Exploiting Asymmetric CPU Performance for Fast Startup of Subsystem in Mobile Smart Devices

Jun Kim, Joonwon Lee, and Jinkyu Jeong

Abstract—*Fast startup of a subsystem, such as an image subsystem, is important in mobile smart devices because the startup time is directly related to the responsiveness of applications, such as a camera. This paper proposes a CPU switch method that offloads the long initialization phase from the subsystem to the fast main CPU in an application processor. By providing a consistent execution environment in the main CPU, the subsystem software can be easily ported without significant engineering costs. The evaluation results show that the proposed scheme greatly reduces the startup time of the image subsystem by up to 78%.*[1]

Index Terms — Subsystem startup, fast startup, snapshot, CPU switch.

I. Introduction

With the tremendous growth of the smartphone market, smartphone shipments surpassed feature phone shipments in the first quarter of 2013 [1]. In such a competitive market, many smartphone vendors strive to include in their phones more attractive and higher performance functions, such as high-resolution cameras, multimedia codecs, and 3D graphics, to increase their market share. To support these increasingly diverse functions, the application processor (AP) in smartphones has become more versatile and complicated.

An AP is a primary chipset in a mobile smart device. It consists of a main CPU, which performs general computations, and several hardware blocks (or intellectual properties (IPs)), which provide hardware-accelerated special-purpose computations. Well-known hardware blocks included in an AP are a 2D/3D graphics processor, image signal processor (ISP), multimedia codecs, and external interfaces. When a mobile device needs to support more sophisticated functions, the functionality of an IP is improved, or additional IPs are integrated in the system.

Owing to the increasing number of IPs in an AP, a subsystem is used to ease manageability of the IPs. A single IP is typically designed to provide a low-level function (e.g., noise reduction, image stabilization, or image resizing), whereas a group of IPs provides a high-level function (e.g., cameras or multimedia codecs). When the number of IPs increases, it is difficult to timely manage multiple IPs in the main CPU because of software contention. To cope with this problem, a control unit (or a sub CPU) and several IPs are brought together to comprise a subsystem. The sub CPU solely handles controlling of the IPs involved in the subsystem. Many modern APs have multiple subsystems, such as an image subsystem and a multimedia codec subsystem.

A subsystem is a small computer system; it runs its own software stack that controls the IPs within the subsystem. This software stack is initialized when an end user requests the operation of the subsystem, such as the launching of a camera application. A problem arises when a subsystem provides more versatile functions because the complexity of its software stack likewise increases. This increasing complexity results in a long initialization time of the subsystem. The long delay directly degrades the responsiveness of applications using subsystems thereby aggravating user experience [2], [3].

A snapshot startup scheme [4] is a well-known approach to speeding up the boot time of systems [5], [6] and subsystems [7]. A snapshot boot scheme provides fast booting of a computer system by exploiting system suspend/resume. In addition, it can be applied to a subsystem because a subsystem is yet another computer system. Previous work in exploiting the snapshot boot, however, either lacks support of all hardware IPs in a subsystem [5] or requires labor-intensive engineering to take snapshots of various IPs [7]. To restore the state of IPs, careful consideration must be given to the characteristics of the register files in the IPs; obtaining snapshots of such IPs requires considerable engineering effort. Moreover, such labor-intensive obtaining of snapshots cannot be easily applied to different types or revisions of subsystems.

This paper presents a CPU switch startup method that accelerates the subsystem startup by offloading the startup phase of the subsystem to the main CPU in the same AP. The main

[1] This research was supported partly by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2014R1A1A2054658) and partly by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (NRF-2014R1A2A1A10049626).

Jun Kim is with the Department of Semiconductor Display Engineering, Sungkyunkwan University, Suwon, South Korea (e-mail: jun7.kim@gmail.com).

Joonwon Lee is with the College of Information and Communications Engineering, Sungkyunkwan University, Suwon, South Korea (e-mail: joonwon@skku.edu).

Jinkyu Jeong is with the College of Information and Communications Engineering, Sungkyunkwan University, Suwon, South Korea (e-mail: jinkyu@skku.edu).

Contributed Paper
Manuscript received 12/30/14
Current version published 03/30/15
Electronic version published 03/30/15.

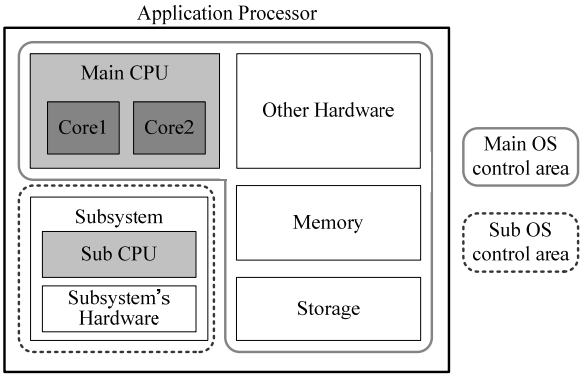

Fig. 1. The architecture diagram of an AP including a subsystem.

CPU typically shows better performance than a sub CPU by exploiting more advanced hardware features, such as larger branch prediction buffers, multiple issue out-of-order execution, larger internal cache memories, higher clock frequencies, and so forth. In addition, both CPUs have the same instruction set architecture; therefore, the initialization software binary of a subsystem can run on the main CPU without any recompilation. To provide the same execution environment, including hardware IPs and memory, the memory management unit (MMU) and interrupt controller unit (ICU) of the main CPU are carefully configured while the subsystem software is running on the main CPU. These additional operations, however, are not dependent on the types or revisions of the subsystem; therefore, they do not affect the portability of the proposed scheme.

Based on the proposed scheme, the prototype was implemented to an image subsystem and evaluated on a smartphone development board. The evaluation results showed that the proposed scheme reduced the startup time of the image subsystem by 79%. In addition, the CPU switch method provides portability of the proposed scheme because it requires minimal device-dependent engineering effort.

The remainder of this paper is organized as follows. In Section II, the background, related work, and motivation of this work are presented. Section III outlines the design and implementation of the proposed scheme. Section IV presents the evaluation results of the proposed scheme. Section V discusses the portability of the proposed scheme to other systems. The conclusions are provided in Section VI.

II. Background and Related Work

A. Subsystem

A subsystem is a group of integrated IPs, each of which provides a low-level single function, and provides one high-level function. Individual execution of each IP is generally useless; however, several IPs operating together provide one high-level function. For example, a hardware codec has a direct memory access (DMA) IP, which reads/writes data from/to memory, an encoding IP, and a decoding IP. The encoding/decoding IPs cannot operate without input data delivered by the DMA IP. In this regard, an application does not separately manage each IP; rather, it interacts with all of the IPs as a hardware codec concept.

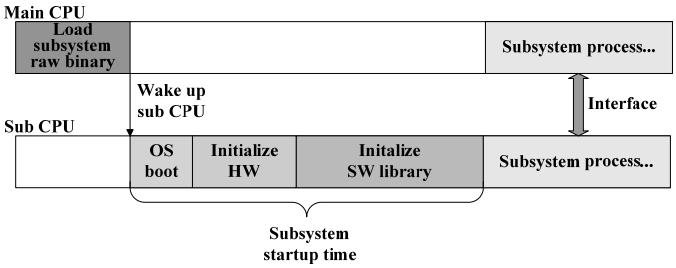

Fig. 2. The startup sequence of a subsystem.

In a subsystem, the sub CPU is integrated to manage several IPs within the subsystem. Device drivers for each IP request I/O operations from/to the IPs and handle interrupts from the IPs. In mobile smart devices, when the device drivers run on the main CPU in an AP, requests/responses to the IPs cannot be timely handled on account of the contention in the main CPU. This is because the main CPU is shared with other software components and applications. When this real-time aspect is not guaranteed, for example, a video playback workload can lose several video frames. Consequently, many subsystems include a low-specification CPU (or a sub CPU) to isolate the controlling domain from the shared main CPU [8], [9]. An AP with multiple subsystems, including multiple sub CPUs, is typically denoted as a multiprocessor system-on-chip (MPSoC) [8].

A subsystem containing a sub CPU is similar to an independent computer system. The sub CPU has memory, which is a part of the main system memory, and IPs are attached as devices. In addition, a subsystem includes dedicated software (e.g., firmware) that executes on the sub CPU to control the IPs in the subsystem. The software runs independently from the main system software. In the case of a complex subsystem, dedicated software includes a simple real-time operating system (RTOS) that manages several tasks while preserving the real-time aspect of handling them. Therefore, when an AP has one subsystem, two operating systems (OSs) run on the AP, as shown in Fig. 1. The main OS is typically a general purpose OS, such as Linux, and a subsystem runs a simple RTOS or firmware-level software.

When the function provided by a subsystem is needed in the main system, the main CPU invokes the initialization of the desired subsystem. The startup of the subsystem, depicted in Fig. 2, begins with the loading of the subsystem software binary at a specific memory address [10]. Then, the main CPU wakes the sub CPU to initiate the subsystem initialization and waits until the initialization completes. During the initialization phase, the sub CPU boots its own OS and then initializes the hardware (i.e., IPs) and software. After the initialization is complete, the subsystem notifies the main system of that the subsystem is ready to function. Finally, the main system continues to the next step, such as launching an application. The initialization phase of a subsystem is similar to the boot sequence of a typical computer system, although the startup repetitively occurs whenever the main system requires a subsystem function.

The startup time of the subsystem refers to the time period between the waking of a sub CPU and the completion of all

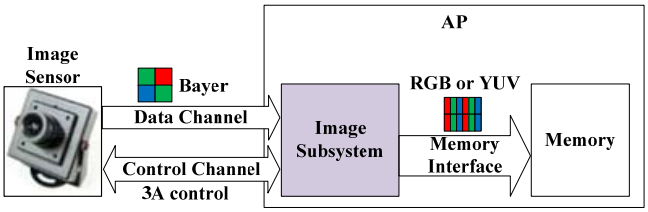

Fig. 3. The overview of an image subsystem.

initialization tasks. The startup time of the subsystem depends on the number and type of hardware and software components within the subsystem. If the subsystem has many IPs to be initialized, the hardware initialization time increases. If the subsystem has complex software libraries, its software initialization time increases. Because smart device vendors are making their APs more versatile and complex, the APs have more complex hardware and software. This results in an increase in the subsystem startup time thereby degrading the responsiveness of applications requiring subsystems.

B. Image Subsystem

An image subsystem is a representative example of a complex subsystem in an AP. An image subsystem performs a camera function in mobile smart devices. Basically, it receives raw image data, called Bayer data, from an external image sensor and converts the data to well-known image formats, such as the YUV or RGB format, as depicted in Fig. 3.

More specifically, when a picture-taking command is issued, an external image sensor captures raw image data. Then, several IPs, such as noise reduction, image stabilization, and image resizing IPs, post-process the raw data to generate Bayer data. From that point, software libraries within the subsystem perform other post-processing algorithms. Well-known post-processing algorithms include those for auto focus, auto exposure, and auto white balance, which are referred to as the 3As. Finally, the processed image data is passed to the camera application in the main system.

One of the main roles of a sub CPU in the image subsystem is to control hardware IPs and software algorithms. Because image capturing is performed in real-time, the sub CPU should schedule each resource at precise intervals. For example, when the application takes a video, the picture that comes from the image sensor should be processed at 30 to 60 frames per second. If the sub CPU fails to timely schedule one of the resources, several frames will be lost and the video quality for end users will be degraded.

The other role of the sub CPU is to initialize the software and hardware within the subsystem, as described in the previous subsection. Devices that the sub CPU initializes include not only internal IPs but also some external devices, such as an external image sensor. The software libraries for the 3As are also initialized during the software initialization stage. All of these initialization phases repetitively occur when the main system requests the camera device to function. Accordingly, the startup (or initialization) time of a subsystem is important from the perspective of end user latency.

The performance of the sub CPU is an important factor for the subsystem startup time because it processes the initialization of all components within the subsystem. The latency to initialize hardware depends on the specific hardware used in the subsystem. However, the latency in initializing the subsystem software is entirely dependent on the performance of the sub CPU. The hardware specification of a sub CPU is, however, determined so that it can timely handle the IPs and software algorithms. The initialization time is excluded from consideration. Consequently, the performance of the sub CPU is typically lower than that of the main CPU. For example, the main CPU is capable of superscalar out-of-order execution with many pipeline stages; moreover, it contains large branch prediction buffers. On the other hand, the sub CPU is capable of only in-order execution with a few pipeline stages; it contains small branch buffers. When the performance of the sub CPU is sufficient to execute the desired functions (i.e., 3As and I/O request/response handling), vendors have no reason to install a more powerful CPU, which would increase the unit cost and power consumption of the system.

C. Related Work

Considering that the startup sequence of the subsystem is similar to that of a conventional computer system, the research most closest aligned to improving subsystem startup time involves fast booting approaches. Bird [11], for example, proposed a number of methods for reducing the startup time of a commodity OS. He analyzed the boot sequence and proposed several methods, such as eliminating probe delay, real-time clock (RTC) synchronization, execution in place (XiP), and avoiding access to a slow serial device by disabling debugging message outputs. These optimizations, however, cannot be applied to the subsystem OS because they are specific to the Linux kernel. A subsystem typically has a very lightweight kernel; such features are already applied or cannot be applied because of the absence of the functions.

The snapshot boot method proposed by Kaminaga [4] exploits the suspend/resume technique of commodity OSs to reduce system boot time. This method generates a snapshot binary once; the snapshot binary is then used for every boot of the system. The software initialization phase includes global structure initialization, object creation, application initialization, and process generation. These are the most time-consuming tasks in the booting time of a complex system. The snapshot boot reduces much of the startup time by skipping these software initialization phases.

The snapshot boot method is applied to many consumer electronics. Jo *et al* [5], for example, exploited the snapshot boot method for a digital television platform. In that study, restoring the state of the hardware IP was difficult during the resume of the system when the IP had registers that changed its value while reading them. Accordingly, after reading the registers to generate the snapshot, the register states differed from the values stored in the snapshot. This value inconsistency was an obstacle to applying the snapshot boot

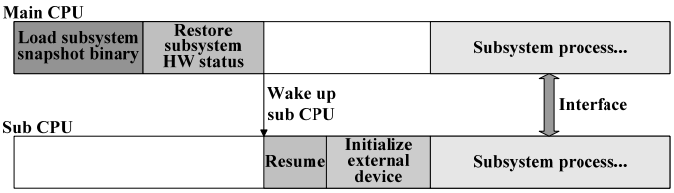

Fig. 4. The startup sequence of a subsystem using snapshot startup method [12].

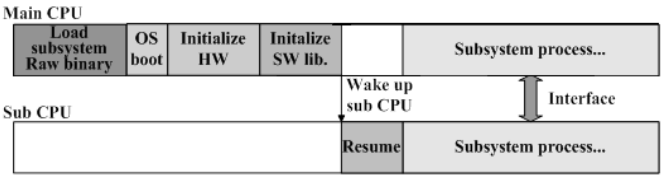

Fig. 5. The startup sequence of a subsystem using the CPU switch startup method.

method to such a system. The solution was to reinitialize the given IPs by again invoking the device initialization phase.

Baik *et al* [6] applied the snapshot method to a commodity OS-based smartphone. In that study, to reduce the footprint of the snapshot binary, unused memory pages were swapped out or reclaimed while taking the snapshot. However, some devices in a smartphone do not support hibernation. Their solution to this problem is to exploit the suspend mode of each device. When a device is suspended, its resume data is stored in memory. Accordingly, the snapshot binary can include the resume data of each device. To initialize these devices during the snapshot boot, the devices are initialized to the suspend state first; the suspended data is then restored to each device.

The main difficulty of applying the snapshot boot method to the subsystem is the complexity of restoring the states of hardware IPs. The device drivers in the subsystem OS typically do not support suspend or hibernation. Accordingly, reading and restoring the states of IPs are complex tasks that require careful consideration of various characteristics of state registers of IPs.

Kim *et al* [7] applied the snapshot boot method to an image processing subsystem in a mobile smart device. The snapshot boot sequence itself is much similar to other approaches, as shown in Fig. 4. For this approach, however, proper methods were suggested to take snapshots of IPs with various characteristics of state restorations. The types of registers targeted were read-only registers, write-clear registers, registers with initialization dependencies, and registers linked to the internal buffer of an IP. The drawback of this approach is that it requires significant engineering effort. An IP generally has thousands of state registers. Applying different read/restore methods to thousands of registers in several IPs incurs high engineering costs.

D. Motivation

Previous work regarding fast system startup that includes a subsystem has primarily focused on applying the snapshot boot method. These approaches, however, have several drawbacks. When device drivers for some IPs do not provide suspend/resume operations, the devices must be again initialized during the snapshot boot phase. This again increases the startup time, which can offset the benefit of the snapshot boot when device initialization takes a long time. Even if the state of IPs can be read and restored by accessing memory-mapped I/O regions of IPs, the characteristics of every single state register should be considered. This incurs the complexity of the snapshot boot approach and cannot be easily applied to another type or revision of a subsystem.

It is therefore important to identify a fast startup method that does not depend on the snapshot boot method. Although the performance of the sub CPU is sufficient for the subsystem to properly function, if a CPU faster than a sub CPU exists, the startup time of the subsystem can be reduced.

Actually, a mobile system already has a more powerful CPU in an AP and hence in a main CPU. Because the main CPU handles the running of many functions in a host system, it is equipped with more advanced hardware features, as discussed in Section II-B. Therefore, if the main CPU can handle the initialization phase of the subsystem, the startup time of the subsystem can be greatly reduced.

III. CPU SWITCH STARTUP

A. Overview

The main idea of the CPU switch startup method is to offload the startup sequence of the subsystem from the sub CPU to the main CPU. Conceptually, this approach is straightforward, as shown in Fig. 5. When a function of the subsystem is initiated, the initialization software, including the OS kernel of the subsystem, runs on the main CPU, not on the sub CPU. It initializes the hardware IPs and software libraries to be used in the subsystem. When the initialization completes, the control of the subsystem is transferred from the main CPU to the sub CPU. Finally, the sub CPU resumes its main function.

Whether the initialization is conducted on a sub CPU or a main CPU does not affect the functionality of the subsystem. When hardware IPs are initialized, their initialized states are preserved as values and states of the registers within the IPs. The states of the subsystem software are also preserved in memory. As a result, when the sub CPU assumes the control of the subsystem, the state of the subsystem can be consistent with the state when a sub CPU conducts the initialization. Accordingly, the sub CPU can continue the function of the subsystem.

The main benefit of this approach is that it can avoid hardware-dependent engineering efforts, as in the related work [5]-[7], while achieving the fast startup of a subsystem. Because this scheme uses the native binary software compiled for running on a sub CPU rather than taking a snapshot of a subsystem, it does not require any snapshot-taking phases. Consequently, it is unnecessary to consider a number of characteristics of hardware registers in hardware IPs [5], [7]. In this regard, the proposed scheme can be applied to any subsystem type or revision, including legacy subsystems.

To transparently run the subsystem software on the main CPU, two issues should be resolved. First, the binary executable that is specialized to a subsystem can run on the

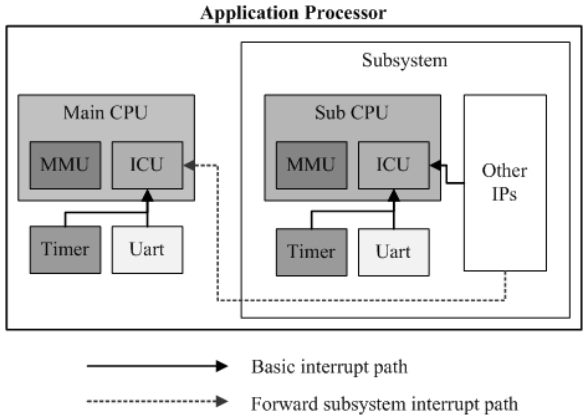

Fig. 6. Subsystem configuration example.

main CPU. This means that both CPUs must have the same instruction set architecture. In many MPSoCs, sub CPUs and the main CPU are built based on the same CPU architecture [9]. In some cases, the main CPU supports more instruction sets than the sub CPU does. The subsystem software binary, however, is still compatible with the main CPU because a high version CPU often provides backward compatibility. Therefore, the binary executable for the sub CPU can run on the main CPU without any modification.

Second, when subsystem software runs on the main CPU, its execution environment should be the same as that of the sub CPU. The execution environment includes hardware devices (or IPs) and a memory system. This means that the main CPU should have a consistent view of internal devices of the subsystem. In addition, the contents of memory pages should be consistent from the perspective of both the main CPU and sub CPU. The following two subsections describe the efforts to provide the consistent views of hardware devices and memory, respectively.

B. Providing a Consistent Hardware View

One of the roles of the subsystem software is to initialize the hardware devices (or IPs) of the subsystem. To transparently initialize the IPs in the main CPU, the main CPU should be able to access the IPs and handle interrupts from the IPs as in the same way that the sub CPU does. In addition, to minimize the modification of the subsystem software, the main CPU should have a consistent interface for accessing the IPs and handling interrupts from them.

In modern MPSoCs, special function registers (SFRs) of each IP are accessed through memory-mapped I/O mechanisms. These SFR regions are usually visible to both CPUs for debugging purposes. Both CPUs, however, have different physical address regions for the same SFR regions. This physical address difference, however, can be easily resolved by using the MMUs of each CPU. The subsystem software basically runs in a protected mode with its MMU enabled. Hence, the SFR region accesses are performed on a virtual address space. In this regard, the MMU of the main CPU is programmed to provide a virtual address space for SFR regions that is consistent with that in the sub CPU.

Whether interrupts from the hardware IPs can be delivered to the main CPU is important because it is hardware dependent. Some IPs specific to a subsystem have interrupt lines that are connected to both CPUs. However, the interrupt lines of some general IPs, such as a timer and universal asynchronous receiver/transmitter (UART), are often not connected to the main CPU, as depicted in Fig. 6. Generally, these IPs are basic IPs; each CPU has its own set of these IPs. Accordingly, whether or not the subsystem software runs on any of the CPUs, it should use local ones.

This limitation of using local IPs, however, does not incur any subsystem software modifications. Instead of a software modification, the SFR region mapping in both CPUs is slightly changed. In both CPUs, the virtual address region that accesses the replicated IPs is configured to access local ones. Hence, from the perspective of the subsystem software, the same virtual address region is linked to different IPs depending on where the subsystem software runs.

C. Providing a Consistent Memory View

During the subsystem startup, software libraries initialize data structures and variables in memory pages allocated to the subsystem. Even if the main CPU initializes the data, the data should remain valid, even when the sub CPU continues execution. The initialized contents may include memory pointers to organize a data structure, such as a linked list or a binary tree. Therefore, the main CPU and sub CPU should have the same virtual address space and physical pages associated with it.

Conceptually, by properly programming the MMU in both CPUs, the same virtual address space can be provided, as shown in Fig. 7. Before starting the initialization, the main CPU knows which physical pages will be used in the subsystem. Accordingly, by carefully programming both MMUs before starting the subsystem startup, both CPUs can have the same virtual pages that are mapped to the same physical pages. The page global directory for the page table in the sub CPU is set when the control transfers to the sub CPU.

In practice, issues for providing consistent virtual memory space to both CPUs are threefold: cache coherency, an input/output memory management unit (IOMMU), and a virtual address range conflict with the kernel in the main CPU.

First, both CPUs have private caches, and the caches are usually not coherent [9]. Operating a cache-coherent protocol between the main CPU and a subsystem is expensive. In addition, the subsystem is regarded as an I/O device from the perspective of the main CPU. Accordingly, the coherence is managed by software only while handling I/O requests and responses. One problem is that when current content of a certain memory region for the subsystem is cached in the main CPU cache, the sub CPU cannot read the current copy of the memory region.

The solution to the above problem is to flush memory pages associated with the subsystem from the main CPU cache when a control is transferred to the sub CPU. This is similar to the coherence management for I/O request/responses (e.g., reading/writing to storage). By flushing the corresponding

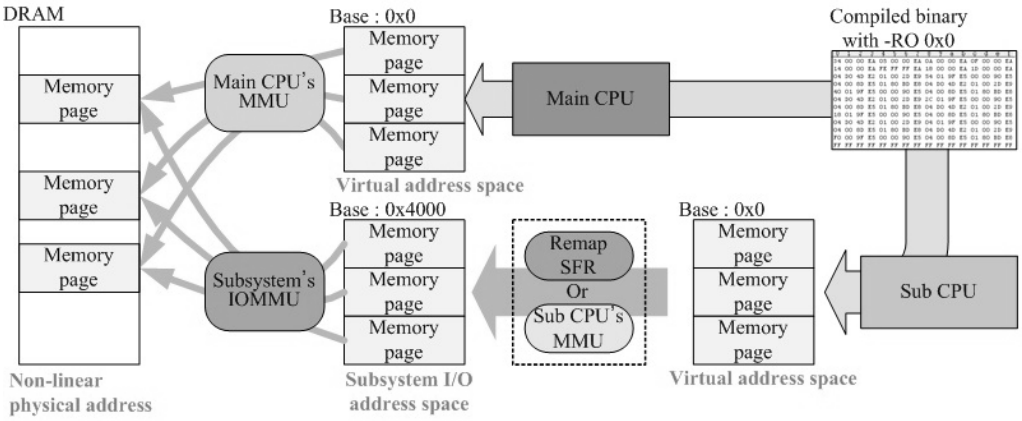

Fig. 7. Providing virtual address space to the subsystem binary on the main CPU and sub CPU with IOMMU enabled.

cache lines in the main CPU, current data is stored in physical memory. When the sub CPU continues execution, it can read the current copy of the data from physical memory.

Second, modern smart devices are integrated with an IOMMU. IOMMU provides address translation between an I/O address space and a physical address space; moreover, it is typically shared across subsystems in an AP. Accordingly, this two-phase address translation should be considered for establishing the page table in the main CPU.

In addition, the subsystem software basically assumes that a few instructions are executed while MMU is disabled. Because the execution of the subsystem software in the sub CPU begins with waking the sub CPU, its MMU is disabled until a page table base address is loaded into the MMU. In this regard, the subsystem binary is programmed to use an address remapping SFR in the sub CPU until its MMU is enabled. For example, as shown in Fig. 7, the start address of the subsystem binary is 0x0000 whereas the subsystem I/O address space starts at 0x4000. The address remapping SFR compensates for this address conflict. When the subsystem software runs on the main CPU, however, the main CPU does not have a time period in which its MMU is disabled. Therefore, use of the address remapping SFR in the main CPU is unnecessary.

The final problem is the virtual address conflict between the OS kernel in the main CPU and the kernel in the sub CPU. An OS kernel typically utilizes a high virtual address space; moreover, the address space is preserved during a context switch. Accordingly, when the subsystem software runs on the main CPU, the high address space is already occupied by the main OS kernel.

To resolve this address space conflict, the main CPU is isolated (e.g., CPU hot-unplugging [12]) from the main OS kernel during the subsystem initialization phase. Since modern APs include multicore processors for the main CPU, isolating one CPU (or a core) does not stop all functions of the main system. Furthermore, the initialization time of a subsystem is only tens of microseconds; therefore, the isolated CPU is unavailable in the main system for only a brief time. Accordingly, it is believed that the performance degradation caused by this CPU isolation would be negligible.

The CPU isolation in the main OS, however, requires additional efforts to enable the main system to correctly run. Any interrupts destined to an isolated CPU should be disabled. When such interrupts are delivered to the isolated CPU, the CPU does not have proper interrupt service routines. Therefore, the ICU of the main CPU should be properly programmed so that any interrupts except for those initiated from the IPs in the subsystem are masked so as not to be delivered to the isolated CPU. After the control of the subsystem is transferred from the isolated main CPU to the sub CPU, the isolated CPU is returned to the main OS kernel (e.g., CPU hot-plugging [12]).

IV. EVALUATION

A. Evaluation Environment

The proposed scheme was implemented and evaluated on a smart device development board. The board was equipped with a dual-core main CPU and had an internal image subsystem. The main CPU and sub CPU in the image subsystem were based on commodity-embedded processor architecture [13]. Table I shows the detailed specifications of both CPUs.

TABLE I
SPECIFICATIONS OF CPUS IN THE TARGET BOARD

Specifications		Main CPU [14]	Sub CPU [15]
Clock frequency		1.6 GHz	400 MHz
L1 Cache	Instruction	32 KB	16 KB
	Data	32 KB	16 KB
L2 Cache		1 MB	N/A
Cache line size		64 bytes	32 bytes
Superscalar		3-way issue	Single issue
Pipeline		15-25 stages	8 stages
Execution type		Out-of-order	In-order
Branch history buffer		1024-16384 entries	256 entries
D(Dhrystone)MIPS/MHz		3.6	1.6

The subsystem software was built on a real-time embedded OS [16]. For evaluation simplicity, the main CPU executed the same OS. Accordingly, the proposed scheme was

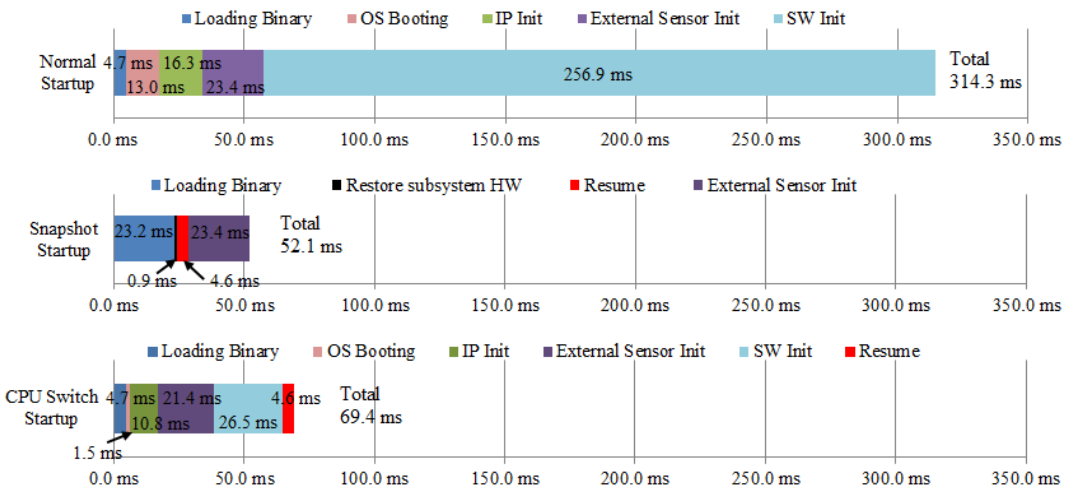

Fig. 8. Startup time of the image subsystem in the three startup methods: normal, snapshot and CPU switch.

implemented on the same OS. The proposed scheme was evaluated in comparison with a normal startup method and a snapshot start method [7] applied to the image subsystem.

B. Evaluation Results

The main goal of the CPU switch method is to accelerate the startup time of a subsystem. To measure how much the proposed scheme reduces the startup time, the startup time of the image subsystem in the target board was measured; the results are depicted in Fig. 8. As shown in the figure, the normal startup took 314.3 ms. During the normal startup, initialization of the software library in the image subsystem took the longest time at 256.9 ms. Because execution time of the software initialization phase depends on the performance of the CPU, the proposed scheme can be efficient for accelerating the startup time. Hardware initialization phases (IP init and external sensor init) comprised the second longest times of the startup.

The snapshot startup method took 52.1 ms (17% of the normal startup). Because this scheme requires no execution of the software, the software initialization phase was eliminated. In addition, by restoring the state of the hardware IPs, the IP initialization phase was likewise eliminated except for external sensor initialization. Note that the state of the external sensor cannot be restored through the snapshot startup method because it does not support suspend or memory-mapped register access. Instead of initializing hardware or software, copying the snapshot image to proper memory regions is the only operation to start the subsystem. This loading binary phase, however, is slightly increased because the size of the snapshot binary is 8 MB larger than the normal binary executable for the subsystem.

The startup time of the CPU switch method was 69.4 ms (22% of the normal startup), which was slightly longer than that of the snapshot startup scheme. More specifically, the loading binary phase was the same as that of the normal startup because both schemes used the same binary. The execution time of the hardware initialization phases was similar to the normal startup time because the times were mostly independent of CPU performance. The main contributor of the reduced initialization time was the software initialization phase because the fast main CPU was used.

The reduced initialization time of the image subsystem can improve the responsiveness of a camera application. Hence, when a user wants to take photographs, a camera application can be more quickly ready, thereby reducing chances to miss desired scenes. Without the fast startup methods, 300 ms of response time is sufficiently noticeable to human eyes. This delay is quantified as noticeable response time in human computer interaction [17]. The proposed scheme, however, shows only 70 ms of response time. This delay is classified to crisp response [17].

Although the proposed scheme was slower than the snapshot startup method by 17.3 ms, this additional delay is not perceivable. In addition, the snapshot method incurs high engineering costs when applied to other system types for several reasons. These reasons are detailed in Section IV-C.

To identify the benefits of using a fast main CPU, additional measurements were conducted. First, to determine the effect of the high clock frequency of the main CPU, the software initialization time on the main CPU at 400 MHz was measured. This clock frequency was the same as that of the sub CPU. As shown in Fig. 9, the software initialization time was decreased fourfold. Because the main CPU ran at a quarter of the maximum clock frequency, the execution time decreased by approximately four times. That time, however, was still more than two times faster than that on the sub CPU. The performance gap is believed to be consistent with the performance of both CPUs in terms of DMIPS/MHz, as depicted in Table I.

The asymmetric performance of the main and sub CPUs was caused not only by the different clock frequencies but also by the advanced hardware features, which were applied only to the main CPU. The main CPU had a three-way superscalar out-of-order execution unit with 15 to 25 pipeline stages, whereas the sub CPU had a single-way in-order execution unit with only 8 pipeline stages, as depicted in Table I. In addition, the main CPU had larger level-1 caches

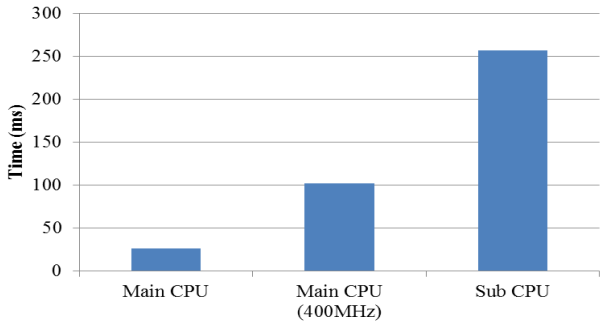

Fig. 9. Execution time of SW library initialization on: the main CPU, main CPU running at 400 MHz, and sub CPU.

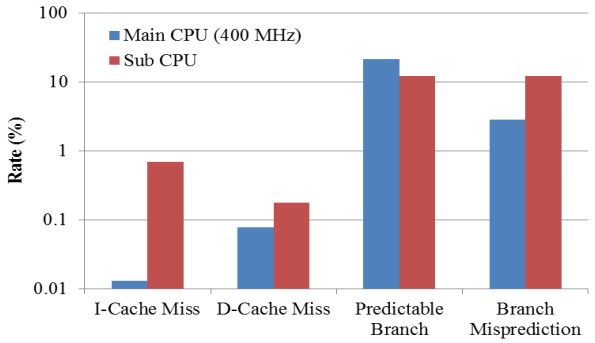

Fig. 10. Architectural performance measurements of the SW library initialization phase on the main and sub CPUs.

than those of the sub CPU; moreover, the main CPU had only 1 MB of level-2 cache. Finally, the main CPU had a more sophisticated prefetching unit, including a larger branch history, target, and return stack buffers.

To determine whether these architectural advances improve performance in runtime, additional performance-related factors were measured using a performance-monitoring unit in each CPU. Figure 10 presents L1 cache miss rates and branch prediction-related metrics. Because the sub CPU had no level-2 cache, that cache miss rate was not measured. As shown in the figure, the sub CPU demonstrated higher cache miss rates, especially in the instruction cache. The main CPU has a larger cache and supports a wider cache line than does a sub CPU. This is believed to be the reason for the high miss rates on the sub CPU.

Regarding branch prediction, the sub CPU showed a low predictable branch rate and a high branch misprediction rate. The set of predictable branch instructions supported in the sub CPU [15] is a subset of those in the main CPU [14]. For example, load instructions to the program counter register are the target of branch prediction in the main CPU; however, they are not the target in the sub CPU. In addition, this embedded processor architecture supports both 32-bit and 16-bit instructions. The change instructions between both modes are the branch prediction targets in the main CPU but not in the sub CPU. For these reasons, the sub CPU showed a lower predictable branch instruction rate.

Finally, the sub CPU showed a higher branch misprediction rate. Since the sub CPU had a smaller branch history buffer than that of the main CPU, as shown in Table I, the branch prediction accuracy was lower.

C. Engineering effort

It is important to know whether the proposed CPU switch method is efficient at accelerating the startup of a subsystem compared to the snapshot startup method. To apply the snapshot boot scheme to a subsystem, taking a snapshot of hardware IPs and restoring the snapshot image are important. The snapshot of hardware IPs consists of SFR areas of each IP in a subsystem. In case of the image subsystem, the snapshot image associated with the SFR areas was 83 KB and contained the state of approximately 5,000 SFRs. For many of the registers, taking a snapshot of and restoring them is easy by simply reading and writing the SFR regions. For some types of registers, however, these processes are difficult [7]. For example, it is difficult to read the state of write-clear registers. If some registers have a programming dependency, the snapshot image should be split and restored in the order of the dependency. More critically, some registers are hidden in a developer's manual because the document describes only registers actually used to program an IP. Missing registers, however, must also be taken as a snapshot and restored when the snapshot boot method is applied. In the tested image subsystem, 10% of the total SFRs were in these categories. In such a case, when a modification of a subsystem is made, every step in taking a snapshot of a register and restoring it should be performed. This comes with a high engineering cost.

Even if a new revision of a subsystem is introduced, a software developer should develop two software binaries. One is the original subsystem software binary; the other is a new snapshot binary made for the new software.

The proposed CPU switch method, on the other hand, does not require consideration of the complex characteristics of thousands of hardware registers. Moreover, a developer must develop only the original subsystem software. Therefore, it is believed that the CPU switch method can be easily applied to other types or revisions of subsystems without much engineering cost.

V. Discussion

Whether the proposed scheme can be extended to other types of systems is an important issue in terms of the portability. Although the proposed scheme is implemented on a real-time OS kernel, it can be deployed to other general-purpose OS kernels such as Linux.

In order to apply the CPU switch method to general OS kernels, the device drivers associated with subsystems should be changed. Basically, such device driver is in charge of allocating memory pages for a subsystem, programming IOMMU, loading the subsystem software binary, and waking up the sub CPU. In this regard, the following three steps should be added to the operations of the device driver before waking up the sub CPU.

First, the device driver needs to create a virtual address space which is identical to that of the subsystem. Since the physical pages are already allocated and the mappings between the I/O address space and the physical pages are already known, creating the identical virtual address space is straightforward. The SFR regions for the local peripherals are

also mapped to the virtual address space in this step. The base address of the page table (e.g., page global directory) is later used to program the MMU of the main CPU.

Second, a core of the main CPU should be isolated from scheduling and interrupt processing in the main OS kernel because during the startup of a subsystem, this core is dedicated to the subsystem. Although, it requires an assistance of the main OS kernel, many commodity OS kernels already support a sort of CPU isolation (e.g., CPU hot-plugging [12]). Since device drivers run in a privileged level, the CPU isolation is viable. The isolated core is later returned to the main OS kernel after the control of a subsystem transfers to the sub CPU. In the isolated core, a thread is polling until required resources (i.e., the subsystem software binary and the base address of the page table) are ready.

Finally, the device driver loads the subsystem binary and lets the thread know the base address of the page table. The polling thread programs the MMU of the isolated core and jumps to the main routine of the subsystem binary. The polling time of the thread can be minimized by overlapping the binary loading, making a page table and preparing the thread. When the subsystem is initialized, the subsystem software raises an interrupt to let the main system known the completion of the initialization. Then, the sub CPU is woken up and the control is transferred. The isolated core is returned to the main OS kernel.

These additional steps are actually independent to the types of subsystems in a system. The subsystem-dependent part is already handled before the beginning of the three steps. In this regard, once a general method is developed, it can be applied to other subsystems in a system by simply porting the three steps into other device drivers.

VI. Conclusion

In mobile smart devices, rapid execution of applications supported by hardware devices, such as a camera or a hardware codec, is important. These hardware devices are not a single device but a complex system regarded as a subsystem. Therefore, fast startup of a subsystem is directly connected to the responsiveness of these applications.

This paper proposed a CPU switch method that exploits the asymmetric performance of CPUs in mobile smart devices. By offloading the slow startup phase of the subsystem to the fast main CPU in an application processor, the execution time of the startup phase of the subsystem is greatly reduced. In addition, by avoiding accesses to complex hardware registers in the internal devices, the proposed scheme incurs only minor engineering efforts compared to application of a snapshot boot method to a subsystem.

References

[1] K. Restivo, R. Llama, and M. Shirer, “More smartphones were shipped in Q1 2013 than feature phones, an industry first according to IDC,” Press Release, Apr. 2013.

[2] J. Jeong, H. Kim, J. Hwang, J. Lee, and S. Maeng, "DaaC: device-reserved memory as an eviction-based file cache," *in Proc. International Conference on Compilers Architecture and Synthesis for Embedded Systems,* Tampere, Finland, pp. 191-200, Oct. 2012.

[3] T. Yan, D. Chu, D. Ganesan, A. Kansal, and J. Liu, “Fast app launching for mobile devices using predictive user context,” *in Proc. International Conference on Mobile Systems, Applications, and Services*, Low Wood Bay, Lake District, UK, pp. 113-126, Jun. 2012.

[4] H. Kaminaga, “Improving Linux startup time using software resume,” in *Proc. Linux Symposium*, Ottawa, Canada, pp. 25-34, Jul. 2006.

[5] H. Jo, H. Kim, H. Roh, and J. Lee, “Improving the startup time of digital TV,” *IEEE Trans. Consumer Electron.*, vol. 52, no. 2, pp. 485-493, May 2009.

[6] K. Baik, S. Kim, S. Woo, and J. Choi, “Boosting up embedded Linux device: experience on Linux-based smartphone,” *in Proc. Linux Symposium*, Ottawa, Canada, pp. 9-18, Jul. 2010.

[7] J. Kim, J. Lee, and J. Jeong, “Design and implementation of snapshot startup method for fast subsystem startup,” KIPS Trans. Comput. and Comm. Sys., vol. 3, no. 7, pp. 209-218, Jul. 2014.

[8] W. Wolf, and A. A. Jerraya, *Multiprocessor Systems on Chips*, Morgan Kaufmann, 2004.

[9] X. Lin, Z. Wang, and L. Zhong, “K2: a mobile operating system for heterogeneous coherence domains,” in Proc. *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, Salt Lake City, Utah, USA, pp. 285-300, Mar. 2014.

[10] J. Jeong, H. Kim, J. Hwang, J. Lee, and S. Maeng, “Rigorous rental memory management for embedded systems,” *ACM Trans. Embed. Comput. Syst.,* vol. 12, No. 1, pp. 43:1-43:21, 2013.

[11] T. R. Bird, “Methods to improve bootup time in Linux,” *in Proc. Linux Symposium*, Ottawa, Canada, pp. 79-88, Jul. 2004.

[12] Z. Mwaikambo, A. Raj, R. Russel, and J. Schopp, “Linux kernel hotplug CPU support,” *in Proc. Linux Symposium*, Ottawa, Canada, pp. 467-480, Jul. 2004.

[13] S. B. Furber, *ARM System Architecture*, Boston, MA: Addison-Wesley Longman, 1996.

[14] *Cortex-A15 Technical Reference Manual*, ARM, 2011, pp. 6:1-6:12.

[15] *Cortex-A5 Technical Reference Manual*, ARM, 2009, pp. 7:1-7:11.

[16] H. Ahn, M. Cho, M. Jung, Y. Kim, J. Kim, and C. Lee, “UbiFOS: a small real-time operating system for embedded systems,” *ETRI J.*, vol. 29, no. 3, pp. 259-269, Jun. 2007.

[17] N. Tolia, D. G. Andersen, M. Satyanarayanan, “Quantifying interactive user experience on thin clients,” *Computer*, vol. 39, no. 3, pp. 46-52, 2006.

Biographies

Jun Kim received a B.S. degree from the Computer Science Department of Yonsei University, Korea, in 2005. He has worked as an engineer in the AP development team of Samsung since 2005 and is an M.S. candidate at Sungkyunkwan University. His research interests include embedded systems, AP architecture, and image signal processing.

Joonwon Lee received a B.S. degree in Computer Science from Seoul National University in 1983 and M.S. and a Ph.D. degree from the Georgia Institute of Technology in 1990 and 1991, respectively. He is currently a professor at Sungkyunkwan University. Before assuming that role, he was a professor at the Korea Advanced Institute of Science and Technology from 1992 to 2008. His current research interests include low-power embedded systems, systems software, and virtual machines.

Jinkyu Jeong received a B.S. degree in Computer Science from Yonsei University and a Ph.D. degree from Korea Advanced Institute of Science and Technology (KAIST) in 2013. He is currently an assistant professor in the department of Semiconductor Systems Engineering at Sungkyunkwan University. His research interests include real-time systems, operating systems, virtualization, and embedded systems.