2011 24th Annual Conference on VLSI Design

Improving Android performance and energy efficiency

Tapas Kumar Kundu
Dept. of Computer Science & Engineering
IIT Delhi
India
e-mail: tapascst@gmail.com

Prof. Kolin Paul
Dept. of Computer Science & Engineering
IIT Delhi
India
e-mail: kolin.paul@gmail.com

Abstract—**Mobile devices and embedded devices need more processing power but energy consumption should be less to save battery power. Google has released an open source platform Android for mobile devices. Android uses new power management framework to save power in mobile devices. Android developers are allowed to build only JAVA applications. In this work, we present benefits of using Android in low power embedded devices. We compared Android JAVA performance with popular Sun embedded JVM running on top of Angstrom linux. Our work shows that Android can be made more energy efficient by improving performance of JAVA applications. We developed a JAVA DSP framework which allows Android JAVA applications to use both ARM & DSP parallely and thus improves performance. We also showed, Android can be made more energy efficient by using our developed framework.**

I. Introduction

An embedded system is a special-purpose computer system designed to perform one or a few dedicated functions, often with real-time computing constraints. Embedded systems control many of the common devices in use today. A key component of such systems is the Operating Systems (OS) which is the interface between hardware and user, it is the critical component responsible for the management and coordination of activities and the sharing of the resources of the computer. Google has developed open source Android operating system for mobile platform which is also used in the net book. In this paper, we describe how we can improve performance and energy efficiency of Android.

In the next section, we briefly review essential parts of the experimental setup. In the Section III, we showed that Android can be made more energy efficient by improving performance. Section IV describes parallel computation framework. Section V & VI describes an example application using our framework and benchmark results. The last section concludes the paper with suggestion of how the energy consumed by applications running on Android can be improved by using our framework.

II. Review

A. Android DALVIK JVM

Android Dalvik JVM is optimized especially for slow CPU, relatively little RAM and to run on OS without any swap space. It is optimized for a system which has 64MB as total RAM. Dalvik is register based VM but Sun JVM is stack based. Generally stack based VM use push, pop instructions to load data on the stack and manipulate that data. Thus stack based VM requires more instructions than register based VM to implement same high level JAVA code. But instructions in a register based VM tends to be larger. Google hasn't released JIT compiler for Dalvik in the initial phase of our work. But Sun embedded JVM contains JIT compiler optimized for embedded platform.

B. Android Power Management Driver

Android power management support sits on top of linux kernel power management. Android implements more aggressive power management policy on top of the standard linux kernel power management. In Android, applications and services must request CPU resources with "wake locks" through the Android application framework and native linux libraries in order to keep power on. Each application informs power management framework its power requirements, which can be viewed as constraints for suspending system components. If there are no active wake locks then Android will suspend CPU and other peripheral devices (e.g. Keyboard).

C. Beagleboard

We used beagleboard as our embedded platform. This board uses up to 2W of power. We run both Android 1.6 (also known as Android Donut release) and Angstrom on this board. This board contains integrated power management/audio codec chip TPS65950 and current resistor R6 to help developer to measure board current through software. The values measured using software were not found to be accurate. For example, before starting any OS, beagleboard software measurement of current consumption varied as 268.55 mA, 214.843 mA, 161.133 mA etc on the uboot boot loader command prompt but our digital multimeter shows 290 mA always. Due to this fluctuating errors in software measurement of current values, we sampled the current values using Agilent 34410A digital multimeter in every 0.5 second. This was used with the board supply voltage to plot the power dissipated.

D. DSP BIOS OS

DSP/BIOS enables our applications to be structured as a collection of threads, each of which carries out a modularized function. Multi threaded programs run on a single processor by allowing higher-priority threads to preempt lower priority threads and by allowing various types of interaction between

1063-9667/11 $26.00 © 2011 IEEE
DOI 10.1109/VLSID.2011.63

threads, including blocking, communication, and synchronization. The thread types (from highest to lowest priority) are: hardware interrupt (HWI), software interrupt (SWI), Task thread (TSK), idle loop (IDL).

E. Dsplink Driver

Dsplink driver facilitates communication between DSP and ARM. The dsplink driver encapsulates low-level control operations on the physical link between ARM and DSP. This module is responsible for controlling the execution of the DSP and data transfer using defined protocol across the ARM-DSP boundary.

III. Analyzing Android Energy Framework

In this section, we discuss how we analyzed Android energy consumption rate and motivation for improving it using parallel framework.

A. Experimental Setup

We run quick sort, heap sort and Caffeine Mark JAVA applications on both Android and Angstrom linux. We also measured energy consumption of those applications. At the time of our work, Android didn't have JIT (Just In Time compiler) support. We considered following versions of quick sort for Android:

- JAVA implementations without JNI and
- Quick sort algorithm written in native C and a JAVA program calls that algorithm through JNI.

In Angstrom, we used four versions of Quick sort:

- Quick sort using Sun JAVA embedded JNI,
- JAVA implementation of Quick sort with JIT enabled,
- JAVA implementation of Quick sort with JIT disabled and
- Native C implementation of Quick sort.

Arrays of random integers were generated and sorted using Quick sort algorithms. Sun JVM uses enhanced hotspot technique to compile JAVA byte code into native code and compiles them by detecting which methods needs to be compiled. G. Chen et al. [4] showed how different number of method invocations affects JIT enabled JAVA compiler performance. We used same Heap sort algorithm to sort 60000 elements in 50 different methods and called these method different number of times to see how Sun JIT compiler affects

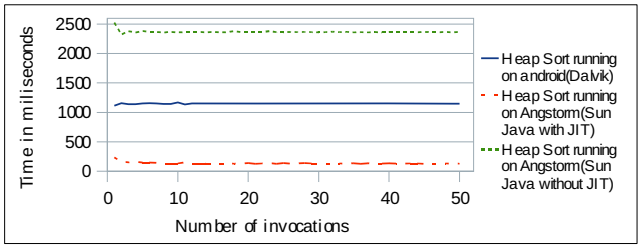

Figure 1: Heap sort performance

TABLE I. CaffeineMark v3.0 Score

	Android (*Dalvik*)	Angstrom *Sun JVM* (with *JIT*)	Angstrom *Sun JVM* (without *JIT*)
Sieve Score	945	3454	576
Loop Score	820	14073	456
Logic Score	929	7779	751
String Score	938	3623	771
Float Score	627	5180	460
Method Score	909	8897	512
Total Score	852	6310	574

TABLE II. Average Energy Consumption Rate

	Angstrom (*Sun embedded JVM*)	Android (*Dalvik JVM*)
Heap sort average energy consumption rate	1.75 watt	1.72 watt
Quick sort average energy consumption rate	1.76 watt	1.73 watt
CaffeineMark v3.0 average energy consumption rate	1.74 watt	1.72 watt

TABLE III. Total Energy Consumption

	Angstrom (*Sun embedded JVM*)	Android (*Dalvik JVM*)
Heap sort total energy consumption	681.64 watt-sec	5064.79 watt-sec
Quick sort total energy consumption	10.56 watt-sec	31.07 watt-sec
CaffeineMark v3.0 total energy consumption	85.18 watt-sec	86.23 watt-sec

on different number of invocations. We used CaffeineMark v3.0 benchmark to benchmark both Dalvik and Sun embedded JVM. CaffeineMark scores roughly correlate with the number of JAVA instructions executed per second, and don't depend significantly on the the amount of memory in the system or on the speed of a computers disk drives or internet connection.

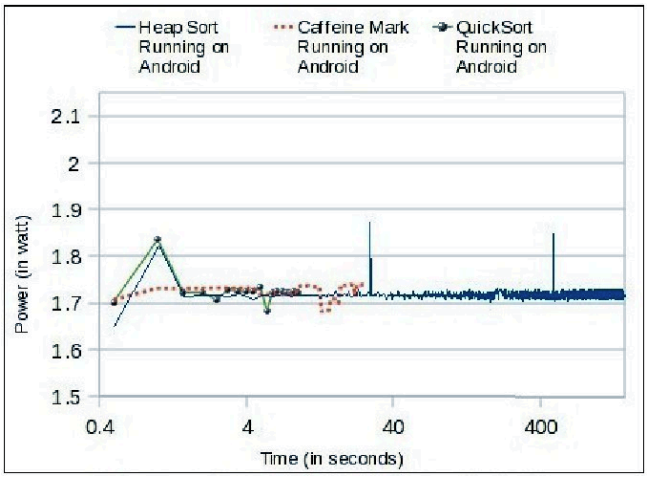

Figure 2(a): Energy consumption on Android

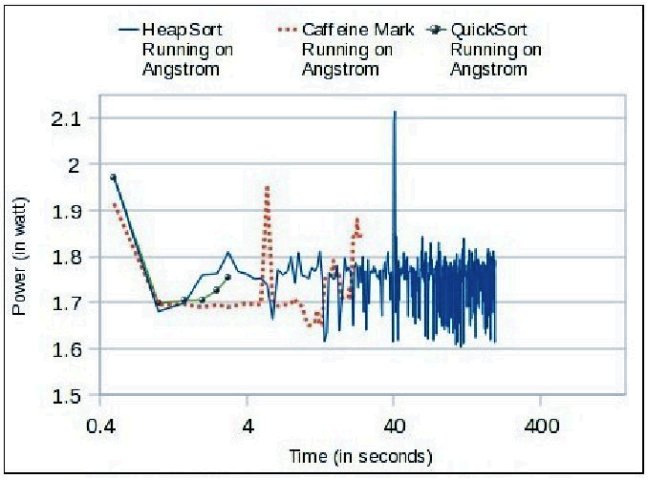

Figure 2(b): Energy consumption on Angstrom

B. *Results*

Android Dalvik VM shows better performance compared to JIT-disabled Sun Embedded JVM running on Angstrom (see Figure-1 and 3). JNI based quick sort also run faster in Android than Angstrom. But JIT enabled Sun embedded JVM performs much better than Android's Dalvik VM for both applications (see Figure-1 and 3). Using register based VM architecture and bionic libc native libraries, Android gets 1.484 times improvement over commercial Sun embedded JVM (with JIT disabled) in CaffeineMark benchmark (see Table-I) total score. In all three cases, Android becomes slower than Sun embedded JVM with JIT. Table-II shows that in all three cases, Android is more power efficient because its power driver saves power on application basis (as explained earlier). Figure-2(a), Figure-2(b), Table-III shows, Android consumes more energy due to longer running duration of JAVA applications: Quick sort and Heap sort. But in case of CaffeineMark, JIT enabled Sun JVM doesn't affects its running time much. So, it consumes almost same energy in Android and Angstrom linux (see Table-III).

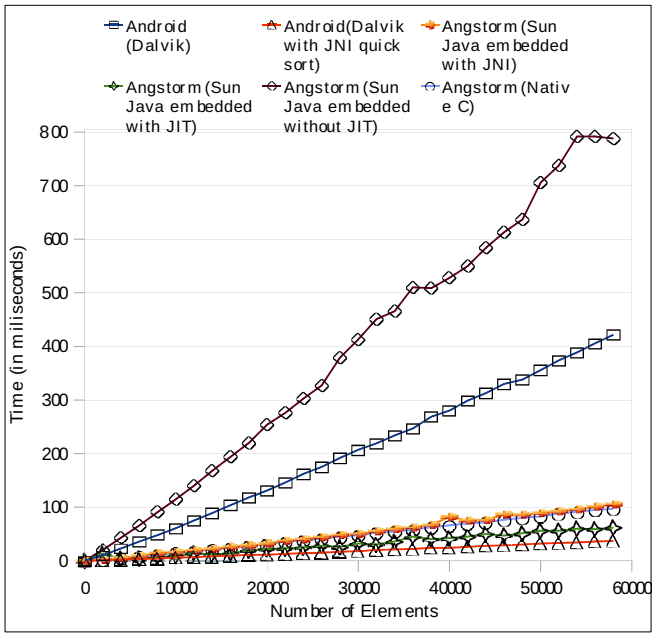

Figure 3: Quick sort performance

C. *Motivation For Parallel Computation Framework*

So, our analysis shows that Android application can be made more energy efficient by improving performance. Google implemented JIT compilers in later version of Android to improve performance. Borkar et al. [2] suggested that a perfect two-way parallelization would lead to one-quarter of the energy consumption compared with the sequential execution given the same execution time constraint. Sangyeun et al. [3] developed an analytical framework to study the trade offs between parallelization, performance and energy consumption for applications. A program has both sequential section and parallel section. They showed that energy consumption can be reduced by using maximum number of processors to execute parallel section. They also proved that energy can be further improved by turning off individual processors (when they don't require to execute any code). Beagleboard has both ARM and DSP processor. They have different clock speed and DSP processor can be shut down if not required. In next section, we showed how we improved performance and energy efficiency for a class of Android JAVA applications using our parallel computation framework.

IV. A Parallel Computation Framework

A. *Porting Dsplink Driver On Android*

We cross compiled TI's dsplink 1.61.03 driver & local power manager driver for beagleboard using Android Tool Chain. Local power manager helps to shutdown DSP processor after completion of Android JAVA Application. The default dsplink memory map for any platform usually assumes the following: 1 MB shared memory between ARM and DSP and 1 MB DSP memory for DSP code & data. General JAVA applications may need more memory than 1 MB. So, we modified that memory map and reserved 30MB memory as shared memory between ARM and DSP. We passed an

initialization parameter to Android linux kernel so that it doesn't manage this reserved memory segment.

B. JAVA DSP API

A typical JAVA application which uses DSP, has two parts: ARM side JAVA application and DSP side application. ARM side JAVA application performs following tasks:

- Initializing dsplink driver components: JAVA application developer creates an array of pool memory buffers with different sizes from ARM-DSP shared memory region. These pool memory buffers are used to allocate shared JAVA data objects in shared memory region. JAVA application loads DSP executables on DSP and starts execution of DSP processor.
- Sharing data with DSP side application: Dalvik VM allocates JAVA data objects from JAVA heap memory which doesn't reside in ARM-DSP shared memory. We have given two options to JAVA developer to share JAVA data objects with DSP. One is to copy JAVA primitive data types (e.g. JAVA integer array, JAVA string object, JAVA float array) from JAVA heap to ARM-DSP shared memory region and informing DSP that. Another method is to directly allocate JAVA data objects from ARM-DSP shared memory region. Both options has some advantages and disadvantages. Developers need to choose both methods judiciously to get required speed up.
- Synchronization between ARM and DSP: To parallely work on both ARM and DSP, ARM side JAVA application must maintain at least 2 JAVA threads. One thread does all the initialization of dsplink and synchronizes with DSP side application as required. Another JAVA thread uses ARM processor to work on shared data objects. Both thread can synchronizes with each other through "JAVA thread synchronization" methods. Synchronization between 1st thread of ARM side JAVA application and DSP side application can be done by passing messages. ARM side JAVA application will issue a message by calling JAVA_MSGQ_put() method. DSP side application's TSK thread can block itself till it gets that message by calling MSGQ_get() method. Similarly, ARM side JAVA application can block itself till it receives messages from DSP by calling JAVA_MSGQ_get() method.
- DSP Debugging support: We provided a logging mechanism which helps developer to print any values in Android logging daemon directly from DSP side application.

V. Benchmarkng Thje Parallel Framework

In this section, we describe how we benchmarked our parallel computation framework. We use image filter JAVA application to benchmark our framework. Image filtering allows us to apply various affects on photos. We use sequential image filter code found in simple scaler benchmark suite[1] for image filter application. In simple scaler benchmark suite, image filter application known as CPU intensive program.

A. Parallelization of Sequential Image Filter Algorithm:

1. *Create input image matrix (imageWdith x imageHeight) and filter matrix (filterWidth x filterHeight).*
2. *Initialize input image matrix and filter matrix.*
3. **For** each pixel in input image matrix
   begin
4. **For** each element in filter *matrix*
   begin
5. *calculate result image pixel value by reading input image and filter values.*
   end
6. *Store calculated value in result image matrix*

end

Steps-1 to 2 are same in both sequential and parallel version. Two threads calculate result image matrix parallely from step-3 to step-6. Each thread calculates result matrix for half of the image size.

B. Issues For Parallel Implementation

Our JAVA DSP API needs to initialize dsplink components before using it. In JAVA, this can be done by using two threads. One JAVA threads always uses ARM processor (let's call it "JAVA ARM thread"). It initializes input image & filter matrix and calculates half of result image matrix. Another thread initializes dsplink components and transfers controls to DSP processor which calculates another half of result image matrix (which is not calculated by ARM).let's call second thread as "JAVA DSP thread". Here, both processors need to know whole input image matrix and filter matrix to calculate result image matrix. We considered two sizes for filter matrix in different versions of our image filter: one is 16x16 and another is 32x32. For filter size >=16x16, steps-3 to 5 needs to read whole image matrix many times and thus need to access image matrix and filter matrix more efficiently. If we allocate whole image matrix from ARM-DSP shared memory region then steps-3 to 5 will become very slow for ARM processor (assuming filter matrix has a size greater than or equal to 16x16) but it will be fast for DSP processor. Reason behind this is, JAVA implementation of bytebuffer class is slow. Solution is to allocate these image matrices & filter matrix from JAVA heap and copy these from JAVA heap to ARM-DSP shared memory region so that DSP can access that. In this way, ARM and DSP both work fast in steps-3 to 5. Sequential algorithm stores result image matrix in steps-6. These statements are executed only once for each pixel values in input image. So, these statements are less computation intensive. So, result image matrix can be directly allocated from ARM-DSP shared memory region. This helps

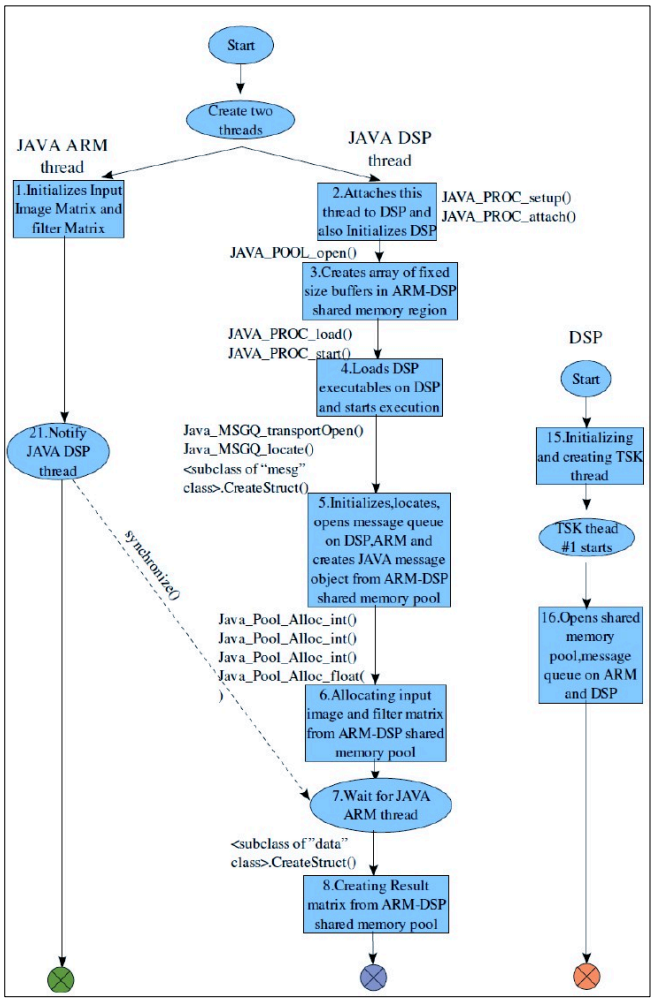

Figure 4: Parallel Implementation Of Image Filter (continued)

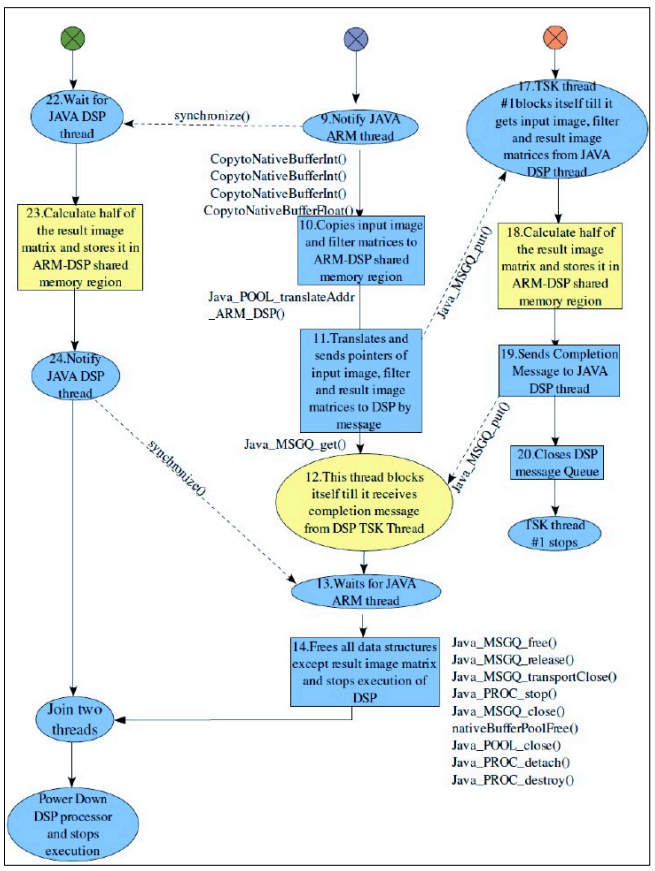

Figure 4: Parallel Implementation Of Image Filter

us not to copy half of result image matrix from ARM-DSP shared memory to JAVA heap.

C. Implementing Parallel Algorithm

Android system server starts image filter JAVA application on ARM processor. JAVA application initializes and loads DSP executable on DSP.

- DSP side application:
  - It initializes dsplink driver and creates a TSK thread.
  - TSK thread waits to get input image matrix, filter matrix, result image matrix pointer from ARM side JAVA DSP thread.
  - TSK thread computes half of the result matrix & stores it in result matrix.
  - TSK thread notifies ARM side JAVA DSP thread about completion of task.
- ARM Side JAVA Application: ARM side JAVA application creates two threads (Figures-4). JAVA ARM thread calculates half of result matrix and waits for DSP processor to finish. JAVA DSP thread initializes various components and asks DSP TSK thread (which runs on DSP) to compute result matrix. JAVA DSP thread loads DSP executable on DSP and start execution of DSP processor in step-4. DSP executables main function creates a TSK thread in step-15. JAVA DSP thread locates DSP message queue identified by a known string and opens ARM side message queue (step-5). DSP message queue helps to send message to DSP and ARM message queue helps to receive message from DSP. DSP TSK thread also opens two message queues, one to send message to JAVA DSP thread and another to receive message from JAVA DSP thread (step-16). JAVA DSP thread allocates buffers for input image, filter and result image matrices in ARM shared memory (step-6 & 8). It copies input image and filter matrices to ARM-DSP shared memory and translates pointer addresses of all 3 matrices (step-10 & 11). It sends those addresses to DSP TSK thread by message protocol and ask DSP TSK thread to calculate half of result image matrix(step-11 & 17). After calculating half of result image matrix, DSP TSK thread informs JAVA DSP thread (step-12 & 19). JAVA DSP thread informs JAVA ARM thread about completion of DSP processing (step-13 & 24). Thus, JAVA imagefilter uses both ARM and DSP parallely to get speed up.

VI. Results

We varied input image size from 64x64 pixels to 512x512 pixels and measured running time & energy consumption for image filter application (see Figures-5 and 6). We get two pairs of graph for filter size 16x16 pixels and 32x32 pixels. Each pair of graphs shows a comparison between sequential and parallel implementation. For filter size 16x16 pixels and image size 64x64 pixels (Figure-6), we get 1.098 times speed up which is very negligible. This happens because, it takes some constant time to start and initialize (known as initialization time) dsplink driver data structures and also takes some constant time to stop DSP after completion of DSP TSK thread (known as deletion time). A program has both sequential and parallel sections. let's consider, a class of CPU intensive programs, which will take longer time to execute parallel section than this initialization & deletion time of dsplink driver. This class of programs will offset this constant time and will achieve better speed up. As we increases image size and filter size, image filter spends more time in parallel section than in initialization and deletion time of dsplink driver. For large image size (512x512 pixels) and large filter size (32x32) , we see good speed up (1.586 times) from Figure-6. With increase of image size and filter size, parallel implementation of image filter shows better energy efficiency than sequential implementation (Figure-5). By comparing Figure-6 with Figure-5, we see parallel implementation of image filter application shows better energy efficiency where it achieves good speedup in execution time. As an example: for filter size 32x32 pixels and image size 512x512 pixels, it consumes 23% less energy while running on both ARM & DSP. (It consumes 1408.14 watt-sec on ARM and 1085.44 watt-sec on both ARM & DSP).

VII. Conclusion

Not all application can take benefit of parallelism. Some application needs more than two processors to improve performance. Moreover, ARM & DSP has different clock speed and different cache size. Our analysis shows that a class of Android applications can take benefit of using our developed framework. This class of Android applications must be CPU intensive and should be capable to achieve significant speed up from two processors theoretically. The more we get speed up, the more we will improve energy efficiency of Android applications. Thus, intelligent use of our framework will improve energy consumption of Android.

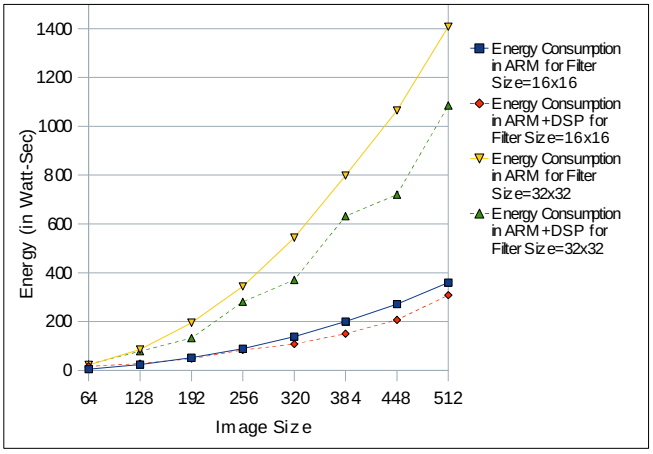

Figure 5: Energy Consumption of Image Filter

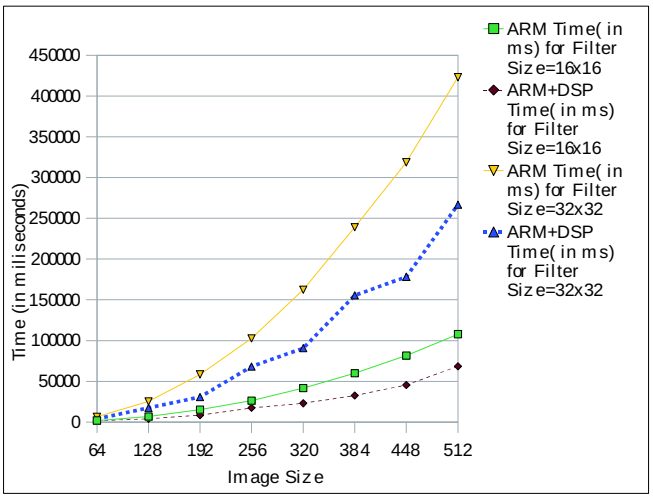

Figure 6: Performance of Image Filter

References

[1] T. Austin, E. Larson, and D. Ernst. “simple scalar: An infrastructure for computer system modeling.” Computer, 35:59–67, 2002.

[2] S. Borkar. “Microarchitecture and design challenges for gigascale integration.” In MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture, pages 3–3, Washington, DC, USA, 2004. IEEE Computer Society.

[3] S. Cho and R. G. Melhem. “On the interplay of parallelization, program performance, and energy consumption.” IEEE Trans. Parallel Distrib. yst., 21(3):342–353, 2010.

[4] G. Chen, M. Kandemir, N. Vijaykrishnan, and M. Irwin. “Pennbench: a benchmark suite for embedded java.” In Workload Characterization, 2002. WWC-5. 2002 IEEE International Workshop on, pages 71 – 80, 25 2002.