Boosting Energy-Efficiency of Heterogeneous Embedded Systems via Game Theory

David Manuel Carvalho Pereira
Email: david.manuel.carvalho.pereira@ist.utl.pt

***Abstract*—The ever-growing demand of mobile devices (e.g smartphones, tablets) with better performance and efficiency led mobile embedded systems to become heterogeneous devices with higher computational power and energy efficiency levels. However, it seems that these energy limited devices, are consuming more energy that it is required in order to meet the performance requirements, leading to discharge the battery more rapidly.**

This paper aims to study and develop a new dynamic energy-aware task scheduling approach for heterogeneous embedded systems, based on Game Theory, in order to reduce the overall energy consumption of the device. The proposed scheduling approach combines an Auction based selection and the Nash Equilibrium concept from Non-Cooperative Game Theory. It develops a game, where players (processors cores) compete with each other in order to acquire the tasks/applications by biding the necessary energy consumption to execute them. The dynamic energy-aware game-theoretic scheduling framework herein proposed has been implemented on ARM Versatile Juno r2 Development Platform, experimentally evaluated and compared with the available ARM big.LITTLE scheduling approaches. The conducted evaluation revealed that the proposed framework can achieve energy savings of up to 36%, 32% and 22% when compared with the Linaro's kernel 3.10, Global Task Scheduling and Energy-Aware Scheduling ARM big.LITTLE approaches, respectively.

***Index Terms*—Game Theory; Global Task Scheduling; ARM big.LITTLE; Mobile Devices; Heterogeneous Embedded Systems; Energy-Efficient;**

I. Introduction

The paradigm of achieving more computational power has led to an evolution in the current computing systems. Heterogeneous embedded systems such as mobile devices (e.g. smartphones, tablets, etc.) are evolving every year aiming to achieve better performance, speed, power consumption and others aspects which can lead to better utilization conditions for the user. However, they are requiring higher power consumption, which leads to the batteries start to unload faster and faster. To solve this problem, it must be found the best way to achieve a good performance level, while at the same time the energy consumption is reduced to the minimum. In order to do so, an energy-aware scheduler must be developed to schedule the tasks to the several cores of a processor in an optimal way.

Several works, for example [2] and [7], have proposed a static scheduling approach based on game theory that generate task-to-machine scheduling maps with the respective executions frequencies for each task. However, these works are focused on solving the energy consumption problem for distributed heterogeneous grids and not for heterogeneous embedded systems. The static scheduling approaches proposed by the authors, also takes into consideration tasks with deadline constraints, which is not common on user's applications. In [3], the authors propose a game-theoretic energy-aware dynamic scheduling approach for multi-core systems to dynamically migrate the applications between cores in order to reduce its temperature and avoid "hot-spots" in the processor's chip. Although considered an energy-aware scheduling approach, it is mainly focused to reduce temperature and not energy consumption at all. The approach proposed by [6] uses a game theoretic concept to adjust at run-time the frequency of each processing unit on a Multi Processing - System on a Chip platform. This approach aims to reduce the system's temperature while maintaining the synchronization between tasks of an application graph.

The works [2], [7] and [3] uses an Auction approach combined with Game Theory concepts such as the Nash Bargaining Solution, Nash Equilibrium and the Prisoners Dilemma, to schedule the tasks to cores/machines. These game theoretic concepts are used to help the players decide the best strategy to select in order to achieve the proposed goals. On the other hand, [6] proposed an iterative algorithm based on Nash Equilibrium to schedule the tasks.

This paper aims to study how game theory can be used to create a new dynamic energy-aware scheduling approach capable of exploring all available computational resources simultaneously and reduce the overall energy consumption in heterogeneous embedded systems. The herein proposed energy-aware game-theoretic scheduling approach will be based on an auction method and the Nash Equilibrium game-theoretic concept, where the processor's cores will compete with each other in order to bid and acquire the tasks/applications to execute. Based on the game-theoretic concept, the cores will select the best frequency to execute the tasks in order to reduce the device's overall energy consumption.

This work is organized as follows. Section II introduces the Nash Equilibrium concept from non-cooperative Game Theory and the Dynamic Voltage and Frequency Scaling power-management technique. Afterwards, Section III presents the proposed energy-aware game-theoretic scheduling approach. In Section IV the proposed scheduling algorithms are evaluated and compared with ARM big.LITTLE scheduling approaches. Finally, the conclusions are presented in Section V.

II. Background

A. Game Theory

Game theory is "the study of mathematical models of conflict and cooperation between intelligent rational decision-makers" [1], and is used in different areas such as economics,

computer science or biology. In game theory there are two main branches: the cooperative and non-cooperative game theory. The objective of this section is to introduce the fundamental concept of non-cooperative game theory, the Nash Equilibrium (NE).

Non-cooperative game theory deals with how individuals interact with one another to achieve their own goals. The players make decisions only by themselves seeking always for the best payoff outcome for themselves. Nash proved that each non-cooperative n-player game has at least one equilibrium point, known as Nash Equilibrium.

Considering a n-player strategic game, let $u_i(s_1, ..., s_N)$ denote the payoff utility of Player i (p_i) that is based on its strategy s_i and the strategy chosen by the other players, s_{-i}. Each p_i has a set of strategies and must choose one of them, $s_i \epsilon \{S_0, S_1, S_2, ..., S_j\}$. Given the others players strategies, $(s_1, ..., s_{(i-1)}, s_{(i+1)}, ..., s_N)$, p_i will choose always the best strategy, s_i^*, which gives the best payoff for him, $u_i(s_1, ..., s_{(i-1)}, s_i^*, s_{(i+1)}, ..., s_N) \geq u_i(s_1, ..., s_{(i-1)}, s_i, s_{(i+1)}, ..., s_N)$.

The strategy set $(s_1^*, ..., s_i^*, ..., s_N^*)$ is a Nash Equilibrium when all players have chosen the best strategy for themselves based on the others players strategies and have no incentive to change their strategy given what the other players are doing. So it means that the best individual payoff to all players was found.

B. Dynamic Voltage and Frequency Scaling (DVFS)

DVFS is a commonly-used power-management technique used to reduce the energy consumption on CPUs. The dynamic CPU power dissipation, P, can be mathematically represented as $P = C * Vdd^2 * f$ [7], where C is the switched capacitance, Vdd is the supply voltage and f is the operating frequency. The supply voltage can be reduced by decreasing the CPU's frequency, which leads to reduce the power consumption of the processor.

In this work, the ARM Juno r2 platform was used to evaluate the proposed scheduling approach. This platform has a dual-core Cortex-A72 (big Cluster) and quad-core Cortex-A53 (LITTLE Cluster), which supports only three different DVFS Operating Performance Points (OPP) as shown in Table I. The ARM Juno r2 platform only supports the DVFS technique at the level of the Clusters.

TABLE I: Cortex-A72 and Cortex-A53 DVFS OPPs [4].

OPP	Voltage [V]	Frequency [MHz]	
		Cortex-A72	Cortex-A53
Underdrive	0.8	600	450
Nominal drive	0.9	1000	800
Overdrive	1.0	1200	950

III. Energy-Aware Game-Theoretic Scheduling Approach For Heterogeneous Embedded Systems

A. Problem Definition

Based on game theory, to solve the scheduling problem, an auction system can be developed as a bargaining game. As already seen, in section I, auctions were used by other researchers and are considered a simple way to assign tasks to cores. In an auction, the auctioneer (scheduler) present tasks to the players (cores) in each round. The player should select the best strategy (frequency), which is decided based on the player's utility function, and must bid on the task in order to acquire it. In the end of the auction, the task is assigned to the winner core. An auction n-player game, based on the Nash Equilibrium concept from game theory, can be formally defined as:

- N Players, $P = \{p_1, p_2, ..., p_N\}$;
- Each player owns a set of strategies. $S = \{strategy_1, ..., strategy_N\}$;
- In a round, each player p_i has a strategy s_i. The set of strategies in a round is $s = \{s_1, s_2, ..., s_N\}$;
- Each player has a payoff function $u_i(\cdot)$ that can be based on other cores strategies and can be the same (or not) to every player.
- The game can be composed by multiple rounds, R rounds.

The most important parts of this game are how the players may choose their strategy and what must be the bid value. The scheduling approach herein proposed must focus on developing an utility function capable to select the best frequency for the player and also, assure that all players compete with each other to bid and acquire the tasks. In the following sections will be explained in detail the proposed auction approach as well as the non-cooperative game-theoretic game approach.

B. Auction Approach

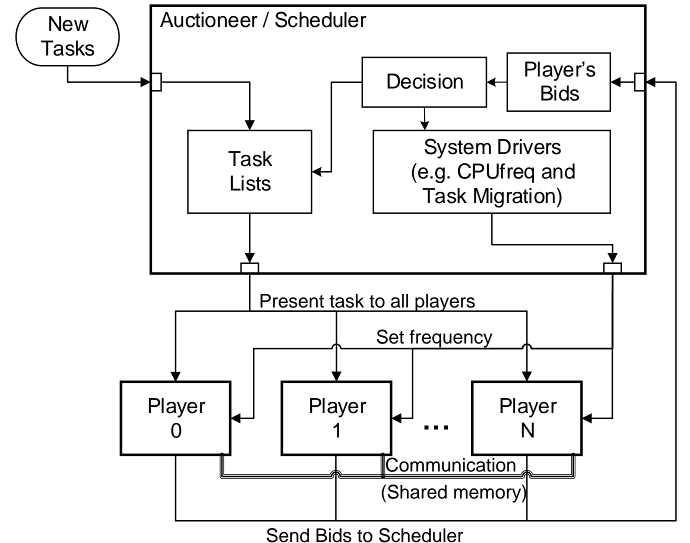


Fig. 1: General structure of the proposed framework.

The auction approach assures that players compete with each other by bidding the tasks in order to acquire them. In each round of the game, one auction is realized, which means that one task is scheduled per round. The auction of each task gives to each player an opportunity to participate and be able to execute the task.

In each auction, the auctioneer (scheduler) present the tasks to the players (cores), which will execute the task for a short amount of time in order to acquire the necessary information about the task in order to select the best strategy and bid the task. The player should also have access to the others players strategies in order to decide its own best strategy using the game theoretic approach. Then, when all players have computed and sent their bids to the scheduler, the winner core will be selected, which will receive and execute the task.

In Figure 1 is shown the multi-player auction scheduling environment herein proposed.

C. Non-Cooperative Game-Theoretic Approach

The Nash Equilibrium concept will be used in this game theoretic approach. As seen in section II-A and in [2], [3] and [6], the utility function is the principal mechanism used by the players to select their own strategy. Using the auction approach together with Nash Equilibrium, it is expected that the players place a bid, for the task to execute, which is based on the decision that maximizes the player's utility function, and also, on the principal goal, to minimize the overall energy consumption of the system. Having said that, it is expected that the player utility function is directly related with the overall energy consumption of the device, and so, in each auction, the winner will be the player who bids the lowest value, which means that it can execute the task with the lowest energy consumption among all players. By scheduling one task in each auction to the core that can execute it with the lowest possible variation in the overall system's energy consumption, this scheduling approach is able to find a local sub-optimal solution.

The scheduling approach herein proposed must consider the overall energy consumption of a system and not just on the processors because this work is focused on energy limited mobile devices, such as smartphones, and thus, the main goal here is to save their battery as much as possible. On a compute system exists mainly two subsystems that consume energy, the processing units (e.g cores of a CPU and GPU) and the remaining units (e.g. memory, system drivers and peripherals). Therefore, the overall player utility function (Equation 1) must be divided in 2 parts: the selected core (u_{player}) utility function, which is divided in two sections to cover the increase of energy consumption in the selected player and its impact on other players ($u_{individual}$), as well as the impact on the remaining units (u_{global}); and the other cores ($u_{other_players}$) utility function, which aims to find if it is possible to change their strategy in order to improve energy savings, taking into account the new strategy of the selected core. These functions are shown in equations 2-5.

In these equations, S_{ij} is the strategy set of player i, and j is one of the M_i available strategies, $S_{ij} = \{s_{i1}, s_{i2}, ..., s_{iM_i}\}$, being s_{i1}, the lowest frequency available. In this game approach exists N players, $i \in \{1, 2, ..., N\}$, and $s_{i'}$ represents the best strategy selected by player i in the last round/auction; s_{i^*} represents the best strategy of player i, that can be found by using the selected player utility function (Equation 2).

$$u_i = u_{player}(s_{1'}, s_{2'}, ..., S_{ij}, ..., s_{N'}) + u_{other_players}(S_{1j}, ..., S_{(i-1)j}, S_{(i+1)j}, ..., S_{Nj}) \quad (1)$$

$$u_{player} = \min_j[(u_{individual}(s_{ij}) + u_{global}(s_{1'}, s_{2'}, ..., s_{ij}, ..., s_{N'})] \quad (2)$$

$$u_{individual} = \Delta E_{player}(s_{ij}, s_{i'}) + \sum_{w=1, w\neq i}^{N} \Delta E_{player}(s_{w'}) \quad (3)$$

$$u_{global} = \Delta E_{system}(s_{1'}, s_{2'}, ..., s_{ij}, ..., s_{N'}) \quad (4)$$

$$u_{other_players} = \sum_{w, w\neq i}^{N} \min_j(\Delta E_{other_player}(s_{i^*}, s_{wj}, s_{w'})) \quad (5)$$

1) Individual Utility Function: The player's utility function, shown in equation 2, is focused on unilaterally minimize the variation of energy consumption of the system due to the selected frequency by player i. To compute the necessary energy consumption to execute a task on a core, is necessary to estimate the task execution time and its associated power consumption. In this work, it is assumed that has been collected some information about the task a priori, such as the total number of instructions of the task, which is essential to estimate the task execution time. Each player can use the performance counters and energy meter registers available on the platform to characterize the task and measure its power consumption, which will be further explained in this document, in order to estimate the necessary energy consumption to execute the task.

In systems with many tasks executing at the same time, to give energy-awareness to the scheduler, it is necessary to know in which cores are the tasks being executed, as well as, their actual frequencies and power consumptions. In this proposed approach, all this information is put together and stored in memory. To simplify the reading, this structure will be referenced as "task execution map". In Figure 2 is shown an example of a task execution map in order to describe how the selected player chooses the best frequency to execute the received task.

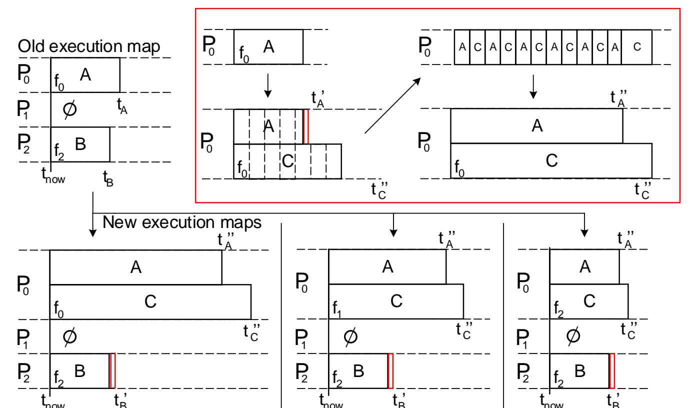


Fig. 2: Individual utility function, usage example.

In this example, player (core) 0, P_0, and player 2, P_2, are already executing task A and B, which will finish approximately at t_A and t_B, respectively. P_1 is empty, and thus,

in idle mode. However, when task C is scheduled to any other core, the execution time of existing tasks A and B, will increase to $t_{A'}$ and $t_{B'}$, respectively, due to conflicts between the tasks inside the shared caches and memory (e.g. cache-miss penalty). Besides that, if task C is scheduled to the same core that is executing task A, then the execution time of task A will increase from $t_{A'}$ to $t_{A''}$ due to the round-robin time-sharing policy, SCHED_OTHER, because just one task will be executed at a time while the remaining ones stay paused, as shown in the red square present on Figure 2. This example shows that an energy-aware approach must be aware of the availability of each core before scheduling the task, because when task C is scheduled to the same core that is executing task A, the amount of context switching between tasks is quite intensive due to the SCHED_OTHER policy, and thus, it will probably be less energy-efficient than if task C was scheduled to an non occupied core, which in this case is P_1.

Following the example, to properly compute the necessary energy consumption to execute task C on core 0, it must be first introduced the following $\Delta Energy$ and the $Energy_{map}$ functions:

$$\begin{aligned} \Delta Energy &= E_{new_map} - E_{old_map} \\ E_{map} &= \sum_{i=1}^{K} Pow_{Task_comb(i)} \times \Delta T_{Task_comb(i)} \end{aligned} \tag{6}$$

In Equation 6, K represents the number of task combinations in the execution map of the selected player. As seen in Figure 2, K is 1 when core 0 is just executing task A (old execution map), and is 2 when core 0 is executing task A and C (new execution map) because there are two time slices with different combinations, first, the time slice from t_{now} to $t_{A''}$ with the task combination A and C and then from $t_{A''}$ to $t_{C''}$ with just task C.

Following the example, in the new execution map, the energy consumption of core 0 is $E_{new_map} = Pow_{AC} * (t_{A''} - t_{now}) + Pow_C * (t_{C''} - t_{A''})$, while previous, the energy consumption of the old execution map was just the energy consumption of task A solo, $E_{old_map} = Pow_A * (t_A - t_{now})$. In this utility function, it is also taking into account the increase of energy consumption in the other players due to the impact of core 0 decisions (increase of task B execution time), and so, these energy consumption variations must be also added in the individual utility function, as seen in the second term of Equation 3. It should be noted that each player must have the ability to access and read its own instantaneous power consumption value in order to compute the $\Delta Energy$.

In these calculations, it can be seen that for each task execution map it must be known several task execution times and powers consumption values. In this example, $Power_{AC}$ can be measured immediately after task C is scheduled to core 0, while $Power_A$ was already been measured when task A was scheduled in the previous round. However, $Power_C$ is not known but can be assumed that its value is already known from previous executions of task C and is stored on memory (task history unit) or it can be measured by pausing task A in order to measure the instantaneous power consumption of task C solo. The executions times can be approximately estimated through the cycles per instruction (CPI) ratio, frequency and total number of instructions as will be explained in section III-E.

There are as many execution maps as frequency levels available on the core. The new frequency (new map) that assures the lowest variation of energy consumption compared to the old frequency (old map), will be the individual decision of the player. These new execution maps should be stored on the memory in order to be used in the next auctions and also because they are used in the global utility function.

2) Global Utility Function: The energy consumption of a mobile device is not only due to the processor, as previously discussed, but also other components that consume energy. For simplicity, this set of components will be referenced as ”system”, which refers to the existing hardware outside the processors (clusters), such as the DRAM memory, buses, power-management subsystems (e.g. DVFS) and other peripherals. On some devices the system energy consumption can be even higher than the processor, and so, it must be considered to the scheduling decisions.

In ARM Juno r2 platform and generally in mobile devices, the variation of power consumption in the ”system” is mainly due to the DRAM memory accesses because it is one of the components whose usage is more dependent on the executing tasks. However, this variation can be slightly insignificant when compared to the static power consumption that the remaining components in the ”system” have.

The global utility function operates similarly to the individual utility function. The difference is that all the players must now be seen as an unique player. This modification is required because the overall instantaneous power consumption of the ”system” is only represented by one power consumption sensor. And so, the task combinations must be aggregated at the level of the whole system and not at the level of the player. In Figure 3 is shown an example to better understand the global utility function. As mentioned before, for each player, there exists as many execution maps as available frequencies. However, to simplify, in this example, it is just shown two new task execution maps when the received task C is scheduled to core 1 and core 2 at frequency 0 and 2, respectively. The $\Delta Energy$ and the E_{map} functions used in this utility function are the same that are used in the individual utility function, Equation 6. However, in this utility function, the power considered will be the ”system” instantaneous power consumption and not the power consumption of each individual player.

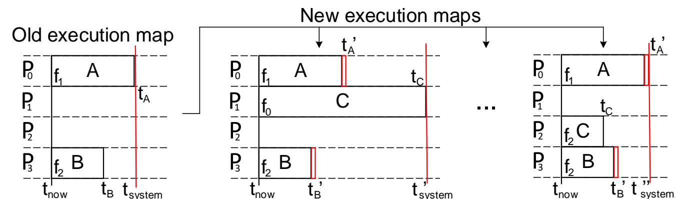


Fig. 3: Global utility function, usage example.

Following the example, the ”system” energy consumption of the new execution map when the task C is scheduled to core 1 with the lowest frequency is $E_{new_map} = Pow_{ABC} *$

$(t_{B'} - t_{now}) + Pow_{AC} * (t_{A'} - t_{B'}) + Pow_C * (t_C - t_{A'})$, while the energy consumption of old execution map was, $E_{old_map} = Pow_{AB} * (t_B - t_{now}) + Pow_A * (t_A - t_B)$. As already mentioned in this document, some of these instantaneous power consumptions are unknown and must be measured. In order to overcome this problem in future auctions, these measure must be stored in the task history unit. However, as already discussed, the "system" power consumption can be approximately constant, and this would let the Pow_{AC} be approximately equal to Pow_{ABC} and not needed to be measured. However, this would be a pessimistic approach and the respective associated errors could influence the overall decision. It should be also noted that if task C was scheduled to core 2 at frequency 2, the only unknown task combination power consumption value is the new ABC, because AB and A were already been measured on previous auctions.

As seen in Figure 2 and 3, different frequencies lead to different execution maps, and hence, different energy consumptions. The relevant information of each execution map should be stored on the memory in order to be used in the next auctions. In this example (Figure 3), it should be noted that depending on the frequency and core selected to schedule the task, the t_{system} varies, and so, in some execution maps, the energy consumption of the "system" can be more relevant than in other execution maps.

Joining the individual utility function with the global (system) utility function it is possible to predict the increase of energy consumption inside the processor and in the "system" when scheduling a task to a specific core. For each frequency available on the core, the one which implies lower variation of the overall (core + system) energy consumption will be selected as the best decision for that player. However, it must be also seen if the remaining players should change (or not) its previous decisions based on the actual selected player decision, which is taken into account in the other players utility function.

3) Other Players Utility Function: As seen in the last two utility functions, the selected core choses the best frequency based on the variation of energy consumption on the processor as well as on the "system". This is done because the scheduler should have an overall device's energy-awareness and not only at the level of the processor. However, the approximately constant power consumption of the "system" can be so higher when compared with the processor/core, that this last can be irrelevant, which can lead to select higher frequencies on the core in order to reduce the task execution time and hence reduce the dominant "system" energy consumption. Having said that, once we have taken into consideration the overall energy consumption of the device, we can then look if it is possible to save energy consumption in the remaining cores by selecting their best frequencies from the individual utility function, as illustrated in Figure 4.

As already seen in the selected core utility function (u_{player}), the best frequency for that player is chosen and the corresponding execution map is stored on memory. This new execution map will be the base for this utility function. The highest execution time of all players is designated the time system, t_{system}, which is marked red on the example. The goal of this approach is to see if the other players can lower

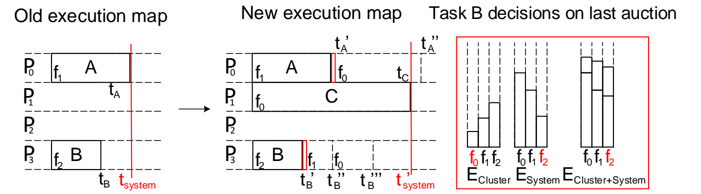


Fig. 4: Other players utility function, usage example.

their frequency in order to achieve energy savings while they are just allowed to select frequencies whose execution time is lower than the t_{system}.

Following the example, task B was scheduled in the last auction to core 3 with the highest frequency, which was not the one that corresponds to the best individual energy saving as can be seen in the energy values represented in the red square. It should be noted that when each task is scheduled to a core, its individually values of necessary energy consumption to execute on that core for all available frequencies are stored on memory in order to be used in this utility function. For task B, the best individually energy consumption would be achieved when selecting the lowest frequency available. However, f_0 was not selected in the last auction because the "system" has higher energy consumption at that frequency, and thus, the selected frequency was f_2, which provides lower overall energy consumption in the device. By using the selected core utility function (u_{player}), the best frequency selected to schedule task C to core 1 was the lowest one, which corresponds to a change from t_{system} to $t^{'}_{system}$. Looking to each execution time for each frequency, one can see that the core 3 can achieve energy savings by selecting the frequency 0, which corresponds to an execution time $t^{'''}_{B}$. However, for task A, frequency 0 will not be selected because the task execution time will be higher than the $t^{'}_{system}$.

D. The Proposed Algorithm

By using these three utility function, one can conclude that, in each auction, the new task can be scheduled to the core that offers the lowest variation of energy consumption on the device. With this game approach, each player will place a bid regarding its own decisions and the other players decisions, which will lead to find local sub-optimal energy savings. In Figure 5 is shown the pseudocode of the player's algorithm based on the proposed game theoretic approach.

As can be seen in the player's algorithm, the player starts to gather information about the new task (Steps 0-2). Afterwards, the player uses this information to estimate the tasks execution times and instantaneous power consumptions (Steps 3-4). These estimations are based just on the current operating frequency, however it is possible to predict them to other frequencies without changing it, which will be explained in sections III-E and III-F. Then, the player selects the best frequency that contributes with the lowest variation of energy consumption to the overall device, and communicate it to the other players. These will see if it is possible to improve their strategies, and communicate them back to the selected player (Step 5-6). Finally, based on all these energy consumption

Player's pseudocode to compute the task's bid

Input: Task t_i
Output: Bid of player j to task i, B_{ij}
0. Read the performance counters and energy meter registers for all existing tasks on player j before executing task t_i.
1. Execute task t_i for a short amount of time.
2. Read the performance counters and energy meter registers again for all existing tasks on player j.
3. For each task executing in player j do:
   3.1 Estimate the task execution time for the actual frequency.
   3.2 Predict the task execution time for the remaining frequency levels.
4. For each available frequency do:
   4.1 Generate the task execution map based on the tasks execution times.
   4.2 Estimate the instantaneous power consumption for each task combination existing in the execution map.
   4.3 Estimate the player and "system" overall energy consumption based on the created execution map.
   4.4 Compute the variation of energy consumption between the new created task execution map and the previous task execution map (without task t_i).
5. Choose the frequency that leads to lower variation of energy consumption when executing task t_i on player j.
6. Optimize the other players decisions through communication of the selected frequency.
7. Compute bid B_{ij} based on all utility functions and send it to the scheduler.

Fig. 5: Player's pseudocode to compute the task's bid.

values, the selected player computes the final bid and sends it to the scheduler (Step 7). Once all players have sent their bids to the scheduler, the player with the lowest bid will be the winner, the task will be schedule to it and the respective operating frequency will be changed.

The players in this approach will be the "big" and "LITTLE" Clusters and not the individual cores because the ARM Juno r2 platform have energy meters only at the level of the Cluster, which means that it can be just known the sum of instantaneous power consumption of all cores on that cluster, and not each one individually. This limitation leads to adopt the notion of "players' representative", which are the clusters and the "sub-players", which are the cores. Basically, the idea is that the player represents a coalition of sub-players, where the player must adopt an non-cooperative game approach in relation with the other clusters, but at the same time, it must exists some cooperation between the cores inside that cluster, because when the frequency is changed in the cluster, all the core's frequency are also changed. That said, in this approach, is now assured cooperation between the sub-players by selecting the new best frequency for all cores that corresponds to the lowest energy consumption on the cluster, and then, is also assured a non-cooperative approach between the players, by using the auction approach in order to compete individually against each other.

To evaluate this approach, real benchmark applications were used. These tasks can have different execution phases until they finish. First, they can start to behave like a memory bounded task, which means that they are mainly dependent on the memory's frequency and not on the processor's frequency, because they do many memory accesses and must wait many cycles to obtain the stored data. And then, they can behave like compute bound tasks, where they just use the caches of the processor, which are much faster than the general DRAM memory. This unpredictable behavior, i.e. different phases (different CPI ratios), can affect the estimation of the task execution time, as it will be explained in section III-E. To overcome this problem, the scheduler must do the estimations frequently in order to detect the task phases. In this approach, when a new task needs to be scheduled, it is done a new estimation of all task's execution times in order to compute the energy consumption accurately. It is also done a reschedule of the already executing tasks when one of the tasks has finished. This contributes to actualize the estimations of execution times as well as to give other opportunity for other cores to acquire those tasks. The pseudocode for the proposed scheduler is shown in Figure 6. The detection of when some task have finished is done through a flag, corresponding to Step 6 of the algorithm.

In the developed algorithm three task lists are maintained: the task_running list, the task_waiting list and the task_paused list. In the task_running list are present the tasks that were scheduled and are being executed on the cores. In the task_waiting list are present the new tasks that arrived to the scheduler and were not yet scheduled. The scheduler starts by picking one task from the task_waiting list and uses the proposed scheduling approach to schedule it. First, the scheduler opts to schedule only to the cores that are empty in order to avoid exhaustive search on all existing cores (Steps 2.3-2.5). However, if there is no available cores, the scheduler has no choice but finding the best of all existing cores (Steps 4). There is also the option to not schedule the task and wait until some task finishes, which could lead to lower energy consumption than scheduling the task to an occupied core (Step 3). If eventually this could be the decision, then the task will be transfered from the task waiting_list to the paused_list and will stay there until some core becomes available (Step 5). As already mentioned, once one task finishes, it is done a reschedule of all executing tasks. To do so, the tasks in the running_list are inserted in the top of the waiting_list followed by the tasks in the paused_list and the remaining tasks already present in the waiting_list. And then, the scheduler executes the tasks reschedule (Step 6.).

E. Execution time estimation and prediction

In order to estimate the task execution time, it will be necessary to know the task actual cycles per instruction (CPI)

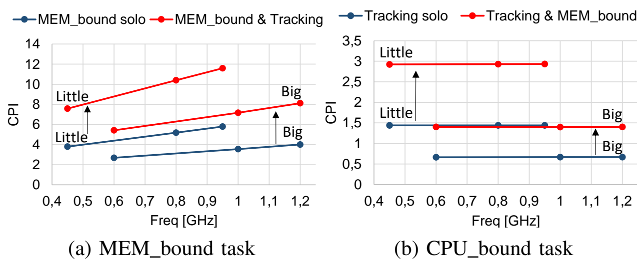
Pseudocode of the Scheduler's algorithm.

Input: Task t_i

Output: Schedule of task t_i to core c_j with freq f_w

0. Scheduler waits until new tasks appear.
1. Scheduler enqueues the new tasks in the waiting list.
2. If there are tasks on waiting list, proceed, otherwise go to Step 0.
   - 2.1. Check all players available, i.e., the ones with at least one core unoccupied.
   - 2.2. If there are no available players then go to Step 3.
   - 2.3. Send the task for each available player and wait to receive all player's bids.
   - 2.4. Select the player with the lowest bid and schedule the task t_i to it.
   - 2.5. Change the players' frequencies according to the bid of the winning player. Go to Step 2.
3. Send the task to the player who will be available sooner and compute the bid as scheduling the task just when the player becomes available, i.e., to not scheduling the task now and wait until some player is available.
4. Send the task and compute the bid for each core in each player.
5. If it is better to not schedule the task now, then send the task to the pause_list and wait until some player becomes available. Otherwise, schedule the task to the winner core. Proceed to Step 2.
6. If some task finishes, insert the already executing tasks on the top of the waiting_list followed by the tasks in the paused_list and proceed to Step 2 (reschedule).

Fig. 6: Pseudocode of the Scheduler's algorithm.



(a) MEM_bound task (b) CPU_bound task

Fig. 7: Comparison and variation of the CPI tendency between MEM_bound and CPU_bound applications.

ratio, frequency and number of total instructions, as shown in Equation 7. The task CPI ratio can be measured by reading the number of cycles and instructions executed, during a time duration, which can be obtained through PAPI [5] or by directly setting and reading the event counter registers. In this work, the task's total number of instructions is assumed that is already known and is stored in the task history unit.

$$Execution_time = \frac{CPI}{freq} \times \#Total_instructions \tag{7}$$

The use of the actual CPI ratio on this function, assumes that the task have always the same behavior when it is being executed. However, as mentioned before, tasks can have different execution phases, and thus, CPI measures must be actualized when a new task is about to be scheduled or when the scheduler effectuates a reschedule when one task finishes.

Generally, memory bounded tasks have higher CPI ratio than compute bounded tasks, because in average it must wait more cycles to load or store data then to compute. Based on two tasks tested on ARM Juno r2 platform, one developed memory bound task and one compute bound benchmark (Tracking), it could be possible to see the CPI variation, of each task, for different frequencies when they are executed solo and together in the same core (Figure 7).

This example shows that the compute bounded task has the same CPI in each frequency, which is true because CPU bounded tasks are mainly dependent on the frequency of the core. And it is also seen, that memory bounded tasks have different CPI values on different frequencies; they are more dependent on the memory's frequency than the frequency of the core, and so, they wait more cycles to get the data from the memory, which is operating at a constant and different frequency than the cores. It can also be seen that when the tasks are executing together in the same core, the CPI tendency tends to be the same, which suggests that this tendency must be preserved when it is necessary to predict the CPI ratio to other frequencies. To do so, it will be assumed that the relation between the new CPI measured and the CPI solo value is the same in each different frequency. And so, based on the CPI measure for the actual frequency and the CPI solo values stored for the same frequency and other frequency to predict, it can be possible to predict the CPI value for that frequency, as shown in Equation 8.

$$CPI_{f_x} = \frac{CPI_{f_y}}{CPI_{solo_{f_y}}} \times CPI_{solo_{f_x}} \tag{8}$$

In the task history unit, it is assumed to be stored the solo CPI value of each task for each available frequency, which were obtained by executing the task solo previously. By predicting the task CPI value for the remaining available frequencies, it is possible to estimate the task execution times for each frequency level without physically changing it (exhaustive search).

F. Power consumption estimation and prediction

The cumulative energy consumption meter register present in the ARM Juno r2 Performance Measure Unit will be used to measure the instantaneous power consumption of a task combination executing on a Cluster at the actual frequency. This register can be accessed, when using an Linaro kernel, through the Hardware Monitoring Driver interface, which is exposed through sysfs (/sys system folder). The instantaneous power consumption estimation can be obtained by dividing

the difference between two cumulative energy measures by the time duration between measures, as shown in Equation 9.

$$Power_consumption = \frac{Energy_{_after} - Energy_{_before}}{\Delta Time} \quad (9)$$

Power dissipation in CMOS circuits can be defined by the processing (dynamic) power dissipation and the transistors static power leakage. However, as referenced in [7], the dynamic power consumption accounts for more than 60-70% of the total power dissipation of a processor, and therefore, it seems reasonable to ignore the static power consumption of the processor. The approximated power consumption of the processor can be now estimated based on the power consuming transitions of all the transistors when the processor is executing tasks, which is shown in equation 10.

$$Power_consumption = \alpha C \times f_{clk} \times V_{dd}^2 \quad (10)$$

In this equation, α represents the activity factor, i.e., the fraction of the circuit that is switching, C is the switched capacitance, f_{clk} is the frequency of the cluster and V_{dd} is the supply voltage of the cluster. However, the existing processors on the market, generally don't have available the data needed to accurately compute the activity factor of the processor, and so, in this work, it will be also assumed that the power consumption can be approximated by using the equation 11 [7]. In this equation is also shown, how it can be predicted the power consumption for frequency 0 by knowing the actual power consumption for frequency 1 and the voltage supply values for each frequency.

$$\begin{aligned} P &\propto f_{clk} \times V_{dd}^2 \\ \frac{P_{f_0}}{P_{f_1}} &= \frac{f_0 \times V_{f_0}^2}{f_1 \times V_{f_1}^2} \end{aligned} \quad (11)$$

As already mentioned, when one task finishes it is done a reschedule of the executing tasks. When this occurs, the tasks must be paused in order for the cluster power consumption return to the value when no task was scheduled. This must be done because the reading of power consumption for the first task to be rescheduled must be only its own power consumption and not the overall cluster power consumption of all executing tasks. That said, the tasks used to evaluate the proposed framework must be modified in order to pause whenever the scheduler sends a reschedule signal.

IV. Experimental Evaluation

A. Experimental Setup

The proposed energy-aware scheduling approach was experimentally tested on ARM Juno r2 platform. This board can run different software stacks developed by both ARM and Linaro, which include an EDK2-based UEFI environment or a Flash with a Linaro kernel, combined with a filesystem (Android / BusyBox / OpenEmbedded (OE)) booted via U-Boot.

To evaluate the proposed scheduler, ARM Juno r2 board was configured with the software combinations present in Table II, by following the instructions present in ARM's "Using Linaro's deliverables on Juno" web document [8]. As already mentioned, this platform have a out-of-order dual-core Cortex-A72 and an in-order quad-core Cortex-53, which have the same instruction set architecture.

TABLE II: Available ARM Juno r2 software's combinations.

Flash	Kernel	PAPI	Energy meters
UEFI 16.04	3.10.0-1-linaro-lt-vexpress64	Yes	Yes
LSK 16.05	3.18.31 (GTS)	No	Yes
LSK 16.06	3.18.34 (EAS)	No	Yes

The Linaro Stable Kernel (LSK) incorporates the big.LITTLE MP patchset produced to support scheduling on heterogeneous multi processor systems. And so, the LSK is the one that supports the Global Task Scheduling (GTS) and the Energy-Aware Scheduling (EAS) ARM big.LITTLE approaches. The available filesystem chosen to configure the board was the OE LAMP 15.09, which is the recommended filesystem by Linaro to use. Each version of the available OE filesystems contains a /boot folder, in which is present the kernel that has the same kernel headers as the filesystem. However, this kernel is not used by the board when it is used the LSK in the Juno's Flash, but it can be loaded when an Unified Extensible Firmware Interface (UEFI) Flash is used. The UEFI Flash contains no kernel and it can be used to load the kernel from the filesystem boot folder, which is inserted in the USB port.

The fact that the filesystem do not contain the correct kernel headers for the LSK 16.05 and 16.06 kernels, some of the modules are not loaded when the board is booting up. Due to this, it was seen that PAPI could not be sucessuflly installed on those LSK configurations. The proposed framework was tested in the Linaro kernel present in the /boot folder of the OE LAMP 15.09 filesystem by using the UEFI 16.04 flash and will be compared with the GTS and EAS scheduling approaches present in LSK 16.05 and LSK 16.06 kernels respectively. It should be noted that each configuration have a different kernel.

B. Benchmarks

To evaluate the developed energy-aware game-theoretic scheduling approach with different applications, the following benchmark suites were considered: the Princeton Application Repository for Shared-Memory Computers (PARSEC) [9]; the Standard Performance Evaluation Corporation (SPEC) CPU 2006 [10]; the San Diego Vision Benchmark Suite (SD-VBS) [11] and the OpenBlas library [12]. These benchmarks were modified to use the necessary PAPI functions to create an event set and start the reading of counters as well as the code to pause the thread when the scheduler sends the reschedule signal. In Table III is shown the single-threaded benchmark applications used to evaluate the proposed framework as well as their respective input sets.

The most delay on the response to the pause signal in all successfully compiled benchmarks, was 10ms, which occurs when the application is being executed at the lowest frequency

TABLE III: Used benchmarks and respective configuration.

SD-VBS			OpenBLAS		
Benchmark name	Input set	Repetitions	Benchmark name	Input set	Repetitions
Disparity	test	8000	Sgemm	random values	2000
Mser	test	3400	Sgemv	random values	2000
Stitch	test	5000	Sscal	random values	200000
Texture Synthesis	test	2000	Saxpy	random values	80000
Tracking	test	500	Sdot	random values	80000
PARSEC			**SPEC CPU2006**		
Benchmark name	Input set	Repetitions	Benchmark name	Input set	Repetitions
Blacksholes	in_4.txt	5000	Bzip2	sample4.ref	2500

on the Cortex-A53. Therefore, the scheduler must wait approximately 10ms after sending the signal to start reading the power consumptions correctly.

C. Experimental Results

The developed MEM_bound and Tracking benchmark were used to evaluate the associated errors of the task execution time and CPI prediction functions. This evaluation revealed a maximum error of 2% between the measured values and the predictions. Afterwards, the Blacksholes benchmark was used to evaluate the power consumption predictions, which revealed a maximum error of 15% when compared with the measured values. These results suggest that future improvements can be done, especially to take into consideration the static power consumption of the processing units on the power predictions.

The proposed scheduler was also evaluated for different benchmark combinations. In Table IV are shown the energy consumption as well as the execution time for different benchmark combinations. It is also presented the achieved energy savings between the proposed scheduling approach and the Linaro's kernel 3.10, GTS and EAS scheduling approaches present in different kernels.

The proposed scheduler controls the frequency scaling and task migrations from the user-space through an algorithm programmed in C language. It also uses the APB interface and PAPI to read the energy meter registers and the performance counters, respectively, from the user-space, which have higher associated overheads than if could be possible to access them directly through the kernel. In order for the proposed scheduler algorithm to have no influence in the task executions, and also, to be as fast as possible like a kernel, the scheduler algorithm was executed in a dedicated Cortex-A53 core. In the experimental results, there are only three Cortex-A53 cores and two Cortex-A72 cores available to execute tasks. The remaining Cortex-A53 core is used to execute the scheduler algorithm of the proposed framework and is shut down in the remaining scheduling approaches of the other kernels. By adopting this, it is achieved fairness in the experimental results between the different evaluated scheduling approaches.

As can be see in Table IV, the proposed framework can achieve energy savings up to 36%, 32% and 22% when compared with the ARM's Linaro, GTS and EAS scheduling approaches. However, it can be unfair to compare the proposed scheduler, evaluated in the Linaro's kernel version 3.10, with the GTS and EAS scheduling approaches, which were evaluated in different kernels versions, 3.18.31 and 3.18.34, respectively. These kernels can have such improvements that are not present in the 3.10 kernel, and so, if one task is selected to be executed on both Linaro's and EAS approaches it can have different energy consumptions. This can be proved by looking at the MEM_bound task experimental results when the interactive governor was selected in both Linaro's and EAS kernel. During these experimental evaluations it could be seen that both kernels execute the MEM_bound task on one Cortex-A72 core at the highest frequency until it finishes.

In order to better understand the scheduling decisions of the proposed scheduler, Figure 8 shows the overall instantaneous power consumption (Cortex-A53 + Cortex-A72 + system) and frequency levels obtained during the execution of the benchmark combination number 4 (see Table IV) for the proposed scheduler as well as for the ondemand governor selected on Linaro's kernel 3.10.

On one hand, the ondemand governor uses the Cortex-A72 during more time and at higher frequencies than the proposed scheduler, which leads to higher power consumption. The ondemand governor also migrates tasks more often than the proposed scheduler, which executes the tasks on the same core until that task (or some other) finishes, in order to perform the task rescheduling. The proposed scheduler tends to select the best frequency for every task combination executing at the time. It should be noted that the ondemand governor took bad migrations decisions between 3-4s. During that time, it executed more than one task at the same core, which increased the frequency level and the power consumption.

V. Conclusion

The energy-aware game-theoretic scheduling approach herein proposed revealed to be capable of use the performance counters and energy meter registers of the system to gather information about the task in order to characterize it, and hence, by following the game-theoretic concepts, to select the best frequency and core to schedule the task, which corresponds to the minimum variation of the overall energy consumption in the device. The conducted evaluation of the proposed framework on ARM Juno r2 platform revealed that both power consumption and execution time predictions have approximated associated errors of 15% and 2%, respectively, and also, that can achieve energy savings of up to 36%, 32% and 22% when compared with the Linaro's kernel 3.10, GTS and EAS ARM big.LITTLE scheduling approaches, respectively.

References

[1] Roger B. Myerson. *Game Theory: Analysis of Conflict*. Harvard University Press, 1nd edition, 1997. ISBN:0-674-34115-5.

[2] Nickolas Bielik and Ishfaq Ahmad. Cooperative versus non-cooperative game theoretical techniques for Energy Aware Task scheduling. *2012 International Green Computing Conference (IGCC)*, pages 1–6, 2012.

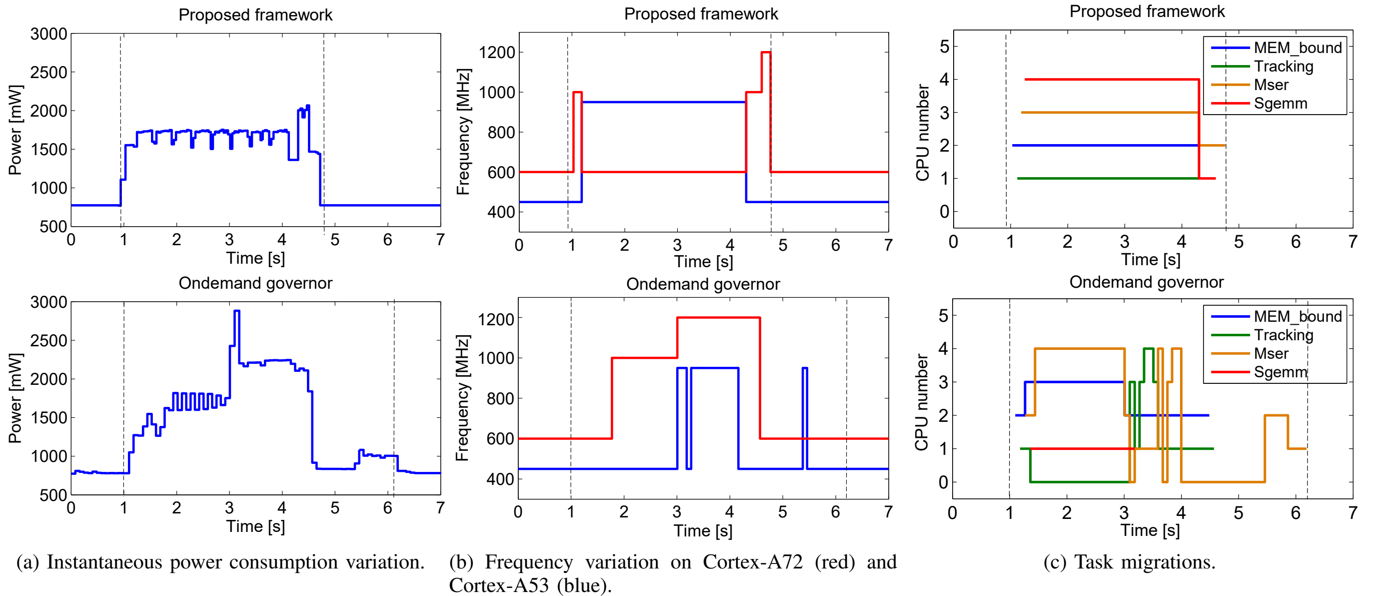
[3] Guowei Wu, Zichuan Xu, Qiufen Xia, and Jiankang Ren. An energy-aware multi-core scheduler based on generalized tit-for-tat cooperative Game. *Journal of Computers*, 7(1):106–115, 2012.

[4] ARM. *ARM Versatile Express Juno r2 Development Platform (V2M-Juno r2) Technical Reference Manual*, November 2015.

[5] Performance Application Programming Interface (PAPI), *https://icl.cs.utk.edu/projects/papi/wiki/ Threads*, Web accessed: 17 of April of 2016.

TABLE IV: Experimental results for each benchmark combination used to evaluate the proposed framework. The energy consumption values represents the overall energy consumed by the ARM Juno r2 platform until all tasks have completed their execution.

Benchmarks	Proposed Scheduler			Linaro Kernel 3.10 - UEFI 16.04			LSK 16.05 (GTS)			LSK 16.06 (EAS)		
	Energy Consumption [mJ] E_0	Execution Time [s]		Energy Consumption [mJ] E_1	Execution Time [s]	Savings (%) $\frac{E_1-E_0}{E_1}\times 100$	Energy Consumption [mJ] E_1	Execution Time [s]	Savings (%) $\frac{E_1-E_0}{E_1}\times 100$	Energy Consumption [mJ] E_1	Execution Time [s]	Savings (%) $\frac{E_1-E_0}{E_1}\times 100$
1 - MEM_bound												
userspace	2837	3,213	ondemand	3116	3,463	8,955	2983	3,428	4,904	3004	3,391	5,568
			interactive	3196	3,128	11,243	2747	2,942	-3,246	2711	2,948	-4,623
2 - Tracking												
userspace	2237	2,503	ondemand	3496	4,514	36,004	3319	4,308	32,582	2447	3,018	8,570
			interactive	2290	2,542	2,282	2206	2,484	-1,429	2198	2,488	-1,772
3 - MEM_bound, MEM_bound, and CPU_bound												
userspace	5801	5,202	ondemand	7560	5,43	23,27	7034	5,104	17,535	5646	4,392	-2,747
			interactive	7986	4,93	27,37	7103	4,624	18,330	5384	3,757	-7,733
4 - MEM_bound, Tracking, Mser, Sgemm												
userspace	6411	5,000	ondemand	7505	5,026	14,579	6650	4,597	3,604	6998	4,589	8,391
			interactive	7429	4,313	13,711	6727	3,868	4,706	7058	4,137	9,173
5 - Blacksholes, Sdot, Bzip2, Saxpy, Texture_Systhesis												
userspace	11936	9,270	ondemand	16133	9,976	26,016	13017	7,907	8,301	15448	8,857	22,733
			interactive	16591	8,952	28,058	12000	6,528	0,532	14083	7,584	15,244
6 - Saxpy, Blacksholes, Texture_Synthesis, Stitch, Sscal, Disparity												
userspace	12813	8,671	ondemand	19969	12,250	35,836	15899	9,837	19,409	16491	9,829	22,302
			interactive	16133	8,606	20,582	14837	8,003	13,646	15595	8,514	17,841
7 - Bzip2, Blacksholes, Sgemm, Stitch, Mser, Disparity, Tracking, Texture Synthesis, Sgemv												
userspace	14094	9,202	ondemand	17790	13,957	20,777	15928	9,955	11,513	15923	10,435	11,485
			interactive	17828	9,251	20,945	16582	8,527	15,005	16482	8,511	14,490



(a) Instantaneous power consumption variation.

(b) Frequency variation on Cortex-A72 (red) and Cortex-A53 (blue).

(c) Task migrations.

Fig. 8: Run-time power consumption, frequency and task migrations comparison between the Linaro's kernel 3.10 with ondemand governor selected and the proposed energy-aware scheduling approach. Cortex-A72 (cores 1 and 2), Cortex-A53 (cores 0, 3 and 4)

[6] Diego Puschini, Fabien Clermidy, C E A Leti Minatec, Pascal Benoit, Gilles Sassatelli, and Lionel Torres. Temperature-Aware Distributed Run-Time Optimization on MP-SoC using Game Theory. pages 375–380, 2008.

[7] Joel Wilkins, Ishfaq Ahmad, Hafiz Fahad Sheikh, Shujaat Faheem Khan, and Saeed Rajput. Optimizing Performance and Energy in Computational Grids using Non- Cooperative Game Theory. 2010.

[8] Using Linaro's deliverables on Juno, *https://community.arm.com/docs/DOC-10804*, Web accessed: 02 of March of 2016.

[9] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[10] Standard Performance Evaluation Corporation (SPEC) CPU 2006, *https://www.spec.org/cpu2006/*, Web accessed: 3 of August of 2016.

[11] Sravanthi Kota Venkata, Ikkjin Ahn, Donghwan Jeon, Anshuman Gupta, Christopher Louie, Saturnino Garcia, Serge Belongie, and Michael Bedford Taylor. SD-VBS : The San Diego Vision Benchmark Suite.

[12] Opensource Basic Linear Algebra Subprograms (OpenBLAS), *http://www.openblas.net/*, Web accessed: 21 of April of 2016.