Automatic Parallelization, Performance Predictability and Power Control for Mobile-Applications

Dominic Hillenbrand, Akihiro Hayashi, Hideo Yamamoto, Keiji Kimura, Hironori Kasahara

Green Computing Systems Research Organization
Waseda University - Tokyo, Japan - Tel./Fax. 81-3-3203- 4485/4523
{dominic, ahayashi, hyamamoto, kimura}@kasahara.cs.waseda.ac.jp, kasahara@waseda.jp

Abstract
Currently few mobile applications exploit the power- and performance capabilities of multi-core architectures. As the number of cores increases, the challenges become more pressing. We picked three challenges: application parallelization, performance-predictability/portability and power control for mobile devices. We tackled the challenges with our auto-parallelizing compiler and operating system enhancements.

1 Introduction
Mobile devices such as smart-phones and tablet PCs have become prevalent. Several mobile SoC (system-on-chip) vendors compete for the lowest power consumption and highest performance. Heterogeneous multi-core architectures allow operating systems to choose between cores that have different performance- and power characteristics. However current process schedulers have been designed with symmetric architectures in mind. Therefore SoC vendors - currently - experiment with custom kernel modifications: NVIDIA “hot-plug” [4] - for example - takes cores on- and off-line during run-time. ARM experimented with hypervisors [5] that automatically switches between pairs of cores so that operating systems perceive symmetric multi-core architectures. For Android an estimated 700,000 applications [3] are available. Few applications take advantage of multi-core architectures. Tools like OpenCL, Intel TBB, Cilk plus, OpenMP and OpenACC - for example - are often not supported on mobile operating systems. On Android - for example - only rudimentary support for the low-level pthreads-library is available to native applications. In this paper we have identified three challenges for mobile platforms: (1) parallelization, (2) performance portability and predictability and (3) power control.

Our contribution is a methodology that tackles these three challenges for some applications: For (1) we use our own auto-parallelizing compiler OSCAR [1]. For (2) and (3) we have experimentally extended and modified the Android kernel. The Android kernel is used in Android itself and also for Tizen, Firefox OS, Chrome OS, WebOS and Ubuntu for mobile. Figures 1, 3a and 3b illustrate that OS-level power control and -scheduling are not suitable in certain situations. The parallelized application in Figure 3a experiences unpredictable execution times. The execution profiles of this application are shown in Figure 1 and 2. The “white gaps” in the execution profile in Figure 4b indicate slack that could have been avoided - if scheduling and DVFS had been applied properly. Figure 3b illustrates OS-level DVFS-control lag and -oscillations. Figure 4a shows the significant latency-reductions of our user-space- and improved kernel interface for DVFS. In the following sections we discuss our approach.

2 Application Parallelization
Our group has developed an auto-parallelizing compiler - called OSCAR [1]. OSCAR accepts sequential C-code as input and generates parallelized C-code with OSCAR API as output. Therefore programmers do not need to bother with intricacies of new software tools and parallel programming. The parallelized C-code output can be compiled with platform specific sequential C-compilers. These platform specific compilers may further take advantage of VLIW- or vector-instructions. OSCAR controls cache-placement, data movements - between processors and accelerators and power. Especially, memory management is crucial for scalability on architectures with hundreds of cores. In the following section we explain how we can maintain performance-portability and -predictability.

3 Performance- portability and predictability
On embedded real-time operating systems applications are isolated and have strong performance guarantees. As we have seen - in Figure 3a - this is not the case on mobile operating systems. Our Android kernel exclusively assigns cores to applications via a new API. Our scheduler modifications were minimal. We
avoid task-creation and -migration on reserved cores. Furthermore, we keep interrupts and kernel threads off reserved cores. During measurements a video player was active in full-screen mode to simulate device usage and disturbances. We executed our benchmarks in the background. Figure 2 shows that our kernel has dedicated cores 1–3 to our OSCAR-parallelized AAC-encoder. All other user- and system processes run on the remaining core 0. Over 20 runs we see execution times vary between 2.71–2.74s on our modified kernel. On the original kernel execution times vary between 3.1–136.82s (2.72-110.36s with thread binding) - see Figure 3a. Occasionally, the vendor “optimized” automatic OS-level power management and -scheduling leads to application-thread synchronization issues. In the following section we discuss the advantages of application-level power control.

4 Application Power Control

Our OSCAR compiler automatically generates power control commands for DVFS, power- and clock gating by utilizing the results of source-program analysis [1]. Manual - fine-granular - application-level power control would take months of profiling and testing to achieve comparable results. On embedded operating systems applications can directly access hardware registers and privileged instructions. However, on mobile operating systems the kernel separates applications from hardware. Clock gating on our research processor RPX [6] - see Figure 4b - takes only 0.002µs, a system-call 3µs and kernel overheads account for more than 20µs. Thus low-latency clock gating cannot be exploited from applications without further hard- and software modifications. DVFS is less time critical than clock gating. To enable low-latency application-level power control we ported DVFS kernel-drivers into user-space. The drivers become part of the application and can directly access memory-mapped hardware registers. In addition to the user-space driver we have introduced a new - very portable - DVFS kernel interface that bypasses the sysfs-file layer [2]. Our OSCAR-compiler supports all presented methods. In the following section we conclude our paper.

5 Conclusion

The OSCAR-compiled AAC-encoder finished within 4±1.27s on the original kernel with thread pinning and excluding one outlier. In the default mode - without thread pinning - execution times are frequently above 10s, sometimes even exceeding hundreds. On our modified kernel - in contrast - AAC finishes within 2.72±0.02s. For 3 threads the speed-up of the OSCAR parallelized AAC encoder is 2.83x. Thus OSCAR in combination with our kernel enhancements is a viable approach for exploiting multi-core architectures on mobile devices. By porting our kernel DVFS-drivers into user-space we could achieve a 155x latency reduction on RPX/Linux. Our new DVFS-kernel interface achieved a 28x latency reduction and is portable across different architectures - see Figure 4a. OSCAR can generate task- and power-control schedules. Therefore OS-power- and scheduling-control is not required - but can even be harmful as our results show. Unfortunately, processors are not designed to provide such support in user-space. Furthermore, some SoCs support only one frequency- and voltage for all cores. Our compiler - however - can choose optimal frequencies and voltages for each core. Application level power- and scheduling-control may also be useful for applications which are not compiled by OSCAR. Therefore, it would be beneficial if such APIs would be standardized.

References

Fig. 1. This thread execution profile shows our ACC-encoder running on the Nexus-7 with the default Android kernel 3.1.10. The compiler was configured to parallelize AAC for 3 cores. The purple colored blocks indicate execution of AAC. Frequently, AAC is interrupted by other applications and system-tasks. Although AAC spawns 3 threads only cores 0 and 2 are utilized, core 3 is idling and core 4 is shut-off. In this particular run the execution time was >100s, instead of roughly 2.7s. The reason is that 2 threads must share one core and burn CPU time in spin-locks. Thus the OS-scheduler in the Nexus-7 fails to efficiently map threads to cores.

Fig. 2. Behavior of our enhanced kernel: The OSCAR-compiled AAC-encoder runs on cores 1–3 without interferences - all other tasks, e.g. movie player, are on core 0. Our enhanced kernel temporarily provides exclusive CPU-core access to applications. Figure 1 shows the behavior of the original kernel.

Fig. 3. Behavior of OS level scheduling and power control: (a) Execution Time Variability (b) Power Profile of two DVFS governors

Fig. 4. DVFS Latencies (a) and Non-OSCAR compiled application (b)