Android Movie Player System Combined with Automatically Parallelized and Power Optimized Code by OSCAR Compiler

Bui Duc Binh† Tomohiro Hirano† Dominic Hillenbrand† Hiroki Mikami†
Keiji Kimura† Hironori Kasahara†

The emergence of multicore processors in smart devices promises higher performance and low power consumption. The parallelization of application enables us to improve the application performance; however, simultaneously utilizing many cores would drastically drain the device battery life. Therefore, power saving technology has become important. This paper shows a realtime video demonstration system for power reduction controlled by the OSCAR Automatic parallelization Compiler on ODROID-X2, an open Android development platform based on Samsung Exynos4412 Prime with four ARM Cortext-A9 cores. The demonstration results show that it can save 18.2% power consumption for MPEG-2 Decoder application and 56.6% power consumption for Optical Flow application by using 2 cores in both applications.

1. Introduction

Smart devices have been becoming the most popular and dominant devices in the electronic market. It is also known that such small hand-held devices are getting rapidly powerful and affordable as well. They are integrated with high performance processors, accelerated graphics processing unit, high resolution display, GPS and so on. These features have turned smart device in a complete work station which is able to compete with laptop as well as desktop computer.

However, in order to achieve such high performance, the hand-held size devices must be able to complete a large number of computations by powerful and sophisticated hardwares which are extremely power consuming. Moreover, the battery size in smart device is limited and not increased as fast as its hardware. Therefore, achieving higher performance in a longer time with a limited energy support has become a very important research issue.

Nowadays, most of commodity smart phone chips are multi-core chips. Moreover, it is also known that a system with more processors can provide better performance than a single core system. Therefore, the applications need to be parallelized in order to take advantage of many cores system. Along with parallelization of application, the power optimization is also required to overcome the battery life constraints.

The Android platform is the most used OS (Operating System) in smart phones with more than 70% in the market share; therefore it is important to focus on improving performance and power consumption in Android devices. Generally, Android applications are developed in Java language. It is possible to parallelize applications in Java as shown in [1]. However, it is also indicated by [1] and [2] that Android applications can be speed up by using Android NDK and JNI. Android NDK and JNI enable Android developer to use native code written in C or C++ which is much faster than Java at doing arithmetic operations. Another way of parallelizing applications is to parallelize them in native language such as C, then build shared library by NDK, finally exchange computed data with Java part through JNI. Android applications can be speed up two times by using Android NDK and parallelized native code.

Parallelization of application is a very effective way to benefit from a multi-core system, however, manually parallelizing a large program is very time consuming and moreover, most of the current applications were not developed with considerations about multi-core system as well as power optimization as a priority. There are some parallelizing compilers, such as OpenMP Compiler [3] and OSCAR compiler [4][5]. For all of these parallelizing compilers, OSCAR Compiler can realize not only application parallelization but also power

† Department of Computer Science and Engineering, Waseda University

ⓒ 2014 Information Processing Society of Japan

optimization [6][7]. [8] shows that by using OSCAR compiler, it can save 86.7% power consumption in case of using 3 cores compared to ordinary case of using 1 core with MPEG-2 Decoder application, and 86.5% power consumption in case of using 3 cores compared to ordinary case of using 1 core with Optical Flow application. The experiments in [8] were conducted on ODROID-X2 [9] board, an open Android development platform based on Samsung Exynos4412 Prime [10]. However, it only showed the execution results of binary files meaning that no realtime video displaying work was done while MPEG-2 Decoder and Optical Flow generate data that should be played on a display.

This paper introduces a full demonstration system of playing video and measuring power consumption simultaneously. In this paper, we show an efficient way of the collaboration between the Java UI thread and parallelized native C modules by core partitioning and thread binding to cores. We also realize a real-time video player system with low power optimization by utilizing per-frequency profiling result in addition to the previously proposed power optimization technique by the OSCAR compiler.

The rest of this paper is structured as follows. Section 2 introduces the power management on Android platform and section 3 introduces the OSCAR compiler. Section 4 and 5 explains the structure of the demonstration system. Section 6 shows the power consumption evaluation results and section 7 gives the conclusion of the paper.

2. Power Management on Android

This section shows a brief introduction about the power management on Android Kernel based on the Linux Kernel.

In the Linux operating system, one of the most effective ways to save power consumption is to use CPU DVFS (Dynamic Voltage and Frequency Scaling). This allows the working frequency to be scaled in the running time. The CPUFreq Governor decides the rules for adjusting frequencies. The governor could be a performance-oriented one which sets the CPU to use the highest supported frequency. It is called “performance”. In contrast, the power-oriented governor, named “powersave”, tries to execute all tasks with the lowest frequency, therefore, save energy as far as possible. In this experiment, we used “ondemand” governor, and “userspace” governor. The ondemand governor dynamically adjusts the working frequency based on the current workload while in userspace governor; users must specify a working frequency and the whole system will run at that frequency.

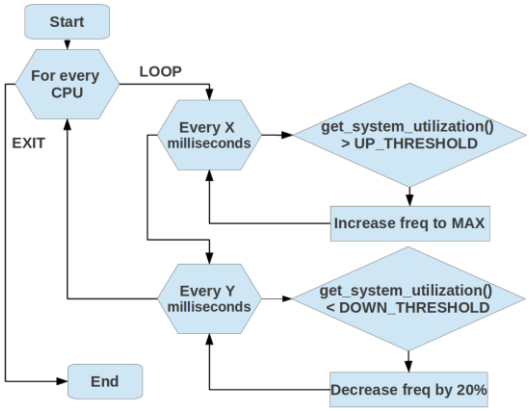

Figure 1: Original ondemand algorithm

In a Linux system, the ondemand governor is enabled by default. The ondemand governor changes CPU frequency based on CPU utilization. Fig. 1 shows the original ondemand power control algorithm [11]. Every X milliseconds, the system checks the current system utilization. If it is larger than the upper bound value, the system will set the working frequency to the maximum value. Likewise, every Y milliseconds, the system checks if the current utilization is smaller than the lower bound, the working frequency will be decreased by 20%. This process is repeated and applied for all available CPU(s). The ondemand governor is very suitable to periodical applications since the operating system can predict the proper frequency based on the previous workloads which are quite stable in case of periodical applications.

In userspace governor, the user changes the working frequency through sysfs interface. The desired frequency value can be applied by writing to sysfs filesystem such as /sys/devices/system/cpu/cpuX/cpufreq/scaling_set_speed

ⓒ 2014 Information Processing Society of Japan

3. OSCAR Compiler

The following briefly describes the OSCAR (Optimally Scheduled Advanced Multiprocessor) Compiler and OSCAR API, which are used for parallelization and power optimization of applications in this paper.

The compiler exploits three kinds of tasks called macro-tasks (MT) from a source program. Each MT can be a basic block, a loop or a function. In constraints of control dependencies and data dependencies, the parallelism among MTs is exploited by the compiler and the result is represented as a hierarchically defined Macro Task Graph (MTG) [4]. Then macro-tasks are scheduled to available processors.

Based on the result of tasks scheduling, the power optimization is applied. In order to save power consumption, OSCAR compiler manages to reduce the working frequency as well as exploit clock gating and power gating. In this paper, four levels of frequency namely HIGH, MID, LOW and VLOW are used [7].

For each macro task, the compiler checks if it can reduce the working frequency of that task given that the application performance is ensured. During the execution time, if there is any CPU which is not assigned with any task, that CPU would be forced to power gating or clock gating mode. With multimedia application, it is required the application to meet the displaying rate of, for example, 30 frames per second. OSCAR also considers this point and make sure that application can run at low frequency but still meet the deadline.

Finally, the parallelized and power optimized C or FORTRAN codes are generated with OSCAR API directives [6]. The results can be improved to be more precise and compatible with the target architecture by providing additional information such as number of cores, cache memory size in form of compiler options.

4. Video Player Demonstration System

The purpose of this paper is to show a demonstration system of automatically parallelized and power optimized realtime video application. This section describes the details of the simple video application we have developed.

Multimedia applications are very computation intensive, therefore, it is common to use fast C code in computing ,then send the result back to a Java thread and let it complete the remaining work, i.e.: displaying the data. Our video application is built based on this observation. The application is divided into two parts: Java part and arithmetic part. The former runs on UI (User Interface) thread which is assigned to CPU 0. This part is responsible for displaying computed data, doing garbage collection and other system related works while waiting for the results from arithmetic thread. Since we should avoid performing long running operations on the UI thread, it is necessary to create new threads and implement heavy jobs on them. By doing so, the UI is not blocked at the calculating time. All long running computations are done on the arithmetic part which is written in C. This part is parallelized, and optimized by the OSCAR compiler. In this experiment, we create new worker threads on different cores to take the advantage of the OSCAR compiler as well as multi-core architecture. This core partitioning can efficiently avoid the interference between Java part and arithmetic part, such as task migration and cache pollution.

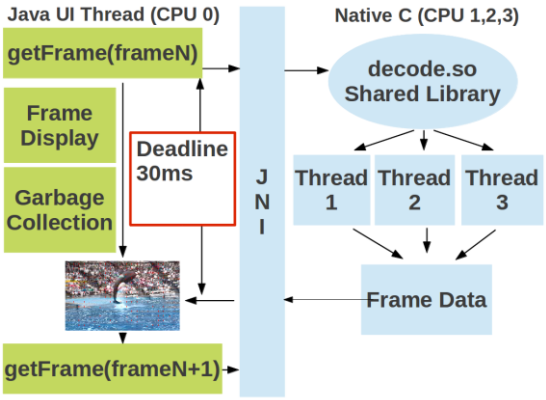

Figure 2: Simple video player model

Figure 2 shows the completed process to compute the data and display it in the device screen. Firstly, an UI Thread invokes a method to request the data of the frame N. This parameter N is passed as input of native method through JNI. Depending on how many cores the developer wants to utilize, it forks into one or two threads working simultaneously. The forked threads will process all arithmetic calculations and join after

ⓒ 2014 Information Processing Society of Japan

finishing all tasks. When all operations have done, the shared C library returns the result and passed them to Java thread through JNI interface. Finally the calculated result will be display to a monitor.

During the time of working on arithmetic computations, the frequency is scaled up and down according to the power optimization result by OSCAR compiler. Since C is a processor bound language, it is possible to programmatically adjust the working frequency at the native level by opening and writing to a specific sysfs. Meantime, the UI Thread on CPU 0 will take care of rendering, displaying one frame of the video, executing garbage collection and so on. This process is repeated until all frames are displayed.

One point should be noticed here is the JNI communication delay between Java part and arithmetic part. [12] shows that it takes about 0.15 microseconds to pass a string from native C library on to the application. Since the deadline of a multimedia application is 33 milliseconds, this delay is negligible.

5. Demonstration Board Setup

5.1. ODROID-X2 Board

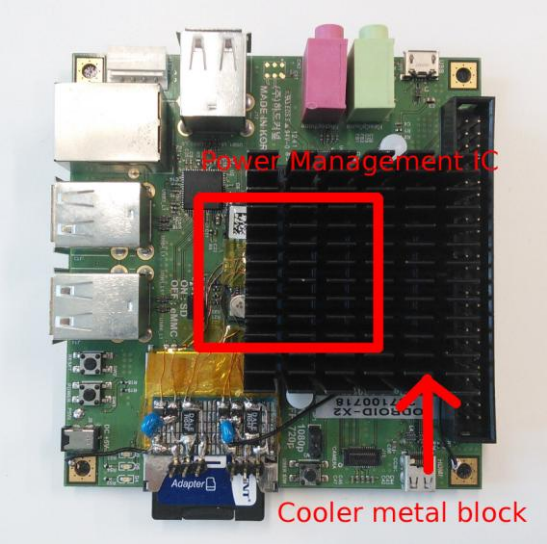

Figure 3: ODROID-X2 Board

In this experiment, the ODROID-X2 is used as the development board. ODROID-X2 has the Samsung Exynos4412 Prime chip which is integrated by four ARM Cortex-A9 cores driven at 1.7GHz and having 2GB main memory. The board is installed with Android 4.1.2. Moreover, in the ODROID-X2 board, all four cores must be switched into a same clock frequency fro DVFS.

Since the ODROID-X2 board does not support power measurement on any part of it, some modifications are implemented in order to measure the power consumption. A circuit is wired near the PMIC (Power Management IC) [13]. That circuit includes a 40[mΩ] shunt resistor and an amplifier. The power consumption is calculated by the following formula:

$$P = \frac{1}{40 \times 10^{-3}} \times dV \times V$$

Where P is power consumption, dV is potential difference and V is supply voltage.

5.2. Experimental Demonstration Structure

The demonstration is arranged as shown in Fig. 4 and the demonstration screen is shown in Fig. 5. The intensive computation is run on ODROID-X2 board and the calculated result is displayed on a separate monitor, simultaneously. The execution time is shown on the screen in the form of fps (frames per second) so that we can keep track to the application performance.

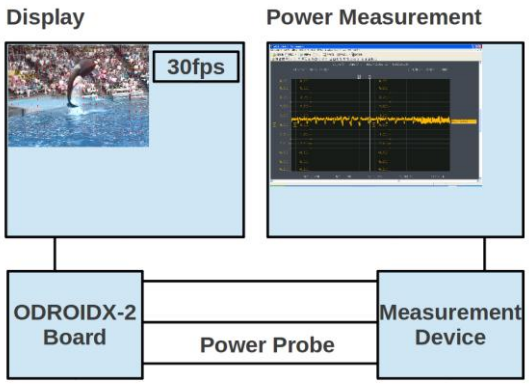

Figure 4: Experimental demonstration structure

In the development board ODROID-X2, Power Management IC part (below the cooler metal block shown in Fig. 3) is connected to an amplifier. The amplifier is then connected to a measurement device whose power consumption information is recorded by a different PC. There are several options set on that PC such as sampling frequency, number of precision digits. In addition, it is also possible to capture the power

ⓒ 2014 Information Processing Society of Japan

wave-form, obtain the average power consumption as well as export data to a CSV file.

Figure 5: Demonstration screenshot

6. Power Consumption Evaluation

6.1. Evaluated Application

In this section we explain 2 realtime video applications used in our demonstration.

6.1.1. MPEG-2 Decoder

MPEG-2 Decoder is a standard video coding application from Mediabench. It converts MPEG-2 video coded bitstream into uncompress video frames. In our experiment, a raw video output ".yuv" extension file is obtained after running the application. We convert frame data into rgb bitstream of the length 352x240 and show that on the device screen by placing the data result into a SurfaceView which is a dedicated drawing surface provided by Android.

The input data of MPEG-2 Decoder application is partitioned into slices and the application decodes the input data slice by slice. The OSCAR exploits the slice level parallelism. The deadline for MPEG-2 Decoder is set to 30[fps] (33[ms] per frame)

6.1.2. Optical Flow

The Optical Flow tracks specific features in an image across multiple frames. In our experiment, Optical Flow is used to draw a vector field of displacement vectors showing the movement of 16x16 blocks from two consecutive frames.

The OSCAR compiler exploits the parallelism on computing the motion vectors of each pixel block in two images. The deadline for Optical Flow is also set to 30[fps] (33[ms] per frame).

6.1.3. Application Parallelization and Power Optimization

After being parallelized by the OSCAR compiler, both two applications are parallelized and the shared libraries are built by ndk-build. The compiler flag is "-O3 -pthread -mfpu=neon -ftree-vectorize", the target CPU is set to "armeabi-v7a". By implementing core partitioning, which means that we separate display and calculation, and assign them to different cores, we can obtained higher performance than doing everything on the core 0. Table 1 shows the performance of MPEG-2 Decoder application with core partitioning and without core partitioning. "Without core partitioning" means that, both UI thread and arithmetic thread are assigned onto core 0. Otherwise, the core 0 is used for UI thread and the core 1 is used for arithmetic thread. From table 1, it can be seen that the number of frames per second in case of using core partitioning is twice as large as that in case of not using core partitioning. In other words, the application can have double-speed with core partitioning.

Table 1: Comparison of application performance in case of implementing core partitioning and not implementing core partitioning

	Performance
With core partitioning	83fps
Without core partitioning	40fps

At this time, it is confirmed that both two applications are obtained speed-up with the OSCAR compiler, we can apply the power optimization. The reason is that once the application is speed up, we have more available time till the deadline. This implies we likely have chance to reduce the working frequency as well as have the CPU stay at idle state longer, therefore, the power consumption can be saved.

The OSCAR pre-calculates the costs of all macro-tasks based on the number of arithmetic operations in them. These data are stored in the data

ⓒ 2014 Information Processing Society of Japan

structure of OSCAR. By using the pre-calculated data and the imported deadline information, the OSCAR estimates the execution time, the cost, the energy of each macro-task in the application and tries to make the best decision of the working frequency for each block.

The OSCAR uses 4 levels of working frequency in this evaluation: HIGH, MID, LOW, VLOW. For instance, in the current target platform ODROID-X2 which supports the frequency in the range of 200MHz to 1700MHz, HIGH is 1700MHz, MID is 800MHz, LOW is 400MHz, VLOW is 200MHz, respectively.

With the help of additional profiler information, the OSCAR determines the most proper frequency for each macro task. It is common to say that for a task, if we reduce the working frequency to half, the execution time of that task will become twice as long as the origin. However, the cycles for cache miss penalty might be reduced when the clock frequency becomes lower. In this case, the execution time of that task will become shorter than expected.

The OSCAR calculates a task cost based on the number of clock cycles in HIGH mode when the profiled feedback is not used. It means, for instance, when the clock cycle is 500 at the half clock frequency of the HIGH mode, the task cost is dealt as 1000 in the OSCAR compiler. In this paper, we measured the actual cost of the specific tasks at every frequency step: HIGH, MIDDLE, LOW and pass that profiling information to the OSCAR. Table 2 shows the profiling measurement results in case of MPEG-2 Decoder application.

Table 2: Measurement results of profiling cost in each frequency level

HIGH (100%)	28.5×10^6cycles	100%
MIDDLE (52%)	19.6×10^6cycles	69%
LOW (23%)	16.4×10^6cycles	57%

From this table, it is clear that the profiling cost is not proportional to the working frequency. Therefore, it is necessary to pass the measured profiling information to the OSCAR in order to obtain better results in power optimization. The OSCAR compiler also computes the idle time until deadline and generates some codes to notify the CPU to go to idle state. Besides that, there are some cases when it is impossible to parallelize a sequential set of tasks, those tasks are assigned to one specific CPU and the OSCAR will force other CPU(s) to idle state while waiting for those tasks completed. Once they are finished, the working CPU will wake all remaining CPU(s) up.

6.2. Power Consumption Evaluation Results

This section shows the results of power measurements on the ODROID-X2. We compare the power consumptions of two applications in case of using the OSCAR compiler and not using the OSCAR compiler. With the OSCAR compiler power control, the cpufreq governor is set to “userspace”. In contrast, the benchmark application without power control is executed with the Linux ”ondemand” governor.

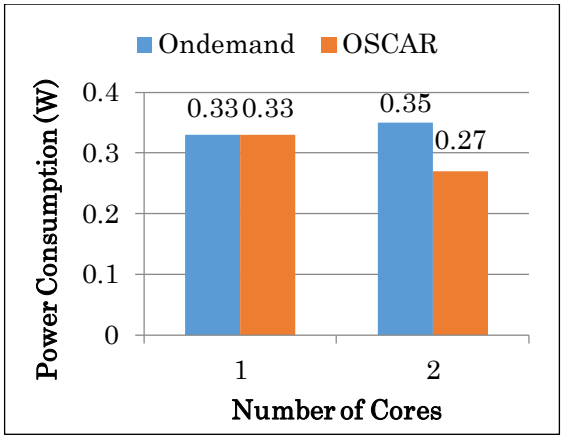

Figure 6: Power consumption of MPEG-2 Decoder

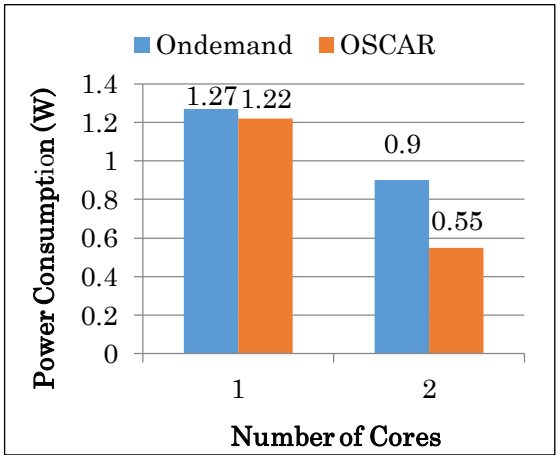

Figure 7: Power consumption of Optical Flow

Fig.6 shows the power consumption results of

ⓒ 2014 Information Processing Society of Japan

MPEG-2 Decoder in case of 1 and 2 cores, respectively. The power consumption in case of 1 core with OSCAR power optimization is 0.33[W] which is the same as that in Linux ondemand governor 0.33[W]. OSCAR and ondemand governor are equal on 1 core (processor element). The power consumption of 2 cores with power control consumes 0.27[W] compared to 0.35[W] with ondemand governor. In this case, the power consumption is saved 22.9%. The power consumption in the case of 2 cores with OSCAR power control 0.27[W] is reduced by 18.2% compared to 1 core in the default Linux ondemand governor 0.33[W].

Fig. 7 shows the power consumption results of Optical Flow application in 3 cases: 1 core and 2 cores. For 1 core, the power consumption is 1.27[W] with OSCAR power optimization. In contrast, with ondemand power control, the result is 1.27[W]. There is no big difference on power consumption in this case. For 2 cores, with power control, the power consumption is 0.55[W] while it is 0.9[W] without using OSCAR power control. The power consumption is reduced 38.9% by implementing OSCAR power optimization. The power consumption in the case of 2 cores with OSCAR power control 0.55[W] is reduced by 56.6% against the execution with 1 core in Linux ondemand governor 1.27[W].

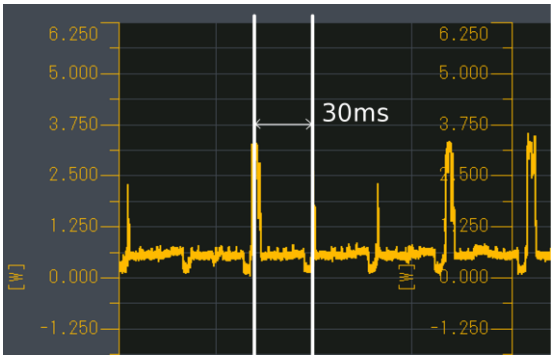

Figure 8: Power waveform with OSCAR power control

Fig. 8 shows the power waveforms with OSCAR power optimization. In this figure, we can observe the peaks in the wave form. These peaks indicate the time when the application finishes calculating 1 frame data and transfer the calculated data to UI thread to display the frame. During this time, the system is running at the highest frequency or in OSCAR's HIGH mode. In other times, OSCAR tries to scale the working frequency as low as possible.

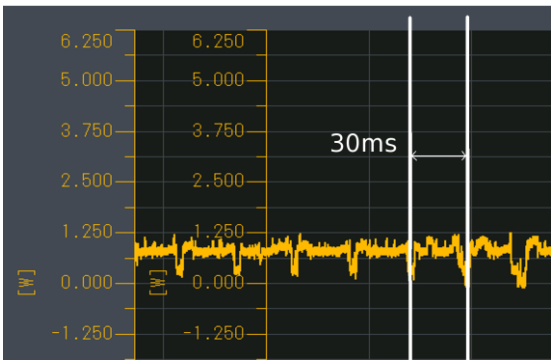

Figure 9: Power waveform with ondemand governor

On the other hands, Fig. 9 points out a characteristic of ondemand power control. Since the ondemand governor decides the working frequency based on the CPU utilization and previous system work-load, it tends to keep frequency stable when dealing with periodical application because there is not much difference between the numbers of computations in 2 consecutive frames. In ondemand governor, the applications run at fixed frequency most of the time except the beginning of the application and the time of garbage collection.

In our experiment, the ondemand governor keeps the system running at the frequency close to OSCAR's MID step. This might be a characteristic of ondemand or most of current architecture which is to run at an average frequency to assure the performance and avoid switching frequency as much as possible. However, DVFS have been showing that it is useful to reduce the power consumption. By making use of DVFS, OSCAR keeps the application running at lower frequency in longer time and it results in the reduction of power consumption.

7. Conclusion

Reducing energy consumption is gradually becoming one of the most important issue in smart device industry and automatically optimize the power consumption is a very promising way in order to attack that with a higher performance as well as lower energy consumption. This

ⓒ 2014 Information Processing Society of Japan

paper shows a realtime video demonstration system for parallelization and power reduction controlled by OSCAR Automatic Parallelization Compiler. MPEG2 Decoder Application showed 18.2% power reduction from 0.33[W] on ordinary execution to 0.27[W] on execution with power optimization by OSCAR compiler using 2 cores and Optical Flow Application showed 56.6% power reduction from 1.27[W] on ondemand Linux governor 1 core to 0.55[W] on execution with power optimization by OSCAR compiler using 2 cores.

Reference

[1] Kundu, T.K. ; Paul, K.: Improving Android Performance and Energy Efficiency, VLSI Design (VLSI Design), 2011 24th International Conference on, On page(s): 256 – 261

[2] Ki-Cheol Son ; Jong-Yeol Lee: The method of android application speed up by using NDK, Awareness Science and Technology (iCAST), 2011 3rd International Conference on, On page(s): 382 – 385

[3] OpenMP: http://openmp.org/wp/

[4] Kasahara, H., Obata, M., Ishizaka, K.: Automatic coarse grain task parallel pro-cessing on smp using openmp. Workshop on Languages and Compilers for ParallelComputing (2001) 1–15

[5] Obata, M., Shirako, J., Kaminaga, H., Ishizaka, K., Kasahara, H.: HierarchicalParallelism Control for Multigrain Parallel Processing. Lecture Notes in ComputerScience2481(2005) 31–44

[6] Kimura, K., Mase, M., Mikami, H., Miyamoto, T., Shirako, J., Kasahara, H.: OS-CAR API for Real-time Low-Power Multicores and Its Performance on Multicoresand SMP Servers. Lecture Notes in Computer Science (2010) 188–202

[7] Shirako, J., Oshiyama, N., Wada, Y., Shikano, H., Kimura, K., Kasahara, H.: Compiler Control Power Saving Scheme for Multi Core Processors. Lecture Notesin Computer Science (2007) 362–376

[8] Yamamoto, H., Hirano, T., Muto, K., Mikami, H., Goto, T., Hillenbrand, D., Takamura, M., Kimura, K., Kasahara, H.: OSCAR Compiler Controlled Multicore PowerReduction on Android Platform, The 26th International Workshop on Languages and Compilers for Parallel Computing (2013)

[9] Samsung Electronics Co., L.: White Paper of Exynos 5.1(1) (April 2011) 1–8

[10] Hardkernel: ODROID-X2 http://www.hardkernel.com/renewal2011/products/prdtinfo.php?gcode=G135235611947

[11] The Ondemand Governor https://www.kernel.org/doc/ols/2006/ols2006v2-pages-223-238.pdf

[12] Sangchul Lee; Jae Wook Jeon: Evaluating performance of Android platform using native C for embedded systems, Control Automation and Systems (ICCAS), 2010 International Conference on, On page(s) 1160 - 1163

[13] SAMSUNG ELECTRONICS: Samsung Semiconductors Global Site https://www.samsung.com/global/business/semiconductor/product/poweric/overview

ⓒ 2014 Information Processing Society of Japan