Performance and energy-aware real-time scheduling for heterogeneous embedded systems

Francisco João Feliciano Gaspar
Email: francisco.gaspar@ist.utl.pt

Abstract—With an ever growing demand for energy efficient systems, heterogeneity has found its way into processors through the combination of high performance and energy efficient cores. The scheduling of tasks on these systems presents a challenge since it is desirable to achieve a good performance while saving power. Therefore, this thesis aims to analyse current scheduling techniques applicable to heterogeneous systems, and to propose a new task management topology that can optimize overall performance and efficiency by maintaining performance fairness among running tasks while limiting energy usage by only allocating the resources needed for the applications to achieve all their performance targets. To achieve this goal, a framework is proposed that gathers data from Quality of Service programs (integrated with a real-time performance reporting Application Programming Interfaces). It achieves task level share control, to ensure task performance fairness, dynamic frequency control (reduces the amount of available system resources which saves energy) and migration controller, enabling the control of single- and multi-threaded applications simultaneously.

The proposed performance and energy aware, real-time scheduling, for heterogeneous embedded systems framework, achieves energy savings of up to 65% while improving the performance error in up to 122x. These results were achieved by comparing the proposed framework with the default system tools on an Odroid-XU+E development board (featuring a big.LITTLE powered system on a chip) with applications selected from the PARSEC benchmark suite that were modified to report their performance.

Index Terms—Heterogeneous multi processor; scheduling; embedded systems; quality of service; big.LITTLE; task migration; dynamic frequency control

I. INTRODUCTION

The dissemination of embedded systems through our society, specially with the widespread use of smartphones and tablets, has created a significant demand for high computational capabilities on mobile platforms. The trends on these systems focus on energy-efficiency where the used Systems on a Chip (SoCs) merge in the same chip different CPUs sharing the same Instruction Set Architecture (ISA) in order for the same system to be able to both achieve efficiency and performance. One example of such an architecture is ARM’s big.LITTLE [1]. The challenge in heterogeneous systems is work distribution, since the decision mechanism should be adaptive and be aware of the performance goals of parallel applications. A controller that considers these factors will result in a system-wide collaborative approach which will facilitate energy savings.

Current Operating Systems (OSs) rely on two main mechanisms to perform task management and frequency scaling, specifically the task scheduler and Dynamic Voltage and Frequency Scaling (DVFS). In the specific case of Linux, whose derivations nowadays power a significant amount of embedded systems through Android, the default system scheduler (as of kernel 2.6.23) is the Completely Fair Schedule (CFS) [2]. Although it tries to equally split the CPU time among the running tasks, thus achieving fairness in the time domain, the applications performance is not considered and might significantly differ from the desired levels. As for DVFS strategies, current dynamic approaches focus solely on CPU load which again does not directly reflect application performance leading to the waste of energy by having tasks over-performing. Some authors are already tackling this issues [3], [4] by developing heterogeneity aware methodologies that also consider energy budgeting, however these implementations are only reported to work with single-threaded programs and require a significant controller overhead and multiple blocks.

This paper extends the previously published work [5] by proposing a new per core adaptive and light-weight task management framework. To achieve this, it uses a set of efficient Controllers together with a performance reporting and monitoring subsystem (Beeps) that communicate with the system and are responsible for:

- Modeling of application’s behavior on a given (heterogeneous) multi-core system;
- Run-time evaluation of current and attainable performance of applications;
- Maintaining performance fairness among all running applications on the system;
- Allowing energy-efficient application execution by considering frequency scaling trough DVFS;
- Considering the available sets of cores (clusters) and allocating tasks to the most appropriate one;

Finally, the performance achieved with the proposed scheduler is experimentally evaluated by measuring performance fairness and power consumption in a real system implementation.

II. RELATED WORK

As previously mentioned, the OS scheduler is responsible to fairly distribute CPU time across concurrent tasks. However, current scheduler implementations are based on load levels and not on the actual application performance requirements which presents a problem because iterative applications easily reach high load levels, but their performance is dictated from the amount of computational cycles they need to achieve a
certain milestone. What this means is that, by looking solely at the load, not only is the current scheduler (CFS) unfair at a performance level but also that too much energy might be being spent on the system, since DVFS, if dynamically adjusted, by default also only bases its decisions on core load.

There is already a significant amount of work in the literature regarding energy budgeting [6], [7], [8], mainly targeted at homogeneous systems. [6] uses offline models to build lookup tables that are used to decide core placement and DVFS levels of the running workloads. [7] focuses on chip-level monitoring, control and dynamic management of power while respecting objectives such as prioritization, power balancing and throughput on different benchmarks. Finally [8], relies on a three step approach to control power budgets and frequency levels per core while considering mixed groups of applications (single- and multi-threaded). Nevertheless, these methods no not consider heterogeneity, rely on performance counters and consider per core frequency scaling, characteristics which are incompatible with the target systems of this work.

More recent works focus on heterogeneous systems that share the same ISA allowing systems to perform migration of tasks across different types of cores without requiring recompilation and achieving different performance levels. Muthukaruppan et al. have developed Quality of Service (QoS) aware methodologies [3], [4] for big.LITTLE heterogeneous systems based both on control theory and price theory, but this methodologies work with single threaded applications and require a significant number of controllers (some requiring kernel recomputations) in order to achieve their goals. Other works targeted at heterogeneous systems [9], [10] aim at finding the best core for each type of thread but fail to consider performance fairness among threads.

III. PERFORMANCE AND ENERGY-AWARE REAL-TIME SCHEDULING FOR HETEROGENEOUS EMBEDDED SYSTEMS

This section details the proposed performance and energy-aware real-time scheduling for heterogeneous embedded systems framework, giving a brief overview of the system resources in which it relies and how it tackles the previously identified performance fairness and efficiency problems. The elements of the proposed framework interface with system tools such as the scheduler (to control the shares), core balancing (by allocating threads to cores) and DVFS (by setting the system frequency). Furthermore, it integrates the Beeps subsystem which provides the means for very accurate application monitoring by interacting with kernel-space scheduling facilities. An illustration of the integration of the proposed framework can be seen in Fig. 1.

A. Background

Achieving the proposed objectives is only possible by tightly coupling the framework with the system tools that control the scheduling, core balancing and frequency scaling (DVFS). To understand how everything comes together, a brief overview of the required system functionalities is herein presented.

\[
s_i = \frac{1024}{\sum_{j=1}^{N} 1.25^{n_j}},
\]

where \( s_i \) is the share attributed task \( i \), \( N \) is the total number of running tasks in the run queue and \( n_i \) is the nice level of task \( i \).

This previous explanation is thus valid for one run queue. In reality, the system may maintain different run queues with a typical case being having a run-queue for every core. The way tasks are balanced across these cores will determine the system fairness in multi-core systems and it is here that, by default, CFS may fail to attain it. This is due to the fact that one of the conditions for migrations to occur in CFS is that the load imbalances across cores must be greater than a certain dynamic threshold. The result of this condition is that, with certain load and number of threads combinations (below the threshold), some threads may accumulate more runtime than similar ones with the same priority. The approach taken to control thread balancing will be to dynamically pin each thread to a core, effectively bypassing CFS’s core balance.

A complementary aspect to the system scheduler is DVFS. This mechanism scales the system frequency and voltage in order to achieve certain power savings (higher power consumption with higher frequencies) and it is usually applied at the level of the clusters of cores. The DVFS functionality is defined by different governors that control the system frequency by relying on different static and dynamic strategies.
Static strategies fix the frequency at certain values, while dynamic strategies allow run-time frequency scaling based on the current system load. Energy-efficient execution is not guaranteed by the default implementation of this mechanism and that is the gap that the proposed work intends to close by scaling the frequency according to the performance needs of the currently running applications.

The frequencies selected by DVFS may be merely virtual, \(i.e.,\) the real frequencies at which the cores operate may be different. This enables the control of cluster migration on heterogeneous platforms though DVFS and creates an implicit performance relation among the clusters. A frequency (and consequently cluster migration) decision immediately takes into account the performance ratio between clusters, even though this value is not tested for specific applications and represents only a global estimation. Figure 2 illustrates such a scaling, where a set of lower virtual frequencies (250 MHz-600 MHz) is used to instruct the execution of threads on a Cortex-A7 cluster (at a real frequency which is twice the virtual frequency). Another higher set of frequencies (800 MHz-1.6 GHz) imposes the tasks to run on the Cortex-A15, cluster at a real frequency which is the same as the virtual one.

Fig. 2. Range of virtual and their correspondent real frequencies for a heterogeneous platform (Exynos 5410 on the Odroid-XU+e) using cluster migration

B. Performance monitoring

In order for the controllers to properly work, applications must report their performance in a standardized way. Application Heartbeats [12] is an Application Programming Interface (API) found on the literature that accomplishes this through shared memory and a set of predefined functions. Unfortunately it reports only per process performance without specific shares or execution time information. For this work the Beeps subsystem was developed which integrates the Beep API and Beep driver to accurately report real time performance and resource utilization of running applications at the thread level.

Beeps was created (based on [13]) to provide an insightful look at the required application data for the proposed framework to properly function. The API is responsible for communicating the beeps triggered by a function call on the application to the driver, which keeps the beeping and timing records for all the monitored tasks. The API is further responsible to register each application (at initialization) and transmit the data stored in the driver to the task management controller.

The Beeps driver is the low-level component responsible not only for keeping track of each target task’s scheduling path and beeping information, but also to provide the required input parameters to the controller and to handle the output requests from the controller. Tasks registration is made in a per-application basis and the leader task must be registered into the driver. When registering an application (or task group), a configuration structure (s beep_attr) is sent to the driver to transmit important data such as application type, target performance and sliding window size (used to average performance). Moreover, all the tasks descending from a registered leader task are automatically registered into the framework driver through the detection of fork events performed by the registered tasks.

Two different types of tasks are identified and registered by the driver: i) leader tasks are registered whenever the framework is initialized; and ii) child tasks are registered automatically by the driver upon being forked from a leader tasks or a task descending from a leader task. Whenever a read request is made to the driver through the provided API functions, the leader tasks are iterated and the timing and beeping information for each individual task is copied to the controller space. The information is sent by using the OS memory calls copy_from_user() and copy_to_user(), which allow copying the necessary data structures containing the required execution information from the user-space to the kernel-space and vice-versa.

As previously referred, the beep information is calculated within the driver each time a beep request is sent from the application (through the beep API). Whenever a beep request is made, the driver updates the beep information of the corresponding task or all the tasks belonging to that task’s group, depending on the application type (i.e., if a leader task is responsible to report overall performance or if individual threads report their own performance).

In order to obtain the accurate timing information within each task’s window, the framework driver interacts directly with the OS scheduler, therefore allowing to detect the exact time when each monitored tasks is scheduled in or out from a specific core. In addition, it is also possible to detect forks, task termination and CPU migration. The interaction between the driver and the OS scheduler is made through system tracepoints. Tracepoints are present in different parts of the OS’s code and allow one to register a custom callback function, which is called whenever the OS reaches a specified point of the execution. The Beeps driver makes use of four main scheduler tracepoints, thus allowing tracing the complete scheduling path of each monitored task:

- **sched_switch()** - tracepoint triggered whenever a task is scheduled into or out from a CPU core. Thus, it provides the accurate timestamps for the time scheduling path, which allow keeping track of each task’s execution time;
- **sched_migrate_task()** - triggered each time a task migrates from one CPU core to another. This allows keeping track of where in the architecture each of the monitored tasks is currently running;
- **sched_process_fork()** - triggered whenever a new task is created. This tracepoint is used for registering tasks descending from previously registered tasks and it marks the newly created task’s execution start;
- `sched_process_exit()` - triggered whenever a task terminates. This tracepoint is used to detect the end of the execution of the monitored tasks.

These tracepoints perform different functions within the Beeps driver. While `sched_migrate_task()`, `sched_process_fork()` and `sched_process_exit()` are used to keep track of the existing target tasks and to mark the beginning and end of their execution, the `sched_switch()` allows keeping track of the accurate execution time of each task. The execution time of a task can therefore be derived by relying on the beep timestamps (i.e., the timestamps for each sbeep_beep() call) and on the `sched_switch()` timestamps, allowing the driver to extract the exact time a thread spent in execution state ($T_{execB_i}$) within the $T_{intB_i}$ interval (i.e., the time interval between two sbeep_beep() calls).

1) **Current performance and share calculation:** Through the times obtained with Beeps, both the current thread share and thread performance can be calculated. The following expressions mathematically express how the per-thread performance and shares are calculated for applications where all threads report performance. In this expression, $T_{intB_i}$ refers to the time interval between Beeps and $W_{s_a}$ is the window size, which represents the number of beeps sampled in order to average performance over a longer period of time. Furthermore, $T_{execB_i}$ is the actual execution time between Beeps.

$$p_i = \frac{W_{s_a}}{T_{intB_i}}, \quad s_i = \frac{T_{execB_i}}{T_{intB_i}} \quad (2)$$

Note that, in this case, the application performance is the sum of the performances obtained in all cores and to get a target performance per core one must simply scale $p_i$ by the global error ($P_{da}/P_a$) of application $a$. The other possible case is that the application’s performance is reported by only one thread (although several of them may be contributing for it).

In this case some adaptations must be made to the previous expressions:

$$P_a = \frac{W_{s_a}}{T_{intB_a}}, \quad s_i = \frac{T_{execB_a}}{T_{intB_a}}, \quad p_i = P_a \frac{T_{execB_i}}{T_{execB_a}} \quad (3)$$

In this case some of the values refer to application times ($a$ index), $T_{execB_a}$ is the sum of the execution time among all threads off application $a$ inside window $W_{s_a}$ (time interval $T_{intB_a}$).

### C. System Controllers

To achieve the proposed goals the developed framework must have monitoring and control capabilities over the shares attributed to each task, thread to core allocation and frequency scaling. This is accomplished by the three block controller illustrated in Fig. 3. At the core level (and for every core) there is a share control step, then, systemwise, there is a frequency and migration controller. The target platform uses DVFS to decide when to change clusters and thus the frequency controller implicitly also manages the migration across different types of cores (clusters). All of this controllers rely on the current application performance acquired through the Beeps interface and are toggled at different rates as explained in the following sections.

In order to provide analytical tractability behind the proposed approach, the **Share controller** relies on a set of dynamically assessed task-specific performance parameters ($c_i$), which are constants that model how a task’s performance ($p_i$) is influenced by its share such that $p_i = c_i s_i$. In detail, to quantify the expected task performance for a fixed operational frequency, the $c_i$ parameters are assessed according to the most recent task sampling information, such that $c_i = p_{curr_i}/s_{curr_i}$, where $p_{curr_i}$ and $s_{curr_i}$ represent the currently attained performance and share of task $i$, respectively. It is worth emphasizing that both $p_{curr_i}$ and $s_{curr_i}$ parameters are reported by the application and measured in the system.

For a set of $N$ parallel tasks, the problem tackled by the **Share controller** can be formalized as follows:

$$\min_{s_1, \ldots, s_N} J(s_1, \ldots, s_N) = \sum_{i=1}^{N} (1 - \frac{c_i s_i}{p_{di}})^2 \quad (4)$$

with

$$\sum_{i=1}^{N} s_i = 1$$

or $s_i = \frac{c_i s_i}{p_{di}}$ for $i, j \in \{1, \ldots, N\}$.
where $s_i$, $p_{d_i}$, and $c_i$ represent the target share, the desired performance and the estimated behavior constant for task $i$, respectively. When minimizing the normalized performance difference of all running tasks, i.e., $1-p_{i}/p_{d_i}=1-c_i s_i/p_{d_i}$, this controller relies on the normalized performance measure in order not to break the linearity and a unitary value will mean the performance target is achieved. Furthermore, the problem expressed in (4) is accompanied by two additional restrictions, which imply that the sum of all per-task shares $s_i$ must be equal to one (i.e., to guarantee utilization of all CPU resources) and that the normalized difference is equalized among the running tasks (which guarantees that a second linear scaling can be applied and also implicitly assures non-negativity of per-application shares). The shares are applied to the tasks via their nice levels as reported in (1) while trying to maintain the levels as close as possible to the default value.

2) Frequency: As previously explained, the share controller guarantees the equalization of the normalized performances of the tasks in a core. This, however, does not guarantee that the tasks are on target since they may be equally above or below performance. The frequency controller aims at correcting this and, by doing it, enables energy savings. Its principle is illustrated in Figure’s 4 last block, Frequency, where the previously over performing tasks had their performance throttled back to the target. This behaviour is modelled by

$$f_n = f_o - \frac{P_{da}}{P_a}$$

where $f_n$ is the new frequency set to all the cores $P_{da}$ is the desired performance of application $a$ and $P_a$ is the current estimated application performance, obtained by adding together all the contributions of the individual threads, i.e., $P_a = \sum c_j s_j$ where $j$ is a thread belonging to application $a$. Applications on the whole system are not necessarily at the same performance level since a combination of multi- and single-threaded applications may make this infeasible. As such, only the application with poorer performance is used (as application $a$) for this controller, meaning that while some tasks may over perform, as long as the requested performances are feasible, all applications will perform at or above their target.

3) Migration: Including a migration supervisor to the controller is fundamental to handle applications that do not spawn threads across all the system cores which includes single- and multi-threaded applications with a number of threads different than the number of cores. This method can further merge threads from the same application on the same core (effectively serializing it) as long as a core alone is expected to provide enough performance to achieve the target.

The controller works by distributing tasks among cores according to the expected share they require to achieve the target performance. It then recalculates the shares in the cores in order to rebalance the performance among the newly allocated tasks.

IV. EXPERIMENTAL EVALUATION

To validate the proposed task management framework, a thorough experimental evaluation was conducted on an Odroid-XU+E development platform together with a set of benchmarks taken from the PARSEC Benchmark Suite [14]. The following sections summarize the platform and benchmark characteristics as well as some significant results.

A. Experimental Platform

The proposed framework was experimentally tested on an Odroid-XU+E platform. This board is powered by a Samsung Exynos 5410 SoC, which features 4 Cortex-A7 (energy efficient) and 4 Cortex-A15 (high performance) cores together with a PowerVR SGX544MP3 GPU and 2GB LPDDR3 RAM. The supported big.LITTLE modes are limited to cluster migration (the tasks either all run on the A7 or in the A15 cores), which is driven by the system DVFS. From the OS point of view, the board features a discrete range of frequencies from 250 MHz to 1.6 GHz, which represents a virtual frequency range, as previously described in Section III-A. In reality, when the system selects a virtual frequency in the range of [250 MHz, 600 MHz] the tasks will be migrated to the A7 cluster with a real operating frequency that is twice as high ([500 MHz, 1.2 GHz]). On the other hand, if the selected virtual frequency is in the range of [800 MHz, 1.6 GHz], tasks will run on the A15 cluster at the same physical frequency. This board also features current and voltage sensors which provide a way to gather the power consumed by the board via a driver. These values are updated at every 0.3 s.

The board is running a custom Ubuntu Linux OS with a 3.4.84 kernel provided by the manufacturer of the board, Hardkernel. Although the board features a heat sink and fan (which starts working when temperatures raise), a common scenario is that, at maximum frequency, when temperatures reach values as high as 100 C, the board throttles down frequency from 1.6 GHz (maximum) to 1.4 GHz. This mechanism is highly relevant, since it might interfere with the functionality of both the system’s default controller and the proposed one. The default system is prepared to expect this thermal emergencies and maintains the system at the lower operating frequency for enough time for the thermal emergency to pass. However, the proposed framework does not currently expect such a behavior and immediately requests an increase of system frequency. This implies that the system spends more time at high operating frequencies with the proposed framework that with the default tools, meaning that in these situations, the proposed framework may consume a greater amount of energy than the default control systems.

B. Benchmark Characterization

To evaluate the proposed framework, a set of multi-threaded programs from the PARSEC Benchmark Suite [14] was used and adapted to report the corresponding performance. For this, they were integrated with Beeps and Application Heartbeats [12], in order for them to report performance both with (Beeps) and without (Heartbeats) the proposed framework. To use these interfaces the selected benchmarks were modified in order to generate a signal at specific representative sections of the application (i.e., an iteration of the main computational loop). Table I summarizes the benchmarks used for assessing
the benefits of the proposed system and what the signals represent. This same table also identifies the inputs used in each benchmark and the average performance obtained with maximum resource allocation (i.e., 4 threads, maximum frequency and no concurrent applications). It is worth recalling that this controller is specifically tailored for non-critical real-time applications whose performance target can be set either by physical or human associated parameters (e.g., perceivable frame rate).

<table>
<thead>
<tr>
<th>Test</th>
<th>Input set</th>
<th>Beep location</th>
<th>Max. Overall Av. Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fluidanimate (F)</td>
<td>PARSEC's 100K Fluid Frames: 150</td>
<td>Every frame</td>
<td>2.5 B/s</td>
</tr>
<tr>
<td>Swaptions (S)</td>
<td>Swaptions: 128 Simulations: 400000</td>
<td>Every swaption</td>
<td>1.0 B/s</td>
</tr>
<tr>
<td>x264 (x)</td>
<td>sint_x264_av_100p4y4m</td>
<td>Every 2 frames</td>
<td>13.1 B/s</td>
</tr>
<tr>
<td>Blacksholes (B)</td>
<td>PARSEC's 5GB</td>
<td>Every 23900 options</td>
<td>23.5 B/s</td>
</tr>
</tbody>
</table>

C. Experimental Results

To test how the proposed framework for performance and energy-aware real-time scheduling for heterogeneous systems handled different workloads, a diversified series of tests with different combinations of the previously introduced benchmarks were run. The invocation periods of the controllers were empirically chosen such that a stable behavior could be obtained from the framework. It was verified that a minimum Share controller invocation period of 0.2 s allows achieving execution control of satisfactory granularity. This interval holds as long as all the threads in a core report their performance in that interval and is automatically extended otherwise, in order for the shares to be calculated based on the most recent data. The Frequency controller was invoked at every 10 share invocations of the share controller. This invocation period was selected to prevent performance instabilities, since there must be a minimum number of share updates performed before the frequency can be scaled. As for the Migration controller, it was activated whenever the total number of threads in the system changes and at every 3 s, in order for it to recognize possible phase changes in the tasks and migrate them accordingly periodically.

The experimentally obtained results with different combinations of the introduced benchmarks are presented in Table II. The proposed framework managed to reduce the Performance relative error in all tested scenarios (compare columns 4 and 7 of Table II). The reduction of the performance relative error ranges from values of around 1.09x (row 6 on Table II) up to 122x (row 3 on Table II), when compared with default system tools. Furthermore, all the test sets that did not require the board to operate at maximum frequency (all the tests that do not have * in their target performance, column 2) achieved energy savings. Specifically, the test set running two instances of Swaptions (row 3) with 4 threads per instance achieved 65.4% energy savings. On the other hand, the test set presented in row 10 of Table II required the full availability of the resources in the board. This happen because the target performance requested of Swaptions corresponds to the maximum performance achievable by one thread of this benchmark when it has exclusive use of the total computational resources of one core (see the maximum performance values obtained with four threads in Table I). In this situation, the previously discussed thermal issues (see Section IV-A) contributed to the increase of energy consumption with the proposed framework by 10.6%.

Figures 5 and 6 illustrates a case where 3 different benchmarks (i.e., Blacksholes, Swaptions and Fluidanimate) were called with 2 threads each (corresponding to row 6 of Table II). The grey bars in the performance graphs (Fig. 5) represent a $\pm 10\%$ deviation from target (dashed line) while the grey area in Fig. 6 corresponds to A7-mapped frequencies. This test configuration presented an interesting case study since the framework was controlling 6 threads, meaning that 2 of the cores were dedicated to the execution of only one thread. However, the system smoothly handled this challenge (Fig. 5) and all applications managed to execute at or above their target performance.

Since different types of workloads were present in the system, it was not possible to exactly match the specified targets (right side of Fig. 5) meaning that both Blacksholes and Fluidanimate had to increase their performance in order for Swaptions to achieve its target. Nevertheless, this presented an improvement both in performance (see left side of Fig. 5, with all applications over-performing) of 32x (see Table II). Blacksholes often presented spikes in its execution which were attributed to the simultaneous reporting of performance across threads. Although graphically this may appear to show irregular performance, it was verified that most of the reports are on the lower contour of the line. Furthermore, while Heartbeats (used on the left side of Fig. 5, without the controller) records every performance communication, Beeps only samples (and stores) the performance at every 0.2 s (minimum Share controller invocation time), which, depending on the application, may happen less often than the performance report.

With this configuration, while using the system’s default controllers, the system frequency often oscillated between 1.6 and 1.4 GHz (see top of Fig. 6) which is a sign of thermal emergency. However, the proposed framework achieved significant energy savings, 38.5%, by performing part of the applications’ execution on the A7 cluster (see bottom of Fig. 6).

V. Conclusions

This paper presents a state of the art performance and energy-aware real-time scheduling for heterogeneous embedded systems framework. It proposed a new method for dynamic scheduling on real-time heterogeneous embedded systems and developed a framework capable of maintaining performance fairness by core and scaling the available system resources in order to follow performance targets of multiple single-threaded and multi-threaded applications without spending unneeded energy. The necessity for such a controller is evidenced by the proliferation of the use of smartphones and tablets and the need to achieve higher performances on these devices with limited energy budgets.
TABLE II
RESULT SUMMARY, WITH PERFORMANCE AND ENERGY DATA FOR A GROUP OF TESTS. THE PERFORMANCE VALUES ARE RELATIVE TO THE PERIODS WHERE ALL APPLICATIONS RUN SIMULTANEOUSLY WHILE ENERGY VALUES CORRESPOND TO THE OVERALL CONSUMPTION UNTIL ALL PROGRAMS CONCLUDE THEIR EXECUTION.

<table>
<thead>
<tr>
<th>Benchmarks and Number of Threads (T)</th>
<th>Without an active controller</th>
<th>With the proposed controller</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Target Performance $P_T$ [B/s]</td>
<td>Observed Performance $P_{Ox}$ [B/s]</td>
</tr>
<tr>
<td>Swaptions (S) 4T</td>
<td>0.3</td>
<td>0.441</td>
</tr>
<tr>
<td>Swaptions (S) 4T</td>
<td>0.1</td>
<td>0.440</td>
</tr>
<tr>
<td>Swaptions (S) 4T</td>
<td>0.4</td>
<td>0.503</td>
</tr>
<tr>
<td>Swaptions (S) 2T</td>
<td>0.2</td>
<td>0.247</td>
</tr>
<tr>
<td>Swaptions (S) 1T</td>
<td>0.1</td>
<td>0.126</td>
</tr>
<tr>
<td>Swaptions (S) 4T</td>
<td>0.2</td>
<td>0.514</td>
</tr>
<tr>
<td>Fluidanimate (FA) 4T</td>
<td>1.0</td>
<td>1.066</td>
</tr>
<tr>
<td>Swaptions (S) 4T</td>
<td>10.0</td>
<td>60.918</td>
</tr>
<tr>
<td>Fluidanimate (FA) 2T</td>
<td>0.2</td>
<td>0.356</td>
</tr>
<tr>
<td>Blacksholes (B) 4T</td>
<td>20.0</td>
<td>30.277</td>
</tr>
<tr>
<td>Fluidanimate (FA) 4T</td>
<td>0.25</td>
<td>0.374</td>
</tr>
<tr>
<td>Swaptions (S) 4T</td>
<td>0.25*</td>
<td>0.084</td>
</tr>
<tr>
<td>Fluidanimate (FA) 4T</td>
<td>0.5</td>
<td>0.791</td>
</tr>
<tr>
<td>Blacksholes (B) 4T</td>
<td>2.0</td>
<td>53.723</td>
</tr>
<tr>
<td>Fluidanimate (FA) 4T</td>
<td>0.25</td>
<td>0.398</td>
</tr>
<tr>
<td>Fluidanimate (FA) 4T</td>
<td>0.5</td>
<td>0.866</td>
</tr>
<tr>
<td>x264 (x264) 4T</td>
<td>1.25</td>
<td>0.700</td>
</tr>
</tbody>
</table>

* Although these performances are feasible, 0.25 B/s is the maximum performance achievable by Swaptions with one thread, meaning that the system will be forced to maintain maximum frequency.

The conducted evaluation revealed that the proposed framework is able to accomplish the defined objectives by providing a fairer execution of tasks across the system while saving energy. These tests relied on the Beeps subsystem, a purpose built performance reporting tool, inserted into four programs from the PARSEC benchmark suite, Blacksholes, Fluidanimate, Swaptions and x264. Depending on the selected targets, results showed energy savings of up to 65% as well as significant reductions on the performance relative error, up to 122x.

In summary, the proposed framework is able to greatly improve the resources’ distribution in a complex heterogeneous...
system, achieving the specified target performances and saving energy while running diverse applications (single- and multi-threaded) without a prior offline evaluation.

REFERENCES