K2: A Mobile Operating System for Heterogeneous Coherence Domains

Felix Xiaozhu Lin, Zhen Wang, and Lin Zhong
Rice University, Houston, TX
{xzl, zhen.wang, lzhong}@rice.edu

Abstract

Mobile System-on-Chips (SoC) that incorporate heterogeneous coherence domains promise high energy efficiency to a wide range of mobile applications, yet are difficult to program. To exploit the architecture, a desirable, yet missing capability is to replicate operating system (OS) services over multiple coherence domains with minimum inter-domain communication. In designing such an OS, we set three goals: to ease application development, to simplify OS engineering, and to preserve the current OS performance. To this end, we identify a shared-most OS model for multiple coherence domains: creating per-domain instances of core OS services with no shared state, while enabling other extended OS services to share state across domains. To test the model, we build K2, a prototype OS on the TI OMAP4 SoC, by reusing most of the Linux 3.4 source. K2 presents a single system image to applications with its two kernels running on top of the two coherence domains of OMAP4. The two kernels have independent instances of core OS services, such as page allocator and interrupt management, as coordinated by K2; the two kernels share most extended OS services, such as device drivers, whose state is kept coherent transparently by K2. Despite platform constraints and unoptimized code, K2 improves energy efficiency for light OS workloads by 8x-10x, while incurring less than 6% performance overhead for a device driver shared between kernels. Our experiences with K2 show that the shared-most model is promising.

Categories and Subject Descriptors D.4.7 [*Operating Systems*]: Organization and Design

General Terms Design, Experimentation, Performance

ASPLOS '14, March 1–4, 2014, Salt Lake City, Utah, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2305-5/14/03…$15.00.
http://dx.doi.org/10.1145/2541940.2541975

Keywords Energy efficiency; heterogeneous architecture; mobile; coherence domains

1. Introduction

Today's mobile devices face a wide range of workloads, from demanding tasks such as interactive applications with rich graphics to light tasks running in the background such as cloud synchronization and context awareness. A recognized, effective approach to achieve high energy efficiency for such diverse workloads is to exploit hardware heterogeneity, i.e., to execute a workload with the hardware component offering the best performance-power tradeoff.

Today, cutting-edge mobile SoCs have already embraced hardware heterogeneity. For instance, the ARM big.LITTLE architecture has cores of different strengths in one cache coherence domain, or *coherence domain*. Cores in the same coherence domain have the same memory view backed by hardware-supported cache coherence. The coherence allows a process to move between big and little cores at the cost close to context switch. Because hardware coherence forbids the little core from being much weaker than the big ones, many mobile SoCs have gone one step further to include multiple coherence domains, where hardware coherence only exists within each domain but not among them. One coherence domain can host high-performance cores for demanding tasks; another domain can host low-power cores for light tasks. In this work, we call them *strong domain* and *weak domain*, respectively. We purposefully use *strong* and *weak* in order to distinguish them from heterogeneous cores in the same domain that are already known as *big* and *little*. The absence of hardware coherence allows cores in the weak domain to be orders of magnitude weaker and lower-power than those in the strong domains, providing much better energy efficiency for light tasks than the big.LITTLE architecture.

In tapping into the energy efficiency benefit of multiple coherence domains, the biggest challenge is *programmability*. Processors from multiple coherence domains essentially constitute a distributed system; programming them is known to be difficult to mass programmers, who must maintain consistency of partitioned program components with messages.

Many have recognized this challenge and proposed a handful of systems solutions [17, 26], which, however, mainly focus on user workloads. They usually implement a small set of OS services from scratch for a weak domain, e.g., sensor reading, and assume that all other OS services are provided by a fixed strong domain. The resulting OS services are limited in functionality, complicate application development because of a fragmented system image, and incur heavy burdens in OS engineering. More importantly, with this design, only light tasks without a need for OS services beyond those crafted for weak domains can benefit from the efficiency of weak domains.

In this paper, we seek to answer the following question: *is it possible to span an OS across multiple coherence domains?* In doing so, we seek to achieve three important goals. (*i*) The resulting OS should preserve the established programming models for mobile applications, as the models have been used in over a million mobile application by a similar number of developers, many of whom do not have sophisticated programming skills. (*ii*) The resulting OS itself shall leverage mature OSes in wide use without reinventing many wheels. An overhaul of existing mobile OSes is increasingly difficult and even undesirable nowadays: they consist of huge codebases and keep receiving contributions from many parties including device vendors. (*iii*) The OS should deliver the same peak performance for demanding tasks.

We show that we can span an OS across coherence domains by properly refactoring, but not overhauling, an existing OS. We identify a *shared-most* OS model where most OS services are replicated in all domains under question with transparent state coherence, while a small, selective set of services operates independently in each domain. We call these services *shadowed services* and *independent services*, respectively. The shared-most model resides midway of two well-studied extremes, *shared-nothing* and *shared-everything*. While the shared-most model promises both benefits of simplified OS engineering and performance for multi-domain mobile SoCs, the other two models, however, provide only one of the two benefits.

By applying the shared-most model, we build K2, an OS that runs multiple kernels on a multi-domain SoC. As an early research prototype, K2 has unoptimized code and is constrained by the limitations of the test platform. Yet, K2 has met the three goals above, showing that the shared-most model is promising. (*i*) K2 presents a coherent, single Linux image to applications, and therefore preserves the widely used programming model; it also provides a familiar abstraction, called NightWatch threads, for distributing user workloads over heterogeneous domains. (*ii*) K2 replicates existing extended OS services, e.g., device drivers, across multiple domains and transparently maintains coherence for them. As a result, K2 is able to reuse most of the kernel source. (*iii*) K2 preserves the current level of OS performance by alleviating inter-domain contention: it creates independent instances of core OS services and properly coordinates them; it avoids multi-domain parallelism within individual processes through scheduling. Throughout its design, K2 adopts asymmetric principles that greatly favor the performance of the strong domain.

In summary, we have made three contributions in this work:

- First, we identify a shared-most OS model for exploiting the energy efficiency potential of multi-domain mobile SoCs, without complicating programming or hurting performance.
- Second, in applying the shared-most OS model, we present a set of guidelines and experiences of refactoring a mature OS, Linux. We classify its OS services as private, independent, and shadowed, in order to replicate them across coherence domains.
- Third, we describe K2, the outcome of refactoring and a working OS prototype for multi-domain mobile SoCs. With the shared-most model applied, K2 reuses most of the Linux kernel source and presents a single system image. The resulting energy efficiency and performance show that our model is promising.

2. Background

To provide a background, we first discuss the characteristics of light tasks in mobile systems, and then review the mobile architectural trend that is moving towards heterogeneity and incoherence.

2.1 Light tasks in mobile systems

Serving as the personal information hub, today's mobile devices execute a rich set of 'background' tasks, or *light tasks*, e.g., sensing user physical activities, monitoring surrounding environment [27], and keeping users connected with social networks and the cloud [41]. Light tasks usually have the following characteristics:

- *Not performance demanding:* Without an awaiting user, light tasks do not directly affect the user experience and thus do not require high processor MIPS.
- *Mostly IO-bound:* Light tasks perform extensive IO operations for exchanging information with the external world. During IO operations, core idle periods are many, however not long enough for a core to become inactive.
- *Requiring both user and OS execution:* Light tasks not only run user code for application logic, but also invoke diverse OS services such as device drivers [18] and page allocator.
- *Scheduled to execute throughout daily usage:* Light tasks are executed not only when a user is interacting with the device, but also during user's think periods between interactions and when a user pays no attention. Since

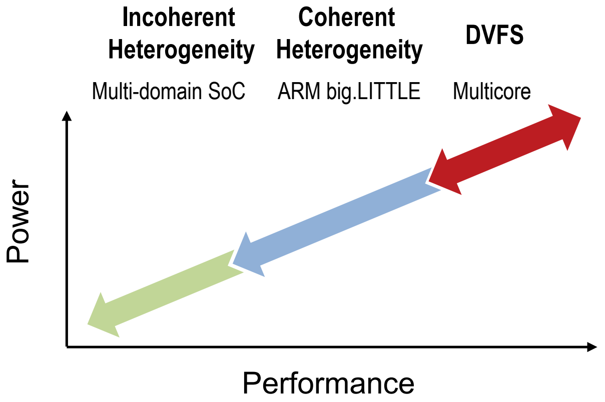


Figure 1. Trend in mobile SoC architectures. Both axes are logarithmic.

user interaction periods are sparse and usually short, most executions of light tasks happen when the entire system is lightly loaded.

- *High impact on battery life:* The nature of frequent executions has a high impact on battery life. For instance, a recent study [41] shows that each run of background email downloading reduces device standby time by 10 minutes.

2.2 Trend in architectures

To meet the compute demand of modern applications, mobile devices are equipped with powerful cores that have gigahertz frequency and rich architectural features. These powerful cores, however, offer poor energy efficiency for light tasks, due to three sources of inefficiency:

- *High penalty in entering/exiting active power state.* Since strong cores are inactive for most of the time, periodic executions of light tasks will inevitably wake them up from time to time.
- *High idle power.* In executing IO-bound light tasks, powerful cores will spend a large number of short periods (of milliseconds) in idle, which are known to consume a large portion of energy in mobile systems [42].
- *Over-provisioned performance.* Although a strong core may increase its energy efficiency by lowering its frequency, shown as 'DVFS' in Figure 1, the lowest possible frequency and active power are limited by its architecture and fabrication process, which still over-provision performance to light tasks with a low performance demand.

Coherent heterogeneous SoC To offer a remedy, today's SoCs incorporate heterogeneous cores with different tradeoffs between performance and power, by placing them in a single or multiple coherence domains.

A handful of heterogeneous SoCs host all heterogeneous cores with a single coherence domain [15, 24]. As shown as 'coherent heterogeneity' in Figure 1, although the introduced heterogeneity indeed widens dynamic power range, the global cache coherence becomes the bottleneck for more aggressive energy efficiency, for a few reasons: *i)* a unified hardware coherence mechanism restricts architectural asymmetry, *ii)* the coherent interconnect itself consumes significant power, and *iii)* cores co-located in the same coherence domain are likely to suffer from similar thermal constraints.

Incoherent heterogeneous SoC For aggressive energy efficiency, several newer SoCs remove chip-level hardware coherence. They embrace multiple coherence domains, each of which can host multiple cores; hardware coherence exists within a domain but not across domains. Examples include OMAP4 [36], OMAP5 [37], and Samsung Exynos [29]. As shown in Figure 1, the absence of hardware coherence enables high asymmetry among cores belonging to different domains. For instance, while the lowest power of different cores in the same domain can differ by 6x [32], that of different domains can differ by up to 20x. This allows using weak cores to greatly reduce the three inefficiencies mentioned above.

3. The case for replicating OS services

To fully exploit multi-domain SoCs for energy efficiency, the key is to enable OS to serve a request from the domain that is able to deliver the needed performance with the lowest power possible. This requires *replicating* OS services on all the candidate domains.

As shown by a recent study of mobile workloads [18], the same OS services are usually invoked with disparate performance and power expectations. For instance, light tasks often access IO regularly, thus sharing the OS services with demanding tasks such as UI rendering. Neither pinning OS services on a single domain nor partitioning them among domains [22] will work here: on one hand, light tasks invoking OS services provided on a strong domain will suffer from all inefficiencies described in §2.1; on the other, demanding tasks invoking OS services provided on a weak domain are likely to fail their performance expectations. Targeting exploiting incoherent heterogeneity for energy efficiency, most prior systems are focused on supporting user workloads and usually adopt ad-hoc solutions for OS services on weak domains [17, 26, 34]. Without systematically replicated OS services, light tasks can only implement a limited set of functionalities.

When running replicated OS services over multiple coherence domains, the OS state is essentially distributed among domains. Inter-domain state synchronization must be done in software explicitly, and is much slower than hardware coherence. This challenges application development, OS engineering, and system performance, as discussed below.

Applications assume a single system image Mobile applications expect the OS to present a single system image, including a unified OS namespace and a global resource management. The single system image has served as the substrate of one million mobile applications and has been accepted by the similar number of application developers.

OS Software assume coherent state All layers of a mainstream mobile OS, from top down, assume hardware coherence. Significant changes to core services, e.g., interrupt and memory management, for coping with incoherent state are difficult and undesirable: their mechanisms and policies have been heavily tuned and proved to work over years. Rewriting extended services, such as device drivers, is costly. For instance, they constitute 70% of the Linux source code [28], and are developed by various software and hardware vendors, such as Google and Samsung.

Performance impact The absence of hardware coherence necessitates explicit inter-domain communication in coordinating OS replicas or keeping their state consistent – a much slower mechanism than hardware coherence. For example, flushing a L1 cache line takes tens of cycles, while sending an IPI usually takes a few thousand cycles. Such communication overheads will be magnified by inter-domain contention for the same shared state.

4. Design

In this section, we sketch our design. We state the design goals, describe the architectural assumptions, and then derive the shared-most OS model.

4.1 Design goals

We seek to design an OS that is capable of replicating its services over multiple coherence domains of a mobile SoC, in order to exploit hardware heterogeneity for energy efficiency. The OS design should meet the following goals:

1. Facilitate application development by relieving app developers from dealing with incoherent program state and fragmented system images.
2. Simplify OS engineering and avoid disruptive changes by maximizing reuse of legacy OS code.
3. Maintain the current performance level of demanding tasks.

4.2 Architectural assumptions

We design our system under the assumption of the following architectural features of a mobile SoC:

- **Heterogeneous cores.** The SoC has multiple types of cores that provide disparate performance-power tradeoffs. Cores may have different ISAs. Process migration is difficult and, if possible, requires sophisticated software [11, 33]. Cores have hardware MMUs.
- **Multiple coherence domains.** All cores on the SoC are isolated into a few (2-3) coherence domains, for the sake of high heterogeneity, decoupled power management and reduced thermal constraints. Within a coherence domain, multiple cores (2-8) function as a traditional multicore. There is no hardware cache coherence among domains.
- **Shared platform resources.** Coherence domains are connected to a chip-level interconnect and thus share all platform resources, including RAM and IO peripherals. Interrupts generated by IO peripherals are physically wired to all domains.
- **Aggressive power management.** Cores are taken online (active) and then offline (inactive) from time to time. The system energy efficiency is highly dependent on core power state, i.e., how long cores remain inactive and how often they are woken up.

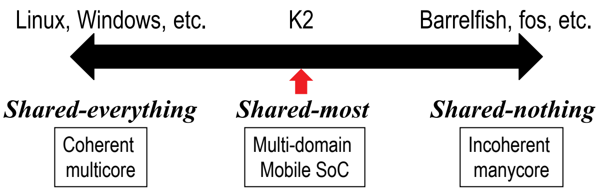


Figure 2. K2 in the spectrum of OS structures

4.3 The shared-most OS model

In order to meet the design goals stated in §4.1, we identify a shared-most OS model: *i)* to transparently maintain state coherence for extended services, or *shadowed services*, *ii)* to coordinate separated instances of core services, or *independent services*, and *iii)* to avoid multi-domain parallelism within individual process. We view our model as an outcome of considering both the underlying incoherent architecture and the programming challenges mentioned in §3. In the spectrum of OS structures shown in Figure 2, the shared-most model sits in the middle, with two extremes being 'shared-everything' monolithic OS and 'shared-nothing' OS. The two extremes, while suiting their respective target architectures well, either incur high performance overheads or heavy programming burdens for multi-domain mobile SoCs. We next discuss the three aspects of the shared-most model in detail, explaining how each of them is connected to the design goals.

Transparent shared state for extended services We argue that the OS should transparently maintain state coherence for replicas of extended services, including device drivers and file systems, for the goal of simplified OS engineering. First, as mentioned in §3, as extended services constitute the majority of an OS codebase and evolve rapidly, it is difficult to manually transform them for multiple coherence domains. Second, extended OS services are less likely to suffer from inter-domain contention, and are more tolerant of the performance overheads of software coherence, such as the inter-domain communication latency.

Independent instances of core services For a core service, the OS creates independent, per-domain instances that share no internal state. As mentioned in §3, core services such as memory management are invoked frequently and are performance-critical. In replicating such services over multiple domains, contention must be carefully avoided for performance.

Creating independent instances of core service raises two design problems: (*i*) How and when the OS should coordinate independent instances of the same service: the coordination is key to maintain a single system image, and has to be efficient as inter-domain communication can be expensive. (*ii*) How to minimize modifications to the mature implementations of core services, as a way to meet the goal of simplified OS engineering. The two problems have to be addressed on a per-service basis. In §6 and §7, we will describe our solutions for two important core services, physical page allocator and interrupt management.

Avoid multi-domain parallelism within individual process Threads belonging to the same process share an extensive set of OS state, e.g., opened files. Running them simultaneously on multiple domains may lead to expensive contention. Since an application would gain little benefit from such parallelism, we argue that such contention should be prevented by always deferring light task execution if the same process has more demanding tasks to run. The extra delay introduced to light tasks is acceptable. First, the delay is unlikely to be long: in principle, mobile demanding tasks should not saturate strong cores for more than a few hundred milliseconds for avoiding sluggish GUI. Second, the deferral only applies to light tasks belonging to the same process.

Multi-domain parallelism, however, should be supported among processes, for fairness in scheduling and encouraging applications to use the weak domain. If strong and weak domains were restricted to run alternately, all light tasks in the system, which likely belong to different processes, will block if any normal task is running, according to our model discussed above. This essentially makes light task execution dependent on other applications' behaviors, discouraging developers from placing code on the weak domain.

5. The K2 OS

In order to test the shared-most model, we build K2, an OS prototype for multi-domain mobile SoC. In this section, we give an overview of K2: we describe the hardware platform, sketch the structure of K2, present our heuristics in refactoring Linux, and discuss how K2 is built from source.

5.1 Hardware platform

We experimentally test our OS model with TI OMAP4, one of the many emerging multi-domain mobile SoCs [23, 29] that has the best public information. OMAP4 has been powering popular mobile devices of various form factors, including Amazon Kindle Fire, Samsung Galaxy S2, and Google Glass.

	Cortex-A9 (strong)	Cortex-M3 (weak)
ISA	ARM	Thumb-2
Freq.	350-1200MHz	100-200MHz
Cache	L1 64KB, L2 1MB	32KB
MMU	One ARM v7-A	Two connected in series

Table 1. Heterogeneous cores in the two coherence domains of OMAP4

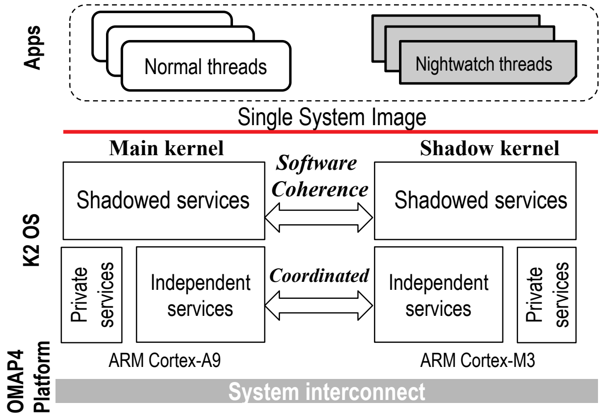


Figure 3. The structure of K2

OMAP4 has two coherence domains, each of which hosts one type of 32-bit ARM core: the performance-oriented Cortex-A9 and the efficiency-oriented Cortex-M3. Their specifications are listed in Table 1. The Cortex-M3 on OMAP4 has a non-standard MMU in which two levels are connected in series, which affects the design of K2 software coherence as will be discussed in §6.3. Each domain has dual cores and a private interrupt controller. Both domains are connected to the system interconnect, which further connects the shared RAM and all IO peripherals.

To support multiple coherence domains, OMAP4 provides two facilities. The hardware spinlocks are for inter-domain synchronization, which are a set of memory-mapped bits that support atomic test-and-set. The hardware mailboxes are for inter-domain communication: cores can pass 32-bit messages across domains with interrupting each other. We measured the message round-trip time as around 5 μs.

While K2 is geared towards OMAP4 with many platform-specific optimizations and workarounds, its design is driven by our OS model and based on the architectural assumptions in §4.2. We expect the design to be applicable to other multi-domain SoCs.

5.2 OS structure

Figure 3 depicts the structure of K2. As shown on the top of the figure, K2 carefully exposes hardware heterogeneity to application developers, by requiring them to statically partition their applications in coarse grains for heterogeneous domains. From an application developer's perspective, they develop the performance-critical parts of their applications as normal threads for execution on the strong domain; they wrap their light tasks in special *NightWatch* threads which will be pinned on the weak domain. We will discuss the details of NightWatch threads in §8.

Under the single system image, K2 runs two kernels on OMAP4: the full-fledged *main kernel* in `ARM` ISA running on the strong Cortex-A9 cores, and the lean *shadow kernel*[1] in `Thumb-2` ISA running on a weak Cortex-M3 core. The two kernels share the same virtual address space (§6.1), the same pool of physical memory (§6.2), and cooperate to handle IO interrupts (§7). As shown in Figure 3, most OS services, including device drivers and file systems, are shadowed between both kernels: they are built from the same source code and K2 transparently keeps their state coherent at run time. A few 'hotspot' core services are independent in both kernels and K2 coordinates them on the meta level. Each kernel has its private services.

5.3 Refactoring Linux

As a key step in applying the shared-most OS model, we refactor a recent Linux kernel 3.4 into both the main and shadow kernels of K2. In particular, in bringing up Linux on the Cortex-M3 of OMAP4, we adopt code from the linux-panda project [21]. The central job of refactoring is to decide how should individual Linux OS services be adopted by K2. We tackle the problem based on their dependence on processor cores, their functionalities, and the performance impact, with the following steps:

1. First, services that are specific to one type of core or that manage domain-local resources are kept private, implemented differently in each kernel, and with different state. Examples include core power management.
2. In the rest of the services, those performing complicated, rarely-used global operations will be kept private only in the main kernel. Examples include platform-level initializations.
3. In the rest of the services, those having high performance impact are replicated as independent instances in each kernel, with K2 coordinating the instances.
4. The remaining services, which manage platform resources and have low to moderate performance impact, are made as shadowed service in both kernels, with K2 automatically keeping their state coherent. In refactoring, we augment their locks by using the hardware spinlocks for inter-domain synchronization. This is the largest category which includes device drivers, file systems, and network service.

5.4 Build K2 from source

We implement both kernels of K2 in a unified source tree. To build K2, we run two passes of compilations, for the main kernel and for the shadow kernel respectively. In compilation, we apply a suite of techniques to bridge the heterogeneity gap between cores, with the help of automated scripts.

First, the compilation process makes sure each shared memory object has identical load addresses in both kernels. For both kernels, it pads data structures, enforces the same output order of memory objects in each object file, and enforces the same link order of object files. Second, the compilation process treats function pointers, which are prevalent in the Linux data structures and thus shared among two kernels. According to its ISA [4], when Cortex-M3 attempts to execute a Cortex-A9 instruction, it falls into an *unrecoverable* core state. To prevent such a situation, the compilation process statically rewrites `blx` instruction – the long jump instruction emitted by GCC for implementing function pointer dereference – with `Undef` instruction. At run time, when Cortex-M3 attempts to dereference a function pointer and thus hits `Undef`, it triggers an exception that is *recoverable*; K2 handles the exception and dispatches the control flow to the Cortex-M3 version of the function. `blx` is sparse in kernel code, constituting 0.1% of all instructions and 6% of all jump instructions.

6. Memory management

In the next three sections, we describe the core components of K2: memory management in §6, interrupt management in §7, and scheduling in §8. Among them, we will focus on memory management, which is the core mechanism in providing state coherence and unified resource management. In this section, we describe the major building blocks of the K2 memory management: the layout of kernel virtual addresses, the physical memory management, and the software coherence.

6.1 Unified kernel address space

K2 creates a unified address space for both kernels, while preserving the key assumption made by the Linux kernel virtual memory. In this section, we briefly recap the background of the Linux kernel virtual memory, describe the design constraints faced by K2, and then present our design.

Linux maps a large portion (sometimes all) of physical memory into kernel space directly. The mapping is linear and permanent: any virtual address is mapped to a physical address that only differs by a constant offset. Such direct-mapping greatly simplifies kernel virtual memory design: accessing direct-mapped memory will never trigger a page fault and conversions between kernel virtual and physical addresses are fast. As a result, direct-mapped memory usually hosts all important kernel data structures. [2]

In managing virtual memory for multiple kernels, K2 has to satisfy the following constraints:

[1] "Shadow" reflects the fact that the two kernels are kept coherent in their extended OS services.

[2] We are aware of the SPARSEMEM feature for servers, which, however, is rarely enabled for mobile devices.

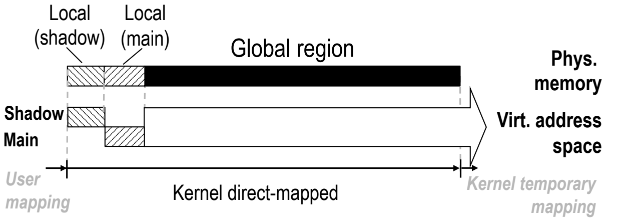


Figure 4. The unified kernel virtual address space of K2

1. Memory objects shared between kernels must have identical virtual addresses in both kernels. In addition, private memory objects should reside in non-overlapping address ranges to help catch software bugs.
2. For each kernel, the assumption of linear mapping holds for the entire direct-mapped memory.
3. Contiguous physical memory should be maximized, in particular for the main kernel which usually needs large physical memory blocks for multimedia applications.

K2 arranges its kernel address space as shown in Figure 4. From each kernel's view, its available physical memory consists of two regions, both direct-mapped. *(i)* A small *local* region hosts executable code (in local ISA) and the memory objects statically allocated for private or independent OS services (e.g., those in bss and data sections). *(ii)* All the rest of the physical memory belongs to the *global* region, which hosts the memory objects for shared OS services and all free physical pages for dynamic allocation. The global region is typically from several hundred MB to one GB.

From the start of physical memory (the left side of Figure 4), K2 populates all local regions: first for the shadow kernel and then for the main kernel. From the end of the last local region, the global region spans to the end of physical memory. By putting the main kernel's local region right before the global region, K2 avoids memory holes in the main kernel. With this memory layout, K2 keeps both kernels' virtual-to-physical offsets identical, thus essentially creating a unified virtual address space.

Temporary mapping In addition to the direct-mapped memory discussed above, the OS may need to establish temporary mapping and thus make changes to the unified kernel address space. In supporting so, K2 treats two major types of temporary mappings differently. First, the OS may need temporary mappings for accessing IO memory. As creations and destructions of such mappings are infrequent, K2 adopts a simple protocol between two kernels for propagating page table updates from one to the other. Second, on platforms with abundant memory resources, the amount of physical pages may exceed that can be directly mapped into kernel space. Thus, temporary mappings are needed on-demand in accessing the extra pages (i.e., highmem in Linux's term). In the current implementation for the 32-bit ARM, K2 does not support highmem; as a workaround, it increases the size of kernel virtual space from the default 1GB to 2GB, which is large enough for directly mapping physical memory available on today's mainstream mobile devices. We do not expect this to be a fundamental limitation, as the emerging 64-bit ARM allows kernel to directly map a much larger amount of physical memory.

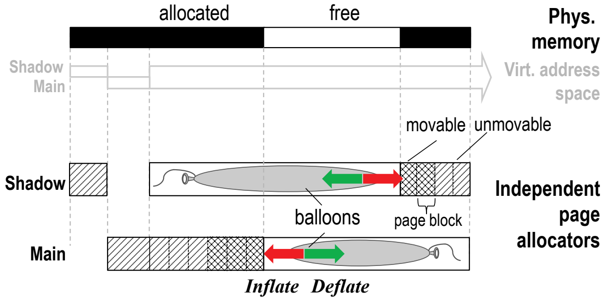


Figure 5. The physical memory management of K2

6.2 Physical memory management

Physical memory, as managed by OS page allocator, is one of the most important resources. The page allocator is frequently invoked, performance-critical, with its state among the hottest data structures in the OS. In managing physical memory for multiple coherence domains, K2 should:

1. Require minimum inter-domain communication in memory allocation and free.
2. Enable both kernels to dynamically share the entire pool of free pages.
3. Avoid disruptive changes to the existing Linux page allocator, which is already a mature design with sophisticated mechanisms and policies.
4. Minimize physical memory fragmentation.

An overview of the K2 memory management is shown in Figure 5. To achieve the first objective above, K2 applies the shared-most model and creates instances of page allocator in both the main and the shadow kernel, with separated state. Allocation requests are always served by the local instance on the same coherence domain; free requests are redirected asynchronously to the instance which allocated the pages, based on a simple address range check implemented as a thin wrapper over the existing free interface.

We next describe how K2 coordinates multiple page allocator instances to achieve objective 2-4 above, by first discussing the mechanism (balloon drivers) and then the policy (the meta-level manager).

Balloon drivers K2 retrofits the idea of balloon driver from virtual machines [39] for controlling the amount of contiguous physical memory available to individual kernels. To individual kernels, balloon driver creates the key illusion of on-demand resizable physical memory.

A balloon driver is a pseudo device driver and each kernel has its own private instance. Controlled by K2, the driver's job is to occupy contiguous physical memory to keep the memory from the local page allocator. As shown in Figure 5, to K2, a balloon driver provides two primitives that operate on physically contiguous blocks of pages, or page blocks: *deflate*, i.e., the driver frees a page block to the local page allocator, which essentially transfers the ownership of the page block from K2 to the local kernel; *inflate*, i.e., the driver allocates a page block from the kernel, by forcing the kernel to evacuate pages from that page block, which essentially transfers the ownership of the page block from the local kernel back to K2. In the early stage of kernel boot, balloon drivers of both kernel are initialized to occupy the entire shared region so that K2 owns all physical pages in that region.

We implement K2 balloon drivers based on Linux's contiguous memory allocation framework. Balloon drivers require no change to the Linux page allocator: from each kernel's perspective, a balloon behaves like a common device driver that reserves a large contiguous memory region at boot time and later allocates and frees page blocks within that region autonomously.

Meta-level manager On top of the balloon drivers, K2 provides a meta-level manager to decide when to take and give page blocks from and to kernels. The meta-level manager is implemented as a set of distributed probes, one in each kernel. Each probe monitors local memory pressure with hooks inserted into the local kernel page allocator; the probes coordinate through hardware messages, and take actions by invoking local balloon drivers. Generally, when memory pressure increases and before the local kernel reacts to the pressure (e.g., activating the Android low-memory killer), the meta-level manager instructs the balloon driver to free page blocks to alleviate the pressure; when free pages run low in the entire system, the meta-level manager instructs a balloon driver to inflate for reclaiming pages. Like the Linux kernel swap daemon, the meta-level manager performs operations in the background when OS is idle in order to minimize performance impact on individual allocations.

Optimizations K2 has applied a few optimizations in physical memory management, as shown in Figure 5. First, to reduce inter-domain communication, the meta-level manager operates on large-grain page blocks, which are 16MB in the current implementation. Second, to maximize the size of physically contiguous area available to the main kernel, the meta-level manager instructs balloons to deflate from the two ends of the free portion in the shared region, and inflates in the reverse directions. Thus, blocks allocated to the main kernel grow from right after its private region. Third, to maximize the chance of successfully reclaiming page blocks from kernels, K2 commands local page allocators to place *movable* pages (e.g., those containing user data) close to the 'frontier' of the balloon with best efforts. The efforts are likely to succeed, as movable pages usually constitute 70%-80% of total pages based on our experiments with mobile systems. As a result, when the balloon inflates, those movable pages can be evacuated from the requested page block to elsewhere.

6.3 Software coherence

As mentioned in §5.2, K2 provides transparent coherence for shadowed services. This is implemented with software Distributed Shared Memory (DSM), a classic approach to hide distributed program state from programmers [16], which also has been applied to recent incoherent architectures such as CPU-GPU systems [14] and smartphones [17]. Like most DSM designs, the K2 DSM maintains the key one-writer invariant [16], by automatically translating memory accesses to inter-domain communication for coherence. While most software DSMs are designed to back user applications [16, 30], the K2 DSM backs OS services. Thus, we carefully construct fault handlers and coherence communication to avoid interference or lockup with other OS components.

Basic design The K2 DSM implements sequential consistency. Although many DSMs [14, 17] implement relaxed consistency which relies on applications' correct locking behaviors, K2 neither relaxes the consistency model assumed by the existing OS, nor makes any assumption of OS locking behaviors which may be complicated. In implementing sequential consistency, the K2 DSM performs coherence operations upon each access fault and keeps the order of coherence messages. The K2 DSM adopts a page-based granularity, using 4KB page as the smallest memory unit that is kept coherent. This is for leveraging MMU to trap accesses to shared memory, as page is the unit used by the MMU for enforcing protection.

The K2 DSM has adopted a simple, standard two-state protocol. For each shared page, every kernel keeps track of its state, being `Valid` or `Invalid`. Kernels communicate with two types of coherence messages: *GetExclusive* and *PutExclusive*. When a page is `Valid`, a kernel can perform read or write of the page. Before performing read or write of a `Invalid` page, a kernel must send *GetExclusive* to the other kernel who currently owns the page; upon receiving *GetExclusive*, the latter kernel flushes and invalidates the requested page from its local cache, and acknowledges with a *PutExclusive* message. After that, the former kernel can proceed to access the memory.

The K2 DSM detects accesses to shared memory as follows: when a page transits from `Valid` to `Invalid` state, the DSM modifies the corresponding page table entry to be ineffective and handles the resulting page fault triggered by the subsequent access to the page. The fault handling is transparent to the OS code that made the memory access.

Coherence communication For performance, the coherence communication directly leverages the OMAP4 hardware mailboxes. Each message is 32-bit, the size of a hardware mail, with 20 bits for page frame number, 3 bits for message type, and the rest for message sequence number. The mailbox will guarantee that messages are delivered in order.

We carefully construct coherence communication to avoid lockup. The communication must be synchronous to the requester: this is because interrupt handlers, which cannot sleep, may access shared state and thus initiates communication. Thus, when a requester kernel sends out a *GetExclusive* message, it spins waiting until the destination kernel sends back a *PutExclusive* message. For the same reason, handling *GetExclusive* must avoid sleeping as well, e.g., it must use atomic memory allocation. More importantly, handling *GetExclusive* must avoid accessing shared state which may trigger new page faults, resulting in an infinite request loop between kernels.

We optimize the communication latency in favor of the main kernel, by setting their priorities of handling coherence messages differently. The main kernel handles *GetExclusive* in bottom halves, and will further defer the handling if under high workloads; in contrast, the shadow kernel handles the request before any other pending interrupt. Through this, K2 reduces the impact of weak cores on system performance.

Optimize memory footprint The K2 DSM incurs a low overhead in storing protocol information, asking for three bits per page. Furthermore, although shared memory areas have to be mapped in 4KB granularity, the K2 DSM allows non-shared areas to be mapped in larger grains (1 MB or 16 MB) as supported by the ARM hardware, thus reducing the size of page tables and alleviating TLB pressure. To achieve this, the K2 DSM turns on software coherence for memory addresses and replaces the existing large-grain mapping on-demand, only when an address (or its neighbouring area) has been accessed by both kernels.

An alternative design While we are aware of a more common three-state protocol [35] that supports read-only sharing, our choice of the two-state protocol is based on the hardware limitation of the Cortex-M3 MMU on OMAP4.

In general, supporting read-only sharing requires DSM to use MMU for differentiating memory reads from writes in order to handle them separately. However, this is expensive on the OMAP4 Cortex-M3, which relies on the first level of its two cascading MMUs for read/write permissions. The first-level MMU has no page table but just a software-loaded TLB which only contains ten entries for 4KB pages. According to our experiment, using the MMU for read access detection puts a high pressure on its TLB and leads to severe thrashing. In contrast, without supporting read-only sharing, the K2 DSM detects both read and write accesses solely with the second-level MMU, which has a larger TLB and a hardware page table walker.

7. Interrupt management

Multiple coherence domains share interrupts generated by IO peripherals. Although any interrupt signal is physically delivered to all domains, K2 must ensure that it is only handled by exactly one kernel. If multiple kernels compete for the same interrupt signal, peripherals may enter incorrect states or cores may have spurious wakeups. The choice of kernel in handling interrupts does not affect OS correctness, thanks to the K2 software coherence; however, it does affect performance and efficiency.

K2 coordinates its two kernels in handling shared interrupts, by following two simple rules. First, for energy efficiency, K2 does not allow shared interrupts to wake up the strong domain from an inactive power state, in which case the shadow kernel should handle the interrupts. Second, for performance, when the strong domain is awake, K2 lets the main kernel handle all shared interrupts.

K2 has implemented the rules on the OMAP4 platform, where coherence domains have private interrupt controllers. K2 inserts a few hooks into the Linux power management code to configure interrupt controllers. When the shadow kernel boots on a weak domain, it masks all shared interrupt locally. When a strong domain transits to an inactive state, K2 unmasks all shared interrupts on the weak domain and masks them on the strong domain; when the strong domain is woken up from the inactive state, K2 reverses such operations, by masking shared interrupts on the weak domain and unmasking them on the strong domain.

8. Nightwatch threads

As mentioned in §5.2, K2 provides an abstraction called *NightWatch* thread for application developers to implement light tasks. NightWatch threads are pinned on a weak domain; after being created, a NightWatch thread enters the shadow kernel runqueue for execution. From a developer's view, a NightWatch thread is identical to a normal thread: for instance, they share a unified process address space and a single system image. The only exception is that in order to limit multi-domain parallelism (the third aspect of the shared-most model in §4.3), K2 enforces that:

A NightWatch thread will only be considered for scheduling when all normal threads of the same process are suspended.

The scheduling strategy, together with the single system image, distinguishes NightWatch threads from other abstractions that also encapsulate code for targeting heterogeneous hardware [17]. It is worth noting that K2 does not change the mechanism or policy of the Linux scheduler; all normal threads are scheduled as they are in Linux.

Preempt NightWatch threads K2 preempts the execution of all NightWatch threads in a process when any normal thread of the same process is about to execute. After the main kernel decides to schedule-in a normal thread, it examines if any NightWatch threads belong to the same pro-

Existing Implementations	**Changed SLoC**	**Original SLoC**
Exception handling	1151	395
Page allocator, interrupt, scheduler	205	12119
New Implementations	**SLoC**	
DSM	883	
Memory management	468	
Bootstrap	1306	
OMAP4-specific Cortex-M3	772	
Debugging, etc.	1362	
Total	4791	

Table 2. Source code changes and additions to the Linux kernel 3.4, which contains around 10 million SLoC. Much of the OMAP4-specific Cortex-M3 code is adopted from the linux-panda project [21].

cess. If so, the kernel sends a `SuspendNW` hardware message to the shadow kernel. Interrupted by the `SuspendNW` message, the shadow kernel immediately responds with an `AckSuspendNW` message and then removes all NightWatch threads of the same process from the local runqueue by flagging them. On receiving the `AckSuspendNW`, the main kernel lands the normal thread on a core for execution.

To reduce latency, the main kernel overlaps the wait for `AckSuspendNW` with the context switch to the schedule-in thread: after sending `SuspendNW`, the main kernel proceeds to context switch, and only waits for `AckSuspendNW` after the context switch is done, before returning to user space. Given that a message round trip takes around 5 μs and a context switch usually takes 3-4 μs, the extra overhead for the main kernel is 1-2 μs for every context switch.

Resume NightWatch threads When the main kernel finds that all normal threads of a process blocked, e.g., waiting for IO, it sends a `ResumeNW` message to the shadow kernel. On receiving the message, the shadow kernel removes flags from all NightWatch threads of the given process and will consider them in future scheduling.

9. Evaluation

We evaluate how well K2 meets the design goals we set in §4.1 by reporting our efforts in refactoring Linux and testing the energy efficiency harvested by K2. We then evaluate the benefits and overheads of the shared-most model by benchmarking the physical page allocator and the DMA device driver.

9.1 Efforts in refactoring

In building K2, we have achieved the goal of reusing mature OS source at the refactoring cost shown in Table 2. K2 introduces small changes to the Linux source. The biggest portion of changes is exception handling, where K2 handles page faults for DSM operations and dispatches function pointers. To core OS components such as the page allocator, K2 only adds a small portion of source lines. Device drivers, such as the one for DMA, can be reused by K2 with few modifications. K2 introduces a set of new software modules to implement its core components like DSM and memory management. It also includes extensive debugging support to help ourselves understand Linux.

	Active	**Idle**
Cortex-M3 (200MHz)*	21.1	3.8
Cortex-A9 (350MHz)*	79.8	25.2
Cortex-A9 (1200MHz)	672	25.2

Table 3. Power consumptions of the heterogeneous OMAP4 cores, in mW. Both cores consumes less than 0.1 mW when inactive. The frequencies with asterisks are used in the benchmarks for testing energy efficiency.

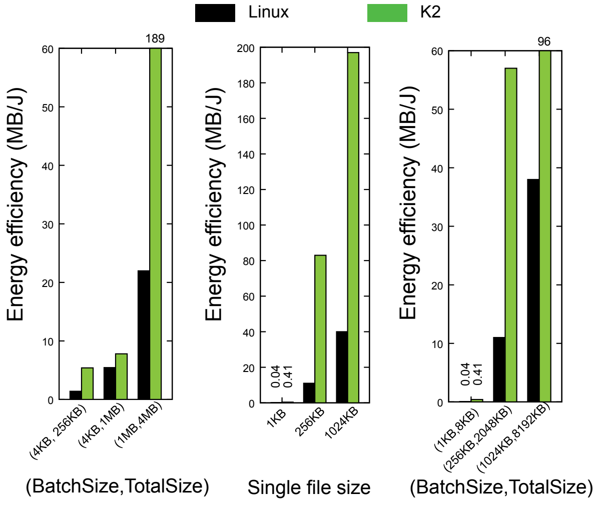
9.2 Energy efficiency benefits

We show that K2 greatly improves energy efficiency for light tasks, by running a series of OS benchmarks to exercise K2 and the original Linux 3.4. In the benchmarks, we measure power consumption of coherence domains by sampling current on their separate power rails [12]. We measure elapsed time using hardware performance counters when cores are active, and using a 32KHz platform timer when idle.

In the benchmarks, K2 is able to use the weak core for OS execution, while Linux can only use the strong core. As summarized in Table 3, we configure the platform to favor the energy efficiency of Linux: we fix the strong core at its *most* efficient operating point while the weak core has to run at its *least* efficient operating point. This is because the OMAP4 implementation limits DVFS on the weak core, by coupling its voltage with that of the system interconnect and RAM. There is no way to scale down the voltage without crashing the entire SoC, to the best of our knowledge.

Our benchmarks test energy efficiency of three representative OS services: device driver, file system, and network stack. The benchmarks mimic mobile light tasks: in each run of a benchmark, cores are woken up, execute the workloads as fast as possible, and then stay idle until becoming inactive. We set the core inactive timeout as 5 sec, as reported in a recent study of mobile device power [41].

DMA driver We choose the DMA driver as a representative device driver for testing, which is used in almost all bulk IO transfers, e.g., for flash and WiFi. The driver executes DMA transfers by operating the OMAP4 DMA engine. As shown in Figure 6(a), we pick a set of DMA transfer sizes typical to light tasks. Each run of the benchmark repeatedly invokes the DMA driver to execute multiple memory-to-memory transfers, where each transfer copies *BatchSize* bytes and for a total of *TotalSize* bytes copied. In each transfer, the DMA driver clears the destination memory region, looks for empty resources, programs the DMA engine and initiates the transfer. When the transfer is done, the DMA



(a) DMA (b) Ext2 (c)UDP loopback

Figure 6. OS benchmarks for testing energy efficiency

driver is interrupted to free resources and complete the transfer. We calculate the energy efficiency as the amount of transferred bytes per Joule.

ext2 file system We choose ext2, a file system widely used for flash storage, to test the energy efficiency of file system workloads. We use ramdisk as the underlying block device, as the SD card driver of K2 is not yet fully functional. This favors the energy efficiency of Linux: ramdisk is a much faster block device than real flash storages; using it shortens idle periods that are more expensive to strong cores. We design the benchmark to mimic a light task synchronizing contents from the cloud. As shown in Figure 6(b), for each run, a NightWatch thread operates on eight files sequentially: it creates a file, writes to it and closes it. We vary the write size to represent different content types: 1KB (emails), 256KB (pictures), and 1MB (short videos). We calculate the energy efficiency as the number of bytes written per Joule.

UDP loopback To test network workloads efficiency, we design a UDP loopback benchmark to mimic the networking activities when light tasks fetch contents from the cloud. In the benchmark, a NightWatch thread creates two UDP sockets; it writes to one and reads from the other one for a total number of *TotalSize* bytes at full speed. Once every *BatchSize* bytes transferred, the thread destroys both sockets and recreates new ones. As shown in Figure 6(c), we choose sizes to represent typical contents as done for the ext2 benchmark. We calculate the energy efficiency as the amount of bytes sent per Joule.

Results analysis In all three benchmarks, K2 shows significant advantages of energy efficiency over Linux, by improving the energy efficiency by up to 9x, 8x, and 10x, respectively. Based such results and the mobile device usage reported in recent literature [41], we estimate that K2 will extend the reported device standby time by 59%, from 5.9 days to 9.4 days.

Allocation Size	**Main**	**Shadow**
4KB	1	12
256KB	5	45
1024KB	13	146
Balloon		
deflate	10,429	12,813
inflate	11,612	20,408

Table 4. Latencies of physical memory allocations in K2, in μs. The performance of K2's main kernel allocator has no noticeable difference from the Linux page allocator.

Overall, K2 gains the advantages mainly by exploiting the much lower-power idle periods of the weak core, which exist within each OS invocation (such as file operations in the ext2 benchmark) as well as in the inactive timeout periods. This is shown by the fact that when workloads are more IO-bound (e.g., the batch size of DMA transfers increases) or when the total task size of each run is smaller (e.g., in UDP loopback with fewer total sent bytes), the advantages of K2 are even more significant. In addition, over all benchmarks, K2 is able to use the weak core to deliver peak performance that is 20%-70% of the strong core performance at 350MHz, which can accommodate a wide range of light tasks that have low to moderate performance expectations.

9.3 Benefits of independent service

In order to evaluate our model of independent core services, we study the performance of the K2 physical memory management, with a microbenchmark to exercise the independent page allocators as well as the meta-level manager. In the benchmark, we measure the latency in allocating memory of different sizes, and in balloon deflating and inflating.

As shown in Table 4, K2 preserves the memory allocation performance of Linux. In comparing the memory allocation latency of the K2 main kernel with that of Linux, we find no noticeable difference: with separate state, the allocators of two kernels can operate independently without communication for most of the time. For each allocation, the pressure-monitoring probes inserted by the K2 meta-level manager incur less than twenty instructions, which is negligible compared to the allocation time. A balloon operation is more expensive: it intensively updates the page allocator state and moves pages, introducing around ten milliseconds latency that may be perceivable to users. This validates our design of triggering them asynchronously in the background. With allocation and free operations interleaved in practice and K2's large-grain page blocks, we expect that balloon operations are triggered infrequently and thus have their overheads amortized over a large number of memory allocations.

To contrast with K2's independent page allocators, we attempted but found it infeasible to implement the page allocator as a shadowed service. The contention between

	GetExclusive **sender**	
Operations	**Main**	**Shadow**
Local fault handling	3	17
Protocol Execution	2	13
Inter-domain communication	5	9
Servicing request	24	7
Exit fault, cache miss	18	2
Total	52	48

Table 5. A breakdown of the latency in a DSM page fault, in μs.

	DMA BatchSize			
	4K	128K	256K	1M
Linux	37.8	40.3	40.3	40.5
K2	35.7	39.9	40.5	43.1
	(-5.5%)	(-1.0%)	(+0.5%)	(+%6.4)
K2:Main	35.6	28.4	28.6	28.8
K2:Shadow	0.1	11.5	11.9	14.3

Table 6. DMA throughputs of K2 when the DMA driver is invoked in both kernels concurrently, in MB/Sec. The throughput differences as compared to the original Linux are also shown.

coherence domains is very high, incurring four to five DSM page faults in every allocation, leading to a 200x slowdown. What is even worse, OS lockups happen frequently and are difficult to debug.

9.4 Performance of shadowed service

With transparent coherence provided by the DSM, K2's shadowed OS services are able to achieve performance very close to Linux. We show the latency of a single DSM page fault and its breakdown in Table 5. Although further optimizations are possible (e.g., local fault handling and protocol execution contribute more than 30% of the latency), the DSM provides acceptable performance to most extended OS services where sharing usually happens at a time scale of milliseconds or seconds.

We examine the performance of shadowed services by running a DMA driver benchmark on the main and the shadow kernel concurrently. In each run of the benchmark, both kernels repeatedly invoke the DMA driver to execute transfers at full speed with a given batch size. The run is repeated for multiple times with different batch sizes. As a comparison, we also run the same benchmark using the original Linux kernel that uses the strong domain only. The DMA throughputs of the Linux and the K2 kernels, together with their differences, are summarized in Table 6.

As shown in the results, in backing the DMA shadowed service with software coherence, K2 incurs low overhead, reducing the throughput by up to 5.5%. Since K2 runs two kernels from two domains simultaneously, the throughput reduction seen by the main kernel is contributed by two factors: software coherence overhead incurred by K2 and the DMA engine bandwidth allocated to the shadow kernel. With small batch sizes such as 4KB, the benchmark is CPU-bound: the contention for shared kernel state is relatively high and the DMA engine is not fully utilized. Due to the asymmetry in processor performance and in K2's design, the coherence overhead is low – a 50 μs DSM page fault happens around every 10 transfers, or 1200 μs. As batch size increases, the benchmark is becoming more IO-bound; the shadow kernel starts to get better chances in competing for the bandwidth of DMA engine. In this situation, as the rate of accessing shared state decreases, the coherence overhead is even lower – one DSM page fault in every 18 ms. Thus, the DMA engine serving two domains has a higher utilization, leading to a small increase in throughput (by up to 6%) as compared to the Linux case.

10. Related Work

K2 is related to a few areas in systems research. Hardware heterogeneity is a recognized way to improve energy efficiency. Driven by the pursuit of heterogeneity, mobile architectures with multiple coherence domains that can operate independently are popular in both industry [23, 36] and academia [1, 17, 26, 34]. However, existing system support has focused on user policies [26] and programming models as in our prior work Reflex [17]; none offers OS support as K2 does. The lack of OS support will defeat the goal of energy efficiency: as shown by a recent study of smartphone workloads [18], demanding and non-demanding tasks share an extensive set of OS services, which must be executed on each coherence domain. In particular, our prior work Reflex [17] features a DSM that transparently maintains state consistency for applications. The user-level DSM of Reflex is complementary to the OS-level DSM of K2; unlike in Reflex, the K2 DSM targets generic OS workloads, and thus has performance as one of the major goals, e.g., by leveraging hardware MMUs.

Programming on top of multiple kernels can be made a lot easier with a single system image. Many systems provide single system images by forwarding service requests among kernels. For example, the V distributed system [10] distributes microkernels over workstations and forwards requests to OS servers via network. Multicelluar OSes such as Hive [9] and Cellular IRIX [31] provide single system images over multiple cells by running independent kernels which cooperate using explicit communication, e.g., RPC. Since kernels are independent, an IO request must be served in the only cell where the IO device is located. Disco [8], which runs different unmodified OSes on a single machine, does coordination at user-level with standard network protocols, thus providing a partial single system image. fos [40], Helios [22], and NIX [5] dedicate a set of cores or hardware accelerators to specific OS services, and send them the corresponding services requests with explicit messages. Unlike all of them, K2 must realize a single system image by replicating OS services, in order to execute the same OS services on each domain for energy efficiency.

To improve resource utilization, unified resource management is relevant when multiple kernels share the same resource pool. Statically partitioning resources is wasteful and thus undesirable. Many research OSes [6, 40] redesign their local resource managers to cooperate at fine grains, which is effective yet difficult to apply to mature mobile OSes. Similarly, Libra [2] enables a general-purpose OS and multiple specialized library OSes to share physical memory dynamically; it also employs redesigned memory managers in the specialized OSes. Virtual machines [39] enable resource sharing among multiple OSes while minimizing modifications to them. For memory management, K2 retrofits the balloon driver idea from virtual machines, and additionally moves physically contiguous regions around individual kernels, without assuming virtualization hardware.

Reducing sharing in OS has been extensively studied for shared memory machines, mostly for scalability. Many argue for structural overhauls. Hurricane [38] organizes OS in a hierarchical way to improve scalability. Barrelfish [6] takes an extreme design point to make all OS state non-shared by default. fos [40] argues to replicate given OS services to a subset of processors and coordinate them with messages. New programming abstractions have been proposed, too. Tornado [13] and K42 [3] argue for object-oriented OS designs, enabling each OS service to have its internal mechanism to distribute over SMP. Inspired by them, K2 treats OS services differently in how to replicate them over coherence domains; unlike them, K2 targets reusing legacy OS code rather than a clean-slate implementation. For manycore machines, Corey [7] enables applications to control sharing within processes. Inspired by it, K2 employs NightWatch thread as a hint of performance expectation.

In reusing mature OSes for new architectures, virtualization and refactoring are two major approaches. While K2 borrows idea from virtual machines for memory management, the latter usually sets out to enforce isolation among OS instances, rather than providing a single system image over them. Refactoring, as done in DrawBridge [25] and Unikernels [19] recently, restructures OS while keeping individual OS components. Their approaches are inspiring, but solve a different problem – rearranging OS services vertically across layers. In comparison, for K2 we refactor Linux by replicating OS services horizontally across coherence domains.

11. Concluding Remarks

Discussion K2 is designed for a typical mobile SoC which consists of a few heterogeneous domains. As compute resources keep increasing, we next discuss how K2 should be adapted for the following architecture trends.

First, individual domains will accommodate more cache-coherent cores. Since individual kernels of K2 already have the Linux SMP support, K2 can (almost) transparently scale with these additional cores.

Second, one system may embrace *more*, but not *many*, types of heterogeneous domains, as determined by the current spectrum of mobile workloads. For N domains (N being moderate), K2 can be extended without structural changes: the DSM (§6.3) will track page ownership among N domains as in [17]; the unified kernel address space (§6.1) will host the additional N-2 local virtual regions; the global physical region will still be managed by balloon drivers (§6.2), yet the relative locations of N local physical regions depend on the intended use of the added domains. Overall, the asymmetric aspects of K2 (e.g., page allocation favoring strong domains) will remain.

Another possibility is that a system incorporates *many* coherence domains of the same type, as seen on some scalability-oriented systems including the 48-domain Intel SCC [20]. Our shared-most model (§4.3) will be useful, while new OS implementations should be engineered for scalability. However, we do not expect to see so many domains on a mobile device in the foreseeable future.

In developing K2, we find that the following architectural features will greatly benefit system performance and efficiency, yet are still missing in today's multi-domain SoCs: direct channels for inter-domain communication that bypass the system interconnect, efficient MMUs for weak domains with permission support, and finer-grained power domains.

Conclusion Multi-domain SoCs promise high energy efficiency to a wide range of mobile applications, yet are difficult to program. To address this challenge, we identify a shared-most OS model for the architecture, and argue for transparently maintaining state coherence for extended OS services while creating per-domain, shared-nothing instances of core OS services. By applying the model, we build K2, a prototype OS on the TI OMAP4 SoC. K2 presents a single system image and reuses most of the Linux kernel source. Although still in an early stage under development, K2 improves energy efficiency for light OS workloads by up to 10x and provides almost the same performance as Linux, showing that the shared-most model is promising. The source of K2 is available from `http://www.k2os.org`.

Acknowledgments

This work was supported in part by NSF CAREER Award #1054693. The authors thank NICTA researchers for publishing their `linux-panda` source code, and the anonymous reviewers for their useful comments.

References

[1] Y. Agarwal, S. Hodges, R. Chandra, J. Scott, P. Bahl, and R. Gupta. Somniloquy: Augmenting network interfaces to reduce PC energy usage. In *Proc. USENIX Symp. Networked*

Systems Design and Implementation (NSDI), pages 365–380, 2009.

[2] G. Ammons, J. Appavoo, M. Butrico, D. Da Silva, D. Grove, K. Kawachiya, O. Krieger, B. Rosenburg, E. Van Hensbergen, and R. W. Wisniewski. Libra: a library operating system for a jvm in a virtualized execution environment. In *Proc. Int. Conf. Virtual Execution Environments (VEE)*, pages 44–54. 2007.

[3] J. Appavoo, D. D. Silva, O. Krieger, M. Auslander, M. Ostrowski, B. Rosenburg, A. Waterland, R. W. Wisniewski, J. Xenidis, M. Stumm, et al. Experience distributing objects in an SMMP OS. *ACM Transactions on Computer Systems (TOCS)*, 25(3):6, 2007.

[4] ARM. ARM v7-M architecture reference manual, 2010.

[5] F. J. Ballesteros, N. Evans, C. Forsyth, G. Guardiola, J. McKie, R. Minnich, and E. Soriano. Nix: An operating system for high performance manycore computing. *Bell Labs Technical Journal*, 2012.

[6] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new os architecture for scalable multicore systems. In *Proc. ACM Symp. Operating Systems Principles (SOSP)*, pages 29–44. 2009.

[7] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, M. F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y.-h. Dai, Y. Zhang, and Z. Zhang. Corey: An operating system for many cores. In *Proc. USENIX Conf. Operating Systems Design and Implementation (OSDI)*, pages 43–57, 2008.

[8] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 15(4):412–447, 1997.

[9] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive: Fault containment for shared-memory multiprocessors. In *Proc. ACM Symp. Operating Systems Principles (SOSP)*, pages 12–25, 1995.

[10] D. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, 1988.

[11] M. DeVuyst, A. Venkat, and D. M. Tullsen. Execution migration in a heterogeneous-isa chip multiprocessor. In *Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, pages 261–272, 2012.

[12] eLinux.org. PandaBoard Power Measurements. `http://elinux.org/PandaBoard_Power_Measurements`.

[13] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proc. USENIX Conf. Operating Systems Design and Implementation (OSDI)*, pages 87–100, 1999.

[14] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W.-m. W. Hwu. An asymmetric distributed shared memory model for heterogeneous parallel systems. In *Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, pages 347–358, 2010.

[15] P. Greenhalgh. Big.LITTLE processing with ARM Cortex-A15 and Cortex-A7. Technical report, 2011.

[16] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, Nov. 1989.

[17] F. X. Lin, Z. Wang, R. LiKamWa, and L. Zhong. Reflex: using low-power processors in smartphones without knowing them. In *Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, pages 13–24, 2012.

[18] F. X. Lin, Z. Wang, and L. Zhong. Supporting distributed execution of smartphone workloads on loosely coupled heterogeneous processors. In *Proc. Workshp. Power-Aware Computing and Systems (HotPower)*, pages 2–2, 2012.

[19] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library operating systems for the cloud. In *Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, pages 461–472. 2013.

[20] T. G. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe. The 48-core SCC processor: the programmer's view. In *Proc. ACM/IEEE Int. Conf. High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–11. 2010.

[21] NICTA. Linux-panda project. `http://www.ertos.nicta.com.au/downloads/linux-panda-m3.tbz2`, 2012.

[22] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt. Helios: heterogeneous multiprocessing with satellite kernels. In *Proc. ACM Symp. Operating Systems Principles (SOSP)*, pages 221–234, 2009.

[23] NVIDIA. Tegra2 Family: Technical reference manual, 2011.

[24] NVIDIA. Tegra3 HD mobile processors: Technical reference manual, 2012.

[25] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt. Rethinking the library os from the top down. In *Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, pages 291–304, 2011.

[26] B. Priyantha, D. Lymberopoulos, and J. Liu. Littlerock: Enabling energy-efficient continuous sensing on mobile phones. *Pervasive Computing, IEEE*, 10(2):12–15, 2011.

[27] M.-R. Ra, B. Priyantha, A. Kansal, and J. Liu. Improving energy efficiency of personal sensing applications with heterogeneous multi-processors. In *Proc. Int. Conf. Ubiquitous Computing (UbiComp)*, pages 1–10. 2012.

[28] L. Ryzhyk, P. Chubb, I. Kuz, and G. Heiser. Dingo: Taming device drivers. In *Proc. The European Conf. Computer Systems (EuroSys)*, pages 275–288. 2009.

[29] Samsung. Exynos 4210 application processor. `http://www.samsung.com/global/business/semiconductor/productInfo.do?fmly_id=844&partnum=Exynos%204210`.

[30] D. Scales, K. Gharachorloo, and C. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. *ACM SIGOPS Operating Systems Review*, 30(5):174–185, 1996.

[31] SGI. Cellular IRIX 6.4 technical report. `http://www.sgistuff.net/software/irixintro/documents/irix6.4TR.html`.

[32] Y. Shin, K. Shin, P. Kenkare, R. Kashyap, H.-J. Lee, D. Seo, B. Millar, Y. Kwon, R. Iyengar, M.-S. Kim, A. Chowdhury, S.-I. Bae, I. Hong, W. Jeong, A. Lindner, U. Cho, K. Hawkins, J. C. Son, and S. H. Hwang. 28nm high-metal-gate heterogeneous quad-core CPUs for high-performance and energy-efficient mobile application processor. In *Proc. IEEE Int. Solid-State Circuits Conf. (ISSCC)*, pages 154–155. 2013.

[33] P. Smith and N. C. Hutchinson. Heterogeneous process migration: The Tui system. *Software-Practice and Experience*, 28(6):611–640, 1998.

[34] J. Sorber, N. Banerjee, M. D. Corner, and S. Rollins. Turducken: hierarchical power management for mobile devices. In *Proc. USENIX/ACM Int. Conf. Mobile Systems, Applications, & Services (MobiSys)*, pages 261–274. 2005.

[35] D. J. Sorin, M. D. Hill, and D. A. Wood. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture*, 6(3):1–212, 2011.

[36] Texas Instruments. OMAP4 applications processor: Technical reference manual. `http://www.ti.com/product/OMAP4470`, 2010.

[37] Texas Instruments. OMAP543x: Technical reference manual. `http://www.ti.com/litv/pdf/swpu249v`, 2010.

[38] R. C. Unrau, O. Krieger, B. Gamsa, and M. Stumm. Hierarchical clustering: A structure for scalable multiprocessor operating system design. *The Journal of Supercomputing*, 9(1-2):105–134, 1995.

[39] C. A. Waldspurger. Memory resource management in VMware ESX server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, Dec. 2002.

[40] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *SIGOPS Oper. Syst. Rev.*, 43(2):76–85, 2009.

[41] F. Xu, Y. Liu, T. Moscibroda, R. Chandra, L. Jin, Y. Zhang, and Q. Li. Optimizing background email sync on smartphones. In *Proc. USENIX/ACM Int. Conf. Mobile Systems, Applications, & Services (MobiSys)*, 2013.

[42] L. Zhong and N. K. Jha. Dynamic power optimization targeting user delays in interactive systems. *IEEE Trans. Mobile Computing*, 5(11):1473–1488, 2006.