Performance-aware task management and frequency scaling in embedded systems

Francisco Gaspar, Aleksandar Ilic, Pedro Tomás and Leonel Sousa
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa
Rua Alves Redol, 9
1000-029 Lisbon, Portugal
Email: {fgaspar,ilic,pfzt,las}@sips.inesc-id.pt

***Abstract*—Due to the dissemination of smartphones and tablets, a constant complexity growth can be observed for both embedded systems and mobile applications. However, this results in an increase in energy consumption. To guarantee longer battery life cycles, it is fundamental to develop system level strategies that allow guaranteeing the applications' required quality of service by managing the available system resources. In this paper a new task management framework is proposed that controls, in real-time, the execution of multi-threaded applications in order to meet their performance targets. For this, we amend the Linux CFS scheduler decisions to efficiently control the shared resource utilization of parallel applications. The proposed framework relies on runtime performance modeling of both the underlying architecture and the running applications to scale the system resource allocation and frequency. As a result, efficient application execution is achieved not only in terms of performance, but also in energy consumption. Experimental results show that the proposed approach satisfies the applications required performance level by decreasing the relative performance error from 2.801 to 0.168, while achieving 49 % energy savings.**

I. Introduction

In recent years, an evident growth of smartphone and tablet markets has lead to the dissemination of complex embedded systems, which have now become intrinsic parts of our society [1]. In order to satisfy the high computational demands of modern real-world mobile applications, current trends in embedded architectures are moving towards increasingly parallel systems whose performance is getting closer to that of general-purpose architectures. These architectural improvements and the advancements seen in manufacturing technology have resulted in a growth of heterogeneity inside a SoC, thus providing the ability to cope with highly demanding workloads while preserving energy-efficiency. In fact, besides incorporating processing elements of different architectural types (such as GPU and CPU), current SoCs contain an additional level of heterogeneity even for elements using a single Instruction Set Architecture (ISA). For example, in order to provide higher performance and energy-efficiency, the ARM big.LITTLE architecture already incorporates separate clusters of out-of-order and in-order CPU cores [2], [3].

Power and dissipation constraints make it impossible to simultaneously employ all the available system resources, a problem which is expected to worsen due to the presence of Dark Silicon [4]. As a consequence, the efficient execution of tasks on these highly heterogeneous systems imposes additional challenges. In fact, to sustain the efficient execution of complex parallel applications, it is essential to provide more advanced task management mechanisms for these types of systems. First, task management needs to be adaptive, in order to allow amending the scheduling decisions according to the current state of the execution platform and during application run-time. Second, in order to guarantee satisfying the target performance levels for a certain set of parallel applications, these mechanisms also need to be application-aware. As a result, by capturing the interaction between the applications and the underlying architecture, these self-learning approaches will allow amending the scheduling decisions according to the realistically assessed application requirements and the capabilities of the architecture to satisfy them. Hence, these scheduling decisions not only need to tackle the adequate thread placement and core migrations, but also the configuration of the execution platform, such that an energy-efficient execution is attained by setting the frequency levels according to the application needs.

Nowadays Operating Systems (OSs) provide two separate mechanisms for task management and frequency scaling, namely: *i*) the default task scheduler, *i.e.*, Completely Fair Scheduler (CFS) [5], which is responsible for workload distribution among computational resources; and *ii*) Dynamic Voltage and Frequency Scaling (DVFS) mechanisms for run-time adjustment of the running voltage and frequency. Current DVFS strategies are nevertheless mostly unaware of the application performance, since they rely on the current system load to set the frequency level. As a result, as long as the running tasks are not significantly I/O bounded, they usually set voltage and frequency levels to the maximum. On the other hand, the CFS [5] tries to equally split the CPU time among the running tasks, thus implying that, although fairness is achieved in the time domain, the application performance might significantly differ from the desired levels. As can be observed, current systems generally do not include mechanisms to adapt the scheduling decisions according to the performance of running workloads. Thus, they usually become bounded by the performance delivered by the system. However, attaining the application performance targets does not always require all available computational resources to be assigned (neither to be equally shared). In fact, the resources need to be allocated according to the application requirements, thus creating the opportunity for further energy savings and to increase resource availability for other tasks.

In the current literature, the state-of-the-art approaches mainly target homogeneous general-purpose multi-core systems, where efficient power/energy management solutions are investigated at the architecture-level via DVFS and scalable online estimation strategies for energy savings [6], [7]. On the other hand, for heterogeneous embedded platforms, studies that improve upon the system scheduler and DVFS by monitoring task performance are surprisingly quite rare [8], [9]. In particular, the existing methods mainly require the implementation and integration of complex scheduling mechanisms that usually impose kernel-level modifications.

In this paper, an adaptive and light-weight task management method is proposed that explicitly takes into account the realistically attainable performance levels of parallel applications on the underlying architecture in order to provide performance fairness among the running tasks. By interacting with the OS scheduler, the proposed method allows attaining a fine-grained control over the allocation of shared computational resources among the parallel tasks, such that the target performance is achieved for the currently running applications. By tracking the relative performance-to-target ratios for all running applications, the proposed approach also allows for energy-efficient execution, where automatic frequency scaling is performed according to the dynamic characterization of the execution of the parallel tasks. As a result, by tightly coupling the DVFS and shared resource allocation decisions, the proposed method also assures that the feasible target performance levels are simultaneously attained for all running application with as little energy as possible.

To provide a comprehensive description of the proposed task management method, we organize the following sections as follows. Section II introduces the basic concepts around modern OS schedulers and provides an overview of the state-of-the-art approaches for OS-level task management. Section III presents the proposed task management system, which provides fairness to the multiple running applications according to the required performance levels. To evaluate the proposed method, it is implemented on an Odroid-XU+E development board featuring an ARM big.LITTLE processor. Experimental results presented in Section IV demonstrate that the proposed system not only allows satisfying the required performance levels lowering the relative performance error from 2.801 to 0.168, but it also brings significant energy gains to the system of up to 49 % on the tested scenarios. Finally, Section V concludes the paper.

II. Background and Related Work

As previously referred, task scheduling is an essential part of modern OSs to manage the execution of multi-threaded applications. In general, the OS scheduler attributes a time slice of the scheduling period to each running thread. On the other hand, when there are no active threads in the system, the OS scheduler is also responsible for maintaining the idle state of the CPU. By default, whenever the active threads do not involve significant amount of I/O operations (*i.e.*, accesses to the disk or other system peripherals), the scheduler usually allocates a similar time slice to every running task in order to maintain fairness. However, this fairness only refers to the time domain and it does not necessarily imply fairness in terms of the attainable application performance. In order to provide the means for achieving energy savings at the architecture level, DVFS is introduced as a complementary subsystem that usually lowers the system frequency (and voltage) whenever the system is not heavily loaded.

A large body of existing state-of-the-art approaches target homogeneous general-purpose systems, when addressing the efficient execution of parallel applications [6], [7]. They mainly investigate advantages of applying DVFS-based approaches for power management [6], as well as scalable modeling of single- and multi-threaded application when targeting energy-efficiency [7]. On the other hand, for modern heterogeneous embedded platforms, there is only a limited number of studies that improve upon the system scheduler and DVFS by monitoring task performance in order to minimize energy consumption and achieve the target performance [8], [9].

For single-ISA systems, techniques that rely on online performance estimation have already been documented [10], as well as solutions that rely on an offline state space analysis to provide the knowledge for the online task to core allocations [11]. Muthukaruppan et al. [8] proposed a method based on control theory to improve management of system resources, while in [9] they propose a dynamic scheduling method based on a price-theory and DVFS to satisfy the user demands. However, the implementation complexity of both proposed solutions requires either a Linux kernel recompilation [8] or additional support at the level of kernel modules [9]. Furthermore, the proposed dynamic scheduling solutions that rely on multiple control agents usually imply significant scheduling overheads due to their implementation and functional complexity.

In contrast to the state-of-the-art approaches, the method proposed herein relies on online application performance monitoring to provide run-time adaptation of shared resource allocations among several parallel applications. In addition, the proposed approach also tightly couples these decisions with DVFS, thus interconnecting the mechanisms to achieve the target application performance at lower energy consumption levels. Furthermore, this method also targets the energy-efficient execution in highly heterogeneous embedded platforms by providing the means for dynamic selection of the most appropriate cluster of multi-cores to efficiently perform the computations of several simultaneously running parallel applications. The proposed method also involves significantly reduced implementation complexity, since it relies on the existing OS-level mechanisms for managing the execution of parallel applications. In order to provide a better understanding of the proposed approach, in the following text the basic functional principles behind the default OS scheduler and DVFS are presented.

A. Scheduler

Current Linux kernels (since version 2.6.23) adopt the CFS [5] as the default system scheduler, which then decides on the actual placement of the running tasks on the available CPU cores. The objective of this scheduler is to maintain fairness among the running tasks while offering a low implementation complexity. The fair concept resembles the idea of providing an opportunity for all running tasks to equally share system resources based on the allocated execution time for the tasks, *i.e.*, the CFS [5] generally does not favor the execution of any particular task in the system. In practice, the execution of multiple independent tasks is managed by giving a small time slice to each running task and by periodically switching their execution on the assigned CPU core. As a result, by

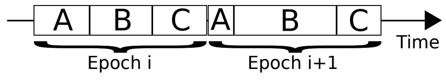


Fig. 1. A graphical interpretation of the basic scheduler functionality

switching the execution of different tasks very quickly, the end-user becomes unaware of these context changes and has a perception of a seamless execution.

Figure 1 presents an example of the execution switch among three tasks (*i.e.*, A, B and C) within a scheduling period (*epoch*). As can be observed, during the first scheduling period (see *epoch i*), all three tasks are assigned with an equal time slice to execute, *i.e.*, the overall scheduling period is equidistantly partitioned among the running tasks and their execution is performed in a round-robin fashion. In fact, this represents the usual method of conducting the execution of multiple tasks in a real system. Hence, the CFS [5] splits an epoch such that each task has a similar virtual runtime, *i.e.*, the relative time measure that represents the weighed time a task has run on the CPU. The weighting factor depends on the assigned *nice* value, and all tasks are usually assigned with the nice value of 0. As a result, no indications of application performance are taken into account when deciding on the time shares given to the tasks.

In order to introduce the performance-aware task distribution by altering the time share of a given task, one should change the task's $nice$ value. The $nice$ values are selected from a predefined range of integer values, namely from -20 to 19. Accordingly, the time share (s_i) of a task i running on a certain core can be calculated as follows:

$$s_i = \frac{\frac{1024}{1.25^{n_i}}}{\sum_{j=1}^{N} \frac{1024}{1.25^{n_j}}}, \quad (1)$$

where N is the total number of running tasks in the core and n_i is the nice level of task i. It is worth noting that the time share of a particular task i is calculated relatively to the $nice$ levels of all j tasks currently assigned to execute at the same core, *i.e.*, n_j nice values. Accordingly, if all tasks j maintain their nice levels, the time share of the single task i can be increased by lowering the nice value; naturally, increasing the nice level implies a smaller time share.

As a result, by controlling the time shares on a per-task basis via the $nice$ values, it is possible to provide a better tuning of the realistically achievable performance of the running tasks. In fact, the method proposed herein relies on this interface to effectively alter application runtime behavior and to reach their target performance. For example, as presented in Fig. 1 for *epoch* $i{+}1$, by attributing different $nice$ values to tasks A, B and C, it is possible to affect their shares in the overall scheduling period, *i.e.*, task B has a larger time share than tasks A and C. Regarding the actual $nice$ values, for *epoch* $i{+}1$, they are set in the following order: $B < C < A$, whereas all tasks have similar nice levels for *epoch i*. As it can be observed, by controlling the task time share (*i.e.*, execution time at the level of the *epoch*), it is also possible to control the achievable performance levels of multiple simultaneously running applications.

B. Dynamic Voltage and Frequency Scaling (DVFS)

A complementary aspect to the system scheduler is DVFS. This mechanism scales the system frequency and voltage in order to achieve certain energy savings and it is usually applied at the level of the clusters of cores. The DVFS functionality is defined by different governors. The governors usually allow controlling the system frequency by relying on different static and dynamic strategies. For example, the static strategies provide the execution modes for achieving the maximum performance or maximum energy savings by fixing the running frequency at the maximum or minimum value, respectively. On the other hand, the dynamic strategies allow run-time frequency scaling based on the current system load.

As can be observed, the current DVFS strategies mainly target energy savings from the architectural point of view, generally not implying an energy-efficient execution of parallel applications with performance constraints. For this, the work proposed herein relies on the ability to set the desired operating frequency, such that the target performance of several simultaneously running multi-threaded applications is automatically achieved while minimizing energy consumption.

III. Performance-aware Task Management

This section provides a detailed description of the proposed adaptive control mechanism to achieve performance-aware task management in highly heterogenous embedded platforms. This mechanism relies on online performance monitoring to explicitly consider the run-time behavior of multiple parallel applications running on the underlying architecture. Based on monitored application-specific parameters, the proposed method provides the decisions regarding the allocation of shared system resources, such that the target performance levels are achieved on a per application basis, as well as to guarantee the performance fairness among the running applications. In addition, the proposed mechanism also relies on DVFS to manage the system energy-efficiency levels and to further augment the scope in which performance can be scaled such that energy consumption savings are achieved.

A. Problem formulation

The execution platforms targeted in this work are heterogeneous embedded systems, which contain several clusters of multi-cores with different micro-architectures that share the same ISA. This means that depending on the core type selected for execution, the behavior and performance levels of currently running applications might significantly vary according to the architecture-specific features, *e.g.*, in-order/out-of-order execution, number of parallel instructions issues, branch prediction mechanisms and/or complexity and capacity of different memory hierarchy levels. As a result, in single-ISA heterogenous platforms, this augmented diversity of architectural features may provide a great opportunity for improving the energy-efficiency, but it also causes a greater challenge when managing the execution of parallel tasks.

The performance of real-world parallel applications is not only limited by these micro-architectural features, but also by application-specific requirements, such as number of memory instructions, integer vs floating point operations, conditional

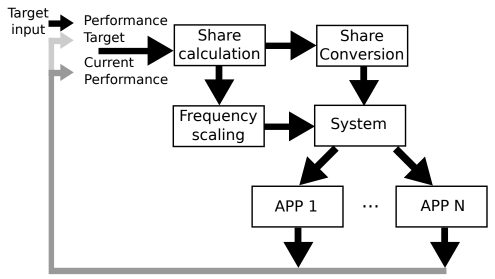


Fig. 2. Block diagram of desired controller modules

statements and/or application input. In fact, nowadays applications may even contain several execution phases with different requirements, thus the achievable performance levels may also significantly vary across different execution periods of the same application. As a result, in order to determine the allocation of available system resources within the proposed controller, it is crucial to provide the means for run-time detection and modeling of this application-architecture interaction.

The adopted modeling concepts do not need to capture all possible details regarding the architecture and application complexity, but they rather have to be insightful enough to allow adequate shared resource allocation, such that the application performance targets are achieved. In addition, in order to provide adaptive decisions during the application runtime, the insightful modeling is of the utmost importance in order to reduce the computational complexity and overheads introduced by the controller. In detail, the main objective of the proposed approach is to determine the CPU share (s_i) and operating frequency (f), such that the given performance target (P_{d_i}) is achieved for the running application i. For this, the control mechanism considers performance to be proportional to both share and frequency ($P \propto s$ and $P \propto f$). In order to capture the influence of architectural and application characteristics on the attainable performance, an application-specific parameter (c_i) is introduced and it is regularly updated by sampling the application performance data at every controller update. More specifically, c_i represents the relative measure used to describe the performance behavior of a process i, such that the application performance can be expressed as $P_i = c_i s_i f$.

In order to simultaneously approach the target performance levels (P_{d_i}) for a set of parallel applications, the proposed mechanism automatically adjusts their CPU time shares (s_i) by relying on the dynamically acquired c_i parameters and by setting the nice levels n_i (see (1)) and operating frequency f. As presented in Fig. 2, the proposed performance-aware task management method incorporates three different steps in order to achieve this functionality. First, the *Share calculation* step is applied to determine the CPU time share s_i for each application i. This step also assures performance fairness among the running applications by equalizing the relative distance between the realistically attained and the target performance for all active tasks. Afterwards, the *Frequency scaling* step is applied to minimize the assessed relative distances by automatically scaling the operating frequency f, such that all the achievable applications' performances are as close as possible to their target performance levels, *i.e.*, $P_i \approx P_{d_i}$, without affecting the previously attained fairness. Finally, in the *Share conversion* step the previously obtained shares s_i

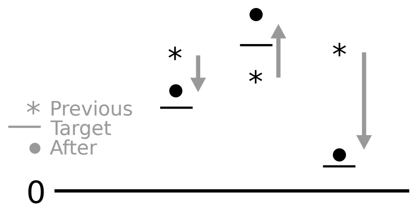


Fig. 3. Graphical interpretation of the share calculator, 3 tasks are brought to within the same error margin

are translated to the nice levels n_i that are attributed to each application. Then, the obtained n_i and f values are fed to the system, and the control mechanism is restarted after the application performance is resampled.

B. Run-time Control of Shared Resources

As previously referred, the proposed method tackles the resource allocation problem by integrating three different functional blocks (steps). In order to provide a detailed description of the proposed approach, the main functional principle behind each individual step is presented in the following text.

1) Share calculation: The first step of the controlling method determines the CPU share for each parallel application, such that their realistically achievable performance is as close as possible to the target performance. In addition, this step also assures the performance fairness among the running applications by equalizing their percentual differences between the achievable and targeted performance. Figure 3 provides the graphical interpretation of this idea for three different applications. As it can be observed, the previously assessed application performance (represented with stars) greatly mismatches the target performance levels (marked with lines). However, after applying the *Share calculation* step, it can be noticed that the newly achieved application performance (represented with dots) is not only inline with the desired performance targets, but also that the percentual difference is equalized among the applications (see the relative difference between lines and dots in Fig. 3).

In order to provide analytical tractability behind the proposed approach, the *Share calculation* step relies on a set of dynamically assessed application-specific performance parameters (c_i) and previously referred application modeling strategy (see Section III-A). In detail, to quantify the expected application performance for a fixed operational frequency, the c_i parameters are assessed according to the most recent application 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 an application 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 applications, the problem tackled in the *Share calculation* step can be formalized as follows:

$$\min_{s_1,...,s_N} \quad J(s_1,...,s_N) = \sum_{i=1}^{N}(1 - \frac{c_i s_i}{P_{d_i}})^2 \qquad (2)$$

$$\text{such that} \quad \sum_{i=1}^{N} s_i = 1$$

$$\tfrac{c_i s_i}{P_{d_i}} = \tfrac{c_j s_j}{P_{d_j}} \text{ for } i,j \in \{1,...,N\},$$

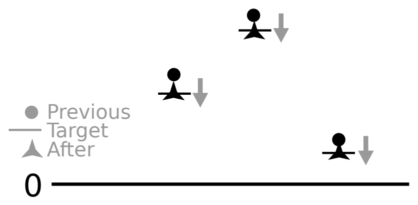


Fig. 4. Graphical interpretation of the frequency scaling effect, 3 tasks are brought to their target performance

where s_i, P_{d_i} and c_i represent the target share, the desired performance and the estimated behavior constant for an application i, respectively. When minimizing the percentual difference of all running applications, *i.e.*, $1-P_i/P_{d_i}$=$1-c_i s_i/P_{d_i}$, the *Share calculation* step 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 (2) is accompanied by two additional restrictions, which imply that the sum of all per-application shares s_i must be equal to one (*i.e.*, to guarantee utilization of all CPU resources) and that the percentual difference is equalized among the running applications (which guarantees that a second linear scaling can be applied in the next step and it implicitly assures non-negativity of per-application shares).

2) Frequency scaling: As previously referred, the first *Share calculation* step guarantees the equalization of the percentual difference between the predicted performance and the target performance for a set of running parallel applications. However, depending on the currently obtained value for percentual difference, the *Frequency scaling* step aims at adjusting the system operational frequency f, such that the percentual difference is further reduced without affecting fairness, thus allowing to achieve the target performance levels while optimizing the energy consumption. Figure 4 graphically represents this idea, where it can be observed that the performance targets (lines) for all three applications can be achieved by decreasing the operational frequency (see arrow in Fig. 4). As a result, the previously assessed performance levels (dots) are scaled such that newly obtained performance for running applications (triangles) closely matches the desired performance targets.

In the *Frequency scaling* step, the operational frequency f is adjusted by relying on a set of dynamically accessed application-architecture interaction parameters (c_i), the previously estimated shares (s_i) and desired performance (P_{d_i}) for all running parallel applications. In detail, the calculation of the new frequency level (f_n) is conducted by considering the above-referred parameters and the previous operating frequency f_o at which they are obtained, such that:

$$f_n = f_o \frac{P_{d_i}}{c_i s_i}. \quad (3)$$

Since the platform where the proposed system was evaluated only allows setting the operational frequency from a predefined range of discrete values, the solution obtained from (3) is further refined by rounding the newly calculated f_n frequency to the nearest value available in the system. Furthermore, when considering heterogeneous systems composed of a set of different cores (which are typically organized in clusters), multiple application-specific performance parameters ($c_i^{coreA}, c_i^{coreB}, \cdots$) can be considered, one per core type, and adjusted in real-time. To allow application migration to a different core, one must evaluate the expected performance drop/increase on the other core, by assuming an application specific relation $r_i^{A/B} = c_i^{coreA}/c_i^{coreB}$ between performance parameters.

3) Share conversion: In order to practically apply the previously calculated s_i shares (from the *Share calculation* step) to all running applications, they must be first translated to the CFS nice values. However, by directly applying (1) (see Section II), the translation between share and nice values may be practically unfeasible. In order to derive a set of nice n_i values for each running application, the *Share conversion* step adopts the following procedure:

$$\frac{s_{i-1}}{s_i} = \frac{1.25^{n_i}}{1.25^{n_{i-1}}} \Leftrightarrow n_i - n_{i-1} = \frac{log(\frac{s_{i-1}}{s_i})}{log(1.25)}. \quad (4)$$

As it can be observed from (4), the *Share conversion* step considers the ratio of share values across different applications, in order to calculate the n_i nice value for each application i. An additional constraint is provided by ensuring that the nice value of the highest priority task is as close as possible to the nice level of 0 (*i.e.*, system default). Given the restriction that the nice values must belong to a predefined integer interval [-20, 19], the obtained values are further rounded to the nearest integer values and scaled.

IV. Experimental results

To validate the proposed performance-aware task management system, an extensive experimental evaluation was performed in an Odroid-XU+E development board. For this, a brief overview of the experimental platform and considered real-world applications is presented, as well as a detailed evaluation of the proposed method.

A. Experimental platform

To experimentally assess the performance and energy savings provided by the proposed controlling mechanism, we performed the evaluation using a state-of-the-art Odroid-XU+E development board with a Samsung Exynos 5410 processor and 2GB of RAM memory. The platform is running the Linux Operating System Ubuntu with a custom 3.4 kernel by the manufacturer, Hardkernel. This Exynos processor is a 28nm implementation of the ARM big.LITTLE architecture [2] composed of 4 Cortex A15 (big) cores and 4 Cortex A7 (little) cores. While each A15 core is an out-of-order microprocessor capable of issuing up to three instructions per clock cycle, the A7 cores use an in-order execution model with a throughput of 2 issued instructions per clock cycle. As a result, by allowing the migration of threads from one core type to the other, the big.LITTLE architecture allows achieving the trade-offs between the high performance of the big A15 cores and reduced energy consumption of little A7 cores.

While the big.LITTLE architecture is developed to provide multiple migration models, the used evaluation board is limited to the cluster migration model [2]. As a consequence only one set of cores can be active at a given time (*i.e.*, either all running threads execute on the A15 cores, or they all execute on the A7 cores). The cluster migration model is built as an extension of the DVFS driver, thus introducing control over the heterogeneity on the system. As a result, two virtual frequency ranges are defined: *i*) a virtual range of [250;600] MHz where all A7 cores are active and all A15 cores are inactive; and *ii*) a virtual range of [600;1600] MHz where only the A15 cores are active. The correspondence between the virtual and physical operating frequencies is as follows: when the A7 cores are active, the physical operating frequency is twice the virtual one; when the A15 cores are active, there is a direct correspondence between virtual and physical operating frequencies.

To provide the means for assessing the energy-efficiency, the Odroid-XU+E development board includes a set of current/voltage sensors to measure the individual power consumption of the big A15 cores and the little A7 cores, as well as for the other system resources. At the software level, a set of facilities are provided to enable reading and converting the sensors readings into power consumption values. Furthermore, by applying periodic sampling over the power consumption it is possible to estimate the energy required to execute any single or multi-threaded application.

B. Benchmark characterization

To show the benefits of the proposed system, a set of multi-threaded programs from the PARSEC benchmark suite [12] was used and adapted to report the corresponding performance. For this, the Application Heartbeats [13] framework was used, which is a generic interface that facilitates the description of both the target and the observed performance specially for iterative single- or multi-threaded applications. To use this interface the selected benchmarks are modified in order to generate a heartbeat at specific representative sections of the application. As an example, by inserting a heartbeat at the end of each iteration one can report the pace at which iterations are executed. Table I summarizes the benchmarks used for assessing the benefits of the proposed system, all running 4 simultaneous threads, and the heartbeat locations. Since application performance can vary with the input sets, we further present, in the same table, the average application heartbeat (HB) rate when the system is running at the highest attainable performance, *i.e.*, when the application has no external interference from other applications and it is executing on the big A15 cores, while operating at 1.6 GHz. It is important to note that this controller will mainly benefit real time applications whose performance target can be set either by physical or human associated parameters (*e.g.*, frame rate).

TABLE I. List of programs used for testing purposes

Test	Input set	HB location	Max. Performance
Fluidanimate	PARSEC's in_300k.fluid Frames: 150	Every frame	2.8 HB/s
Swaptions	Swaptions: 128 Simulations: 400000	Every swaption	1 HB/s
x264	sintel_trailer_2k_480p24	Every 5 frames	4.6 HB/s
Blackscholes	PARSEC's in_10M.txt	Every 500000 options	7.7 HB/s

C. Task Management of Applications

The proposed task management system was implemented in the system and optimized such as to reduce the performance impact to the overall execution. Furthermore, since the Odroid-XU+E updates its power readings at a significantly low frequency, the system was configured to periodically wake-up and automatically apply nice and frequency levels (via system calls) at each 1 s time interval. However, it should be noticed that, since the controller imposes a low overhead to the system (typically less than 1 ms), a higher wake-up frequency could be easily used, thus providing higher control accuracy. Finally, other system parameters were also set, namely related to the feedback loop and error compensation. The maximum deviation from the application specified target was set to 50%, in order not to strangle the current application or the execution of other concurrent processes. Furthermore, the second limit was set to restrict the amount of the cumulative error per iteration (5 %), in order to avoid oscillating behaviors. These parameters are summarized on Table II.

TABLE II. Controller parameters

Parameter	Value
Controller sampling rate	1 s
Power sampling rate	1 s
Allowed deviation from target	50 %
Maximum performance scaling per iteration	5%
Controller execution time	< 3 ms

To validate the proposed method, we evaluate the experimentally achieved benefits by comparing them with the execution in a standard system (without the proposed controller being active). We specifically consider: *i*) the relative error between the target and observed performance (i.e., the percentual difference); and *ii*) the energy consumed during the execution of a set of considered applications. Table III presents the obtained experimental results when different combinations of the previously mentioned multi-threaded PARSEC benchmarks were simultaneously run according to the predefined target performance levels (in heartbeats per second, HB/s). Since the aim is to achieve realistic performance levels these targets were chosen close to the applications' typical performance. It should be noticed that, while all applications were set to start executing at the same time, they have different initialization phases. As a result, while some applications start reporting their HB rate sooner, the others may take considerably longer to make the first HB report. As such, the performance results presented in Table III (with and without the proposed controller in the system) refer to the time intervals where all considered applications are reporting their HBs.

As it can be concluded by analyzing the values presented in Table III, the proposed task management mechanism is capable of providing substantial benefits in terms of the relative performance difference between the achieved and the targeted performance levels (compare columns 4 and 7 marked with "error" in Table III). The advantage of the proposed system is even more evident when all four applications are simultaneously executed in the system. In this case, it is possible to observe a significant decrease ($16\times$) in the performance relative error. In particular, it can be observed that, with the controller, there is an almost exact match between the target and observed performance levels for the Fluidanimate, Blackscholes, and Swaptions benchmarks. There is however a slight error for the x264 application, which is mostly because its performance rapidly changes during the execution (*e.g.*,

TABLE III. RESULT SUMMARY, WITH PERFORMANCE AND ENERGY DATA FOR THE SETS OF TESTS USED

Benchmark settings		Without an active controller			With the proposed controller			
Benchmarks	**Target Performance** P_T [HB/s]	**Observed Performance** P_{O_0} [HB/s]	**Perfomance Relative Error** $\sum_{apps}\left(1-\frac{P_{O_0}}{P_T}\right)^2$	**Energy Consumption** E_0 [J]	**Observed Performance** P_{O_1} [HB/s]	**Perfomance Relative Error** $\sum_{apps}\left(1-\frac{P_{O_1}}{P_T}\right)^2$	**Energy Consumption** E_1 [J]	**Energy Savings** $\frac{E_0-E_1}{E_1}\times 100\%$
Fluidanimate (FA) *Swaptions (S)*	1.2 0.4	1.103 0.477	0.044	1116	1.169 0.429	0.006	1032	8.1 %
Blackscholes (B) *Swaptions (S)*	3.0 0.6	3.367 0.529	0.029	2022	2.849 0.625	0.004	2406	-16.0 %
Fluidanimate (FA) *x264 (x264)*	1.1 2.1	1.981 1.589	0.701	634	1.198 3.069	0.221	464	36.6 %
Fluidanimate (FA) *Swaptions (S)* *x264 (x264)*	0.8 0.3 1.4	1.006 0.447 0.268	0.961	1318	0.790 0.296 1.845	0.101	1178	11.9 %
Fluidanimate (FA) *Blackscholes (B)* *Swaptions (S)* *x264 (x264)*	0.6 1.0 0.2 1.0	0.796 2.223 0.372 0.319	2.801	2701	0.600 1.079 0.251 1.311	0.168	1805	49.6 %

by switching between intra and inter frames) and it highly dependents on the content of the input video sequence.

Figure 5 provides further details regarding the execution of a set of considered benchmarks by reporting the achieved uncontrolled (left side) and the controlled (right side) performance at different execution time intervals. For uncontrolled data set, Fluidanimate (FA) presents the earliest finish followed by Swaptions (S) and x264 (x264). Although the obtained per-application performance is irregular, it is clearly evident that the target performance (represented with the dotted line) is not achieved for Fluidanimate and Swaptions (real performance is higher than target) and x264 (lower than target). Also, it can be noticed that, as soon as one application completes, the remaining applications experience a performance boost which may result in delivering performance that is even further away from the target (as is the case of Swaptions).

On the other hand, the right side of Fig. 5 presents the achieved performance levels for the same set of benchmarks when the runtime controller is enabled. As it can be observed, their realistically achieved average performance is much closer to target (as also reported in Table III). This means that the proposed method was capable of optimizing the run of each parallel benchmark, such that the target performance is achieved for all tests simultaneously. The presented relative performance values in Table III also confirm this behavior, where a significant reduction can be observed for different sets of three and four simultaneously run benchmarks, i.e., the reduction ranges from 0.961 to 0.101 and from 2.801 to 0.168, respectively. As for the applications with rapid performance changes (such as x264), the proposed controller also incorporates the performance compensation mechanism in order to guarantee that the expected performance targets are achieved at the level of the overall application execution.

D. Energy Savings

As previously referred, controlling the relative performance difference and operating frequency opens the possibility of reducing the energy consumption of running (parallel) applications. In fact, as presented in Table III, significant energy savings (up to 49%) can be observed for different sets of considered benchmarks. However, since the system may automatically lower down the frequency in thermal emergency scenarios, setting the performance target limits to very high values might also result in an increased energy consumption. For example, in the case of B+S benchmark set in Table III, with the controller enabled the system was able to sustain maximum frequency for a longer period, achieving the expected performance level, but at a cost of increasing the energy consumption.

Figure 6 shows the power consumption and virtual runtime frequency when executing FA+S+x264 benchmark set. As it can be observed, in contrast to the execution without controller, reaching the target performance levels with the proposed method does not always require execution at the maximum frequency. This allows achieving significant energy savings, since during certain application phases the execution was even migrated from the A15 to the A7 cluster (see $f\leq$600MHz). In this particular case, the execution with the proposed controller was capable of attaining the energy consumption reduction of about 11.9% (on average) when directly compared with the execution that relies on default CFS and DVFS decisions (see Table III). However, when all evaluated benchmark sets are considered, the proposed method may provide even higher energy savings, namely up to 49%.

V. CONCLUSIONS

In this paper an adaptive performance-aware task management and frequency scaling method is proposed for modern heterogenous embedded systems. This mechanism relies on online performance monitoring to explicitly capture the runtime behavior of multiple parallel applications running on the underlying architecture. Based on the monitored application-specific parameters, the proposed method provides the decisions regarding the allocation of shared system resources, such that the target performance levels are achieved, as well as to guarantee the performance fairness among the running applications. The proposed mechanism also relies on DVFS to manage the system energy-efficiency levels and to further augment the scope in which performance can be scaled such that energy consumption savings are achieved. Furthermore, the proposed lightweight method allows amending the heterogeneous system's task scheduling, resource utilization and DVFS decisions, coupled with cluster migration, without the need to perform any kernel level modifications.

The experimental results show that a better performance fitting is achieved with the proposed method when several real-world parallel applications were simultaneously executed in the state-of-the-art embedded platforms. In particular, significant

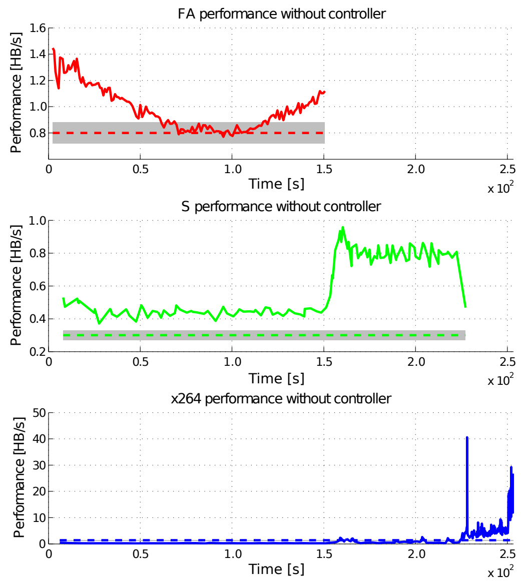
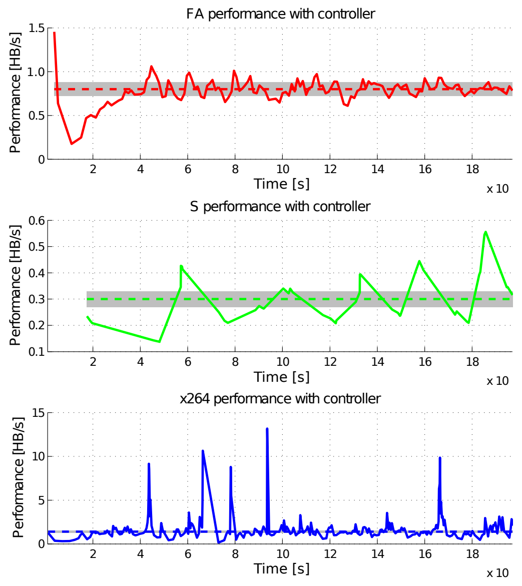


Fig. 5. Application performance during the simultaneous execution of Fluidanimate, Swaptions and x264, the dashed line represents the target while grey bars present a 20% deviation area from target



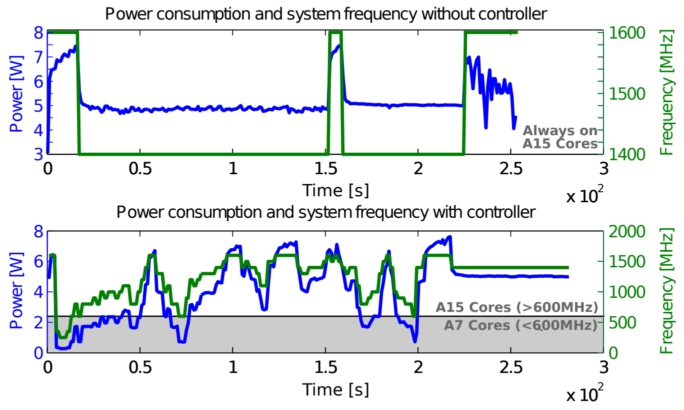


Fig. 6. Frequency level and energy consumed during the simultaneous execution of Fluidanimate, Swaptions and x264

reductions in the relative difference between the target and realistically achievable performance were attained when four different benchmarks from the standard PARSEC suite were simultaneously executed, *i.e.*, the relative performance error was reduced from 2.801 to 0.168, a 16× drop. The proposed controller also provides the additional energy savings at the level of the overall parallel application execution, thus allowing to achieve up to 49% reduction in the overall energy consumption for the presented test cases.

Acknowledgment

This work was supported by national funds through FCT – Fundação para a Ciência e a Tecnologia, under projects PEst-OE/EEI/LA0021/2013, PTDC/EEI-ELC/3152/2012 and PTDC/EEA-ELC/117329/2010.

References

[1] "Android, the world's most popular mobile platform," http://developer.android.com/about/index.html, accessed:17/06/2014.

[2] P. Greenhalgh, "Big.little processing with arm Cortex-A15 & Cortex-A7," *ARM White Paper*, 2011.

[3] "Qualcomm snapdragon 810," http://www.qualcomm.com/snapdragon/processors/810, accessed:18/06/2014.

[4] H. Esmaeilzadeh, E. Blem, R. St.Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *38th Annual International Symposium on Computer Architecture (ISCA)*, June 2011, pp. 365–376.

[5] "CFS Scheduler," https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt, accessed:17/06/2014.

[6] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi, "An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget," in *39th IEEE/ACM International Symposium on Microarchitecture*, 2006, pp. 347–358.

[7] X. Wang, K. Ma, and Y. Wang, "Adaptive power control with online model estimation for chip multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 10, pp. 1681–1696, 2011.

[8] T. S. Muthukaruppan, M. Pricopi, V. Venkataramani, T. Mitra, and S. Vishin, "Hierarchical power management for asymmetric multi-core in dark silicon era," in *50th Annual Design Automation Conference*, 2013, p. 174.

[9] T. Somu Muthukaruppan, A. Pathania, and T. Mitra, "Price theory based power management for heterogeneous multi-cores," in *19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014, pp. 161–176.

[10] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, "Scheduling heterogeneous multi-cores through performance impact estimation (PIE)," in *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3, 2012, pp. 213–224.

[11] A. Schranzhofer, J.-J. Chen, and L. Thiele, "Dynamic Power-Aware Mapping of Applications onto Heterogeneous MPSoC Platforms," *IEEE Transactions on Industrial Informatics*, vol. 6, no. 4, pp. 692–707, 2010.

[12] C. Bienia and K. Li, *Benchmarking modern multiprocessors*. Princeton University USA, 2011.

[13] H. Hoffmann, J. Eastep, M. D. Santambrogio, J. E. Miller, and A. Agarwal, "Application Heartbeats: A Generic Interface for Specifying Program Performance and Goals in Autonomous Computing Environments," in *7th International Conference on Autonomic Computing*, New York, NY, USA, 2010, pp. 79–88.