Electronic Preprint for Journal of Information Processing Vol.24 No.3

Regular Paper

Android Video Processing System Combined with Automatically Parallelized and Power Optimized Code by OSCAR Compiler

Bui Duc Binh[1,a)] Tomohiro Hirano[1] Hiroki Mikami[1]
Hideo Yamamoto[1] Keiji Kimura[1] Hironori Kasahara[1]

Received: May 19, 2015, Accepted: January 12, 2016

Abstract: The emergence of multi-core processors in smart devices promises higher performance and low power consumption. The parallelization of applications enables us to improve their performance. However, simultaneously utilizing many cores would drastically drain the device battery life. This paper shows a demonstration system of real-time video processing combined with power reduction controlled by the OSCAR automatic parallelization compiler on ODROID-X2, an open Android development platform based on Samsung Exynos4412 Prime with 4 ARM Cortext-A9 cores. In this paper, we exploited the DVFS framework, core partitioning, and profiling technique and OSCAR parallelization - power control algorithm to reduce the total consumption in a real-time video application. The demonstration results show that it can cut power consumption by 42.8% for MPEG-2 Decoder application and 59.8% for Optical Flow application by using 3 cores in both applications.

Keywords: low-power consumption, automatic parallelizing compiler, multi-core processor, Android

1. Introduction

Recently, smart devices have become progressively affordable and powerful with significant growth in the number of consumers. Users' expectation of higher performance leads to demands on the speed of processors as well as requiring efficiency when using limited energy resource. In smart devices, video applications including video players and image recognition make one of the most used application areas. However, it consumes a huge amount of energy due to high calculation complexity. Since the growth in capacity of the battery is limited, research into optimizing the power consumption of video applications in Android devices is significantly important.

Generally, Android applications are developed in Java language. It is possible to parallelize applications in Java as shown in Ref. [1]. However, it is also indicated that Android applications can be sped up by using Android NDK and JNI [1], [2]. 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 a native language such as C, then build shared library by NDK, finally use that library to compute the data passed by Java side through JNI.

One way of reducing power consumption is to make use of the power management classes provided in the application layer of Android platform framework [14]. However, this method is limited by a few of functionalities such as switching between on/off statuses of the screen, adjusting the brightness of the screen or applying and releasing the WakeLock. These operations are not enough to give an optimized solution for power consumption issue in Android smart devices. Therefore, utilizing DVFS, clock gating and power gating by native code written in C or C++ is required in order to control the power consumption of Android application.

Parallelization of applications 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 the multi-core system and 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 achieve not only application parallelization but also power optimization [6], [7]. In Ref. [8], it is showed that on ODROID-X2 [10] 4-cores (processor element - PE) board running Android 4.1.2, using OSCAR compiler and its power control module can save 86.7% power consumption when utilizing 3 cores versus the ordinary 1 core execution without OSCAR power control in case of MPEG-2 Decoder application, and 86.5% power consumption when utilizing 3 cores against the ordinary 1 core execution without OSCAR power control in case of Optical Flow application, respectively. However, this paper only measured the power consumption of computation part in those applications. In other words, no real-time display task had been done in spite of both experiment applications are multimedia applications. In order to implement real-time video processing task,

[1] Department of Computer Science and Engineering, Waseda University, Shinjuku, Tokyo 162–0042, Japan
[a)] binh@kasahara.cs.waseda.ac.jp

it is necessary to separate displaying work from calculating work and consider the application performance at different frequency steps.

This paper shows a full implementation of a demonstration system of low power real-time video processing applications by using OSCAR compiler. The completed Android video applications are built with Android NDK, core partition and OSCAR power optimized shared library. Moreover, the power-optimized result is improved by providing more precise profiling information feedback to OSCAR compiler power control module.

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

2. CPU Frequency Scaling in Linux System

This section shows a brief introduction about the power management framework of the Linux Kernel in an Android system, which is used by the OSCAR compiler for low power optimization in the proposed demonstration system.

The linux kernel provides a DVFS framework for controlling DVFS, namely CPUFreq. There are five kinds of governors, each of them offers its own strategy of working frequency in the followings.

- performance – sets the frequency to the highest supported CPU frequency.
- powersave – sets the frequency to the lowest supported CPU frequency.
- userspace – sets the frequency from a userspace program, our work uses this governor.
- ondemand – adjusts the frequency based on system utilization, our work uses this governor for comparison.
- conservative – adjusts gradually based on utilization.

In a Linux system, the ondemand governor is enabled by default. The ondemand governor changes the CPU frequency based on the CPU utilization. **Figure 1** shows the original ondemand power control algorithm [11]. For every CPU, the kernel checks the current utilization. If it is larger than the upper bound value, the kernel will set the working frequency to the maximum value. Likewise, if the current utilization is smaller than the lower bound, the working frequency will be decreased by 20%. The ondemand governor is suitable to periodical applications since the operating system can predict the proper frequency based on the previous CPU utilization, which is quite stable in case of periodic applications.

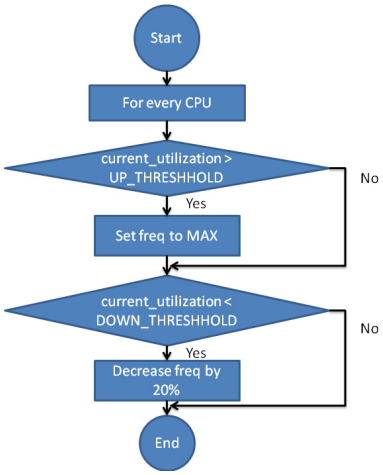

Fig. 1 Original ondemand algorithm.

In userspace governor, the user changes the working frequency through the sysfs interface. The desired frequency value can be explicitly set by writing to a sysfs file system such as /sys/devices/system/cpu/cpu<n>/cpu freq/scaling_set_speed.

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 3 kinds of tasks called macro-tasks (MT): basic block (BB), loop (RB), and subroutine call (SB) from a sequential source program. 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, 4 levels of frequency namely HIGH (100% of the highest frequency), MID (52% of the highest frequency), LOW (23% of the highest frequency) and VLOW (11% of the highest frequency) are used [7].

In the OSCAR compiler, a task cost is defined as the number of clock cycles needed to finish the task. Moreover, for each kind of hardware architecture, it is necessary to specify configuration information so that the compiler can give more precise power optimization for the target hardware architecture. The CPU static energy, dynamic energy, leak current, CPU frequency switching latency time are examples of the configuration information.

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 the form of compiler options.

4. Video Processing Demonstration System

The purpose of this paper is to show a demonstration system of automatically parallelized and power optimized real-time video applications. This section describes the details of the demonstration that has been developed in this paper.

The applications are divided into 2 parts: Java part and arithmetic part with the JNI interface. The former runs on UI (User Interface) thread which is programmatically assigned to CPU 0. This core is responsible for displaying computed data, doing garbage collection and other system related works while waiting for the results from the arithmetic thread. Since the long-running operations should not be performed on the UI thread, it is neces-

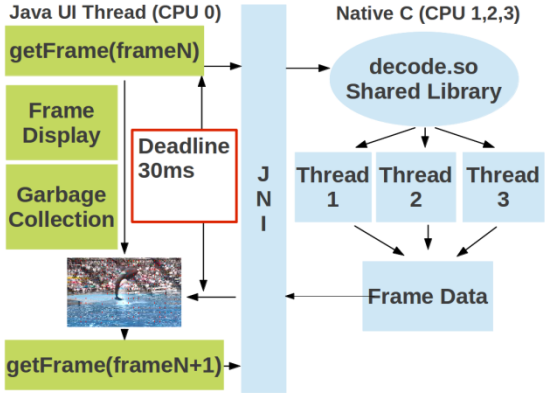

Fig. 2 Simple video player model.

sary to create new threads and process intensively heavy calculation by those threads, asynchronously. By doing so, the UI can still operate during processing. Arithmetic threads are assigned to the remaining cores by using the CPU affinity mask. The C source code for those computations is generated by the OSCAR compiler. In this experiment, we create new worker threads on different cores to take advantage of the OSCAR compiler as well as multi-core architecture. This core partitioning implementation can efficiently avoid the interference between Java part and arithmetic part, such as task migration and cache pollution. Here, we try to restrict the CPU affinity to a certain core to take advantage of CPU cache and therefore, maintain the application performance.

Figure 2 shows the completed process to compute the data and display the result on the device screen. Firstly, the UI Thread invokes a method to request the data of the frame N. This parameter N is passed as an input parameter of the native method through JNI. Depending on how many cores the developer wants to utilize, it forks into 1 or more threads working simultaneously. The forked threads will process all arithmetic calculations and join after finishing all tasks. When all operations are completed, the shared C library returns the result and passes them to Java thread through JNI interface. Finally, the calculated result will be displayed on a screen.

During the time of working on arithmetic computations, the frequency is scaled up and down according to the power optimization result by the OSCAR compiler. Since C is a processor bound language, it is possible to adjust the working frequency programmatically at the native level by opening and writing to a specific sysfs interface as described in Section 2. Meantime, the UI Thread on CPU 0 will take care of rendering, displaying 1 frame of the video, executing garbage collection and so on. This process is repeated until all frames are displayed.

One point to be noticed here is the JNI communication delay between Java part and arithmetic part. It is shown in Ref. [12] that it takes about 0.15 microseconds to pass a string from native C library on to the application. Since the deadline for a multimedia application is larger than 10 milliseconds, this delay is negligible.

5. Demonstration Board Setup

5.1 ODROID-X2 Board

In this experiment, the ODROID-X2 is used as the development board. ODROID-X2 has the Samsung Exynos4412 Prime chip [9] which is integrated by 4 ARM Cortex-A9 cores driven at 1.7 GHz and having 2 GB main memory. The board is installed with Android 4.1.2. Moreover, in the ODROID-X2 board, all 4 cores must be switched into the same clock frequency by DVFS.

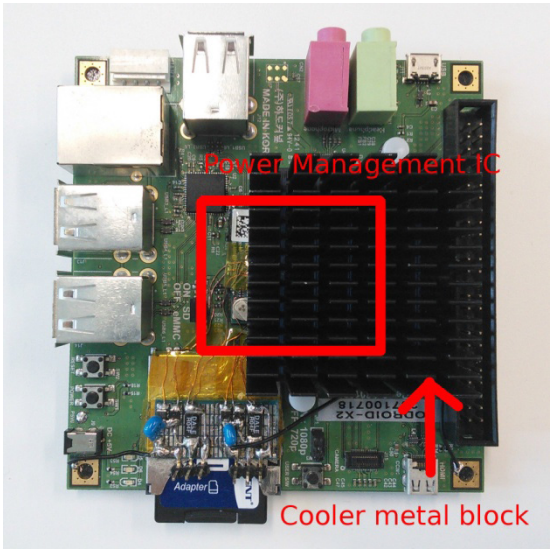

Fig. 3 Modified ODROID-X2 Board.

Since the ODROID-X2 board does not support power measurement on any part of it, some modifications are made 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 the potential difference, and V is the supply voltage. This formula calculates the instantaneous power consumption coming to the CPU of the development board. When we collect the data, we actually use the average power consumption over the execution time by using the measurement software for the evaluation.

5.2 Experimental Demonstration Structure

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

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 on a different PC. There are several options for the measurement software such as sampling frequency or number of precision digits. In addition, it is also possible to capture the power waveform and obtain the average power consumption as well as export data to a CSV file.

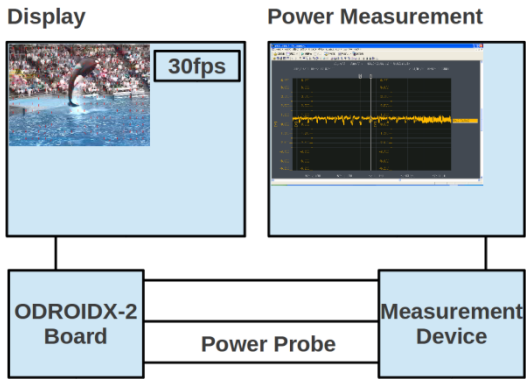

Fig. 4 Experimental demonstration structure.

Fig. 5 A screenshot of the demonstration system.

6. Power Consumption Evaluation

6.1 Evaluated Applications

In this section, we explain 2 real-time video applications used in our demonstration. This paper used the video that was contributed by the Institute of Image Information and Television Engineers. Among about 70 video samples, we selected one video for the purpose of the demonstration. It is a 15 seconds yuv video of the size 352×240. The video shows an orca jumping out of the pool in front of a large audiences in a public performance in an aquarium. It was taken using a pan to follow the shot of the orca moving from left side to right side. This situation makes it difficult to compress as well as decode the video because of the fine pattern of the audiences. Since this is a highly intensive processing video, we chose this video as the input of our experiment. The information about the video sample can be found at Ref. [15].

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 an RGB bitstream of the 352×240 frame size and show that on the device screen by placing the data result into a SurfaceView, which is a dedicated drawing surface by the Android system.

The input data of MPEG-2 Decoder application is partitioned into slices, and the application decodes the input data slice by slice. The OSCAR compiler exploits the slice level parallelism. The deadline for MPEG-2 Decoder is set to 60 [fps] (16 [ms] per frame).

Table 1 Comparison of application performance in case of implementing core partitioning and not implementing core partitioning (MPEG-2 Decoder).

	Performance
With core partitioning	83fps
Without core partitioning	40fps

Table 2 Comparison of application performance in case of implementing core partitioning and not implementing core partitioning (Optical Flow).

	Performance
With core partitioning	22fps
Without core partitioning	10fps

6.1.2 Optical Flow

The Optical Flow tracks specific objects in an image across multiple frames. In our experiment, the Optical Flow is used to draw a vector field of displacement vectors showing the movement of 16×16 blocks from 2 consecutive frames. In our experiment, the frame size is 640×360 pixels.

The OSCAR compiler exploits the parallelism on computing the motion vectors of each pixel block in 2 images. The deadline for Optical Flow is also set to 10 [fps] (100 [ms] per frame) in this evaluation.

6.1.3 Application Parallelization and Power Optimization

After being parallelized by the OSCAR compiler, both 2 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 run display task and calculation tasks on separated threads, and assign those threads onto different cores, we can obtain higher performance than doing everything on the core 0. **Table 1** and **Table 2** show the performance of MPEG-2 Decoder and Optical Flow applications, respectively, 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 those 2 tables, it can be seen that the number of frames per second in case of using core partitioning is more than twice as large as that in the case of not using core partitioning. In other words, the application can have double-speed with core partitioning.

At this time, it is confirmed that both 2 applications have room for power optimization by the OSCAR compiler since both applications attain higher FPS than the target FPS. This implies we likely have a chance to reduce the working frequency as well as keeping the CPU at the idle state longer. Therefore, the power consumption can be saved.

The OSCAR compiler precalculates the costs of all macro-

Table 3 Measurement results of profiling cost in each frequency level for the main loop in MPEG-2 Decoder.

Ratio of Clock Frequency	Profiling Cost (Normalized to HIGH mode)	Ratio of Profiling Cost
HIGH (100%)	28.5×10^6cycles	100.0%
MIDDLE (52%)	19.6×10^6cycles	68.8%
LOW (23%)	16.4×10^6cycles	57.5%
VLOW (11%)	14.2×10^6cycles	49.8%

Table 4 Measurement results of profiling cost in each frequency level for the main loop in Optical Flow.

Ratio of Clock Frequency	Profiling Cost (Normalized to HIGH mode)	Ratio of Profiling Cost
HIGH (100%)	141.7×10^6cycles	100.0%
MIDDLE (52%)	140.2×10^6cycles	98.9%
LOW (23%)	140.1×10^6cycles	98.8%
VLOW (11%)	140.0×10^6cycles	98.8%

tasks based on their number of arithmetic operations. These data are stored in the data structure of the OSCAR compiler. By using the pre-calculated data and the imported deadline information, the OSCAR compiler estimates the execution time, the cost, the energy of each macro-task in the application and tries to make the best decision on 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 200 MHz to 1,700 MHz, HIGH is 1,700 MHz (100%), MID is 800 MHz (52%), LOW is 400 MHz (23%), VLOW is 200 MHz (11%), respectively.

Since our applications are soft real-time applications, we applied profiling technique to improve power control result. With the help of additional profiling information, the OSCAR determines the most proper frequency for each macro task. It is usual 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 relatively reduced when the clock frequency becomes lower. In this case, the execution time of that task will become shorter than expected.

The OSCAR compiler calculates a task cost based on the number of clock cycles normalized to HIGH mode when the profiled feedback is not used. It means, for instance, when the number of 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 several frequency steps: [HIGH, MIDDLE, LOW, VLOW] and pass that profiling information to the OSCAR compiler. **Table 3** shows the profiling measurement results at the sequential execution in the case of MPEG-2 Decoder application and **Table 4** shows the profiling measurement results in the case of MPEG-2 Decoder application, respectively. In these tables, all profiling costs are normalized to the HIGH mode.

In Table 4, the profiling cost is almost the same in all working frequencies. On the other hand, in Table 3, it is clear that the profiling cost is not the same in all working frequencies. Therefore, it is necessary to pass the measured profiling information to the compiler in order to obtain better results in power optimization, especially for the case of MPEG-2 Decoder. If there is no feedback information given to OSCAR compiler, the compiler will use the default task cost which is calculated based on the number of arithmetic operations in the task. The feedback information is obtained by profiling the application that means measuring the cost or the number of clock cycles taken by the related code blocks under multiple frequency steps such as HIGH, MIDDLE, LOW, VLOW. The profiled block costs are then passed to OSCAR compiler by compiler directives in the form of pragma parameter. For instance, we might have a line of code as the following.

```
#pragma BLOCK COST 100000 70000 50000 20000
```

Those 4 values correspond to the block costs when the working frequency is HIGH, MID, LOW, and VLOW respectively. The OSCAR compiler uses that profiling information instead of the default block cost information to optimize the power control of the application.

The OSCAR compiler also computes the idle time until the 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 2 applications in the 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 shows the power consumption results of MPEG-2 Decoder in the case of 1, 2 and 3 cores, respectively. The power consumption in case of 1 core with OSCAR power optimization is 1.1 [W] which is almost the same as that in Linux ondemand governor 1.31 [W]. The power consumption of 2 cores with power control consumes 0.81 [W] compared to 2.2 [W] with ondemand governor. In the case of 3 cores, the power consumption with OSCAR is 0.76 [W] versus 3.15 [W] with ondemand governor. The power consumption in the case of 3 cores with OSCAR power control 0.76 [W] is reduced by 42.8% compared to 1 core in the default Linux ondemand governor 1.31 [W].

Figure 7 shows the power consumption results of Optical Flow application in 3 cases: 1 core, 2 cores, and 3 cores. For 1 core, the power consumption is 1.04 [W] with OSCAR power optimization. In contrast, with ondemand power control, the result is 1.07 [W]. There is no big difference in power consumption in this case. For 2 cores, with power control, the power consumption is 0.55 [W] while it is 1.74 [W] without using OSCAR power

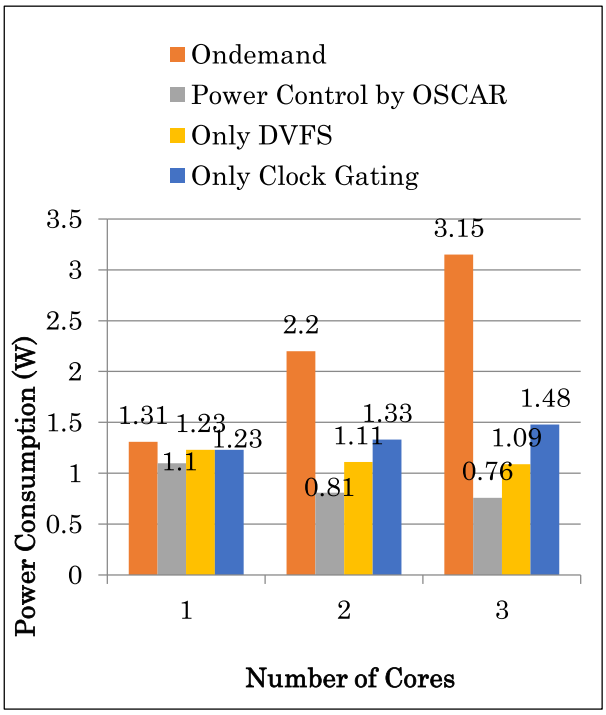

Fig. 6 Power consumption of MPEG-2 Decoder.

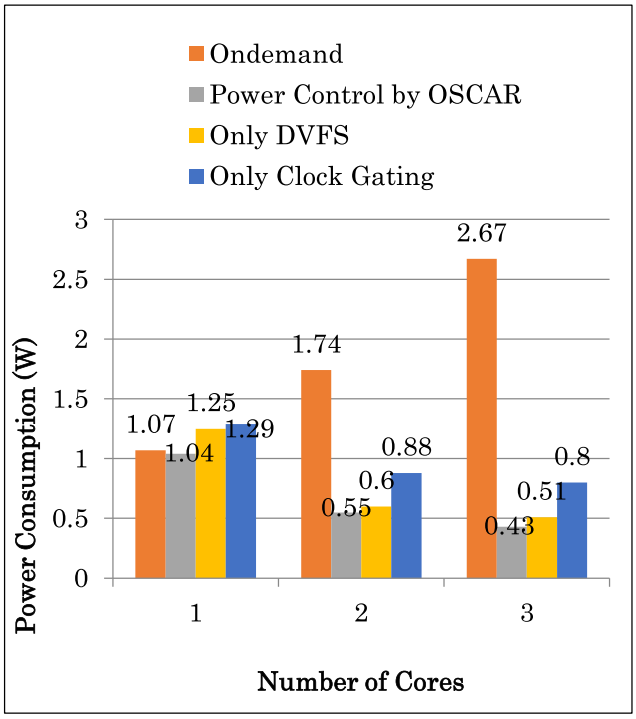

Fig. 7 Power consumption of Optical Flow.

control. In the case of 3 cores, the power consumption is 0.43 with OSCAR and 2.67 [W] with ondemand governor. The power consumption in the case of 3 cores with OSCAR power control 0.43 [W] is reduced by 59.8% versusu the execution with 1 core in Linux ondemand governor 1.07 [W].

Figures 8 and **9** show the power waveforms with OSCAR power optimization. In this figure, we can observe the peaks in the waveform. These peaks indicate the time when the application finishes calculating 1 frame data and transfers the calculated data to UI thread to display the frame. During this time, the system

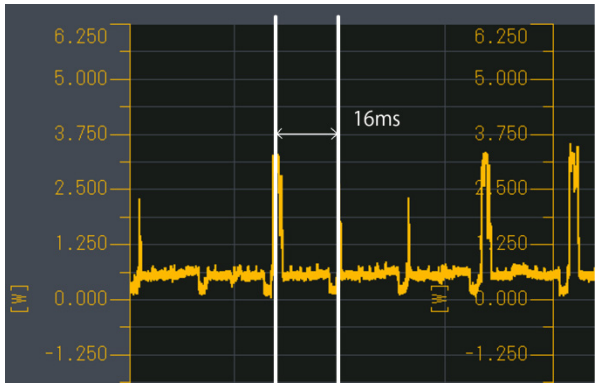

Fig. 8 Power waveform with OSCAR power control (MPEG-2 Decoder – 1 core).

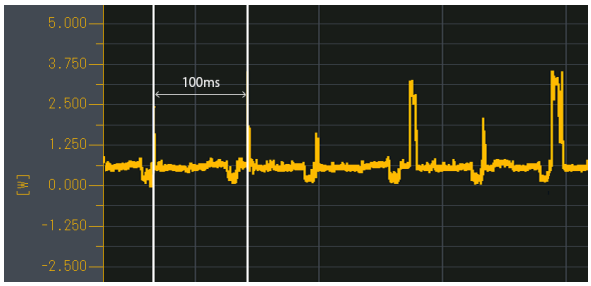

Fig. 9 Power waveform with OSCAR power control (Optical Flow – 1 core).

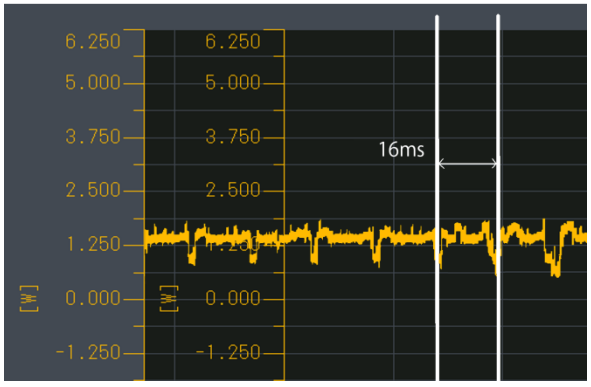

Fig. 10 Power waveform with ondemand governor (MPEG-2 Decoder – 1 core).

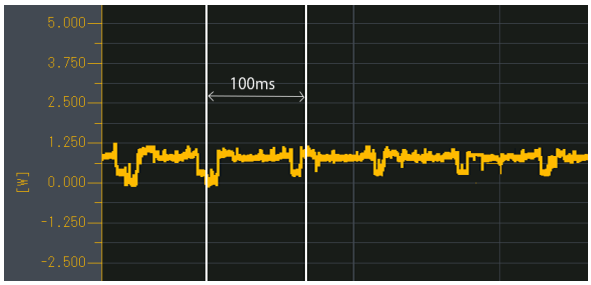

Fig. 11 Power waveform with ondemand governor (Optical Flow – 1 core).

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 at the calculation time. As the results, MPEG-2 Decoder is executed with 1.1 [W] and Optical Flow is executed with 1.04 [W] as shown in Fig. 6 and Fig. 7, respectively.

On the other hand, **Figs. 10** and **11** point out a characteristic of ondemand power control. Since the ondemand governor decides

the working frequency based on the CPU utilization and previous system workload, it tends to keep frequency stable when dealing with periodic applications because there is not much difference between the numbers of computations in 2 consecutive frames. With the ondemand governor, the applications run at a fixed frequency most of the time except the beginning of the application and the time of garbage collection. As a result, MPEG-2 Decoder is executed with 1.3 [W] and Optical Flow is executed with 1.07 [W] as shown in Fig. 6 and Fig. 7, respectively.

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 a lower frequency for longer time, and it results in the reduction of power consumption.

The power control by the OSCAR compiler includes DVFS, clock gating and power gating as described in Section 3. In order to make a comparison between CPU clock frequency scaling and clock gating, we evaluated more experiments under the following two scenarios. Firstly, we measure the power consumption by using only CPU clock frequency and voltage scaling (only DVFS). In this scenario, DVFS is utilized appropriately by the OSCAR compiler. However, when one frame is decoded, all the cores must wait for the next frame at VLOW frequency. Secondly, we measure the power consumption by using only clock gating (only clock gating). In this scenario, the cores run at HIGH frequency for decoding one frame then wait for the next frame by utilizing clock gating. The measurement results are also shown in Figs. 6 and 7. Note that the OSCAR compiler did not utilize power gating in our experiment due to the expensive overhead for the ODROID-X2 running Android 4.1.2.

Two figures show that CPU clock frequency scaling is more efficient than clock gating in the case of MPEG2-Decoder and Optical Flow applications executed on the Odroid-X2 board running Android 4.1.2. These figures also show that only DVFS and only clock gating consume higher power that the power control by the OSCAR compiler, which combines both of DFVS and clock gating. It means that we first apply CPU clock frequency to run the system at low frequency as long as possible, then we apply the clock gating to the remaining time until the deadline if any. As the result, for the case of MPEG-2 Decoder, the power control by OSCAR with 3 cores attains 0.76 W while the power consumption by DFVS and clock gating are 1.09 W and 1.48 W, respectively. Similarly, for the case of Optical Flow with 3 cores, the power consumption by the OSCAR compiler, DFVS, and clock gating are 0.43, 0.51, 0.80 respectively.

7. Conclusion

Reducing energy consumption is gradually becoming one of the most important issues in smart device industry and automatically optimizing the power consumption by a compiler is a very promising way to attack that issue. This paper shows a real-time video demonstration system for parallelization and power reduction controlled by the OSCAR Automatic Parallelization Compiler. In addition, core partitioning and per-frequency profiling are applied to maximize the efficiency of the power control by the OSCAR compiler. MPEG-2 Decoder Application showed 42.8% power reduction from 1.31 [W] on ordinary execution to 0.76 [W] on execution with power optimization by OSCAR compiler using 3 cores. Similarly, Optical Flow Application showed 59.8% power reduction from 2.67 [W] on ondemand Linux governor 1 core to 0.43 [W] on execution with power optimization by OSCAR compiler using 3 cores.

References

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

[2] Son, K.C. and Lee, J.Y.: The method of android application speed up by using NDK, 2011 3rd International Conference on Awareness Science and Technology (iCAST), pp.382–385 (2011).

[3] OpenMP, available from ⟨http://openmp.org/wp/⟩.

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

[5] Obata, M., Shirako, J., Kaminaga, H., Ishizaka, K. and Kasahara, H.: Hierarchical Parallelism Control for Multigrain Parallel Processing, *Lecture Notes in Computer Science*, Vol.2481, pp.31–44 (2005).

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

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

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

[9] Samsung Electronics Co., Ltd.: White Paper of Exynos 5, Vol.1, No.1, pp.1–8 (Apr. 2011).

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

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

[12] Lee, S. and Jeon, J.W.: Evaluating performance of Android platform using native C for embedded systems, *2010 International Conference on Control Automation and Systems* (*ICCAS*), pp.1160–1163 (2010).

[13] SAMSUNG ELECTRONICS: Samsung Semiconductors Global Site, available from ⟨https://www.samsung.com/global/business/semiconductor/product/poweric/overview, http://www.hardkernel.com/main/products/prdt_info.php?g_code=G13523⟩.

[14] Android Develop Site, available from ⟨http://developer.android.com/guide/basics/what-is-android.html, http://developer.android.com/images/system-architecture.jpg⟩.

[15] Video Sample, available from ⟨http://www.nes.or.jp/gaiyo/index.html, http://www.nes.or.jp/gaiyo/pdf/ite_hyoujundouga_sample.pdf⟩.

Bui Duc Binh is currently a Master student in Department of Computer Science and Communications Engineering, Graduate School of Fundamental Science and Engineering, Waseda University, Tokyo, Japan. He received his B.S. degree of Computer Science from Waseda University in 2014. His research interests include automatic parallelizing compiler, green computing, operating system and applications in Android platform.

Tomohiro Hirano received his M.S. degree in Fundamental Science and Engineering of Waseda University, Tokyo, Japan and B.S. degree in Fundamental Science and Engineering from the same University. His research interests include automatic parallelizing compiler, green computing, operating system, reduction of power consumption.

Hiroki Mikami received his M.S. degree in Computer Science and Engineering from Waseda University in 2009. He was a research associate of Department of Computer Science and Engineering in 2011 and has been a research associate of Green Computing Systems R&D Center since 2014 at Waseda University. His research interests include parallelizing applications, automatic parallelizing compiler, and multicore architecture.

Hideo Yamamoto is Adjunct Researcher in Department of Computer Science and Communications Engineering, Graduate School of Fundamental Scienceand Engineering, Waseda University, Tokyo, Japan. His research interests are in the areas of parallel computing, embedded computing and compilers.

Keiji Kimura received his B.S., M.S. and Ph.D. degrees in electrical engineering from Waseda University, in 1996, 1998, 2001 respectively. He was an assistant professor at 2004, an associate professor at 2005, and has been a professor of Department of Computer Science since 2012 at Waseda University. His research interest includes multicore processor architecture and parallelizing compilers. He is a member of IPSJ, IEICE, ACM and IEEE.

Hironori Kasahara received his Ph.D. degree in electrical engineering from Waseda University, Tokyo in 1985. He has been a professor of Department of Computer Science and Engineering since 1997 and a director of the Advanced Multicore Processor Research Institute since 2004, Waseda University via an assistant professor in 1986 and an associate professor in 1988. He was a visiting scholar at University of California, Berkeley and University of Illinois at Urbana-Champaign's Center for Supercomputing R&D. He received IEEE Computer Society Golden Core Member Award, IFAC World Congress Young Author Prize, IPSJ Fellow and Sakai Memorial Special Research Award and a Science and Technology Prize in the commendation by Minister of Education, Culture, Sports, Science and Technology. He has served as a chair or a member of 220 society and government committees including IPSJ, IEEE Computer Society, ACM, METI, MEXT and so on.