Supporting Distributed Execution of Smartphone Workloads on Loosely Coupled Heterogeneous Processors

Felix Xiaozhu Lin, Zhen Wang, and Lin Zhong

Rice University

Abstract

Modern smartphones are embracing asymmetric, loosely coupled processors that have drastically different performance-power tradeoffs. To exploit such architecture for energy proportionality, both application and OS workloads need to be distributed. We propose Kage, a combination of runtime and OS support, to replicate application execution and OS functions over asymmetric processors. Kage selectively creates replicas of application and OS services and maintains state consistency for them with low overhead. By doing so, it is able to reduce processor energy consumption of light-loaded smartphones manyfold. While enabling energy-proportionality, Kage simplifies application programming by providing the illusion of a single system image and per-process address spaces.

1. Introduction

Mobile systems, most notably smartphones, are seeing a great variety of workloads, from intensive ones, e.g., web browsing and gaming, to light ones, e.g., receiving instant messages. At the same time, users expect their smartphones to deliver high performance for intensive workloads while consuming low energy for light workloads. This essentially calls for *energy-proportionality*: scale down energy consumption at the same rate as workloads scale down. Today, mobile systems are embracing asymmetric processors with different performance-power tradeoffs, in hopes of executing any given workload on a processor that has the best tradeoff while keeping all other processors in a low-power state. As an example scenario, the system could periodically fetch emails with its low-power (small) processors while supporting interactive email composition with its high-performance (big) ones.

The key characteristic of such an asymmetric architecture is how processors are coupled, i.e., the presence or absence of hardware cache-coherence. Tightly coupled processors share a unified, coherent address space backed up by hardware, e.g., ARM big.LITTLE [5] and NVIDIA Tegra 3 [13]. The unified coherence mechanism restricts asymmetry among processors and therefore limits the energy-proportionality resulted from the asymmetry. In addition, tightly coupled processors are likely to be co-located on chip and therefore experience similar thermal constraint.

In contrast, loose coupling offers better energy-proportionality for a wider range of workloads. System-on-Chip (SoC) examples include TI OMAP4 [16], OMAP5 [17], and Samsung Exyons [15]. The authors are also aware of smartphones that employ 16-bit microcontrollers to deal with sensor data processing on behalf of the main SoC. Without hardware coherence, processors are allowed to be highly asymmetric, where performance and power consumption may differ by orders of magnitude. For instance, on OMAP4 [16] as illustrated in Figure 1, the clock rates of loosely coupled Cortex-A9 and Cortex-M3 differ by ~4x, much higher than the tight coupling case, e.g., up to 1.7x in big.LITTLE [5]. Although attractive, loosely coupled processors poses serious programming challenges to the entire mobile software stack because program state in both user and OS code becomes distributed and potentially inconsistent. In this paper, we use the term OS to refer to software infrastructure that supports the execution of user code.

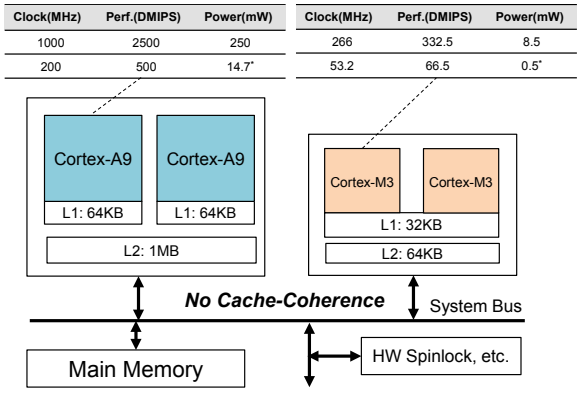


Figure 1: The architecture of TI OMAP4 SoC, which features two types of asymmetric processors that are not cache-coherent. Data with asterisks are estimated based on a DVFS providing 1:5 performance scaling and 1:17 power scaling [12]

Targeting energy-proportionality, we seek to support a broad range of smartphone workloads over loosely

coupled processors. In particular, our system must adapt to two major characteristics of smartphone workloads: (*i*) high *temporal* variation, i.e., workloads with time-varying computational demands and (*ii*) wide *spatial* distribution, in particular, the use of diverse OS functions.

To achieve the goals above, we propose Kage, a suite of runtime and OS support. Kage distributes both application and OS workloads over loosely coupled processors, while presenting a single system image and per-process unified address spaces to applications.

Kage greatly improves system energy-proportionality. Estimated based on our observations of smartphone CPU usage and clock rates, any workloads using less than 12% of a Cortex-A9 core of OMAP4 SoC, can be executed on a single Cortex-M3 core without overrunning the maximum instruction throughput. By enabling this, Kage reduces the processor energy consumption of such light workloads by four times. Note that this estimation is conservative: we only considered workloads that can be executed on Cortex-M3 at its lowest possible clock rate and ignored energy savings from its other clock rates.

2. Related Work

Hardware heterogeneity is widely used for energy-proportionality. Heterogeneous processors may be tightly coupled [5, 8, 13], which eases programming but limits energy saving. Alternatively, loosely coupled heterogeneous processors offer much higher efficiency, but *directly* programming them is difficult due to separate address spaces and separate system images [1, 7, 14].

Function replication is common in distributed systems, usually for the sake of scalability. Back in the 1980s, the V distributed system [4] replicates OS functions among a group of homogeneous workstations, by implementing software distributed shared memory (DSM) below the OS. More recently, factored OS (fos) [18] parallelizes individual OS services over multiple cores with explicit message passing. Although inspiring, they do not account for architectural asymmetry, processor power state changes, or legacy OS code, the major challenges we face on smartphones.

Aiming at performance or isolation, many heterogeneous systems split their functions into complementary partitions and pin such partitions on different processors [2, 11]. However, under highly time-varying workloads, partitioned but not replicated functions can easily become either performance or energy-efficiency bottleneck. In multicore systems, device virtualization [3] supports distributing OS workloads, by allowing multiple instances of the same OS service (e.g., network stack) to execute concurrently. However, without sharing state, these instances do not collaboratively provide a single system image, making it difficult for an application to span across processors.

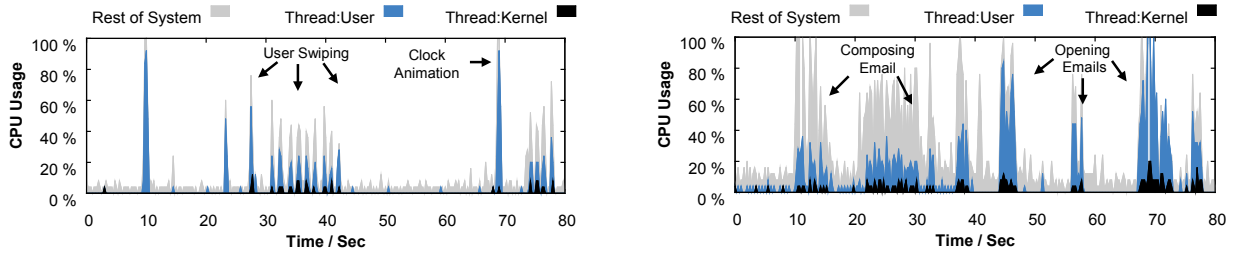
We have built Reflex [9], a system that supports executing simple, periodic tasks on low-power processors and keeps distributed application state consistent. Such user tasks can be pinned on low-power processors due to their low temporal variation; besides, systematic replication of OS functions is unnecessary in Reflex since these tasks only access a few OS functions. In building Kage, we leverage our experiences with Reflex to explicitly treat both temporal variation and spatial distribution that exist in a much broader range of workloads.

3. Observations of Workloads

The unique characteristics of smartphone workloads challenge existing system support and motivate Kage, as we will see in the following measurements. All data are collected on a combination of Android ICS (4.0.3) and Samsung Galaxy S2, a mainstream smartphone.

Workloads Change over Time: Many smartphone workloads are known to have high temporal variations. To experimentally show this, we measure the CPU usage from `/proc` in two benchmark applications, the Android home screen and the Google Gmail client. As shown in Figure 2, for each benchmark application, we sample the main thread's CPU usage, which is broken down into user and kernel parts, denoted as *thread:user* and *thread:kenrel* in the figure; meanwhile, we sample CPU usage of the rest of the system other than the main thread, denoted as *rest of system*. Results show that both benchmarks have time-varying workloads. During user inactivity, the CPU usage is low to moderate (5%-20% globally). Once interactive workloads emerge, the global CPU usage usually spikes by a few orders of magnitude, up to 100%.

Same Application Threads Experience Varying Workloads: Existing smartphone applications do not necessarily organize their program code into separate threads or even separate functions based on the computational demands of the code. As shown in the measurements above, the CPU usage of a single thread can vary by orders of magnitude. Sometimes the variation is purely temporal: the same code segment can have highly varying demand, e.g., application UI renderer.



(a) Home screen **(b) Gmail client**

Figure 2: CPU usage of main threads in two smartphone benchmarks

***Many OS Functions Experience Varying Workloads*:** Overall, smartphone OS workloads exhibit significant temporal variations: in both benchmarks, when total workloads increase, the CPU usage of OS usually scales up accordingly. More importantly, the same OS function can experience highly varying workloads. Figure 3 shows system-wide invocations of OS kernel functions during the executions of two benchmarks: *browser*, an intensive workload incurring 82% global CPU usage on average, in which the user is actively using Chrome browser for browsing; *home-idle*, a light workload incurring 7% global CPU usage on average, in which the smartphone is idle and the home screen is shown. As shown in the figure, many OS functions are invoked with both intensive and light workloads.

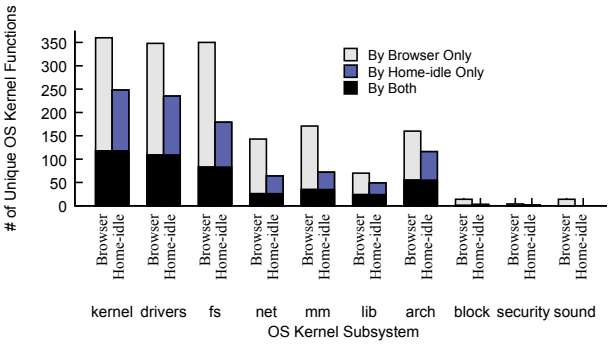


Figure 3: Diverse OS kernel functions invoked under a typical intensive benchmark (*Browser*) and a typical light benchmark (*Home-idle*), with each benchmark runs for 300 seconds. The invocations are reported by Linux `perf` tool

Applications Use Diverse OS Functions: Results in Figure 3 also indicates that smartphone applications usually use a variety of OS functions, even when the workloads are light. This is because when the user is inactive and the system is lightly loaded, various 'background' tasks are maintaining the smartphone's presence by interacting with the external world, e.g., cloud servers or the physical environment, which naturally invokes various OS functions.

4. Implications on Design

We next discuss the implications of the workloads characteristics on the design of Kage.

4.1.Replicating Application Execution

To achieve energy-proportionality with time-varying workloads described above essentially calls for a way of spreading applications on heterogeneous processors, which (*i*) enables a single thread to use different processors, with choices made at run time and (*ii*) preserves the existing program structure and therefore maximizes the reuse of existing application code. Note that (*i*) is useful even to threads with sustained light workloads: when big processors are active and the energy saving from using small processors diminishes, such threads can switch to big processors to avoid expensive inter-processor communication.

On cache-coherent multicores, the two requirements are naturally met with transparent thread migration, which, however, becomes difficult in face of extremely architectural asymmetry and lack of cache coherence. Partitioning a single thread into a set of 'heterogeneous' threads is infeasible because of that a single thread may have varying workloads (Section 3) as well as the tedious programming efforts in handling extra concurrency.

Execution Replication: To meet such requirements on asymmetric architecture, we argue that the OS/runtime must provide an abstraction of *execution replication*. From a programmer's view, execution replication enables a thread to switch its execution among multiple replicas on heterogeneous processors. The control flow is still one: all replicas cooperatively execute and 'yield' to each other. The possible yielding locations in code are defined by programmer and are chosen dynamically, under decisions collectively made by user code, runtime, and OS.

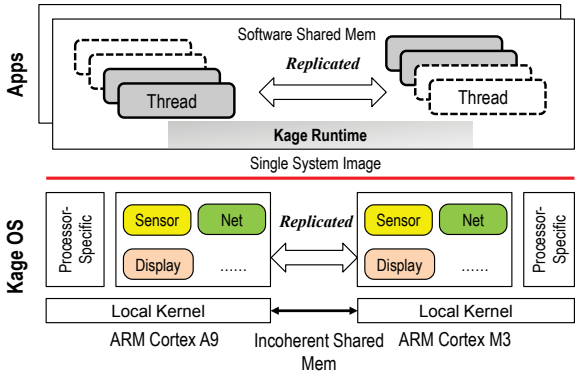


Figure 4: The structure of Kage

RPC Considered Inadequate: the abstraction of Remote Procedure Call (RPC), which encapsulates code executed on other processors as subroutines, is inadequate in our case: upon returning from a remote routine, a local routine must resume from exactly where it invokes the remote routine. However, replicas, as cooperative peers, do not fit into the caller/callee pattern. As a result, using RPC usually requires overhauling existing applications. For instance, in enabling two processors to independently execute the main event loop of an application, RPC requires programmers to manually split the loop into two, with one calling into the other.

4.2.Replicating OS Functions

In face of high variations in OS workloads, some OS functions need to be replicated across heterogeneous processors. The rationales are twofold: (*i*) to present a single, coherent system image to physically distributed applications, in particular, references of OS objects, e.g., file descriptors, should remain valid across processors; (*ii*) to execute given OS workloads on processors that provide the best performance-power tradeoff. While (*i*) may be easily achieved by merely replicating OS *interface*, for example, redirecting all syscalls to a specific processor, only by replicating the actual OS functions can we achieve (*ii*), the key to energy-proportionality for OS workloads. Furthermore, the use of diverse OS functions warrants a system support for replicating a large set of OS functions, rather than hand-crafting a few specific ones.

4.3.Limiting Inter-Processor Sharing

Distributed workloads extensively share state: within an application, threads and their replicas share user and OS state; globally, multiple applications can share OS state, e.g., opened files. Due to the high communication overhead among loosely coupled processors, it is crucial for the system to limit inter-processor sharing and access contention. As an example, a Cortex-A9 core of OMAP4 can fetch 32 byte data from the other A9 core through their shared L2 cache in tens of nanoseconds [6]; however, it takes 1000 times longer or 70 µs for a Cortex-A9 core to fetch the same data from a Cortex-M3 core: the inter-processor sharing requires software to write back and then invalidate cache lines, and to use a hardware spinlock on a slow peripheral bus (shown in Figure 1) to synchronize accesses of the main memory.

4.4. Ensuring System Responsiveness

Interactive smartphone workloads are usually intensive, and they must be executed on the big processor as soon as they emerge. Although loose coupling and aggressive processor power management introduces delay in starting the execution, the system can still appear responsive to the user. The introduced delay consists of three parts: inter-processor interrupt, the big processor waking up from a low-power state, and software synchronizing state shared between processors. In our measurements of OMAP4, it takes 20-30 µs to deliver an inter-processor interrupt from Cortex-M3 to Cortex-A9 through a hardware mailbox; it takes up to 2 ms for the Cortex-A9 core to wake up from the retention power state [10]. Ultimately, it is the responsibility of the state synchronization scheme to keep the total delay much lower than the latency-perception threshold of human, i.e., several tens of milliseconds.

5. Kage Overview

We next sketch Kage design. In the discussion, we assume a hardware architecture consisting of two loosely coupled, heterogeneous processors, a popular design among modern mobile SoCs. Figure 4 illustrates the structure of Kage.

Application and runtime: From a programmer's view, an application can launch threads on both processors, big and small. Conceptually, each thread has a replica that is always available on the other processor; a thread and its replica voluntarily *yield* their own executions to each other for using different processors alternately. In this way, an application is free to choose processors according to its own performance-power goal. Under the hood, a thread on the other processor, i.e., a remote thread, implements the replica; the OS schedules the original thread and the remote one alternately: when the original thread *yield*s, the OS suspends it and resumes the remote thread, and vice versa.

Within an application, the Kage runtime library creates a unified address space across all the threads, with a user-level software shared memory. Given limited inter-processor concurrency per application, the shared memory design is able to perform coherence operations

less frequently, keeping the communication overhead low.

OS structure: As shown in Figure 4, Kage OS is distributed. On each heterogeneous processor, Kage OS runs a small local kernel, which only consists of minimum functions to abstract hardware heterogeneity. For any OS service that is not tied to a specific processor and deserves energy-proportional execution, Kage OS creates per-processor replicas, each of which realizes the function. All these OS replicas act as peers: locally, they serve requests from application threads on the same processor; globally, they cooperate to create a single, coherent image to all applications.

To maintain the single system image illusion, Kage OS keeps the state of OS replicas consistent with an OS-level software shared memory. Unlike in a single application, the OS shared memory design needs to handle much higher inter-processor concurrency from multiple applications on different processors. In order to reduce the communication overhead, the shared memory tracks and only maintains consistency for OS objects shared across application boundaries.

6. Open Questions

Proper Programming Abstractions should be carefully devised for both application and OS code, so that replicas can be easily created and their state is maintained as consistent. While arguing for a structural change to software stack, we target at maximizing reuse of existing smartphone application and OS codebases which are already huge. We believe that single system image and unified address space are essential foundations of higher-level abstractions.

Implementation of Replicas brings the key challenge of transferring stack content across heterogeneous processors. We plan to retrofit ideas from distributed systems, including lightweight virtualization and linked stack. We will co-design the replica internals with the programming abstractions, in order to leverage programmers' assistance for reducing overhead.

Scheduling of Application Threads determines the inter-processor concurrency within an application, a key factor to the complexity and overhead of the shared memory design. In particular, the OS has to decide which thread to pick when multiple ones are ready on both big and small processors. We plan to experiment with various scheduling strategies.

7. Conclusion

Kage is a suite of OS and runtime that can greatly improve the energy proportionality for a wide range of smartphone workloads. In order to achieve such a goal, Kage distributes both user and OS workloads over loosely coupled processors by selectively replicating application execution and OS functions. A full implementation of Kage is in progress.

Reference

[1] Agarwal, Y., Hodges, S., Chandra, R., Scott, J., Bahl, P., and Gupta, R., "Somniloquy: augmenting network interfaces to reduce PC energy usage," in *Proc. USENIX NSDI*, April 2009.

[2] Baumann, A., Barham, P., Dagand, P., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A., and Singhania, A., "The Multikernel: A new OS architecture for scalable multicore systems," in *Proc. ACM SOSP*, October 2009.

[3] Boyd-Wickizer, S., Chen, H., Chen, R., Mao, Y., Kaashoek, F., Morris, R., Pesterev, A., Stein, L., Wu, M., Dai, Y., Zhang, Y., and Zhang, Z., "Corey: an operating system for many cores," in *Proc. USENIX OSDI*, December 2008.

[4] Cheriton, D., "The V distributed system," in *Commun. ACM*, vol. 31, 1988.

[5] Greenhalgh, P., "Big.LITTLE Processing with ARM Cortex-A15 and Cortex-A7," ARM White Paper, 2011.

[6] Gutierrez, A., Dreslinski, R. G., Wenisch, T. F., Mudge, T., Saidi, A., Emmons, C., and Paver, N., "Full-system analysis and characterization of interactive smartphone applications," in *Proc. IEEE Intl. Symp. on Workload Characterization*, November 2011.

[7] Han, C. C., Goraczko, M., Helander, J., Liu, J., Priyantha, B., and Zhao, F., "CoMOS: An operating system for heterogeneous multi-processor sensor devices," Technical Report MSR-TR-2006-117, Microsoft Research, 2006.

[8] Kumar, R., Farkas, K. I., Jouppi, N. P., Ranganathan, P., and Tullsen, D. M., "Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction," in *Proc. IEEE/ACM MICRO*, December 2003.

[9] Lin, F. X., Wang, Z., LiKamWa, R., and Zhong, L., "Reflex: using low-power processors in smartphones without knowing them," in *Proc. ACM ASPLOS*, April 2012.

[10] linux-omap: arch/arm/mach-omap2/cpuidle44xx.c.

[11] Nightingale, E. B., Hodson, O., McIlroy, R., Hawblitzel, C., and Hunt, G., "Helios: heterogeneous multiprocessing with satellite kernels," in *Proc. ACM SOSP*, October 2009.

[12] Nowka, K. J., Carpenter, G. D., MacDonald, E. W., Ngo, H. C., Brock, B. C., Ishii, K. I., Nguyen, T. Y., and Burns, J. L., "A 32-bit PowerPC system-on-a-chip with support for dynamic voltage scaling and dynamic frequency scaling," in *IEEE J Solid-State Circuits*, vol. 37, 2002.

[13] NVIDIA, *NVIDIA Tegra 3*: http://www.nvidia.com/object/tegra-3-processor.html.

[14] Priyantha, B., Lymberopoulos, D., and Liu, J., "LittleRock: Enabling Energy-Efficient Continuous Sensing on Mobile Phones," in *Pervasive Computing*, vol. 10, 2011.

[15] Samsung, *Exynos 4210 Application Processor*: http://www.samsung.com/global/business/semiconductor/productInfo.do?fmly_id=844&partnum=Exynos%204210.

[16] Texas Instruments, "OMAP4 Applications Processor: Technical Reference Manual," 2010.

[17] Texas Instruments, *OMAP5 Platform - OMAP5430*: http://www.ti.com/ww/en/omap/omap5/omap5-OMAP5430.html.

[18] Wentzlaff, D. and Agarwal, A., "Factored operating systems (fos): the case for a scalable operating system for multicores," in *SIGOPS Oper. Syst. Rev.*, vol. 43, 2009.