If any of the tasks currently executing is not a successor of the Previous_Task (line 33), then the Previous_Task is again allocated an execution slot on the same processor in the current scheduling instant (lines 34–35). The result of applying this algorithm to the example schedule in Table 4 is shown in Table 5, where the changes are highlighted in blue. Moreover, the LFT of this task is adjusted accordingly. The Analyze_Schedule procedure comprises of two nested "for loops"; the first loop iterates over the number of entries of input schedule and the second loop traverses the number of allocated processors. The number of entries depends on the number of tasks, the DAG, and the available processors. The upper bound case occurs if the processor count is set to one, where the number of entries in the schedule is equal to the task count. Thus, the complexity of this function is $O(t*p)$

ALGORITHM 1: Off-line analysis

```
1:  Notations:
2:  P_base : Baseline number of processors
3:  P_Max : Maximum available processors
4:  Task_program : Total tasks in program
5:  DAG : Directed acyclic graph
6:  LFT_base : Task LFTs for baseline processors adjusted for user-specified QoS.
7:  LFT_Table[][] : LFT table.
8:  ST : Start times of tasks from the base schedule
9:  RT : Ready times of tasks from the base schedule
10: UBET : Array containing task UBETs
11: Deadline_QoS : User specified deadline
12:
13: function OFF-LIN_ANALYSIS
14:     [Schedule_base, LFT_base, RT, ST] = COMPUTE_BASE_SCHEDULE(UBET, P_base, DAG)
15:     LFT = ANALYZE_SCHEDULE(Task_program, LFT_base, Schedule_base)
16:     LFT_Table[][P_base] = LFT
17:     for all P ∈ [P_base + 1 : P_Max] do
18:         [Schedule, LFT] = COMPUTE_SCHEDULE(Task_program, P, LFT_base, RT, ST)
19:         LFT = ANALYZE_SCHEDULE(Task_program, LFT, Schedule)
20:         LFT_Table[][P] = LFT
21:     end for
22:     makespan = LFT_Table[−1][P_base]
23:     QoS_Scaling = Deadline_QoS/makespan
24:     ADJUST_LFT(makespan, Deadline_QoS, LFT_Table)
25: end function
26:
27: function ANALYZE_SCHEDULE(Task_program,LFT_in,Schedule_in)
28:     LFT_new = LFT_in
29:     for all Entry ∈ Schedule_in do
30:         for all Processor ∈ Entry do
31:             if Processor.Allocation == −1 then
32:                 Previous_Entry ← Schedule_in[Entry − 1]
33:                 Previous_Task ← Previous_Entry[Processor]
34:                 if any Task ∈ Entry is not successor of Previous_Task then
35:                     LFT_new[Previous_Task] = Schedule_in[Entry + 1][Time]
36:                     Schedule_in[Entry][Processor] = Previous_Task
37:                 end if
38:             end if
39:         end for
40:     end for
41:     return LFT_new
42: end function
```

The next step involves finding the task LFTs for the range between the base processor count and the maximum processor count as depicted by the For loop on line 16. Every iteration of this loop computes a schedule (line 17) for the processor count between $P_{\text{base}} + 1$ and P_{Max}. The schedule is also analyzed for task imbalance using the already discussed ANALYZE_SCHEDULE function (line 18), and the LFT is compiled and recorded into the LFT table (line 19).

The user-provided program deadline can be equal to or greater than the makespan. The task LFTs must be adjusted according to the user-defined deadline. This is accomplished by merely scaling the LFTs (lines 21–23). First, the deadline specified by the user is divided by the makespan to get an *extension factor*. Completion times of all tasks are multiplied with the *extension factor* to compute the final LFT table.

ALGORITHM 2: Off-line analysis (continued)

```
1:  Notations:
2:  Task_program : Total tasks in program
3:  DAG : Directed acyclic graph
4:  UBET : Array containing task UBETs
5:  LFT_Table[][] : LFT table.
6:  ST : Start times of tasks from the base schedule
7:  RT : Ready times of tasks from the base schedule
8:  function COMPUTE_SCHEDULE(Task_program,DAG,P_IN,RT,ST)
9:      Task_Completed = []
10:     LFT = []
11:     Schedule = []
12:     Time = 0
13:     ALLOCATE_TASK(P0, T0, UBET[T0])
14:     while Task_Completed == Task_program do
15:         if PROCESSOR_FREE() then
16:             [Task_No, T_Completion, Schedule] = CHECK_COMPLETED_TASKS(DAG)
17:             LFT[Task_No] = T_Completion
18:             P_Free = GET_FREE_PROCESSORS()
19:             Tasks_Ready = GET_READY_TASKS(sizeof(DAG))
20:             for all P ∈ P_Free do
21:                 Task_HP = GET_HIGH_PRIORITY_TASK(Tasks_Ready)
22:                 if ( (Time ≥ ST[Task_HP]) OR (ST[Task_HP] ≠ RT[Task_HP]) ) then
23:                     if ( Time + UBET[Task_HP] ≥ LFT_Table[Task_HP][P_base] ) then
24:                         Allocation_Time = Time + UBET[Task_HP]
25:                     else
26:                         Allocation_Time = LFT_Table[Task_HP][P_base]
27:                     end if
28:                     ALLOCATE_TASK(P, Task_HP, Allocation_Time)
29:                 end if
30:             end for
31:         else
32:             Time = Time + 1
33:         end if
34:     end while
35:     return [LFT, Schedule]
36: end function
```

The algorithm for COMPUTE_SCHEDULE function is depicted in Algorithm 2. This function primarily simulates the execution using task UBETs but with some conditions. The function ALLOCATE_TASK kicks off the execution by assigning task T0 to processor P0 (line 15). Then the "while loop" (line 16) executes until all the tasks are executed. In each iteration of the loop, the availability of free processors is first evaluated, i.e., PROCESSOR_FREE (line 14). If processors are free for allocation, the allocation procedure is initiated (lines 15–24); otherwise, the time is incremented (line 26). The allocation procedure consists of several steps. First the tasks are checked for completion by CHECK_COMPLETED_TASKS function (line 15). This function updates the schedule and returns the completed task numbers and the tasks' completion times. The completion times are recorded in the LFT array (line 16). Next, the available free processors are checked (line 17), after which the tasks are checked for their readiness because the successors of the newly completed tasks are also ready for execution now (line 18).

The next step is to allocate every free processor with a task (lines 19–24), where first, the highest priority task is evaluated, i.e., GET_HIGH_PRIORITY_TASK() and then this task is allocated to a free processor, provided that one of the two conditions is met. As for the first condition, the elapsed

Table 4. Base Schedule Using Task UBETs

Entry	Time	P1	P2	P3	P4
1	0	T1	−1	−1	−1
2	2	T4	T3	T6	T5
3	3	T2	T3	T6	T5
4	4	T2	−1	T7	−1
5	5	T8	−1	T7	−1
6	6	T8	−1	−1	T9
7	7	T8	−1	−1	T11
8	8	T10	−1	−1	T11
9	9	T10	−1	−1	−1
10	10	T12	−1	−1	−1
11	12	−1	−1	−1	T13
12	14	−1	−1	−1	T13

Table 5. Schedule Adjusted for Task Imbalance

Entry	Time	P1	P2	P3	P4
1	0	T1	−1	−1	−1
2	2	T4	T3	T6	T5
3	3	T2	T3	T6	T5
4	4	T2	T3	T7	T5
5	5	T8	−1	T7	T5
6	6	T8	−1	−1	T9
7	7	T8	−1	−1	T11
8	8	T10	−1	−1	T11
9	9	T10	−1	−1	T11
10	10	T12	−1	−1	T11
11	12	−1	−1	−1	T13
12	14	−1	−1	−1	T13

Table 6. Ready, Start and Completion Times of Tasks from Base Schedule

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13
Ready Time (RT)	0	2	2	2	2	2	2	5	6	8	7	10	12
Start Time (ST)	0	3	2	2	2	2	4	5	6	8	7	10	12
Completion Time	2	5	4	3	4	4	6	8	7	10	9	12	14

time must be greater than the task's start time as recorded in the *ST-table*. The second condition evaluates whether the task started as soon as it became ready, by comparing the "start time" and "ready time" of the task. In case both the times are equal, the "start time (ST)" of this task must be kept the same. On the contrary, if the task started later after being ready, then it can start execution early. In short, this allows the early start of only those tasks which executed later due to lack of available processors (refer to Table 6). Moreover, the task LFTs are also adjusted to harness additional energy savings. If the finish time (i.e., Start Time + UBET) based on the new start time is less than the "base LFT" (line 23), then the LFT of the task is kept the same (line 24). Alternatively, if the task's finish time (i.e., Start Time + UBET) is greater than the "base LFT," then the LFT of the task is changed to "Start Time + UBET" (line 26). The task is then allocated to the processor for this time duration, i.e., "ST till LFT" calculated above, and the control returns to the start of the loop to schedule the next task. This process is repeated until all tasks have completed execution. The output of this process is the final LFT table, which, if adhered to, would lead to energy minimization and satisfaction of QoS constraints. The time complexity of the COMPUTE_SCHEDULE procedure is $O(t)$, where t is task count in the program.

3.3 Run-Time Resource Management

The objective of the run-time resource management is to allocate enough resources to the tasks so that the program finishes as close to the deadline, specified by QoS, as possible. This is accomplished by allocating appropriate resources to each task so that it finishes before its LFT. Note that the run-time system will only use the final LFT-table produced at the end of the off-line analysis, and no further analysis is done. The *Task-RM* will allocate resources so as the tasks finish before their soft deadlines, as depicted in the LFT table. Every time a task completes or processors become available, the run-time system module is invoked as shown in Figure 4.

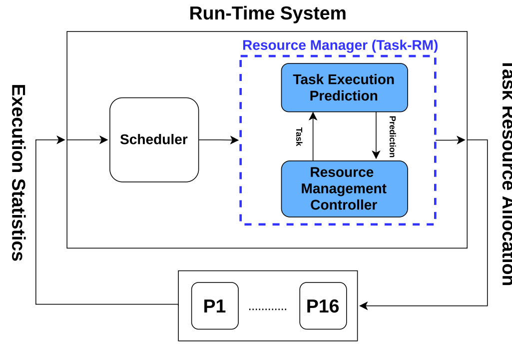


Fig. 5. Block diagram of resource manager.

A block diagram of run-time system detailing the internals of *Task-RM* is show in Figure 5. Here, first the on-line scheduler finds the high-priority tasks next in line for execution, and the *Task-RM* finds an appropriate resource allocation for the tasks. The *Task-RM* consist of two components; first, a resource manager controller as discussed in Section 3.3.1, and second, an architectural prediction mechanism that is used to predict the task execution behavior as presented in Section 3.3.3. The RM controller invokes the predictor for each task and receives the prediction that is used for resource allocation.

3.3.1 Resource Management Controller. The algorithm for the run-time resource manager is depicted in Algorithm 3 in the RESOURCE_MANAGER procedure. This function takes the set of ready tasks to be scheduled ($\text{Task}_{\text{Schd}}$) and the set of available processors ($\text{Proc}_{\text{Free}}$) as input. Note that the scheduler provides as many tasks as the number of free processors, so $\text{Task}_{\text{Schd}}$ and $\text{Proc}_{\text{Free}}$ have equal size.

The RM loops over all the tasks in $\text{Task}_{\text{Schd}}$ and, for each task, it finds a suitable processor. First, the LFT for the task is read (line 10), and processor types of free processors, i.e, big/LITTLE; are identified (line 11). The minimum frequency that can meet the LFT target is predicted for each of the processor types. The energy consumption is predicted using the predicted V-F pair and the execution time. The processor that provides the lowest energy is picked and allocated for the task. The processor and task are consequently removed from the list of available processors, i.e., $\text{Proc}_{\text{Free}}$ and the list of tasks to be scheduled $\text{Task}_{\text{Schd}}$, respectively, (lines 19–20). The process is repeated until tasks are assigned to all the processors.

The outer loop iterates for the number of tasks to be scheduled in a given system epoch and the inner loop iterates over the number of processor types that are constant in a given architecture. This function will be invoked at run-time to find the resource allocation for $\text{Task}_{\text{schd}}$, which is a subset of T. For a given system epoch, the complexity of RESOURCE_MANAGER can be expressed as $O(n)$, where n represents $\text{Task}_{\text{schd}}$. Considering the total run-time of the application the complexity of RESOURCE_MANAGER is $O(t)$, where t denotes the task count in the program.

3.3.2 Execution Time and Energy Prediction Model. The PREDICT_FREQUENCY() and PREDICT_ENERGY() functions are used to predict the frequency and energy consumption. In order to predict the minimum frequency, a simple mathematical model is developed that is based on the execution-time model in Equation (1), where T_{exe}, I, F, CPI_0, MPI_{LLC}, and MP_{LLC} represent the execution time, instruction count, frequency, cycle per instruction base, misses per instruction for last-level cache (LLC), and miss penalty for LLC, respectively. Equation (1) represents the execution time as a combination of compute and memory components. Since a task must finish execution

ALGORITHM 3: Algorithm for resource management controller

```
1: Notations:
2: Task_Schd : Set of ready tasks scheduled to execute next
3: Proc_Free : Set of available processors
4: LFT_Table[][] : LFT table.
5: F_Legal[Core Type] : Available legal frequencies for each core type
6: function RESOURCE_MANAGER(Task_Schd,Proc_Free)
7:     Processor_Count = SIZE(Proc_Free)
8:     Task_Count = SIZE(Task_Schd)
9:     Resource_Allocation[Task_Count] = []
10:    for all Task ∈ Task_Schd do
11:        [Proc_Allocation, Freq_Prediction, Energy_Prediction] ← [Null, , Null, ∞]
12:        LFT = LFT_Table[Task][Processor_Count]
13:        Proc_Types = FIND_PROCESSOR_TYPES(Proc_Free)
14:        for all Proc ∈ Proc_Types do
15:            Freq_temp ← PREDICT_FREQUENCY(TASK, LFT)
16:            Energy_temp ← PREDICT_ENERGY(Task, Freq_temp)
17:            if Energy_temp ≤ Energy_Prediction then
18:                [Proc_Allocation, Freq_Prediction, Energy_Prediction] ← [Proc, Freq_temp, Energy_temp]
19:            end if
20:            Resource_Allocation[Task] = [Proc_Allocation, Freq_Prediction, Energy_Prediction]
21:        end for
22:        Proc_Free.remove(Proc_Allocation)
23:        Task_Schd.remove(Task)
24:    end for
25:    return Resource_Allocation
26: end function
```

before its LFT, the execution time T_{exe} can be equated with a deadline D.

$$T_{\text{exe}} = \frac{I \times CPI_0}{F} + MPI_{\text{LLC}} \times MP_{\text{LLC}} = Deadline \quad (1)$$

Furthermore, solving the equation for frequency, i.e., F gives us Equation (2), where D represents the deadline. The deadline in our case can also be represented as the difference between the task LFT and the current system time leading to Equation (3). The PREDICT_FREQUENCY function implements Equation (3), where the input values of I, CPI_0, and MPI_{LLC} are predicted using a concept discussed later in Section 3.3.3.

$$F = \frac{CPI_0 \times I}{D - MPI_{\text{LLC}} \times MP_{\text{LLC}}} \quad (2)$$

$$F = \frac{CPI_0 \times I}{LFT_{\text{task}} - T_{\text{current}} - MPI_{\text{LLC}} \times MP_{\text{LLC}}} \quad (3)$$

Energy prediction in the PREDICT_ENERGY() function is based on the energy model [21] for dynamic energy as depicted in Equation (4), where α, C, V, F, and T_{pred} represent the activity factor, the capacitance, the voltage, the frequency, and the predicted execution time, respectively.

$$E = \alpha \times C \times V^2 \times F \times T_{\text{pred}} \quad (4)$$

$$E = C_{\text{eff}} \times V^2 \times F \times T_{\text{pred}} \quad (5)$$

Energy consumption depends on the predicted execution time, which depends on the predicted frequency computed in the last step. This equation can be further simplified by replacing the product of the activity factor α and the capacitance C with the effective capacitance C_{eff} resulting in Equation (5). The capacitance is a property of the circuit, and the activity factor primarily depends

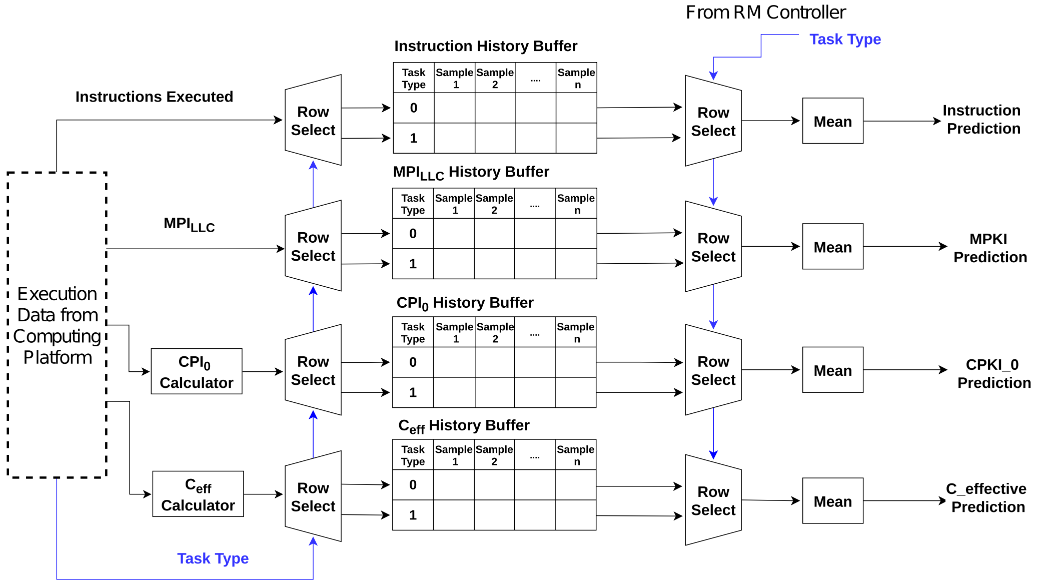


Fig. 6. Prediction Mechanism.

on the application program and the micro-architecture. This article proposes that the effective capacitance values measured from completed tasks can be used to predict the value for future tasks in the same application. Thus, C_{eff} will be predicted for every application at run-time, where details will be provided in Section 3.3.3.

3.3.3 Predictor Architecture. The prediction model discussed in Section 3.3.2 requires prediction of a number of parameters. The block diagram of the predictor is shown in Figure 6 that is used to predict the instruction count (I), the number of misses per instruction (MPI_{LLC}), the base cycle count per instruction (CPI_0) and the effective capacitance C_{eff}. The prediction mechanism is based on storing the old samples of the measured statistics, i.e., I, MPI_{LLC}, C_{eff}, CPI_0 and use averaging for prediction.

The prediction of the parameters, as mentioned above, is made per task type. The task type is defined as the "task functions" in the application source code. These "task functions" are generally different blocks of the algorithm that are invoked repeatedly with a different set of data. For example, the *SparseLU13* application has four tasks that are repeatedly invoked, resulting in a total of 146 tasks in the entire DAG. Hence, the number of task types is four. The reason behind such a design is that the separate instances of a task with different data are likely to exhibit similar execution behavior. Therefore, it is intuitive to design the prediction mechanism accordingly.

The predictor performs two essential tasks. First, it inserts the old sample into the history buffers, and second, it predicts the execution behavior of tasks. As for the first duty, the measured values of I, CPI_0, MPI_{LLC}, and C_{eff} are stored in the history tables using the **first-in-first-out (FIFO)** principle after the completion of each task. Since the values of CPI_0, MPI_{LLC} and C_{eff} can be different for each core type, there exist separate tables for each core type for these statistics. However, the instruction count I is indifferent to the core type, so there is no need to separate tables. Moreover, each table contains separate rows for each task type; thus, the values are stored per task type. CPI_0 and C_{eff} are derivative values, so they are computed on-the-fly using Equations (6) and (7), respectively, where $MP_{\text{LLC(cycles)}}$ and T_{exe} represent the miss penalty for LLC misses in cycles and

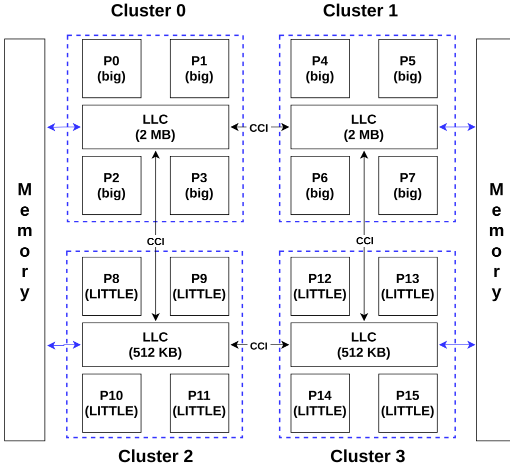


Fig. 7. Hardware platform.

Table 7. Voltage-Frequency for Exynos-5422

Frequency (M-Hz)	big-core Voltage (mV)	LITTLE-core Voltage (mV)
2000	1300	-
1900	1225	-
1800	1175	-
1700	1137.5	-
1600	1100	-
1500	1062.5	-
1400	1037.5	1250
1300	1025	1200
1200	1000	1150
1100	975	1112.5
1000	950	1075
900	925	1037.5
800	900	1000
700	-	962.5
600	-	925
500	-	900

the execution times of the tasks, respectively:

$$CPI_0 = CPI - MPI_{\text{LLC}} \times MP_{\text{LLC}} \tag{6}$$

$$C_{\text{eff}} = \frac{E}{V^2 \times F \times T_{\text{exe}}} \tag{7}$$

The second duty is to predict the execution behavior of the tasks. First, *RM controller* inputs the task type to the predictor as shown in Figure 6. As a result, the multiplexers choose the appropriate row, i.e., task type, from the table that holds the old samples. Next, an averaging unit computes the mean from all the valid samples in the row. This mean value is used as a prediction by the RM controller to find a suitable configuration to execute the given task. Finally, the task executes at the predicted configuration, and the execution statistics of the task are fed back to the prediction history buffer as explained above.

Please note that, at the start of the execution, the history buffers are empty and, once the first task completes an entry in the history buffer, is filled for a specific core and task type. The RM executes the first task at the maximum frequency available on a given core type. Subsequent tasks of the same type and the same core can be predicted afterwards.

4 EXPERIMENTAL METHODOLOGY

4.1 Hardware Platform

This article assumes a hardware platform inspired from the ARM big.LITTLE [25] architecture and is shown in Figure 7. Eight big and eight LITTLE cores are arranged in four homogeneous clusters, where two clusters consist of big cores and two consist of LITTLE cores. Cores within a cluster share the LLC and the **cache-coherent interconnects (CCI)** connect the neighboring LLCs to allow fast data transfer between the clusters. The big and LITTLE cores are assumed to be *ARM* CORTEX A15 and A7, respectively, same as in the Exonys 5422 [10] SoC available on an *ODROID-XU3* board. The A15 is a performance-oriented, out-of-order core while the A7 is an energy-efficient, in-order core. Each core has a 32-KB private L1 cache. The *big cluster* has a 2-MB shared LLC while the *LITTLE cluster* has a 512-KB shared LLC. In short, one can assume this

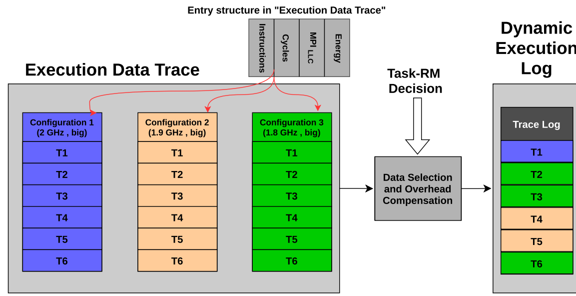


Fig. 8. Simulation methodology.

hardware as an extension of ARM big.LITTLE that offers a higher level of parallelism. All the cores are assumed to be able to operate on voltage-frequency levels that are the same as available in the *ODROID-XU3* board, as shown in Table 7.

4.2 Simulation Methodology

The central theme of our simulation methodology is to closely emulate the application's execution behavior on real hardware, i.e., odroid XU3. Thus, as a first step, we execute the applications on real hardware to record the *execution data trace* for all possible distinct energy footprint scenarios. In our case, these scenarios or so-called configurations refer to unique combinations of core types and voltage-frequency (V-F) pairs. In short, each DAG is executed on the odroid board on all configurations, and an *execution data trace* at a granularity of a task is recorded.

An example scenario depicting this is shown in Figure 8. The measured values include the instruction count, cycles, LLC misses, and energy consumption. The second step is to simulate a dynamic execution using above-mentioned execution data trace employing the *Task-RM* scheme.

Execution data for each task is picked from one of the execution traces based on *Task-RM* decisions, as shown in Figure 8. For example, the first task T1 executes at the fastest configuration, i.e., big core - 2 GHz, and data for T1's execution at the corresponding configuration is selected from the execution data trace.

Then, overheads associated with V-F switching, core migration, and *Task-RM* are added, and the resultant data is both recorded in a *dynamic execution log* and fed back to *Task-RM* for prediction purposes, as shown earlier in Figure 5. Finally, *Task-RM* uses it to predict the resource allocation for the next-in-line task, i.e., T2. As shown in Figure 8, *Task-RM* predicts to execute T2 at 1.8 GHz-big core; hence, the data for T2 at 1.8 GHz - big core is selected from the execution data trace. This data is recorded in the dynamic execution log and fed back to *Task-RM*. This process repeats until the completion of the application. For each task, the data for a predicted configuration by *Task-RM* is read from the execution data trace and recorded in the dynamic execution log. At the end of the application, the dynamic execution log is aggregated to compute the result for the complete execution.

4.3 Simulation Setup

The simulation framework employed in this article is outlined in Figure 9. The first step is the instrumentation of the application source code for two purposes. The first purpose is to extract the task dependency information to generate a DAG, while the second is to record the execution data trace. The DAG is essential for the scheduler to execute the application with correct functional behavior. The source code is transformed to print out task dependency information while executing on a faster non-native machine. The *DAG generator* processes the information

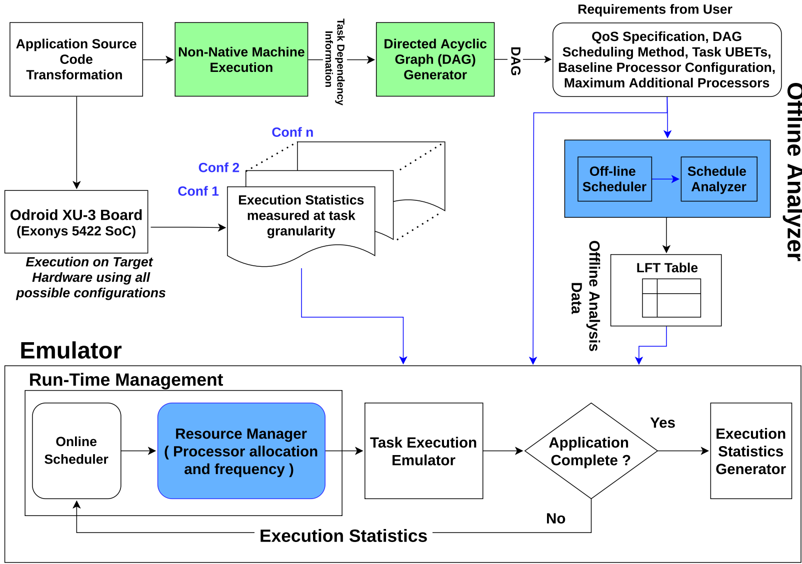


Fig. 9. Experimental methodology.

to generate a DAG for the application. The DAG is both used for off-line analysis and run-time resource management. Both these steps are highlighted in green in the figure. Please note that this is an optional step, because if the user provides the DAG, then this step becomes redundant.

The second transformation of the source code is done to measure the task execution statistics by inserting routines to read the hardware performance counters before and after task execution. The modified applications are executed on the ODROID-XU3 board containing the Exonys 5422 [10] SoC on all the configurations. In context of the chosen hardware, there are thirteen V-F settings for the big-cores and ten for the LITTLE-cores, which amounts to 23 configurations in total. The execution statistics are measured for each task while executing applications using a single thread.

This execution data trace is then used for off-line analysis and run-time simulation. In context of our proposal, the user provides the UBETs and the baseline configuration, both of which are then used to carry out off-line analysis and establish the LFT table. Off-line analysis, as discussed earlier, computes the LFT for a range of processor counts from the baseline configuration to the maximum processor count as specified by the user.

The run-time simulation uses the execution data trace, DAG, and LFT to compute the modified schedule. The *resource manager* block is the same as explained in Section 3. Once the *Task-RM* determines the task mapping and V-F assignment, the "Task Execution Simulation" block records the task's execution from the real execution data trace in the schedule. Concerning the resource sharing (i.e., shared LLC), the execution statistics might change as sharing could induce destructive/constructive interference that will result in different LLC misses, task execution time and energy. The exact effect of this interference is difficult to establish considering the scope of this paper. Instead, we argue that this interference will be the same for all the schemes, i.e., Race-to-Idle, Oracle, and *Task-RM*; thus, it is fair to ignore its effect. In short, this simplistic model allows us to carry out the experiments quickly.

The process of task scheduling and resource allocation is continued until all the tasks are completed. Once the application is complete, the schedule is analyzed by the *Execution Statistics Generator* (see Figure 9) to compute the evaluation statistics.

Table 8. Applications Used Along with Their Characteristics

Application	Kernel	Input/Size	Task Count	Task Types	Execution Types
SparseLU		block size=13	146	4	0.29
SparseLU		block size=20	395	4	0.84
SparseLU		block size=30	1255	4	2.82
SparseLU		block size=40	2890	4	6.65
SparseLU		block size=50	5550	4	12.92
Heat Diffusion	gauss	antoniu	810	3	9.09
Heat Diffusion	gauss	testgrind2	50	3	15.6
Heat Diffusion	gauss	testgrind2-regions	50	3	0.54
Heat Diffusion	jacobi	test-jacobi	1010	4	80.1
Heat Diffusion	jacobi	testgrind	260	4	79.29
Heat Diffusion	redblack	test-redblack	810	5	34.86
Heat Diffusion	redblack	test-redblack2	2010	5	22.39

4.4 Implementation & Overheads

The run-time resource manager, i.e., *Task-RM,* is implemented in C++ and executed on real hardware to establish its execution time and energy overheads. These values are recorded as overheads whenever the *Task-RM* is invoked in the simulation framework.

Other overheads include DVFS switching (assumed to be 10 microseconds [26]) and performance-counter reading overhead (measured to be 500 cycles per counter). Since the off-line analysis is done once for the lifetime of the application, its overheads are not considered.

4.5 Benchmarks

This article employs applications from the BSC Application Repository (BAR) [9] for evaluation. These applications are written in OmpSs, where successive iterations of algorithms are parallelized using precedence constrained tasks, and thus are a good fit for our proposed scheme. However, we have modified these benchmarks to comply with the dependence model of OpenMP 4.0 for evaluation. This includes syntax transformation of the task specifications and adding data dependency clauses. Moreover, OmpSs implementation lacks the specification of parallel regions as parallelization is done by *nanos* at run-time. In case of OpenMP, one needs to specify the parallel regions that encapsulate the tasks and the said modification is also carried out in the code. The resultant code is published on-line [5] and available for future use.

The applications used in the evaluation are listed in the Table 8 and include four different kernels with various inputs. In total, there are twelve unique workloads that are used for evaluation. To demonstrate the variety of the workloads considered, we also provide a number of workload characteristics, i.e., the number of tasks, the number of task types, and the total execution times. Here, it can be seen that workloads cover a wide spectrum in terms of the above-mentioned characteristics to evaluate our proposed scheme.

4.6 Systems Evaluated

Three schemes are evaluated in this article, the details of which are explained below:

Race-to-Idle. The race-to-idle policy (for example, [2], [11], and [28]) is one of the established methods for energy saving, where the application is executed at the fastest possible configuration and processors are powered down after finishing until the deadline. In context of this article, race-to-idle is assumed as the scheme that executes applications using all the available cores on the fastest frequency available. This scheme will be referred to as *RTI* in the context of this article.

Dynamic Slack Allocation (DSA). The Dynamic Slack Allocation (DSA) [20] scheme from state-of-the-art is used for comparison. This proposal initially allocates DVFS settings offline and later readjusts them if a task finishes earlier or later than anticipated while keeping the processor allocation unchanged. The slack gained by early finish is allocated to a small selection of descendants by recomputing their DVFS settings. In this context, we use the *K3Dependent* variant where descendants up to three levels are selected for slack allocation. This scheme will be referred to as DSA in text and figures.

Oracle. Oracle is used for comparison to evaluate the potential of energy savings and efficacy of the proposed *Task-RM*. Here, Oracle has perfect prediction of execution time and energy. Thus, it can determine the optimal processor allocation and V-F setting.

Task-RM. Our proposed *Task-RM* scheme is compared against the above-mentioned schemes.

4.7 Assumptions: User Provided Inputs & Design Parameters

Our proposed scheme requires certain parameters/inputs from the user which must be assumed in order to carry out the evaluation. In the following, we provide details of these user specifications used for evaluation.

Baseline Configuration. The baseline configuration is chosen to be two big-cores.

UBET. The tasks' UBETs are also required. However, in this paper, they are established using measurement-based deterministic timing analyses (MBDTA) [1] that is an acceptable approach in systems with low criticality. The tasks are instrumented to measure the execution times using hardware performance-counters while executing the application several times (i.e., 10) at the big-core using 2-GHz frequency. The highest observed execution time (HOET) per task type is used as the UBET for that task. Note that applications typically consist of a limited set of tasks that are instantiated multiple times. Thus, the sample size for the measurement is quite big (i.e., No of runs × Number of task instances per execution).

QoS Specification. Specification of QoS is a key requirement for evaluation and in this context the applications' deadline is set to the WCSL, which is established using simulation of the execution on the baseline configuration using the tasks' UBETs.

Scheduling Method. The scheduling methods determine the next task to be executed among the available tasks, in case the ready tasks are greater than the free processors. Heterogeneous Earliest Finish Time (HEFT) [32] is used in this evaluation to schedule tasks. In the case of the off-line scheduler, the task UBETs determine the finish time, and the task with the earliest finish time is prioritized. On the other hand, the run-time scheduler assumes the tasks' LFTs as the finish time.

History Buffer Size. The size of the history buffers in the predictors is set to ten.

5 EVALUATION

In this section, we evaluate the merits of *Task-RM* in terms of the energy savings, its ability to finish close to the deadline, the overheads incurred and the accuracy of the predictors. Energy savings will be presented for two cases. First, for the case of a fixed processor count in Section 5.1 and second, for the case where additional processors are allocated in Section 5.4. The energy savings are computed using Equation (8), where $\text{Energy}_{\text{race-to-idle}}$ and $\text{Energy}_{\text{scheme}}$ refer to the energy consumption of the race-to-idle and the other scheme, i.e., Oracle or Task-RM under consideration, respectively. The prediction accuracy and analysis of the ability to finish close to the deadline, i.e.,

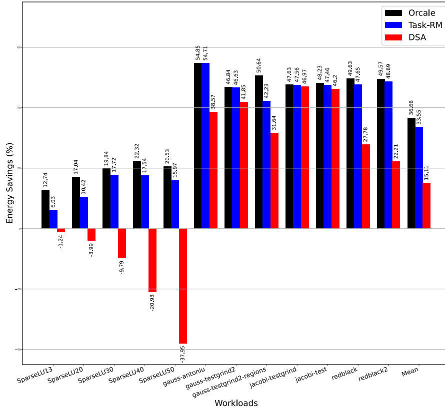


Fig. 10. Energy savings for the baseline processor allocation.

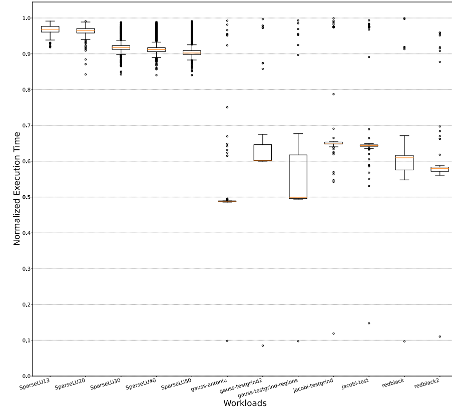


Fig. 11. Execution time distribution compared to UBET.

slack, is presented in Sections 5.2 and 5.3, respectively.

$$\text{Energy Savings}(\%) = 100 \times \frac{\text{Energy}_{\text{race-to-idle}} - \text{Energy}_{\text{scheme}}}{\text{Energy}_{\text{race-to-idle}}} \tag{8}$$

5.1 Energy Savings - Fixed Processor Count

In this section, we present the energy savings when the processor count is set equal to the baseline allocation, i.e., 2-big cores. The results are shown in Figure 10 where the x-axis shows different workloads, and the y-axis represents the percentage energy savings that are computed with respect to the energy consumption of *Race-to-Idle* scheme. Three schemes, i.e., Oracle, *DSA*, and *Task-RM*, are represented through bar graphs, as indicated in the legend. As expected, Oracle is shown to have maximum energy savings, i.e., 36.66% on average. Our proposed *Task-RM* scheme performs very close to Oracle with energy savings of 33.55% on average, showing its effectiveness in harnessing the energy savings on offer. The *DSA* scheme shows an energy saving of 15.11% with respect to *Race-to-Idle*. Moreover, *Task-RM* provides approximately 22% more energy savings compared to *DSA*.

The energy savings for various workloads show a big variation. As discussed earlier in Section 2, one of the sources of energy savings is the deviation of the execution time of the task from the UBET of that same task (i.e., early finish). To verify this reasoning, we derive a box plot of the execution times of the tasks in Figure 11. Here, the execution times are normalized to the UBETs of the tasks. A box in a *boxplot* represents the **interquartile range (IQR)**, i.e., the 25th to 75th percentile, and the orange line in the box represents the *median*. The top end of the box in the boxplot represents the third quartile (Q3). Similarly, the bottom end of the box represents the first quartile (Q1).

The *upper* and *lower whisker* are at $Q3 + 1.5 \times IQR$ and $Q1 - 1.5 \times IQR$, respectively. The range between the lower whisker to Q1 represents the lower 25th percentile, and the range between Q3 to the upper whisker represents the last upper 25th percentile. If more samples, i.e., task execution times, are far below one, i.e., UBET, the more energy savings will be on offer as more tasks have slack that can be used to slow down the execution.

Examining the boxplot, one can find a correlation between the execution time distribution and energy savings of the tasks. For example, *SparseLU13* have most of the samples pretty close to UBETs and thus have relatively small energy savings, i.e., 12.49% for Oracle and 6.03% for Task-RM.

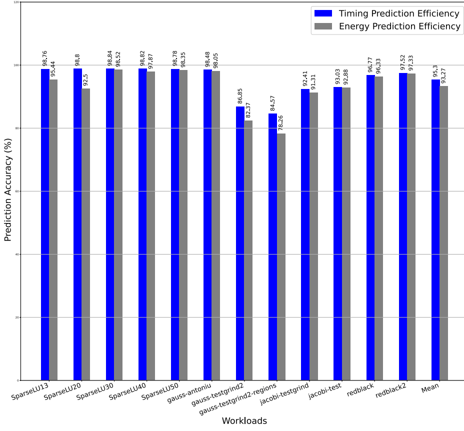


Fig. 12. Timing and energy prediction accuracy.

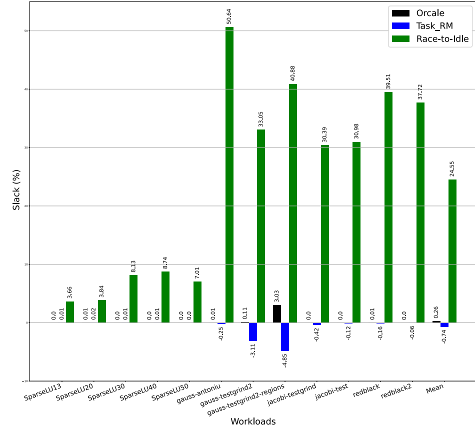


Fig. 13. Percentage slack.

Similarly, the box plot shows that execution times of SparseLU20-50 lie close to their UBET (i.e., upper whisker to lower whisker lie between 0.9 and 1 on the y-axis). This means that there isn't a lot of available slack for these particular benchmarks and, consequently, they show relatively low energy savings.

On the other hand, for example, *gauss-antoniu* has most of the samples (i.e., *IQR*, the lower whisker to Q1, Q3 to the upper whisker) around 0.5 approximately. This means that most of the tasks have 50% slack. The energy savings of 54.85% for Oracle and 54.11% for Task-RM endorse the hypothesis that farther the tasks' execution times from UBET more will be energy savings (see Figure 10). The remaining benchmarks, i.e., gauss-tesgrind2 – redblack2, also show similar behavior where most of their task execution times are far less than their respective UBETs (as evident from the box plot). Thus, they have far higher energy savings. Therefore, it is fair to conclude that the more significant deviation is between the UBET and execution time, the more energy savings will be on offer.

In the same context, the *DSA* scheme consumes more energy than race-to-idle for benchmarks that have less available slack, i.e., SparseLU13-50, and performs better for the benchmarks with more available slack. The reason behind it is the high overheads involved in repeatedly determining task descendants and applying static scheduling algorithms for slack re-allocation at run-time. If higher energy saving is available, it can offset the overhead, but if the available slack is limited, less energy saving is on offer, and hence the overheads can offset the savings resulting in higher energy consumption. One key factor in this regard is the size of the DAG, as it affects the number of descendants in each step and thus affects the overheads. This is the reason why DSA performs worst for the "SparseLU 50" workload as it has the biggest DAG with 5,550 tasks and very low (i.e., ≈10%) slack as shown in the box plot.

5.2 Prediction Accuracy

The accuracy of the timing and energy prediction mechanism is key to analyze the effectiveness of the proposed *Task-RM* scheme, so the prediction accuracy is presented in Figure 12. Here, the workloads are depicted on the x-axis and the percentage accuracy is depicted on the y-axis. The accuracy is computed by using "$\text{Accuracy}(\%) = 100 \times (1 - \text{ABS}(1 - \frac{X_{predicted}}{X_{real}}))$" equation, where X_{real} and $X_{predicted}$ represent the real and predicted values, respectively, and ABS() means the absolute function.

Energy and timing predictions show a considerable high accuracy with an average of 95% and 93% for timing and energy predictions, respectively. Please note that the prediction accuracy depends on the variance in task execution times. If execution times are spread over a large range, predictions will be less accurate and vice-versa. Comparing the *boxplot* in Figure 11 and prediction accuracy in Figure 12, one can draw some correlations. For example, the *IQR* range (shown by the *box*) is pretty narrow for most benchmarks except *gauss-testgrind* and *gauss-testgrind-regions*. Therefore, the prediction efficiency for *gauss-testgrind* (Timing-90%, Energy-87%) and *gauss-testgrind-regions* (Timing-89%, Energy-80%) is minimum among the workload set and below average.

5.3 Deadline

In this section, the ability of various schemes to finish close to the application deadline is analyzed as shown in Figure 13. Here, the percentage of slack normalized to the application deadline i.e., $100 \times \frac{T_{\text{deadline}} - T_{\text{completion}}}{T_{\text{deadline}}}$ is presented on the y-axis, and the x-axis shows the workloads. As expected, Oracle mostly finishes close to the deadline with an average percentage slack of 0.26%, and *Task-RM* performs well with an average of -0.74% that means *Task-RM* misses the deadline by a small margin. The race-to-idle scheme finishes way before the deadline (i.e., average slack of 24%) as this scheme does not consider slowing down to save energy. It is interesting to correlate that workloads showing the lowest timing and energy prediction accuracy, i.e., *gauss-testgrind* and *gauss-testgrind-regions* miss the deadline by the highest margins, i.e., −3.11% and −4.85%, respectively. Thus, it is fair to conclude that the execution behavior of the tasks affects the prediction accuracy and, consequently, affects the energy savings and deadline accomplishment. However, since the *Task-RM* is continuously adjusting the resource allocation, it effectively mitigates the loss of efficiency to finish close to the programs' deadline and saves considerable energy.

5.4 Overheads

The proposed *Task-RM* scheme has minimal overheads owing to its hybrid approach, where most of the analysis is carried out offline, and resource allocation is done at run-time as shown in Figure 14. The timing and energy overheads of the *Task-RM* scheme are 0.6% and 0.7% compared to the execution time and energy consumption, respectively. In contrast, the *DSA* scheme shows average timing and energy overhead of 9% and 6%, respectively. The significantly higher overheads of *DSA* are due to the repeated application of static scheduling algorithms. Thus, static scheduling algorithms, i.e., DSA, are not suitable for runtime use, and our scheme *Task-RM* is much better in limiting the overheads while providing significant energy savings.

5.5 Energy Savings with Additional Processor Allocation

Next, we evaluate the energy savings with the allocation of additional processors. The results are shown in Figure 15. Here the x-axis shows the processor allocations, and the y-axis represents the percentage of energy savings. The labels on the x-axis represent the processor allocation, where "B" and "L" stand for big and LITTLE cores, respectively. The number after the letters "B" and "L" represents the processor count, and for example, label *B4L2* means an allocation of four big and two LITTLE cores. The core allocations are sorted in increasing number of cores, and energy savings increase as more cores are allocated, but they saturate at *B8L4* with maximum energy savings of 58.8% for Oracle and 55.6% for Task-RM, respectively. Note that the availability of additional cores only increases energy savings if there are ready tasks to execute (i.e., parallelism in DAG). That is the reason that energy savings almost saturates beyond an allocation of four (i.e., *B4L0*) or six cores (i.e., *B4L2*).

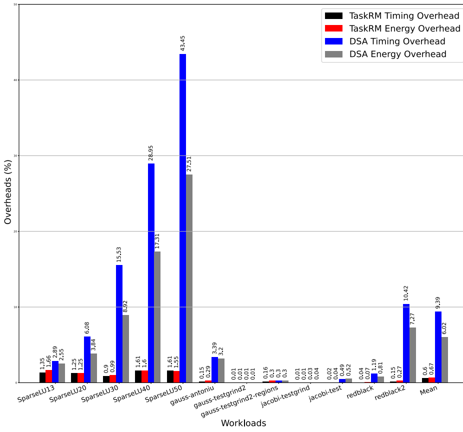


Fig. 14. Timing and energy overhead for baseline processor count.

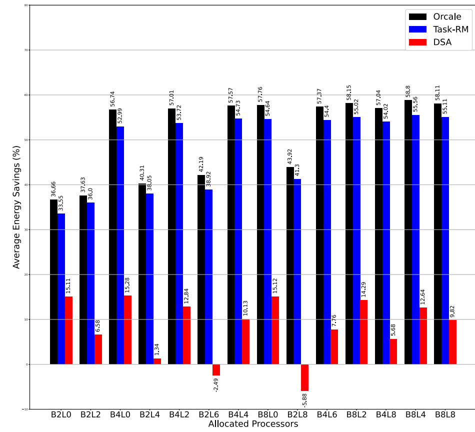


Fig. 15. Energy savings for additional processors.

The *DSA* scheme is unable to use additional processor allocation for energy savings as it sticks to the processor allocation from the off-line schedule and only modifies the DVFS settings. Hence, the energy savings for the DSA are the same as for the baseline processor allocation. However, as shown in Figure 15, DSA has diminishing energy savings in comparison to the **race-to-idle (RTI)** scheme when additional "LITTLE" cores are allocated. The reason for this is that the energy consumption for RTI changes with increased processor allocation. Remember that RTI uses all the available cores at the highest frequency. The energy consumption at the highest frequency on the "LITTLE" core is less than the energy consumption of the same task at the highest frequency on the "big" core. Thus, with the allocation of additional "LITTLE" cores, race-to-idle energy consumption tends to reduce. In contrast, the energy consumption of DSA remains the same; thus, the energy savings diminish.

The energy savings remain the same for the case when only additional "big" cores are allocated to the RTI, i.e., "B4L0" and "B8L0". However, DSA performs worse compared to RTI for the processor allocations with more "LITTLE" cores than "big" cores, e.g., "B2L6", "B2L8". This is understandable as the more the "LITTLE" cores, the less the energy consumption of the RTI; consequently, the DSA scheme will have more energy consumption. This fact that energy consumption of the RTI decreases with allocation of additional "LITTLE" cores further strengthens the effectiveness of our scheme, i.e., *Task-RM*, as it consistently performs better.

In short, the evaluation shows that our proposed scheme *Task-RM* is effective in exploiting energy savings (33%) of the Oracle (36%) with negligible overheads. *Task-RM* performs considerably better (i.e., 22%) than run-time use of static scheduling algorithms as proposed by *DSA*. Moreover, the allocation of extra processors provides additional opportunities to save energy, and the ability of *Task-RM* to adjust to these changes is critical to the objective of saving energy. Other state-of-the-art schemes, i.e., *DSA* lack the ability to harness the opportunities that arise with additional processor allocation. Offline analysis of these situations allows the compilation of a *LFT table* that enables such intelligent resource allocation at run-time without the need for re-scheduling or re-computation of the LFT-table.

6 RELATED WORK

Task scheduling and resource allocation has been a well-studied subject over the years [35], in which the primary objective is the reduction of the program's completion time. However, this

article targets the energy efficiency, which has emerged as a vital design metric in recent years [36]. In this context, a popular approach is not to explicitly consider the QoS requirements and apply energy-saving optimizations while maintaining the same performance level [29, 30, 33]. However, this article focuses on saving energy under a QoS constraint.

Considering QoS, the initial proposals target sequential applications and mainly employ DVFS [16, 18, 31], thread placement [13, 27] on **heterogeneous multi cores (HMP)**, or a combination of DVFS and thread placement [7] to save energy. In contrast, this article targets parallel applications.

Data and task parallelism are two popular parallel programming models. For example, *OpenMP for loops* has attained considerable attention. In this regard, De Sensi et al. [12] use a linear-regression-based prediction model for controlling thread-to-socket allocation, core count, and DVFS to provide desired performance or power consumption. Similarly, Donyanavard et al. [14] propose a method to reduce power for a given performance target by mapping the threads/processes to cores using an off-line trained binning-based prediction model. On the other hand, the proposal [6] provides a solution that analyzes the slack-energy tradeoff at run-time to allocate the DVFS, core-type, and core-count to iterative data-parallel applications. All these proposals cater to the data-parallel model (e.g. OpenMP for loop); however, this article mainly targets precedence-constrained task-parallel applications.

Existing solutions comprise two broad categories: static techniques that make off-line/static resource management decisions (i.e., task scheduling, processor mapping, and DVFS assignment) and on-line/dynamic techniques that make decisions at run-time. In static methods, Xie and Qin [34] provide an algorithm for mapping the tasks on a distributed heterogeneous platform. Kumar and Vidyarthi [22] provide an off-line method to assign DVFS to tasks under a QoS constraint on the completion time. Similarly, Baskiyar and Abdel-Kader [8] offer an off-line technique to slowdown the tasks that do not belong to the critical path using DVFS. Jejurikar and Gupta [19] propose an off-line technique that employs a combination of processor sleep and DVFS to reduce energy consumption. Ali et al. [3] present an off-line approach utilizing a genetic algorithm to efficiently map tasks on a shared-memory HMP. Lee and Zomaya [23] propose an off-line scheduling method that minimizes the computational energy by DVFS and reduces the communication energy by using processor mapping.

Off-line techniques use **worst-case execution times (WCET)** of tasks; however, their run-time behavior could differ. Thus, off-line resource management decisions based on WCET can lead to over-provisioning and, consequently, more energy consumption. On the other hand, dynamic on-line techniques, which are the particular foci of this article, make resource management decisions at run-time while monitoring application behavior and thus are better suited to exploit the energy savings. Li et al. [24] propose a hybrid method, where execution starts with an energy-efficient off-line schedule based on slowing down tasks using DVFS. Later, if the execution behavior of tasks deviates from the assumed average-case execution time, the scheduling algorithm is re-run. A similar proposal is provided by Kang and Ranka [20] where an off-line schedule determines the initial DVFS setting that is later re-evaluated by employing an exhaustive algorithm on a subset of future tasks in case the program is progressing ahead of the deadline. These techniques incur considerable overheads by repeatedly re-running the scheduling algorithm at run-time for the remaining portion of the DAG. Moreover, they fail to realize the full potential of energy saving, only employing DVFS and ignoring heterogeneity. Second, they can also not deal with the scenario of changing computational resources that further limits its energy-saving potential. Our proposal only computes the necessary soft deadlines for the tasks offline, but the scheduling, processor mapping, and DVFS assignments are evaluated on-line using predictions based on hardware performance counters.

7 CONCLUSION

In this article, we have demonstrated that through appropriate allocation of resources to the tasks in a precedence-constrained task-parallel application, the resource manager can save considerable energy, i.e., 33% and 22% compared to race-to-idle and dynamic slack allocation [20] schemes respectively, yet meeting the applications' deadlines. The critical insight is that the execution behavior of the tasks may change at run-time resulting in an earlier completion compared to its UBET. Consequently, the program finishes before the deadline.

Our proposed framework identifies this variance in execution time by monitoring each task and employs energy-saving optimizations, including the DVFS and core processor mappings in heterogeneous platforms. In order to estimate the performance requirements of the task, a thorough analysis of the application DAG is required, and doing that at run-time could lead to increased energy consumption. Thus, a novel scheme is proposed where the run-time resource manager is assisted by offline analysis that provides the soft deadlines, i.e., LFTs for the tasks, by analyzing the application DAG and UBETs of the tasks. The run-time resource manager uses the tasks' LFTs and behavioral prediction based on hardware performance counters to allocate an appropriate amount of resources to the tasks. The proposed resource manager tries to adhere to task LFTs as soft deadlines, ensuring the program completion is close to the deadline.

Moreover, offline analysis computes the LFTs for the scenario of additional processor availability, which opens the door for extra energy savings of up to 55.6% compared to race-to-idle. Last but not least, the resource manager incurs less than 1% of timing and energy overheads.

REFERENCES

[1] J. Abella, C. Hernandez, E. Quiñones, F. J. Cazorla, P. R. Conmy, M. Azkarate-askasua, J. Perez, E. Mezzetti, and T. Vardanega. 2015. WCET analysis methods: Pitfalls and challenges on their trustworthiness. In *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*. 1–10. https://doi.org/10.1109/SIES.2015.7185039

[2] Susanne Albers and Antonios Antoniadis. 2014. Race to Idle: New algorithms for speed scaling with a sleep state. *ACM Trans. Algorithms* 10, 2, Article 9 (Feb. 2014), 31 pages. https://doi.org/10.1145/2556953

[3] H. Ali, X. Zhai, U. U. Tariq, and L. Liu. 2018. Energy-efficient heuristic algorithm for task mapping on shared-memory heterogeneous MPSoCs. In *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. 1099–1104.

[4] P. Alonso, M. F. Dolz, R. Mayo, and E. S. Quintana-Ortí. 2011. Improving power efficiency of dense linear algebra algorithms on multi-core processors via slack control. In *2011 International Conference on High Performance Computing Simulation*. 463–470.

[5] M. Waqar Azhar. 2021. BSC application repository OpenMP transformation. (2021). https://github.com/waqarazhar/BAR_OpenMP.

[6] M. Waqar Azhar, Miquel Pericàs, and Per Stenström. 2019. SaC: Exploiting execution-time slack to save energy in heterogeneous multicore systems. In *Proceedings of the 48th International Conference on Parallel Processing (ICPP 2019)*. ACM, Article Article 26, 12 pages. https://doi.org/10.1145/3337821.3337865

[7] M Waqar Azhar, Per Stenström, and Vassilis Papaefstathiou. 2017. SLOOP: QoS-supervised loop execution to reduce energy on heterogeneous architectures. *ACM Transactions on Architecture and Code Optimization (TACO)* 14, 4 (2017), 1–25.

[8] Sanjeev Baskiyar and Rabab Abdel-Kader. 2010. Energy aware DAG scheduling on heterogeneous systems. *Cluster Computing* 13, 4 (1 Dec 2010), 373–383. https://doi.org/10.1007/s10586-009-0119-6

[9] BSC. 2021. BSC application repository. (2021). https://pm.bsc.es/projects/bar.

[10] Hongsuk Chung, Munsik Kang, and Hyun-Duk Cho. 2013. Heterogeneous multi-processing solution of Exynos 5 Octa with ARM big.LITTLE TM technology. (2013). https://www.arm.com/files/pdf/Heterogeneous_Multi_Processing_Solution_of_Exynos_5_Octa_with_ARM_bigLITTLE_Technology.pdf.

[11] Stephen Dawson-Haggerty, Andrew Krioukov, and David E. Culler. 2009. *Power Optimization- A Reality Check*. Technical Report UCB/EECS-2009-140. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-140.html.

[12] Daniele De Sensi, Massimo Torquati, and Marco Danelutto. 2016. A reconfiguration algorithm for power-aware parallel applications. *ACM Trans. Archit. Code Optim.* 13, 4, Article 43 (Dec. 2016), 25 pages. https://doi.org/10.1145/3004054
[13] Christina Delimitrou and Christos Kozyrakis. 2013. QoS-aware scheduling in heterogeneous datacenters with paragon. *ACM Trans. Comput. Syst.* 31, 4, Article 12 (Dec. 2013), 34 pages. https://doi.org/10.1145/2556583
[14] B. Donyanavard, T. Mück, S. Sarma, and N. Dutt. 2016. SPARTA: Runtime task allocation for energy efficient heterogeneous manycores. In *2016 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. 1–10.
[15] Alejandro Duran, Roger Ferrer, Eduard Ayguadé, Rosa M. Badia, and Jesus Labarta. 2009. A proposal to extend the OpenMP tasking model with dependent tasks. *Int. J. Parallel Program.* 37, 3 (June 2009), 292–305. https://doi.org/10.1007/s10766-009-0101-1
[16] Stijn Eyerman and Lieven Eeckhout. 2011. Fine-grained DVFS using on-chip regulators. *ACM Trans. Archit. Code Optim.* 8, 1, Article 1 (Feb. 2011), 24 pages. https://doi.org/10.1145/1952998.1952999
[17] Priyanka Ghosh, Yonghong Yan, Deepak Eachempati, and Barbara Chapman. 2013. A prototype implementation of OpenMP task dependency support. In *OpenMP in the Era of Low Power Devices and Accelerators*, Alistair P. Rendell, Barbara M. Chapman, and Matthias S. Müller (Eds.). Springer Berlin, Berlin, 128–140.
[18] C. J. Hughes, J. Srinivasan, and S. V. Adve. 2001. Saving energy with architectural and frequency adaptations for multimedia applications. In *34th ACM/IEEE International Symposium on Microarchitecture (MICRO-34)*. 250–261. https://doi.org/10.1109/MICRO.2001.991123
[19] Ravindra Jejurikar and Rajesh Gupta. 2005. Dynamic slack reclamation with procrastination scheduling in real-time embedded systems. In *42nd Annual Design Automation Conference (DAC -05)*. Association for Computing Machinery, New York, 111–116. https://doi.org/10.1145/1065579.1065612
[20] Jaeyeon Kang and Sanjay Ranka. 2010. Dynamic slack allocation algorithms for energy minimization on parallel machines. *J. Parallel Distrib. Comput.* 70, 5 (May 2010), 417–430. https://doi.org/10.1016/j.jpdc.2010.02.005
[21] S. Kaxiras and M. Martonosi. 2008. Computer Architecture Techniques for Power-Efficiency. https://doi.org/10.2200/S00119ED1V01Y200805CAC004
[22] Neetesh Kumar and Deo Prakash Vidyarthi. 2019. A green SLA constrained scheduling algorithm for parallel/scientific applications in heterogeneous cluster systems. *Sustainable Computing: Informatics and Systems* 22 (2019), 107–119. https://doi.org/10.1016/j.suscom.2019.02.001
[23] Young Choon Lee and Albert Y. Zomaya. 2009. Minimizing energy consumption for precedence-constrained applications using dynamic voltage scaling. In *2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID'09)*. IEEE Computer Society, 92–99. https://doi.org/10.1109/CCGRID.2009.16
[24] Y. Li, M. Chen, W. Dai, and M. Qiu. 2017. Energy optimization with dynamic task scheduling mobile cloud computing. *IEEE Systems Journal* 11, 1 (2017), 96–105.
[25] I. Lin, B. Jeff, and I. Rickard. 2016. ARM platform for performance and power efficiency - Hardware and software perspectives. In *2016 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*. 1–5. https://doi.org/10.1109/VLSI-DAT.2016.7482541
[26] S. Park, J. Park, D. Shin, Y. Wang, Q. Xie, M. Pedram, and N. Chang. 2013. Accurate modeling of the delay and energy overhead of dynamic voltage and frequency scaling in modern microprocessors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32, 5 (May 2013), 695–708. https://doi.org/10.1109/TCAD.2012.2235126
[27] V. Petrucci, M. A. Laurenzano, J. Doherty, Y. Zhang, D. Mossé, J. Mars, and L. Tang. 2015. Octopus-Man: QoS-driven task management for heterogeneous multicores in warehouse-scale computers. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 246–258. https://doi.org/10.1109/HPCA.2015.7056037
[28] A. Raghavan, L. Emurian, L. Shao, M. Papaefthymiou, K. P. Pipe, T. F. Wenisch, and M. M. K. Martin. 2013. Utilizing dark silicon to save energy with computational sprinting. *IEEE Micro* 33, 5 (Sept. 2013), 20–28. https://doi.org/10.1109/MM.2013.76
[29] V. Spiliopoulos, S. Kaxiras, and G. Keramidas. 2011. Green governors: A framework for continuously adaptive DVFS. In *2011 International Green Computing Conference and Workshops*. 1–8. https://doi.org/10.1109/IGCC.2011.6008552
[30] B. Su, J. Gu, L. Shen, W. Huang, J. L. Greathouse, and Z. Wang. 2014. PPEP: Online performance, power, and energy prediction framework and DVFS space exploration. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 445–457. https://doi.org/10.1109/MICRO.2014.17
[31] Jinho Suh, Chieh-Ting Huang, and Michel Dubois. 2015. Dynamic MIPS rate stabilization for complex processors. *ACM Trans. Archit. Code Optim.* 12, 1, Article Article 4 (April 2015), 25 pages. https://doi.org/10.1145/2714575
[32] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. 2002. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems* 13, 3 (2002), 260–274.
[33] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer. 2012. Scheduling heterogeneous multi-cores through performance impact estimation (PIE). In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*. 213–224. https://doi.org/10.1109/ISCA.2012.6237019

[34] T. Xie and X. Qin. 2008. An energy-delay tunable task allocation strategy for collaborative applications in networked embedded systems. *IEEE Trans. Comput.* 57, 3 (March 2008), 329–343. https://doi.org/10.1109/TC.2007.70809

[35] Sergey Zhuravlev, Juan Carlos Saez, Sergey Blagodurov, Alexandra Fedorova, and Manuel Prieto. 2012. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Comput. Surv.* 45, 1, Article 4 (Dec. 2012), 28 pages. https://doi.org/10.1145/2379776.2379780

[36] Sergey Zhuravlev, Juan Carlos Saez, Sergey Blagodurov, Alexandra Fedorova, and Manuel Prieto. 2013. Survey of energy-cognizant scheduling techniques. *IEEE Trans. Parallel Distrib. Syst.* 24, 7 (July 2013), 1447–1464. https://doi.org/10.1109/TPDS.2012.20

Received April 2021; revised August 2021; accepted October 2021