Performance Monitor Based Power Management for big.LITTLE Platforms

Simon Holmbacka, Sébastien Lafond, Johan Lilius
Department of Information Technologies, Åbo Akademi University
20530 Turku, Finland
firstname.lastname@abo.fi

ABSTRACT

Recently new heterogeneous computing architectures, coupling low-power low-end cores with powerful, power-hungry cores, appeared on the market. From a power management point of view, and compared to traditional homogeneous multi-core architectures, such architectures provide one more variable: the core type to map applications on. At the same time conventional power managers drives the DVFS mechanism based on the notion of workload. This means that as long as the CPU is capable of executing work, a workload increase will result in a frequency increase. In practice this results in a Race-to-Idle execution which mostly uses high clock frequencies. In this work we propose a performance monitor based power manager for cluster switched ARM big.LITTLE architectures. The proposed power manager allocates resources based on application performance rather than workload levels, which allows the hardware to adapt closer to software requirements. The presented power manager is capable of saving up to 57% of energy with the addition of one line of c-code in legacy applications.

1. INTRODUCTION

The big.LITTLE architecture [1] using one cluster of high performance cores and one cluster of energy efficient cores is becoming popular in mobile devices such as mobile phones and tablets. The big cores are designed for high performance calculations using high clock frequencies, deep pipelines, large caches and out-of-order execution. In case high performance is not required, the system can shut down the big cores and activate the energy efficient LITTLE cores. The LITTLE cores in e.g. the Exynos 5410 SoC utilizes four A7 cores with short pipelines, small caches and in-order execution, which reduces the power dissipation significantly. The core selection – in this SoC which we consider – is based on cluster switching [12], which enables either the cluster of A7 or A15 cores, but not both types simultaneously. The core types are automatically switched from LITTLE to big as the clock frequency of the CPU is increased beyond a certain threshold.

While the hardware shows a great potential in energy savings, the software is usually unable to utilize such an architecture efficiently. Optimally, the system should not allocate more CPU resources than what the application requires. This aim can be achieved with a power manager monitoring the system and adjusting the clock frequency accordingly. Currently, the power managers controlling CPU resources use only *workload* as the metric for resource allocation [2]. Workload is defined in Linux as the ratio between CPU execution and CPU idle states for a given time window. It is, however, a poor metric for controlling CPU resources because it does not describe application *performance*. When applying high workload on a CPU, power managers will increase the clock frequency as long as the workload remains high, and since applications execute as long as work is available for execution, the workload will remain high for the whole execution. This leads to Race-to-Idle [11] conditions, in which the CPU is executing the work as fast as possible in order to reach the idle state. Consequently, by using high clock frequencies, it leads to unnecessary execution on the big cores after which the system idles until more work is available.

In this work we investigate whether a power manager driven by performance monitoring in the applications is able to more efficiently manage big.LITTLE architectures. The CPU allocation is directly based on application performance monitored by a new kind of power manager. We use a big.LITTLE power model created from real-world experiments to obtain the most power efficient execution at run-time. The model is able to predict the optimal clock frequency to satisfy the performance requirements of the applications.

We evaluate the system with typical legacy applications, and up to 57% of energy savings have been obtained with executions on real hardware using an unmodified Linux OS.

2. RELATED WORK

Power optimization of DVFS in multi-core systems has been extensively studied in the past [5, 10, 15]. A critical difference between traditional multi-cores and big.LITTLE multi-cores is the significant power reduction potential of executing tasks on the LITTLE cores. A utilization-aware load balancer for big.LITTLE system was presented in [9]. The balancer implemented a processor utilization estimator for determining the most optimal clock frequency for a given set of tasks without loosing performance. We argue that a utilization-based metric alone is not sufficient to efficiently control big.LITTLE power management. Instead we focus on performance monitoring *in the applications* in order to allocate the resources directly based on software demands.

The work in [3] presents the partition of real-time tasks onto heterogeneous cores such that energy is minimized by an optimal load distribution. The scheduling decisions were based on an analytical power model and an energy model based on

the load distribution of tasks. Minimum energy consumption was calculated by modeling tasks executing on cores with given clock frequencies. Our work is focused on non real-time or soft real-time tasks without a given deadlines but with performance requirements in the applications. The power model we rely on is, in contrast to [3], derived from real-world experiments and not from analytical bottom-up models.

C-3PO [13] is a power manager used to maximize performance under power constraints and minimize peak power to reduce energy consumption. Applications are given a power budget, which is used for resource allocation in form of clock frequency and the number of cores. Orthogonally, we aim to minimize power under performance constraints. This means that our notion of constraints relate to the execution of applications rather than the power dissipation of the hardware. We further aim to implement this practice on big.LITTLE CPUs on which power is significantly reduced as long as the execution can take place on the LITTLE cores.

3. EXECUTION MODEL

The consequence of using workload-based power management is in often an execution model called "Race-to-Idle" [11]. Its behavior is to execute a job as fast as possible in order for the CPU to minimize the execution time and to maximize the idle time. The popularity of this execution model relates to simple programming; the programmer specifies only the program functionality, and the OS scales the clock frequency indirectly according to the workload.

Ondemand power management. Clock frequency in Linux based systems is driven by a kernel module called *frequency governor*. A frequency governor is monitoring the workload of the system and adjusts the clock frequency according to the policy for the governor in question. A number of different governors can be installed on a system, but usually the default governor is called *Ondemand* [14]. The Ondemand governor monitors an *upthreshold* value after which the workload is considered "too high". As the threshold value is reached, the governor switches the clock frequency automatically to the highest value (as illustrated in Figure 1). After the maximum value is reached, the governor decreases the clock frequency step-wise in order to find the most suitable frequency.

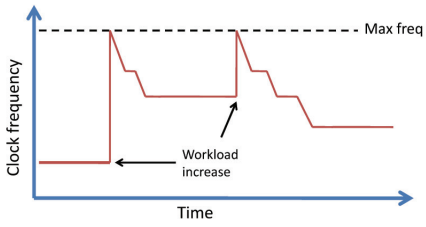

Figure 1: Illustration of the clock frequency scaling strategy of the Ondemand governor

The strategy of the governor was designed to rapidly respond to changes in workload without performance penalty, and to save power by step-wise scaling down. However, this strategy **a)** forces the CPU to always execute some part of the workload on the maximum clock frequency if the threshold is reached and **b)** for Race-to-Idle conditions, most of the workload will execute on the maximum (or a high) frequency since the workload will remain high as long as jobs are available for execution. For big.LITTLE systems, this strategy is contradictory to the intentions of the hardware since much time is spent on executing on high frequencies (with big cores) even if the system has significant idle time.

QoS driven power management. We argue that workload alone is not a sufficient metric to efficiently control big.LITTLE systems, instead the system should measure application performance for driving the power management.

As example illustrated in Figure 2, a video decoder decodes a number of frames and puts them in a display buffer. When the buffer is full, the decoder waits until the buffer is emptied. Since the output is usually determined by a fixed framerate, e.g. 25 frames per seconds (fps), the decoder is only required to decode frames at the same rate as the output display is using. Part (A) illustrates the Race-to-Idle strategy in which the CPU executes on maximum clock frequency for half a time window, after which it idles on the lowest clock frequency. The decoding process is hence producing 50 fps while the required rate would be 25 fps. Even though the power dissipation of the CPU is low on the idle part, the decoding part uses only the big cores even if the LITTLE cores would be sufficient when stretching the execution.

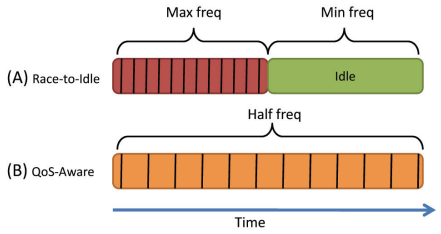

Figure 2: Illustration of (A) Race-to-Idle strategy and (B) QoS-Aware strategy

To create a system controlled by software requirements, we implemented a framework [6] to inject application specific performance directly into a new type of power manager (further explained in [7]). The power manager monitors the *performance* of the applications to determine the magnitude of the CPU resource allocation.

The power manager supports an execution strategy called *QoS-Aware*. The strategy is illustrated in Figure 2 (B), in which the execution time is stretched out over the whole time window. By executing only at the *required* clock frequency, the LITTLE cores are utilized as long as the performance is sufficient. The power manager is re-evaluating the performance measurements periodically, and the effort of the programmer is to suitably assist the power manager with the *performance* parameter. Practically, one line of c-code must be added to the applications:
`fmonitor(<performance>);`. This function calls the power management library and provides the run-time information, for example the current decoding framerate (fps).

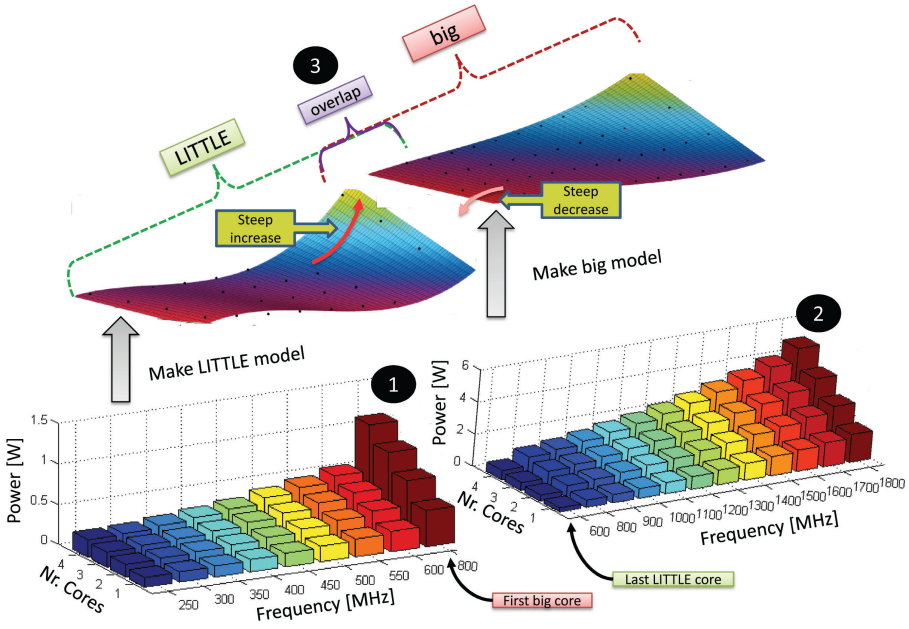

Figure 3: Creation of big.LITTLE power model. Separate reference measurements on the LITTLE and the big cores are used to generate a mathematical model which overlaps in the [600 800] MHz range.

4. BIG.LITTLE POWER MODEL

The power manager uses a power model to determine the increase in power by increasing/decreasing the clock frequency one step. The performance values given by the `fmonitor` library call are compared against a power model in order for the power manager to determine the power output caused by the CPU allocation.

As an application demands more resources, the aim is to chose a frequency which results in minimum power increase and sufficient performance increase. In contrast to our previous work on homogeneous systems [7], we require a dynamic model for describing the big.LITTLE architecture in which two types of cores can be used. As the model is constructed by mathematical expressions including architecture based parameters, the power manager must be able to adjust the dynamic parameters based on the core type currently in use. Since we use a big.LITTLE system with cluster switching [12], we consider only one type of core active at one time.

Similarly to [7], we stressed the physical system to maximum CPU load with the *stress* benchmark under Linux. Under full load we increased the number of cores and the clock frequency step-wise until all configurations were obtained. The power was physically measured after each step by reading internal power measurement registers in the chip.

By using the real-world measurements, we transformed the results into two mathematical functions using plane fitting methods [8] into a third degree polynomial[1]: $P(q,c) = p_{00} + p_{10}q + p_{01}c + p_{20}q^2 + p_{11}qc + p_{30}q^3 + p_{21}q^2c$ where P is the power, q is the clock frequency and c is the number of cores. With traditional non-linear optimization methods [4], we can minimize the cost (power) by selecting the optimal clock frequency for a given application based on performance requirements and the number of cores in use.

The studied architecture is a big.LITTLE configuration with two different types of cores, and the types are selected based on the clock frequency transition between 600 MHz and 800 MHz. We therefore created two separate power models for each core type based on the *stress* measurements. Figure 3 **(1)** shows the LITTLE measurements from 250 MHz to 600 MHz and **(2)** the big measurements from 800 MHz to 1800 MHz.

Because the aim is to keep the system executing on the LITTLE cores as much as possible, we overlapped the LITTLE and the big power models by including the lowest frequency of big cores in the LITTLE measurements (seen in Figure 3 **(1)**). This generates a steep cost increase when transitioning from the LITTLE to the big model (Figure 3 **(3)**), and pushes the optimizer to avoid the big cores if possible. Similarly, the highest clock frequency setting (600 MHz) of the LITTLE cores was included in the big-core measurement profile (seen in Figure 3 **(2)**), which drives the optimizer to descend to this setting if performance is sufficient. The result is a surface defined by the previously described third degree polynomial with one step overlapping (seen in Figure 3 **(3)**). The selected model (and p_{xy} parameters defined in the polynomial) is chosen based on the current core type in use, which can be monitored with Linux `sysfs`.

5. PRIORITY WEIGHT INTERFACE

As long as only one application has exclusive control over the power manager, no control conflicts can occur. However, as soon as several applications compete over the same resources, two applications could output conflicting execution conditions to the power manager. Conflicting information can result in wrong control settings for both appli-

[1]Further details in [7]

cations, instability in the resource allocation or diverging control output favoring one of the applications.

In order to increase the predictability of the control output which allocates CPU resources to the applications, the notion of priority weights in the applications was included in case several applications input conflicting information. The basic notation behind CPU allocation is the measured performance P_n of application n. P_n is compared to a user defined *setpoint* S_n, which marks the *desired* performance of application n. In case $P_n < S_n$, the application is given a positive *error value* E_n by the power manager, which signals for increased resource allocation. Similarly, in case $P_n > S_n$ the application is given a negative error value, which corresponds to resource waste and resource deallocation.

The magnitude of the error values determines the amount of resources to allocate/deallocate. With no notion of priority, the difference between setpoint and measured performance alone determines the error. By manipulating the magnitude of the error values, it is hence possible to alter the priority weight of an application error E_n, and increase the influence of important applications.

Application priorities in the Linux kernel are set by manipulating the run-time information of the tasks. The execution time of a task is simply replaced by a virtual time, which is manipulated according to priority weights. In other words, a high priority task will receive a slowly incrementing virtual time, which means that the scheduler will keep the task under execution for a longer "real" time.

We applied the same concept by replacing the error values with virtual errors vE_n to increase the influence of important tasks. The virtual errors of the applications were determined by sending all errors E_n and their respective priorities R_n to an error transformation function. Listing 1 outlines this procedure: **(2)** The system is monitoring all applications and calculate their respective error values based on the performance, **(3)** error values are replaced with virtual errors based on priorities, **(4)** the virtual errors are sent to the power manager which allocates the resources. Listing 2 shows the algorithm: **(1–4)** All applications are iterated over and a sum of all weights (priorities) for the current applications is calculated, **(5–6)** for each application, the virtual error is determined as the error multiplied with a weight determined by the priority in relation to all other applications (`weightsum`).

```
1  LoopForever{
2    <Apps><Errors><Priorities> = getMeasurements()
3    <vErrors> = veTrans(<Apps><Errors><Priorities>)
4    PowerManagement(<vErrors>)
5  }
```

Listing 1: Pseudo code for measurement procedure

The weight values were extracted from the Linux kernel source and are shown in Table 1. There are currently forty different priority levels defined by the weights where a higher weight means higher priority.

```
1  for(j=0; j<num_apps; j++){
2    weightsum = 0.0;
3    for(i=0; i<num_apps; i++){
4      weightsum += weights[priorities[i]];
5    }
6    verrors[j] = 2*errors[j]*weights[priorities[j]]/
         weightsum; }
```

Listing 2: c-code for generating virtual errors

Table 1: Weight values

15	18	23	29	36	45	56
70	87	110	137	172	215	272
335	423	526	655	820	1024	1277
1586	1991	2501	3121	3906	4904	6100
7620	9548	11916	14949	18705	23254	29154
36291	46273	56483	71755	88761		

6. EXPERIMENTAL RESULTS

For evaluation we required a benchmark with variable load, yet repeatable and multi threaded.

We chose video decoding using Mplayer[2] as basis for the evaluation. Further, we added a Facedetection application sharing the resources with Mplayer to create a mixed-priority scenario. Both applications were run with the Ondemand governor and with our optimized power manager under Linux 3.7.0. Our test platform was the 28 nm octa-core Exynos 5410 SoC based on the big.LITTLE configuration with four ARM Cortex-A15 cores and four ARM Cortex-A7 cores.

Mplayer. The first experiment was set up to use only the Mplayer application. Mplayer was set to decode and play a 720p video for 5 minutes using the h.264 video codec. Since the playback is executed with a steady framerate of 25 fps, we added a QoS requirement of 30 fps on the decoder by using our power management library. This means that the decoding process is slightly faster than the playback in order to keep up with occasional buffer underruns.

Figure 4 (A) shows the power dissipation for using Ondemand with a power sample rate of 250 ms. The dark gray curve is the A15 power, the black curve is the A7 power and the light gray curve is the memory power. With the resource requirement for decoding the 720p video, the workload exceeds the threshold used by the governor. Because of the Race-to-Idle strategy, the system is forced to stress the CPU to decode the frames as fast as possible and the core type in use is mostly the big A15 even though the performance of using a lower clock frequency would be sufficient.

By regulating the system according to the application specific performance (fps) instead of the workload, the CPU is allowed to stretch the decoding window while the output framerate is still met. Instead of racing to idle, a clock frequency below the core transition limit (800 MHz) is used which allows the system to execute on the LITTLE A7 cores. With this strategy there is almost no idle time in the system, but the execution is performed more energy efficiently and the performance requirements are met. Figure 4 (B)

[2]http://www.mplayerhq.hu/

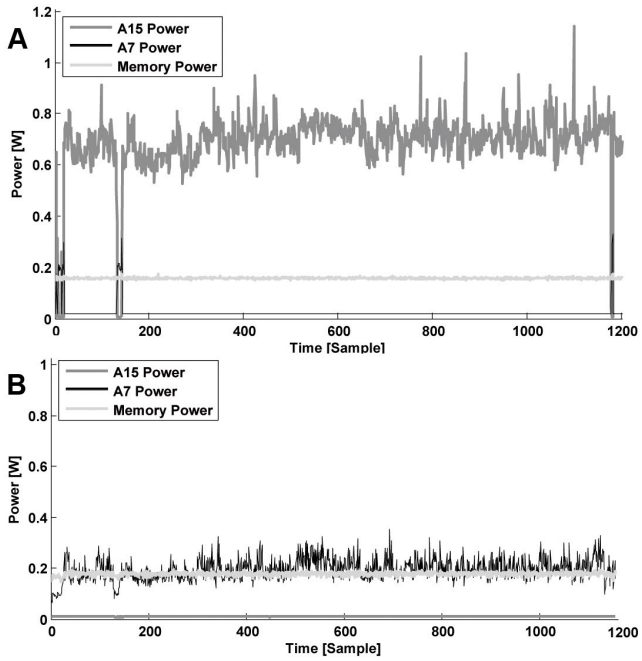

Figure 4: Power dissipation for Mplayer using (A) Ondemand (B) Optimizer

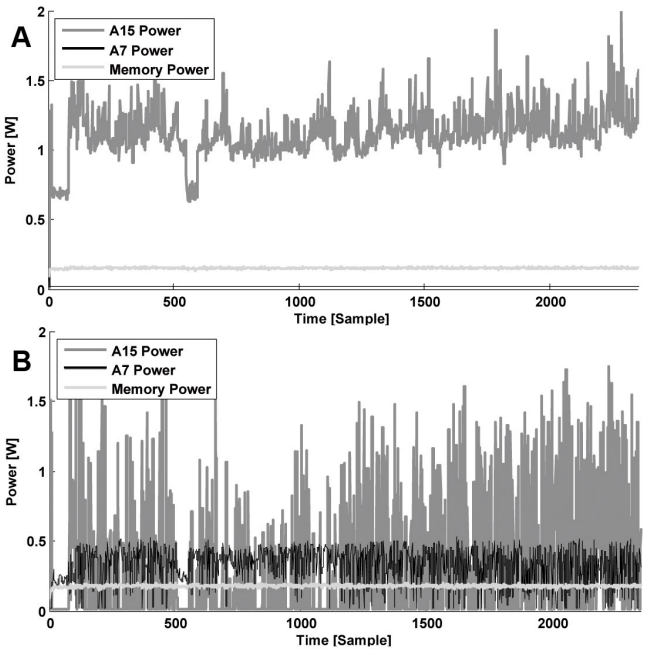

Figure 5: Power dissipation for Mplayer and Facedetection using (A) Ondemand (B) Optimized

shows the optimized execution in which the A7 cores are mostly used for processing the same video as in Figure 4 (A). The Ondemand governor consumed in total 103.96 Joules of energy while the optimized power manager consumed only 43.88 Joules, which is a reduction of 57% for executing the same amount of work with 0% performance degradation.

Mplayer + Facedetection. In the second evaluation we extended the use case to a mixed-priority scenario with several applications. Similarly to the previous evaluation we executed a 720p video with a required decoding rate of 30 fps. Furthermore, we added a Facedetection application used for video surveillance. The Facedetection application reads the input of a video stream, scans the current frame for the occurrence of one or more faces and draws a rectangle of the found face on the video stream. The QoS requirements added to this application was to scan 10 video frames per second for faces i.e. "10 Scans per Second (SPS)".

Since this application was used for surveillance, its performance was more critical than the video player. The priority for Facedetection was therefore set to 30 while Mplayer used a priority of 9. With a higher priority on Facedetection, it was expected for framedrops to occasionally occur in the video playback. We therefore executed Mplayer with parameters `-framedrop` and `-benchmark` in order to measure the number of dropped frames as well as the power.

Figure 5 (A) shows the power dissipation for using Ondemand and Figure 5 (B) for using the optimizer. Similarly to the Mplayer-only use case, the Race-to-Idle conditions of Ondemand forces a mostly high clock frequency and the workload is executed exclusively on the big A15 cores. The optimizer (in part (B) of Figure 5) shows a rather spiky output since the added Facedetection application occasionally requires more resources than what can be achieved on the LITTLE A7 cores. The system rushes to meet the performance requirements by temporarily using the A15 cores after which is it able to scale down to the A7 cores.

The mixed-application scenario occasionally imposes conflicting control signals based on the performance requirements. For example, while Mplayer is decoding very light frames and measures a "too high" framerate, Facedetection is under utilized and requires more resources. In order to verify the priority interface, we also plotted the scanrate for Facedetection during the whole experiment. Figure 6 (A) shows the scanrate for using Ondemand and (B) for using the optimizer. With a setpoint of 10 SPS we marked our acceptable lower and upper QoS limits for the application at 9 SPS and 11 SPS respectively. Since Ondemand is able to use the full power of the CPU all the time, it is expected to reach a more stable scanrate than the optimizer which can be seen in the figure. In case a better QoS is required using the optimizer, the user can either increase the performance setpoint to e.g. 12 SPS or increase the application priority with the cost of increased power dissipation.

Table 2 finally summarizes the mixed-scenario experiments. The optimized power manager was able to save roughly 40% of energy while imposing only a 1% QoS degradation on Mplayer and 6% QoS degradation on Facedetection compared to Ondemand.

Table 2: Energy (in Joules) and QoS (in %)

	Energy	QoS Mplayer	QoS Facedetection
Ondemand	334.3	100 (1 drop)	92 (52 late frames)
Optimized	201.5	99 (97 drops)	86 (108 late frames)

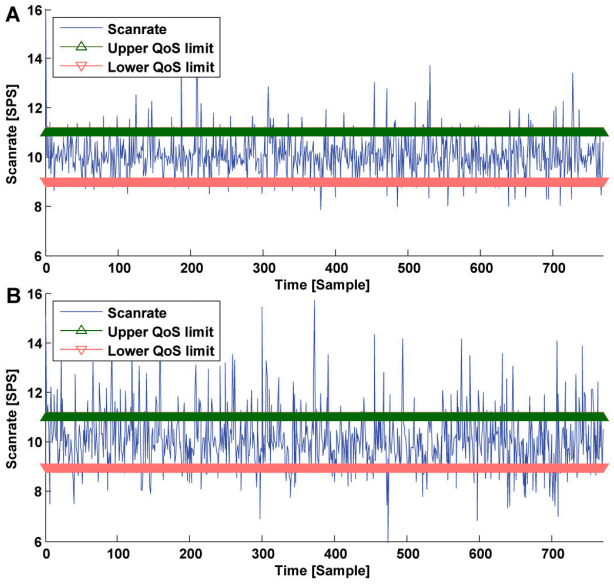

Figure 6: Scanrate and QoS for Facedetection using (A) Ondemand (B) Optimized

7. CONCLUSION

Workload alone is not a sufficient metric for driving power management in modern big.LITTLE systems. Since workload only expresses CPU utilization and not application performance, the execution is forced to Race-to-Idle as long as the workload remains high. By measuring the application performance and regulating the CPU allocation based on application requirements, the system is able to keep the execution of jobs on the energy efficient LITTLE cores for a longer time. We have presented a power manager utilizing a dynamic big.LITTLE power model for maximizing the LITTLE core usage. The usage is maximized by minimizing the idle time; allowing the system to execute on the lowest possible clock frequency without performance penalties. With an implemented library, applications can set performance requirements and input run-time information to influence the control decisions. Applications are further able to express their importance and the relation to CPU allocation in resource sharing scenarios involving several applications.

With real-world measurements using Linux running on big.LITTLE hardware we have obtained up to 57% of energy reduction for decoding typical HD videos with no performance degradation. Further on a mixed-priority scenario using one critical and one best effort application, we obtain energy savings up to 40% with minor QoS degradation compared to the default power management system. We plan to integrate the system into embedded devices such as mobile phones to increase the battery time when using typical every-day applications. We are also targeting *global task scheduling* systems in which both the big and the LITTLE cores are available at the same time.

8. REFERENCES

[1] ARM Corp. big.little processing witharm cortex-a15 & cortex-a7. `http://www.arm.com/files/downloads/big_LITTLE_Final_Final.pdf`, 2011.

[2] D. Brodowski. Cpu frequency and voltage scaling code in the linux(tm) kernel. `https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt`, 2013.

[3] A. Colin, A. Kandhalu, and R. Rajkumar. Energy-efficient allocation of real-time applications onto heterogeneous processors. In *RTCSA, 2014 IEEE 20th International Conference on*, pages 1–10, Aug 2014.

[4] P. E. Gill, W. Murray, Michael, and M. A. Saunders. Snopt: An sqp algorithm for large-scale constrained optimization. *SIAM Journal on Optimization*, 12:979–1006, 1997.

[5] M. Haque, H. Aydin, and D. Zhu. Energy-aware task replication to manage reliability for periodic real-time applications on multicore platforms. In *Green Computing Conference (IGCC), 2013 International*, pages 1–11, 2013.

[6] S. Holmbacka, D. Ågren, S. Lafond, and J. Lilius. Qos manager for energy efficient many-core operating systems. In *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, pages 318–322, 2013.

[7] S. Holmbacka, E. Nogues, M. Pelcat, S. Lafond, and J. Lilius. Energy efficiency and performance management of parallel dataflow applications. In A. Pinzari and A. Morawiec, editors, *The 2014 Conference on Design & Architectures for Signal & Image Processing*, pages 1 – 8, 2014.

[8] K. Iondry. *Iterative Methods for Optimization*. Society for Industrial and Applied Mathematics, 1999.

[9] M. Kim, K. Kim, J. Geraci, and S. Hong. Utilization-aware load balancing for the energy efficient operation of the big.little processor. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pages 1–4, March 2014.

[10] T. Rauber and G. Runger. Energy-aware execution of fork-join-based task parallelism. In *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2012 IEEE 20th International Symposium on*, pages 231–240, 2012.

[11] B. Rountree, D. K. Lownenthal, B. R. de Supinski, M. Schulz, V. W. Freeh, and T. Bletsch. Adagio: Making dvs practical for complex hpc applications. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 460–469, New York, NY, USA, 2009. ACM.

[12] Samsung Corp. Heterogeneous multi-processing solution of exynos 5 octa with arm big.little technology. `https://events.linuxfoundation.org/images/stories/slides/elc2013_poirier.pdf`, 2013.

[13] H. Sasaki, S. Imamura, and K. Inoue. Coordinated power-performance optimization in manycores. In *Parallel Architectures and Compilation Techniques (PACT), 2013 22nd International Conference on*, pages 51–61, 2013.

[14] V. P. A. Starikovskiy. The ondemand governor. In *Proceedings of theLinux Symposium*, 2006.

[15] I. Takouna, W. Dawoud, and C. Meinel. Accurate mutlicore processor power models for power-aware resource management. In *Dependable, Autonomic and Secure Computing (DASC), 2011 IEEE Ninth International Conference on*, pages 419–426, 2011.