Android Demonstration System of Automatic Parallelization and Power Optimization by OSCAR Compiler

BUI DUC BINH[†1] TOMOHIRO HIRANO[†2] DOMINIC HILLENBRAND[†3]
HIROKI MIKAMI[†4] KEIJI KIMURA[†4] HIRONORI KASAHARA[†5]

The emergence of multicore processors in smart devices promises higher performance and better user experience. 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 report shows a realtime video demonstration system for power reduction controlled by OSCAR Automatic parallelization Compiler on ODROID-X2, an open Android development platform based on Samsung Exynos4412 Prime. 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

Recently, smart devices have been becoming the most popular and dominant devices in the electronic market. Smart phone and tablet sales overtook the global PC sales in 2013 [1].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.

This leads to a significant growth in the number of smart device users and sequentially, the demands on mobile device are getting more concerned. It is required by users that smart devices should be able to provide application performance at desktop-like performance, however, in order to achieve that 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. To be able to reach higher performance in smart device, it is necessary to fully utilize all available processors while still considering about power consumption. 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 [2] 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, the applications in Android are developed in Java language. It is possible to parallelize the applications in Java as shown in [3]. However, it is also indicated by [3] and [4] 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 in doing arithmetic operations. One more way of parallelizing applications is to parallelize them in native language such as C, 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 firstly using Android NDK and secondly using parallelized native code.

Parallelization of application is a very effective way to benefit from multi-core system, however, manually parallelizing a large program is very time consuming and most of current applications are written for single core architecture. There are some parallelizing compilers, such as OpenMP Compiler [5] and OSCAR compiler [6][7]. For all of these parallelizing compilers, OSCAR Compiler can realize not only application parallelization but also power optimization [8][9]. [10] 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 [10] were conducted on ODROID-X2[11] board, an open Android development platform based on Samsung Exynos4412 Prime [12]. 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 report introduces a full demonstration system of playing video and measuring power consumption simultaneously. In this report, 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 play system with low power optimization by utilizing per-frequency profiling result in addition to the previously proposed power optimization technique by OSCAR compiler.

The rest of this report 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 evaluation results and section 7 gives the conclusion of the report.

2 Power Management on Android

Android Kernel or Linux Kernel supports a set of basic functions of DFS (Dynamic Frequency Scaling) by which users can adjust the system frequency in their own way. Besides, the Android Kernel also provides an APCI (Advanced Configuration and Power Interface) with five frequency adjustment schemes. Respective to each of these schemes, there is one “CPUFreq Governor” that decides which frequency should be used. The governor could be a performance-oriented one which sets the working frequency to the highest supported value. It is called “performance”. In contrast to the governor performance, the CPUfreq governor called “powersave” tries to execute all tasks at the lowest frequency and save energy as far as possible. It is power-oriented governor. Between these two governors, other three governors are “ondemand”, “conservative” and

ⓒ 2014 Information Processing Society of Japan

"userspace". Ondemand will dynamically change the working frequency depending on the current workload while in userspace governor, users must specify a working frequency and the whole system will run at that frequency. The conservative governor is similar to the ondemand governor. The only difference is that the conservative governor gradually increases and decreases the frequency rather than jumping from one frequency to another one.

In Linux system, the ondemand governor is enabled by default. In response to the ACPI event, the ondemand governor changes CPU frequency depending on CPU utilization. Fig. 1 [13] shows the original ondemand power control algorithm. After every X milliseconds, the system checks if the current system utilization is larger than the defined upper bound value, it will set the working frequency to the maximum available frequency value. Likewise, after every Y milliseconds, the system calculates the current utilization and compares that to the lower bound value. 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.

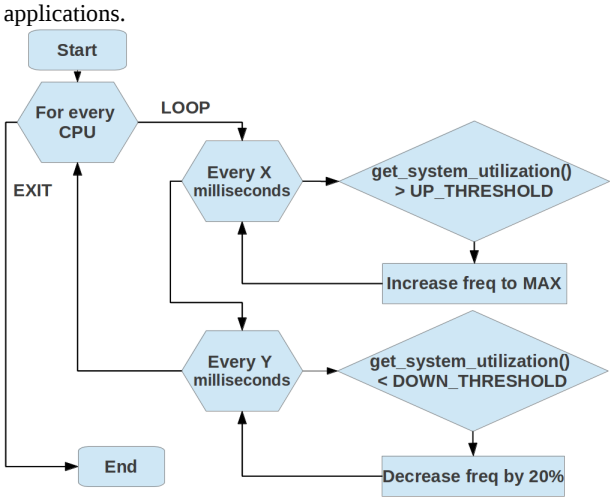

Figure 1: Original ondemand algorithm

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

3 OSCAR Compiler

In order to fully utilize all available processors, the applications need to be multi-threaded. Most of the current applications were not developed with considerations about multi-core system as well as power optimization as a priority. The following briefly describes OSCAR Compiler and OSCAR API, which is used for parallelization and power optimization of applications in this report.

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) [6].

Then macro-tasks are scheduled to available processors. A task is called ready task if it satisfies the earliest executable condition. According to the priorities, a macro-task is picked from a list of ready tasks and assigned to an appropriate core. This process is repeated until all macro tasks are scheduled.

Since the application is speed up after the parallelization, it is possible to execute it by a lower frequency at some point while maintaining the performance as good as in sequential case. 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 report, four levels of frequency namely HIGH, MID, LOW and VLOW are used [9].

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 by OSCAR API. The results can be improve 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 Demonstration System

The purpose of this report is to show a demonstration system of automatic parallelization and power optimization realtime video application and this section describes the details of the simple video application we have developed.

Android is a powerful Operating System supporting a large number of mobile applications which are mainly developed in Java language and Android SDK. However, only Android SDK is not enough if the developer wants an application to utilize all the cores, or to speed up an application by making use of the C code he has got. Android NDK is distributed to enable developers to program directly into Android native interface, hence, developers can overcome the limitations of Java, such as performance or memory management.

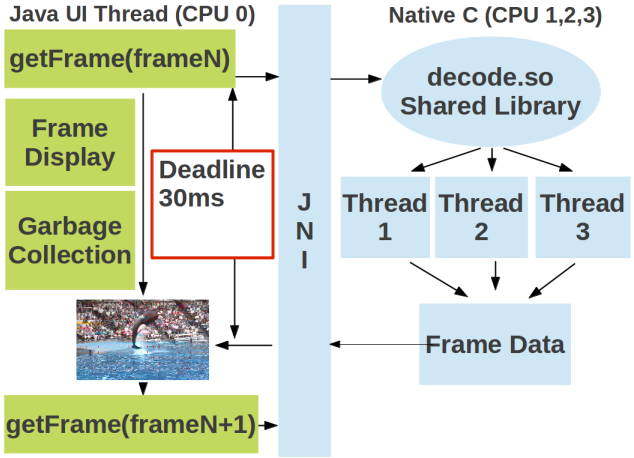

Figure 3: Simple video player model

Multimedia applications are often required to finish a large number of computations in a restricted time, therefore, it is common to use fast C code in computing, then send the result back to 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 can be divided into two parts: arithmetic part and Java part. The Java part runs on UI (User Interface) thread which is assigned to CPU 0. This part will be responsible for displaying computed data, doing garbage collection and other system related works. Since we should avoid

ⓒ 2014 Information Processing Society of Japan

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 and it remains responsive. All long running computations are done on the arithmetic part which is written in C. This part is parallelized, and optimized by OSCAR compiler. The UI thread keeps running to perform all basic tasks while waiting for the results from arithmetic part. In Android application, it is possible by using AsyncTask [14] to perform background operations and then pass the results to the UI thread. However, in this experiment, we create new worker threads on different cores to take the advantage of 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.

```
FOR i from 0 to FRAME_NO

    CALL native method to get frame[i] data
    WHILE waiting for frame[i] data

        Display previous frame data
        Execute garbage collection
        Run system related works

    Display frame[i]

ENDFOR
```

Figure 4: UI thread algorithm

```
FRAME_NO = i
FORK thread1, thread 2, … thread CPU_NO
FOR each thread

    SET affinity to core
    Get frame[i] data partly
    Synchronize with other threads

ENDFOR
JOIN thread1, thread2, … thread CPU_NO
RETURN frame[i] data
```

Figure 5: Native thread algorithm

Figure 3 shows the completed process to compute the data and display it in the device screen. Figure 4 and 5 describes the algorithm in UI thread and native thread, respectively. Firstly, UI Thread invokes a method to require the data of the frame N. This parameter N is passed as input of native method through JNI. At JNI, the DalvikVM will map the method invoked by UI Thread to a native target method which is prepared in a shared C library. Depending on how many cores the developer want to utilize, it forks into one, two or three threads working simultaneously. Forked threads will process all arithmetic calculations and join after finishing all tasks. When all operations have done, the shared C library returns the result and again, this result is passed to Java part through JNI interface, finally the calculated result will be display to the device screen.

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 by opening and writing to a specific sysfs. Besides that, during this time, 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. [15] 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 around 30 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 which is integrated by four ARM Cortex-A9 cores and 2GB memory. Each of these cores supports the maximum frequency clock of 1.7GHz. In addition, this board comes with six USB 2.0 ports, micro HDMI. The board is installed with Android 4.1.2 Operating System. Moreover ODROID-X2 board is homogeneous multi-core architecture, meaning that all four cores always have the same behaviors when the working frequency is changed.

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) [16]. 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.

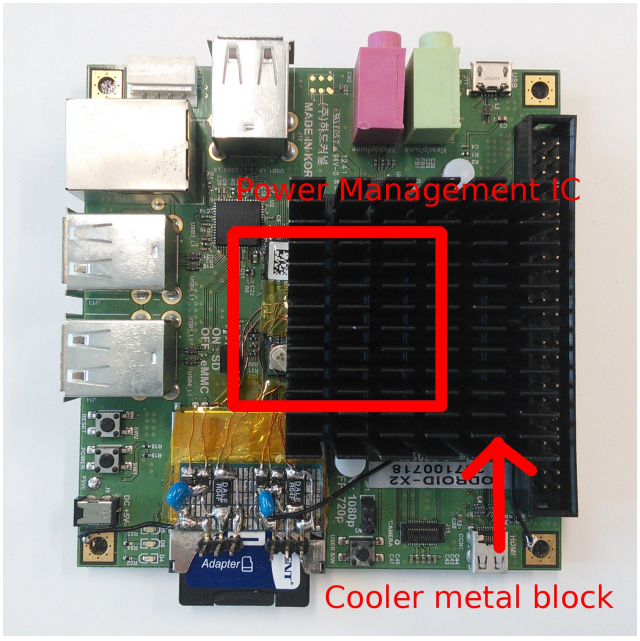

Figure 4: ODROID-X2 Board

5.2 Experimental Demonstration Structure

The demonstration is arranged as shown in Fig. 6 and the demonstration screen is shown in Fig. 7. Since the ODROID-X2 is a micro sized development platform, it does not come with a screen and therefore, it is necessary to connect it to an external screen. The main computation is run on the main ODROID-X2 board and the result will be displayed on the connected screen, simultaneously. The execution time is shown on the screen in the form of fps (frames per second). By this we can keep track to the application performance.

Meantime, the development board Power Management IC part (below the cooler metal block shown in Fig. 4) is connected to an amplifier. The amplifier is then connected to a measurement

ⓒ 2014 Information Processing Society of Japan

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

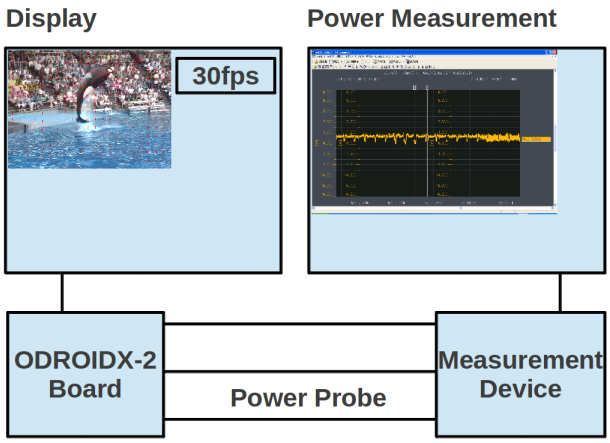

Figure 6: Experimental demonstration structure

Figure 7: Demonstration screenshot

6 Evaluation

6.1 Evaluated Application

In this section we explain 2 multimedia 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 this application. We load the requested frame data from that raw file and convert it to rgb bitstream of the length 352x240 and finally show that on the device screen by placing the data result into a SurfaceView which is a dedicated drawing surface supported by Android.

The input data of MPEG-2 Decoder application is partitioned into slices and the application decodes the input data slice by slice. 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 is a benchmark application tracking 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.

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

6.1.3 Application Parallelization and Power Optimization

After being parallelized by OSCAR compiler, the shared libraries are built for both two applications. The compiler flag is "-O3 -pthread -mfpu=neon -ftree-vectorize", the target CPU is set to "armeabi-v7a". By using shared libraries in case of utilizing 1, 2 and 3 cores, the performance of two applications is observed and when it is confirmed that both two applications are speed up with OSCAR compiler, we 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 applications are compiled by OSCAR compiler. Firstly, 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 structure of OSCAR. By using the pre-calculated data and the imported deadline, OSCAR estimates the execution time, the cost, the energy of each block in the application and tries to make the best decision of the working frequency for each block.

OSCAR supports 4 levels of working frequency: HIGH, MID, LOW, VLOW. Depending on the architecture, each of those steps will correspond to one specific value of frequency. For example, 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. These would be different if an Intel CPU is used instead of our ARM-core board.

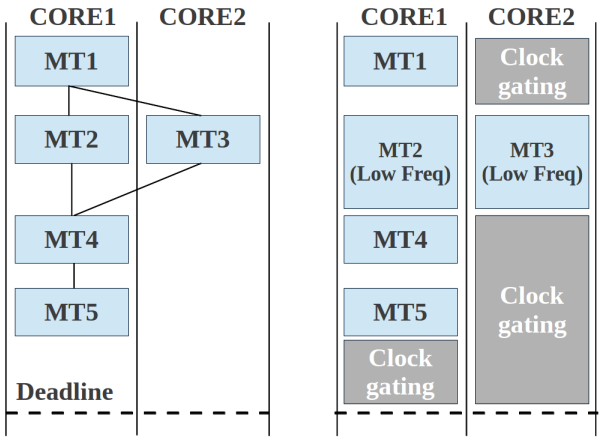

Figure 8: OSCAR power control

With the help of additional profiler information, OSCAR determines the most proper frequency for each macro task. It 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 OSCAR will force the 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. Fig. 8 shows an example of OSCAR

ⓒ 2014 Information Processing Society of Japan

power optimization in case of using 2 cores. There are 5 macro tasks from MT1 to MT5. MT2 and MT3 belong to the same loop in the sequential program but work on different cores after the sequential program is parallelized by OSCAR compiler. While core 1 is executing MT1, core 2 is waiting for core 1 to finish MT1. At this time, since there is no load on core 2, it is possible to apply clock gating on core 2 in order to save energy. Then, for MT2 and MT3, which are simultaneously executed on core 1 and core 2 respectively, the working frequency is reduced while assuring the deadline. The execution time of these tasks will be longer but they still finish before the deadline. We assume that it is impossible to parallelize MT4 and MT5. During the execution time of these tasks, core 2 has no task on it, hence, we can apply clock gating to core 2 until the deadline. Meantime, core 1 is running MT4 and MT5 and it finishes these two tasks before the deadline. Again, we apply clock gating to core 1 on the remaining time until the deadline.

6.2 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 OSCAR compiler and not using OSCAR compiler. With 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.

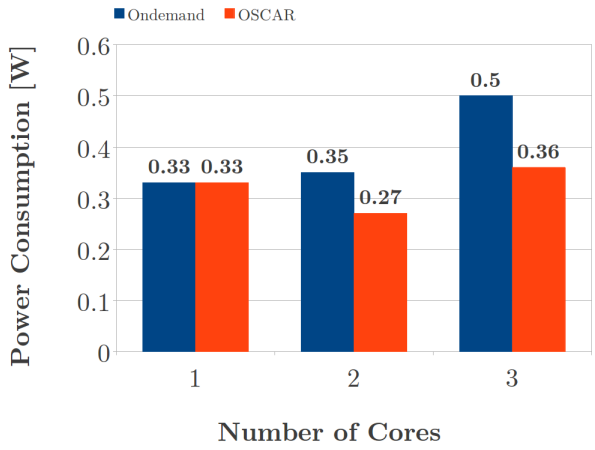

Figure 9: Power consumption of MPEG-2 Decoder

Fig.9 shows the power consumption results of MPEG-2 Decoder corresponding to number of processors (1, 2 and 3). The power consumption in case of 1 core with 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 1PE (processor element). The power consumption of 2PE with power control consumes 0.27[W] compared to 0.35[W] in case of not be implemented by OSCAR compiler. In this case, the power consumption is saved 22.9%.Using 3PE with power optimization consumes 0.36[W] while using 3PE in ondemand governor consumes 0.5[W], it can save up to 28% energy in case of 3PE. The best result is 2PE with power reduction control 0.27PE which reduced 18.2% compared to 1PE in the default Linux ondemand governor 0.33[W].

Fig. 10 shows the power consumption results of Optical Flow application in 3 cases: 1PE, 2PE and 3PE. For 1PE, 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 2PE, 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 about 38.9% by implementing OSCAR power optimization. The power consumption of 3PE with OSCAR power control consumes 0.66[W] compared to 0.92[W] without using OSCAR power control. It is saved 28.2% by OSCAR power reduction control. In the best case (2PE with OSCAR compiler power optimization), the power consumption is reduced 56.6% against the execution with 1PE in Linux ondemand governor.

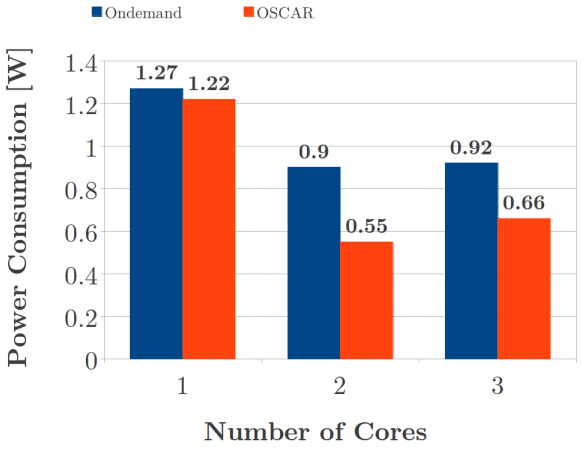

Figure 10: Power consumption of Optical Flow

Fig. 11 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 that time, the system is running at the highest frequency or in OSCAR's HIGH mode. In other times, since OSCAR tries to keep the working frequency as low as possible, as a result, the application will be running at lower frequencies.

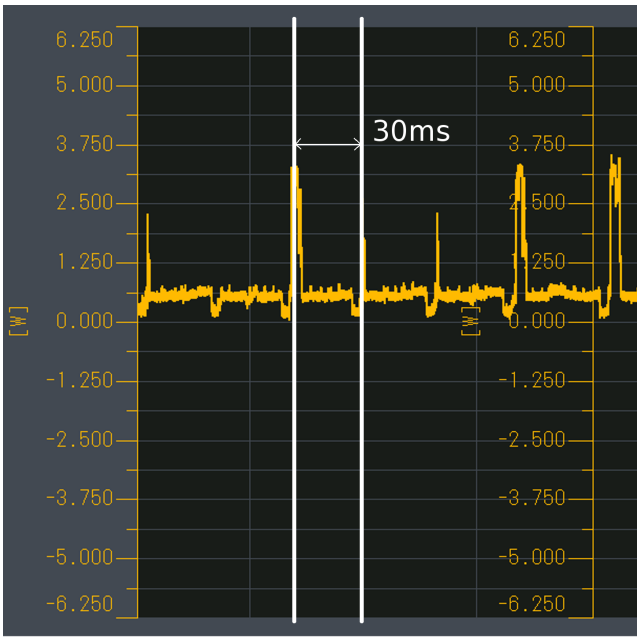

Figure 11: Power waveform with OSCAR power control

On the other hands, Fig. 12 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 the frequency stable when dealing with periodical application such as multi-media application because there is not much difference between the numbers of computations in consecutive frames. In ondemand

ⓒ 2014 Information Processing Society of Japan

governor, the applications run at fixed frequency most of the time except the beginning of the application and the time of garbage collection.

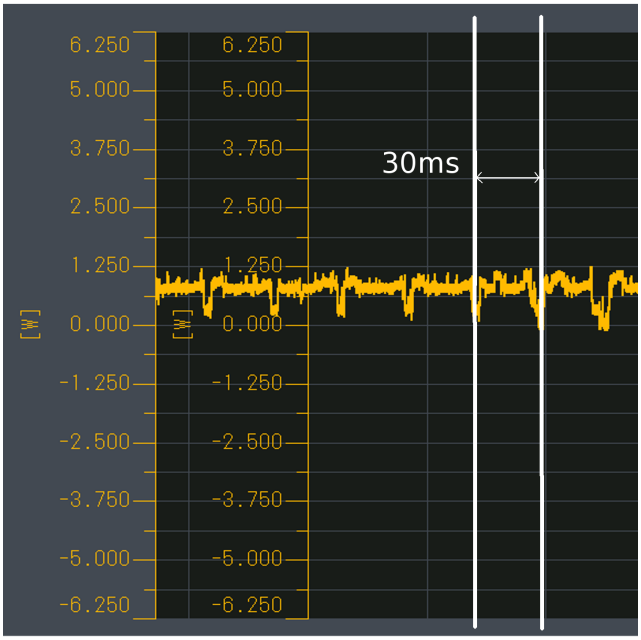

Figure 12: Power waveform with ondemand governor

In our experiment, the ondemand governor keeps the system running at the frequency which is 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 somehow 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 this results in the reduction in power consumption.

Table 1: CPU idle time

	2PE	3PE
MPEG-2 Decoder	4.615 ms	5.418 ms
Optical Flow	5.013 ms	5.406 ms

One more thing need to be noticed is that with OSCAR compiler, even though using 3PE is faster than using 2PE, the power consumption in case of 2PE is better than that in case of 3PE. Table 1 shows the idle time of 2PE and 3PE until deadline. From this table, it is observed that there is no crucial difference in the idle time between 2PE and 3PE. Since there is one more PE used in case of 3PE, the power consumption in case of 2PE is lower than in case of 3PE. [17] also points out that for small applications, it is more efficient when using less number of cores while for large applications, it might be more efficient with a larger number of cores.

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 time efficiency as well as lower energy consumption. This report shows a realtime video demonstration system for parallelization and power reduction controlled by OSCAR Automatic Parallelization Compiler. With MPEG2 Decoder Application, in case of using 2PE, it can save 18.2% power consumption comparing with the case of using 1PE in ondemand governor. With Optical Flow Application, the best result is in case of 2PE with OSCAR Compiler Power Control, which saves 56.6% power consumption comparing with the case of 1PE, ondemand governor.

Reference

1) Louis Columbus "IDC: 87% Of Connected Devices Sales By 2017 Will Be Tablets And Smartphones" forbes.com 9 Dec. 2013. Mon. 30 May. 2014 <http://www.forbes.com/sites/louiscolumbus/2013/09/12/idc-87-of-connected-devices-by-2017-will-be-tablets-and-smartphones/>
2) Android platform http://developer.android.com/tools/revisions/platforms.html
3) 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
4) 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
5) OpenMP: http://openmp.org/wp/
6) 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
7) Obata, M., Shirako, J., Kaminaga, H., Ishizaka, K., Kasahara, H. “HierarchicalParallelism Control for Multigrain Parallel Processing” Lecture Notes in ComputerScience2481(2005) 31–44
8) Kimura, K., Mase, M., Mikami, H., Miyamoto, T., Shirako, J., Kasahara, H. “OSCAR API for Real-time Low-Power Multicores and Its Performance on Multicoresand SMP Servers” Lecture Notes in Computer Science (2010) 188–202
9) 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
10) 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)
11) Samsung Electronics Co., L.: White Paper of Exynos 5.1(1) (April 2011) 1–8
12) Hardkernel: ODROID-X2 http://www.hardkernel.com/renewal2011/products/prdtinfo.php?gcode=G135235611947
13) AsynTask: http://developer.android.com/reference/android/os/AsyncTask.html
14) The Ondemand Governor https://www.kernel.org/doc/ols/2006/ols2006v2-pages-223-238.pdf
15) 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
16) SAMSUNG ELECTRONICS: Samsung Semiconductors Global Site https://www.samsung.com/global/business/semiconductor/product/poweric/overview
17) Mikami, H., Kitaki S., Mase, M., Hayashi, A., Shimaoka, M., Kimura, K., Edahiro, M., Kasahara, H. “Evaluation of Power Consumption at Execution of Multiple Automatically Parallelized and Power Controlled Media Applications on the RP2 Low-power Multicore”, Proc. of LCPC 2011(The 24th International Workshop on Languages and Compilers for Parallel Computing), Colorado State University, Fort Collins, Colorado, Sept 8-10, 2011.

ⓒ 2014 Information Processing Society of Japan