Transkernel: Bridging Monolithic Kernels to Peripheral Cores

Liwei Guo, Shuang Zhai, Yi Qiao, and Felix Xiaozhu Lin, *Purdue ECE*

https://www.usenix.org/conference/atc19/presentation/guo

This paper is included in the Proceedings of the
2019 USENIX Annual Technical Conference.

July 10–12, 2019 • Renton, WA, USA

ISBN 978-1-939133-03-8

Transkernel: Bridging Monolithic Kernels to Peripheral Cores

Liwei Guo, Shuang Zhai, Yi Qiao, and Felix Xiaozhu Lin

Purdue ECE

Abstract

Smart devices see a large number of ephemeral tasks driven by background activities. In order to execute such a task, the OS kernel wakes up the platform beforehand and puts it back to sleep afterwards. In doing so, the kernel operates various IO devices and orchestrates their power state transitions. Such kernel executions are inefficient as they mismatch typical CPU hardware. They are better off running on a low-power, microcontroller-like core, i.e., peripheral core, relieving CPU from the inefficiency.

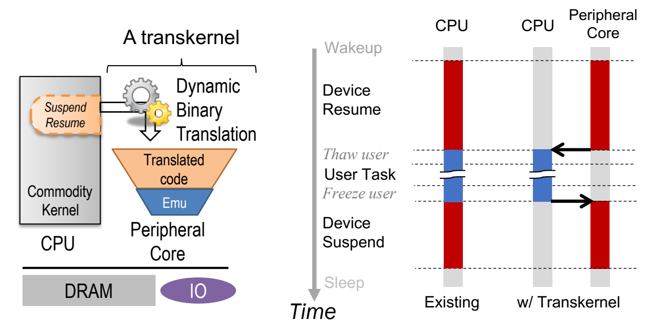
We therefore present a new OS structure, in which a lightweight virtual executor called *transkernel* offloads specific phases from a monolithic kernel. The transkernel translates stateful kernel execution through cross-ISA, dynamic binary translation (DBT); it emulates a small set of stateless kernel services behind a narrow, stable binary interface; it specializes for hot paths; it exploits ISA similarities for lowering DBT cost.

Through an ARM-based prototype, we demonstrate transkernel's feasibility and benefit. We show that while cross-ISA DBT is typically used under the assumption of efficiency *loss*, it can enable efficiency *gain*, even on off-the-shelf hardware.

1 Introduction

Driven by periodic or background activities, modern embedded platforms[1] often run a large number of ephemeral tasks. Example tasks include acquiring sensor readings, refreshing smart watch display [44], push notifications [38], and periodic data sync [91]. They drain a substantial fraction of battery, e.g., 30% for smartphones [13, 12] and smart watches [45], and almost the entire battery of smart things for surveillance [84]. To execute an ephemeral task, a commodity OS kernel, typically implemented in a monolithic fashion,

[1]This paper focuses on battery-powered computers such as smart wearables and smart things. They run commodity OSes such as Linux and Windows. We refer to them as embedded platforms for brevity.



(a) The transkernel model (b) System execution workflow

Figure 1: An overview of this work

drives the whole hardware platform out of deep sleep beforehand (i.e., "resume") and puts it back to deep sleep afterwards (i.e., "suspend"). During this process, the kernel consumes much more energy than the user code [44], up to 10× shown in recent work [38].

Why is the kernel so inefficient? Recent studies [26, 92, 44] show the bottlenecks as two kernel phases called *device suspend/resume* as illustrated in Figure 1. In the phases, the kernel operates a variety of IO devices (or *devices* for brevity). It invokes device drivers, cleans up pending IO tasks, and ensures devices to reach expected power states. The phases encompass concurrent execution of drivers, deferred functions, and hardware interrupts; they entail numerous CPU idle epochs; their optimization is proven difficult (§2) [92, 50, 46].

We deem that device suspend/resume mismatches CPU. It instead would be much more efficient on low-power, microcontroller-like cores, as exemplified by ARM Cortex-M. These cores are already incorporated as *peripheral cores* on a wide range of modern system-on-chips (SoCs) used in production such as Apple Watch [59] and Microsoft Azure Sphere [52]. On IO-intensive workloads, a peripheral core delivers is much more efficient than the CPU due to lower idle power and higher execution efficiency [42, 43, 54, 1]. Note that running *user code* (which often builds atop POSIX) on peripheral cores is a non-goal: on one hand, doing so would gain much less efficiency due to fewer idle epochs in user execution; on the other hand, doing so requires to support a much more complex POSIX environment on peripheral cores.

Offloading the execution of a commodity, monolithic kernel raises practical challenges, not only i) that the peripheral core has a different ISA and wimpy hardware but also ii) that the kernel is complex and rapidly evolving [68]. Many OS proposals address the former while being inadequate in addressing the latter [5, 61, 6, 48, 75]. For instance, one may refactor a monolithic kernel to span it over CPU and a peripheral core; the resultant kernel, however, depends on a wide binary interface (ABI) for synchronizing state between the two ISAs. This interface is brittle. As the upstream kernel evolves, maintaining *binary compatibility* across different ISAs inside the kernel itself soon becomes unsustainable. Instead, we argue for the code running on peripheral cores to enjoy firmware-level compatibility: developed and compiled *once*, it should work with *many* builds of the monolithic kernel – generated from different configurations and source versions.

Our response is a radical design called *transkernel*, a lightweight virtual executor empowering a peripheral core to run specific kernel phases – device suspend/resume. Figure 1 overviews the system architecture. A transkernel executes unmodified kernel binary through cross-ISA, dynamic binary translation (DBT), a technique previously regarded as expensive [5] and never tested on microcontroller-like cores to our knowledge. Underneath the translated code, a small set of emulated services act as lightweight, drop-in replacements for their counterparts in the monolithic kernel. Four principles make transkernel practical: i) translating stateful code while emulating stateless kernel services; ii) identifying a narrow, stable translation/emulation interface; iii) specializing for hot paths; iv) exploiting ISA similarities for DBT.

We demonstrate a transkernel prototype called ARK (**A**n a**R**m trans**K**ernel). Atop an ARM SoC, ARK runs on a Cortex-M3 peripheral core (with only 200 MHz clock and 32KB cache) alongside Linux running on a Cortex-A9 CPU. ARK transparently translates unmodified Linux kernel drivers and libraries. It depends on a binary interface consisting of only 12 Linux kernel functions and one kernel variable, which are stable for years. ARK offers complete support for device suspend/resume in Linux, capable of executing diverse drivers that implement rich functionalities (e.g., DMA and firmware loading) and invoke sophisticated kernel services (e.g., scheduling and IRQ handling). As compared to native kernel execution, ARK only incurs 2.7× overhead, 5.2× lower than a baseline of off-the-shelf DBT. ARK reduces system energy by 34%, resulting in tangible battery life extension under real-world usage.

We make the following contributions on OS and DBT:

• We present the transkernel model. In the design space of OSes for heterogeneous multi-processors, the transkernel represents a novel point: it combines DBT and emulation for bridging ISA gaps and for catering to core asymmetry, respectively.

• We present a transkernel implementation, ARK. Targeting Linux, ARK presents specific tradeoffs between kernel translation versus emulation; it identifies a narrow interface between the two; it contributes concrete realization for them.

• Crucial to the practicality of ARK, we present an inverse paradigm of cross-ISA DBT, in which a microcontroller-like core translates binary built for a full-fledged CPU. We contribute optimizations that systematically exploit ISA similarities. Our result demonstrates that while cross-ISA DBT is typically used under the assumption of efficiency *loss*, it can enable efficiency *gain*, even on off-the-shelf hardware.

The source code of ARK can be found at `http://xsel.rocks/p/transkernel`.

2 Motivations

We next discuss device suspend/resume, the major kernel bottleneck in ephemeral tasks, and that it can be mitigated by running on a peripheral core. We show difficulties in known approaches and accordingly motivate our design objectives.

2.1 Kernel in device suspend/resume

Expecting a long period of system inactivity, an OS kernel puts the whole platform into deep sleep: in brief, the kernel synchronizes file systems with storage, freezes all user tasks, turns off IO devices (i.e., device suspend), and finally powers off the CPU. To wake up from deep sleep, the kernel performs a mirrored procedure [11]. In a typical ephemeral task, the above kernel execution takes hundreds of milliseconds [31] while the user execution often takes tens of milliseconds [44]; the kernel execution often consumes several times more energy than the user execution [38].

Problem: device suspend/resume By profiling recent Linux on multiple embedded platforms, our pilot study [92] shows the aforementioned kernel execution is bottlenecked by device suspend/resume, in which the kernel cleans up pending IO tasks and manipulates device power states. The findings are as follows. i) ***Device suspend/resume is inefficient.*** It contributes 54% on average and up to 66% to the total kernel energy consumption. CPU idles frequently in numerous short epochs, typically in milliseconds. ii) ***Devices are diverse.*** On a platform, the kernel often suspends and resumes tens of different devices. Across platforms, the bottleneck devices are different. iii) ***Optimization is difficult.*** Device power state transitions are bound by slow hardware and low-speed buses, as well as physical factors (e.g., voltage ramp-up). While Linux already parallelizes power transitions with great efforts [50, 46], many power transitions must happen sequentially per *implicit* dependencies of power, voltage, and clock. As a result, CPU idle constitutes up to 68% of the device suspend/resume duration.

SoC	Cores	ISAs	Shared DRAM?	Mapping kern mem?	Shared IRQ
OMAP4460 [83] (2010)	A9+M3	v7a+v7m	Full	Yes. MPU	39/102
AM572x [81] (2014)	A15+M4	v7a+v7m	Full	Yes. MPU	32/92
i.MX6SX [62] (2015)	A9+M4	v7a+v7m	Full	Yes. MPU	85/87
i.MX7 [65] (2017)	A7+M4	v7a+v7m	Full	Yes. MPU	88/90
i.MX8M [63] (2018)	A53+M4	v8a+v7m	Full	Yes. MPU	88/88
MT3620 [52] (2018)*	A7+M4	v7a+v7m	Full	Likely. MPU	Likely most

Table 1: Our hardware model fits many popular SoCs which are used in popular products such as Apple Watch and Azure Sphere. Section 7.5 discusses caveats. *: lack public technical details.

Challenge: Widespread, complex kernel code Device suspend/resume invokes multiple kernel layers [68, 32]. Specifically, it invokes functions in individual drivers (e.g., MMC controllers), driver libraries (e.g., the generic clock framework), kernel libraries (e.g., for radix trees), and kernel services (e.g., scheduler). In a recent Linux source tree (4.4), we find that over 1000 device drivers, which represent almost all driver classes, implement suspend/resume callbacks in 154K SLoC. These callbacks in turn invoke over 43K SLoC in driver libraries, 8K SLoC in kernel libraries, and 43K SLoC in kernel services. The execution is control-heavy, with dense branches and callbacks.

Opportunities We observe the following kernel behaviors in device suspend/resume. i) ***Low sensitivity to execution delay*** On embedded platforms, most ephemeral tasks are driven by background activities [38, 53, 13]. This contrasts to many servers for interactive user requests [93, 53]. ii) ***Hot kernel paths*** In successful suspend/resume, the kernel acquires all needed resources and encounters no failures [41]. Off the hot paths, the kernel handles rare events such as races between IO events, resource shortage, and hardware failures. These branches typically cancel the current suspend/resume attempt, perform diagnostics, and retry later. Unlike hot paths, they invoke very different kernel services, e.g., syslog. iii) ***Simple concurrency*** exists among the syscall path (which initiates suspend/resume), interrupt handlers, and deferred kernel work. The concurrency is for hardware asynchrony and kernel modularity rather than exploiting multicore parallelism.

Summary: design implications Device suspend/resume shall be treated systematically. We face challenges that the invoked kernel code is diverse, complex, and cross-layer; we see opportunities that allow focusing on hot kernel paths, specializing for simple concurrency, and gaining efficiency at the cost of increased execution time.

2.2 A peripheral core in a heterogeneous SoC

Hardware model We set to exploit peripheral cores already on modern SoCs. Hence, our software design only assumes the following hardware model which fits a number of popular SoCs as listed in Table 1.

1. ***Asymmetric processors***: In different coherence domains, the CPU and the peripheral core offer disparate performance/-efficiency tradeoffs. The peripheral core has memory protection unit (MPU) but no MMU, incapable of running commodity OSes as-is.
2. ***Heterogeneous, yet similar ISAs***: The two processors have different ISAs, in which many instructions have similar *semantics*, as will be discussed below.
3. ***Loose coupling***: The two processors are located in separate power domains and can be turned on/off independently.
4. ***Shared platform resources***: Both processors share access to platform DRAM and IO devices. Specifically, the peripheral core, through its MPU, should map all the kernel code/-data at identical virtual addresses as the CPU does. Both processors must be able to receive interrupts from the devices of interest, e.g., MMC; they may, however, see different interrupt line numbers of the same device.

How can peripheral cores save energy? They are known to deliver high efficiency for IO-heavy workloads [42, 54, 78, 1, 76]. Specifically, they benefit the kernel's device suspend/resume in the following ways. i) A peripheral core can operate while leaving the CPU offline. ii) The idle power of a peripheral core is often one order of magnitude lower [43, 64], minimizing system power during core idle periods. iii) Its simple microarchitecture suits kernel execution, whose irregular behaviors often see marginal benefits from powerful microarchitectures [58]. Note that a peripheral core offers much higher efficiency than a LITTLE core as in ARM big.LITTLE [24], which mandates a homogeneous ISA and tight core coupling. We will examine big.LITTLE in Section 7.

ISA similarity On an SoC we target, the CPU and the peripheral core have ISAs from the same family, e.g., ARM. The two ISAs often implement similar instruction *semantics* despite in different *encoding*. The common examples are SoCs integrating ARMv7a ISA and ARMv7m ISA [62, 65, 81, 52, 83]. Other families also provide ISAs amenable to same-SoC integration, e.g., NanoMIPS and MIPS32. We deem that the ISA similarities are *by choice*. i) For ISA designers, it is feasible to explore performance-efficiency tradeoffs within one ISA family, since the family choice is merely about instruction *syntax* rather than *semantics* [8]. ii) For SoC vendors, incorporating same-family ISAs on one chip simplifies software efforts [40], silicon design, and ISA licensing.

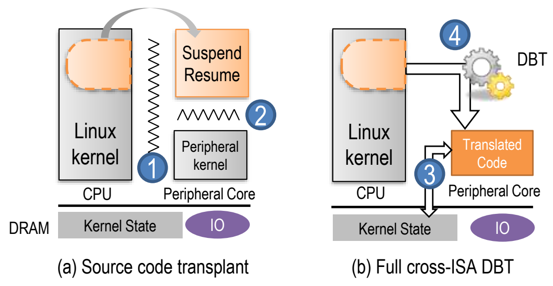
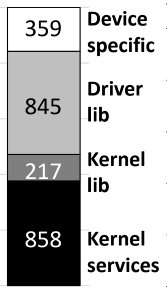
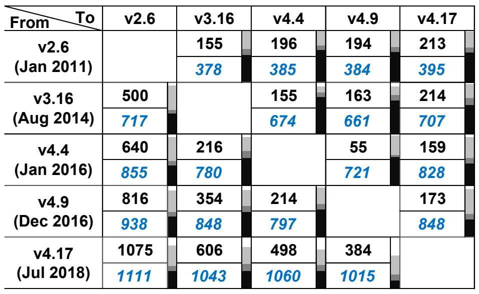


Figure 2: Alternative ways for offloading kernel phases





From \ To	v2.6	v3.16	v4.4	v4.9	v4.17
v2.6 (Jan 2011)		155 / *378*	196 / *385*	194 / *384*	213 / *395*
v3.16 (Aug 2014)	500 / *717*		155 / *674*	163 / *661*	214 / *707*
v4.4 (Jan 2016)	640 / *855*	216 / *780*		55 / *721*	159 / *828*
v4.9 (Dec 2016)	816 / *938*	354 / *848*	214 / *797*		173 / *848*
v4.17 (Jul 2018)	1075 / *1111*	606 / *1043*	498 / *1060*	384 / *1015*	

(a) # of functions (b) # of functions (upper) & types (lower) w/ changed ABI across kernel versions

Figure 3: Counts of Linux kernel functions referenced by device suspend/resume, showing (a) the functions are rich and diverse and (b) their ABI change is substantial over time. Exported functions only. Build config: omap2defconfig. ABI changes detected with ABI compliance checker [70]

2.3 OS design space exploration

We set to realize heterogeneous execution for an *existing* monolithic kernel.

How about refactoring the kernel and cross-compiling statically? One may be tempted to modify a monolithic kernel (we use Linux as the example below) [43, 5] to be one unified source tree; the tree shall be cross-compiled into a kernel binary for CPU and a "peripheral kernel" for the peripheral core. This approach results in an OS structure shown in Figure 2(a). Its key drawback is the two interfaces that are difficult to implement and maintain, shown as ∿∿∿ in the figure.

① The interface between two heterogeneous ISAs, as needed for resolving inter-kernel data dependency. Through the interface, both kernels synchronize their kernel state, e.g., devices configurations, pending IO tasks, and locks, before and after the offloading. Built atop shared memory [43, 5, 23], the interface is essentially an agreement on thousands of shared Linux kernel data types, including their semantics and/or memory layout. The agreement is brittle, as it is affected by ISA choices, kernel configurations, and kernel versions. Hence, keeping data types consistent across ISAs entails tedious tweak of kernel source and configurations [22, 23]. As Greg Kroah-Hartman puts, "you will go insane over time if you try to support this kind of release, I learned this the hard way a long time ago." [36]

② The interface between the transplant code and the peripheral kernel, as needed for resolving functional dependency. In principle, this interface is determined by the choice of transplant boundary. In prior work, the example choices include the interface of device-specific code [22, 23, 80], that of driver classes [10, 79], or that of driver libraries [43]. All these choices expose at least hundreds of Linux kernel functions on this interface, as summarized in Figure 3(a). This is due to Linux's diverse, sophisticated drivers. Implementing such an interface is daunting; maintaining it is even more difficult due to significant ABI changes [37] as shown in Figure 3(b).

In summary, all these difficulties root in the peripheral kernel's *deep dependency* on the Linux kernel. This is opposite to the common practice: heterogeneous cores to run their own "firmware" that has little dependency on the Linux kernel. This is sustainable because the firmware stays compatible with many builds of Linux.

How about virtual execution? Can we minimize the dependency? One radical idea would be for a peripheral core to run the Linux kernel through virtual execution, as shown in Figure 2(b). Powered by DBT, virtual execution allows a *host* processor (e.g., the peripheral core) to execute instructions in a foreign *guest* ISA (e.g., the CPU). Virtual execution is free of the above interface difficulties: the translated code precisely reproduces the kernel behaviors and directly operates the kernel state (③). The peripheral core interacts with Linux through a low-level, stable interface: the CPU's ISA (④).

The problem, however, is the high overhead of existing cross-ISA DBT [4]. It is further exacerbated by our *inverse* DBT paradigm: whereas existing cross-ISA DBT is engineered for a brawny host emulating a weaker guest (e.g., an x86 desktop emulating an ARM smartphone) [17, 87], our DBT host, a peripheral core, shall serve a full-fledged CPU. A port of popular DBT exhibits up to 25× slowdown as will be shown in §7. Such overhead would negate any efficiency promised by the hardware and result in overall efficiency *loss*. Furthermore, cross-ISA DBT for the *whole* Linux kernel is complex [7]. A peripheral core lacks necessary environment, e.g., multiple address spaces and POSIX, for developing and debugging such complex software.

2.4 Design objective

We therefore target threefold objective.

G1. Tractable engineering. We set to reuse much of the kernel source, in particular the drivers that are impractical to build anew. We target simple software for peripheral cores.

G2. Build once, work with many. One build of the peripheral core's software should work with a commodity kernel's binaries built from a wide range of configurations and source versions. This requires the former to interact with the latter through a stable, narrow ABI.

G3. Low overhead. The offloaded kernel phases should yield a tangible efficiency gain.

3 The Transkernel Model

Running on a peripheral core, a transkernel consists of two components: a DBT engine for translating and executing the unmodified kernel binary; a set of emulated, minimalistic kernel services that underpin the translated kernel code, as will be described in detail in Section 4. A concrete transkernel implementation targets a specific commodity kernel, e.g., Linux. A transkernel does not execute user code in ephemeral tasks as stated in Section 1.

The transkernel follows four principles:

1. Translating stateful code; emulating stateless services By *stateful code*, we refer to the offloaded code that must share states with the kernel execution on CPU. The stateful code includes device drivers, driver libraries, and a small set of kernel services. They cover the most diverse and widespread code in device suspend/resume (§2). By translating their binaries, the transkernel reuses the commodity kernel without maintaining wide, brittle ABIs. (objective G1, G2)

The transkernel emulates a tiny set of kernel services. We relax their semantics to be stateless, so that their states only live within one device suspend/resume phase. Being stateless, the emulated services do not need to synchronize states with the kernel on CPU over ABIs. (G2)

2. Identifying a narrow, stable translation/emulation ABI The ABI must be unaffected by kernel configurations and unchanged since long in the kernel evolution history. (G2)

3. Specializing for hot paths In the spirit of OS specialization [20, 71, 51], the transkernel only executes the hot path of device suspend/resume; in the rare events of executing off the hot path, it transparently falls back on CPU. The transkernel's emulated services seek *functional equivalence* and only implement features needed by the hot path; they do not precisely reproduce the kernel's behaviors. (G1)

4. Exploiting ISA similarities for DBT The transkernel departs from generic cross-DBT that bridges arbitrary guest/host pairs; it instead systematically exploits similarities in instructions semantics, register usage, and control flow transfer. This makes cross-ISA DBT affordable. (G3)

Limitations First, across ISAs of which instruction semantics are substantially different, e.g., ARM and x86, the transkernel may see diminishing or even no benefit. Second, the transkernel's longer delays (albeit lower energy) may misfit latency-sensitive contexts, e.g., for waking up platforms in response to user input. Our current prototype relies on heuristics to recognize such contexts and falls back on the CPU accordingly (Section 4).

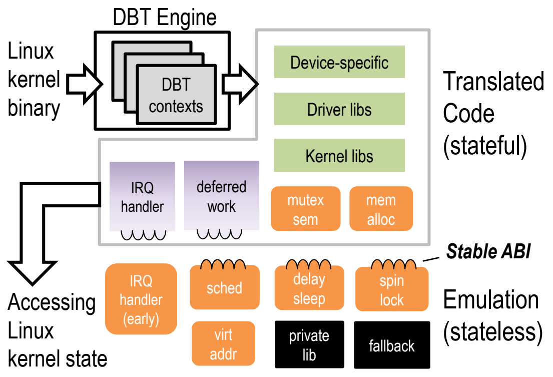


Figure 4: The ARK structure on a peripheral core

In Section 4 below we describe how to apply the model to a concrete transkernel, in particular our translation/emulation decisions for major kernel services, and our choices of the emulation interface. We will describe DBT in Section 5.

4 ARK: An ARM Transkernel

Targeting an ARM SoC, we implement a transkernel called ARK. The SoC encompasses a popular combination of ISAs: ARMv7A for its CPU and ARMv7m for its peripheral core. The CPU runs Linux v4.4.

Offloading workflow ARK is shipped as a standalone binary for the peripheral core, accompanied by a small Linux kernel module for control transfer between CPU and the peripheral core. We refer to such control transfer as *handoff*. Prior to a device suspend phase, the kernel shuts down all but one CPU cores, passes control to the peripheral core, and shuts down the last CPU core. Then, ARK completes the device phase in order to suspend the entire platform. Device resume is normally executed by ARK on the peripheral core; in case of urgent wakeup events (e.g., a user unlocking a smart watch screen), the kernel resumes on CPU with native execution.

System structure As shown in Figure 4, ARK runs a DBT engine, its emulated kernel services, and a small library for managing the peripheral core's private hardware, e.g., interrupt controllers. The emulated services serves downcalls ($\frown\!\frown\!\frown\!\_$) from the translated code and makes upcalls ($\_\smile\!\smile\!\smile$) into the translated code. Table 2 summarizes the interfaces. Upon booting, ARK replicates Linux kernel's linear memory mappings for addressing kernel objects in shared memory [43, 23]. ARK maps I/O regions with MPU and time-multiplexes the regions on the MPU entries.

To support concurrency in the offloaded kernel phases, ARK runs multiple DBT contexts. Each context has its own DBT state (e.g., virtual CPU registers and a stack), executing DBT and emulated services independently. Context switch is as cheap as updating the pointer to the DBT state.

ARK executes the hot paths. Upon entering cold branches pre-defined by us, e.g., kernel `WARN()`, ARK migrates all the

Kernel services	Implementations & reasons
Scheduler (§4.1)	Emulated. Reason: simple concurrency.
IRQ handler (§4.2)	Early stage emulated; then translated
HW IRQ controller (§4.2)	Emulated. Reason: core-specific
Deferred work (§4.3)	Translated. Reason: stateful
Spinlocks (§4.4)	Emulated. Reason: core-specific
Sleepable locks (§4.4)	Fast path translated. Reason: stateful
Slab/Buddy allocator (§4.5)	Fast path translated. Reason: stateful
Delay/wait/jiffies (§4.6)	Emulated. Reason: core-specific

jiffies	udelay()	msleep()	tasklet_schedule()	irq_thread()
ktime_get()	queue_work_on()	worker_thread()	run_local_timers()	
generic_handle_irq()	schedule()	async_schedule()*	do_softirq()*	

*=ABI unchanged since 2014 (v3.16); others unchanged since 2011 (v2.6).

Table 2: Top: Kernel services supported by ARK. Bottom: Linux kernel ABI (12 funcs+1 var) ARK depends on. ARK offers complete support for device suspend/resume in Linux.

DBT contexts of *translated* code back to the CPU and continues as *native* execution there (§6).

4.1 A Scheduler of DBT Contexts

ARK emulates a scheduler which shares no state, e.g., scheduling priorities or statistics, with the Linux scheduler on the CPU. Corresponding to the simple concurrency model of suspend/resume (§2), ARK eschews reproducing Linux's preemptive multithreading but instead maintains and switches among cooperative DBT contexts: one primary context for executing the syscall path of suspend/resume, one for executing IRQ handlers (§4.2), and multiple for deferred work (§4.3). Managing no more than tens of contexts, ARK uses simple, round-robin scheduling. It begins the execution in the syscall context; when the syscall context blocks (e.g., by calling `msleep()`), ARK switches to the next ready context to execute deferred functions until they finish or block. When an interrupt occurs, ARK switches to the IRQ context to execute the kernel interrupt handler (§4.2).

4.2 Interrupt and Exception Handling

During the offloaded device phase, all interrupts are routed to the peripheral core and handled by ARK.

Kernel interrupt handlers ARK emulates a short, early stage of interrupt handling while translating the kernel code for the remainder. This is because this early stage is ISA-specific (e.g., for manipulating the interrupt stack), on which the CPU (v7a) and the peripheral core (v7m) differ. Hence, the emulated services implement a v7m-specific routine and install it as the hardware interrupt handler. Once an interrupt happens, the routine is invoked to finish the v7m-specific task and make an upcall to the kernel's ISA-neutral interrupt handling routine (listed in Table 2), from where the ARK translates the kernel to finish handling the interrupt.

Hardware interrupt controller ARK emulates the CPU's hardware interrupt controller. This is needed as the two cores have separate, heterogeneous interrupt controllers. The CPU controller's registers are unmapped in the peripheral core; upon accessing them (e.g., for masking interrupt sources) the translated code triggers faults. ARK handles the faults and operates the peripheral core's controller accordingly.

Exception: unsupported We don't expect any exception in the offloaded kernel phases. In case exception happens, ARK uses its fallback mechanism (§6) to migrate back to CPU.

4.3 Deferred Work

Device drivers frequently schedule functions to be executed in the future. ARK translates the Linux services that schedule the deferred work as well as the actual execution of the deferred work. ARK chooses to translate such services because they must be *stateful*: the peripheral core may need to execute deferred work created on the CPU prior to the offloading, e.g., freeing pending WiFi packets; it may defer new work until after the completion of resume.

ARK maintains dedicated DBT contexts for executing the deferred work (Section 4.1). While the Linux kernel often executes deferred work in kernel threads (daemons), our insight is that deferred work is oblivious to its execution context (e.g., a real Linux thread or a DBT context in ARK). Beyond this, ARK only has to run the deferred work that may *sleep* with separate DBT contexts so that they do not block other deferred work. From these DBT contexts, ARK translates the main functions of the aforementioned kernel daemons, which retrieve and invoke the deferred work.

Threaded IRQ defers heavy-lifting IRQ work (i.e., deferred work) to a kernel thread which executes the work after the hardware IRQ is handled. A threaded IRQ handler may sleep. Therefore, ARK maintains per-IRQ DBT contexts for executing these handlers. Each context makes upcalls into `irq_thread()` (the main function of threaded irq daemon, listed in Table 2).

Tasklets, workitems, and timer callbacks The kernel code may dynamically submit short, non-sleepable functions (tasklets) or long, sleepable functions (workitems) for deferred execution. Kernel daemons (softirq and kworker) execute tasklets and workitems, respectively.

ARK creates one dedicated context for executing all non-sleepable tasklets and per-workqeueue contexts for executing workitems so that one workqueue will not block others. These contexts make upcalls to the main functions of the kernel daemons (`do_softirq()`, `worker_thread()`, and `run_local_timers()`), translating them for retrieving and executing deferred work.

4.4 Locking

Spinlocks ARK emulates spinlocks, because their implementation is core-specific and that ARK can safely assume all spinlocks are free at handoff points: as described in early

Section 4, handoff happens between one CPU core and one peripheral core, which do not hold any spinlock; all other CPU cores are offline and cannot hold spinlocks. Hence, ARK emulates spinlock acquire/release by pausing/resuming interrupt handling. This is because ARK runs on one peripheral core and the only hardware concurrency comes from interrupts.

Sleepable locks ARK translates sleepable locks (e.g., mutex, semaphore) because these locks are stateful: for example, the kernel's clock framework may hold a mutex preventing suspend/resume from concurrently changing clock configuration [56]. Furthermore, mutex's seemingly simple interface (i.e., compare & exchange in fast path) has *unstable* ABI and therefore unsuitable for emulation: a mutex's reference count type changes from `int` to `long` (v4.10), breaking the ABI compatibility. The translated operations on sleepable locks may invoke spinlocks or the scheduler, e.g., when updating reference counts or putting the caller to sleep, for which the translated execution makes downcalls to the emulated services.In practice, no sleepable lock is held prior to system suspend.

4.5 Memory Allocation

The device phase frequently requests dynamic memory, often at granularities of tens to hundreds of bytes. By Linux design, such requests are served by the kernel slab allocator backed by a buddy system for page allocation (fast path); when the physical pages runs low, the kernel may trigger swapping or kill user processes (slow path).

ARK provides memory allocation as a stateful service. It translates the kernel code for the fast path, including the slab allocator and the buddy system. In the case that the allocation enters the slow path (e.g., due to low physical memory), ARK aborts offloading; fortunately, our stress test suggests such cases to be extremely rare, as will be reported in Section 7. With a stateful allocator, the offloaded execution can free dynamic memory allocated during the kernel execution on CPU, and vice versa. Compare to prior work that instantiates per-kernel allocators with split physical memory [43], ARK reduces memory fragmentation and avoids tracking *which* processor should free *what* dynamic memory pieces. Our experience in Section 7 show that ARK is able to handle intensive memory allocation/free requests such as in loading firmware to a WiFi NIC.

4.6 Delays & Timekeeping

Delays ARK emulates `udelay()` and `msleep()` for busy waiting and sleeping. ARK converts the expected wait time to the hardware timer cycles on the peripheral core. ARK implements `msleep()` by pausing scheduling the caller context.

jiffies The Linux kernel periodically updates jiffies, a global integer, as a low-overhead measure of elapsed time. By consulting the peripheral core's hardware timer, ARK directly updates the jiffies. It is thus the only shared variable on the kernel ABI that ARK depends (all others are functions).

5 The Cross-ISA DBT Engine

A Cross-ISA DBT Primer DBT, among its other uses [60, 49, 27], is a known technique allowing a *host* processor to execute instructions in a foreign *guest* ISA. In such cross-ISA DBT, the host processor runs a program called DBT engine. At run time, the engine reads in guest instructions, translates them to host instructions based on the engine's built-in *translation rules*, and executes these host instructions. The engine translates guest instructions in the unit of translation block – a sequence (typically tens) of guest instructions that has one entry and one or more exits. After translating a block, the engine saves the resultant host instructions to its *code cache* in the host memory, so that future execution of this translated block can be directed to the code cache.

Design overview We build ARK atop QEMU [7], a popular, opensource cross-ISA DBT engine. ARK inherits QEMU's infrastructure but departs from its generic design which translates between arbitrary ISAs. ARK targets two well-known DBT optimizations: i) to emit as few host instructions as possible; ii) to exit from the code cache to the DBT engine as rarely as possible. We exploit the following similarities between the CPU's and the peripheral core's ISAs (ARMv7a & ARMv7m):

1. Most v7a instructions have v7m counterparts with identical or similar semantics, albeit in different encoding. (§5.1)
2. Both ISAs have the same general purpose registers. The condition flags in both ISAs have same semantics. (§5.2)
3. Both ISAs use program counter (PC), link register (LR), and stack pointer (SP) in the same way. (§5.3)

Beyond the similarities, the two ISAs have important discrepancies. Below, we describe our exploitation of the ISA similarities and our treatment for *caveats*.

5.1 Exploiting Similar Instruction Semantics

	Category	**Cnt**	**v7m**
w/ CNTPRT	Identity	447	1
	Side effect	52	3-5
	Const constraints	22	2-5
	Shift modes	10	2
w/o counterparts		27	2-5
	Total (v7a)	558	

Table 3: Translation rules for v7a instructions. Column 3: the number of v7m instructions emitted for one v7a instruction

We devise translation rules with a principled approach by parsing a machine-readable, formal ISA specification recently published by ARM [72]. Our overall guideline is to map each v7a instruction to one v7m instruction that has identical or similar semantics. We call them *counterpart* instructions. For a counterpart instruction with similar (yet

ARMv7a	ARMv7m (by ARK)
G1: `ldr r0, [r1], r2, lsr #4`	**H1:** `ldr.w r0, [r1]` **H2:** `lsr.w t0, r2, 0x4` **H3:** `add.w r1, r1, t0`
G2: `adds r0, r1, 0x80000001`	**H4:** `mov.w t0, 0xc0` **H5:** `ror.w t0, t0, 0x7` **H6:** `adds.w r0, r1, t0`
G3: `sub r0, r1, r2`	**H7:** `sub.w r0, r1, r2`

Table 4: Sample translation by ARK. By contrast, our baseline QEMU port translates G1–G3 to **27** v7m instructions

non-identical) semantics, ARK emits a few "amendment" v7m instructions to make up for the semantic gap. The resultant translation rules are based on individual guest instructions, different from translation rules based on one or more translation blocks commonly seen in cross-ISA DBT [86]. This is because semantics similarities allows identity translation for most guest instructions. Amendment instructions are oblivious to interrupts/exceptions: as stated in §4.2, ARK defers IRQ handling to translation block boundary and expects no exceptions.

Table 3 summarizes ARK's translation rules for all 558 v7a instructions. Among them, 80% can be translated with identity rules, for which ARK only needs to convert instruction encoding at run time. 15% of v7a instructions have v7m counterparts but may require amendment instructions, which fortunately fall into a few categories: i) ***Side effects.*** After load/store, v7a instructions may additionally update memory content or register values (shown in Table 4, G1). ARK emits amendment instructions to emulate the extra side effect (H3). ii) ***Constraints on constants.*** The range of constants that can be encoded in a v7m instruction is often narrower (Table 4, G2). In such cases, the amendment instructions load the constant to a scratch register, operate it, and emulate any side effects (e.g., index update) the guest instruction may have. iii) ***Richer shift modes.*** v7a instructions support richer shift modes and larger shift ranges than their v7m counterparts. This is exemplified by Table 4 G1, where a v7m instruction cannot perform LSR (logic shift right) inline as its v7a counterpart. Similar to above, the amendment instructions perform shift on the operand in a scratch register.

Beyond the above, only 27 v7a instructions have no v7m counterparts, for which we manually devise translation rules.

In summary, through systematic exploitation of similar instruction semantics, ARK emits compact host code at run time. In the example shown in Table 4, three v7a instructions are translated into seven v7m instructions by ARK, while to 27 instructions by our QEMU baseline.

5.2 Passthrough of CPU registers

General purpose registers Both the guest (v7a) and the host (v7m) have the same set (13) of general-purpose registers. In allocating registers of a host instruction, ARK follows guest register allocation with best efforts (e.g., one-to-one mapping in best case, as in Table 4, G1). ARK emits much fewer host instructions than QEMU, which emulates all guest registers in host memory with load /store.

Caveats fixed The amendment host instructions operate scratch registers as exemplified by `t0` in Table 4, H2-H6. However, the wimpy host faces higher register pressure, as it (v7m) has no more registers than the brawny guest (v7a). To spill some registers to memory while still reusing the guest's register allocation, we make the following tradeoff: we designate one host register as the *dedicated* scratch register, and emulates its guest counterpart register in memory. We pick the *least* used one in the guest binary as the dedicated scratch register, which is experimentally determined as R10 by analyzing kernel binary. We find most amendment instructions are satisfied by *one* scratch register; in rare cases when extra scratch registers are needed, ARK follows a common design to allocate dead registers and spill unused ones to memory.

Condition flags Both the guest and the host ISAs involve five hardware condition flags (e.g., zero and carry) with identical semantics; fortunately, most guest (v7a) instructions and their host (v7m) counterparts have identical behaviors in testing/setting flags per the ISA specifications [72]. ARK hence directly emits instructions to manipulate the host's corresponding flags. Such flag passthrough especially benefits control-heavy suspend/resume, which contains extensive conditional branches (§2); we study its benefits quantitatively in §7.3.

Caveats fixed Amendment host instructions may affect the hardware condition flags unexpectedly. For amendment instructions (notably comparison and testing) that *must* update the flags as mandated by ISA, ARK emits two host instructions to save/restore the flags in a scratch register around the execution of these amendment instructions.

5.3 Control Transfer and Stack Manipulation

Function call/return Both guest (v7a) and host (v7m) use PC (program counter) and LR (link register) to maintain the control flow. QEMU emulates guest PC and LR in host memory. As a result, the return address, loaded from stack or the emulated LR, points to a guest address (i.e., kernel address). Each function return hence causes the DBT to step in and look up the corresponding code cache address. This overhead is magnified in the control-heavy device phase.

By contrast, ARK never emits host code to emulate the guest (i.e., kernel) PC or LR. For each kernel function call, ARK saves the return addresses within *code cache* on stack or in LR; for each kernel function return, ARK loads the return address (which points to code cache) to hardware PC from the stack or the hardware LR. By doing so, ARK no longer participates in all function returns. Our optimization is inspired by same-ISA DBT [34].

Stack and SP QEMU emulates the guest (i.e., kernel) stack and SP with a host array and a variable. Each guest push/pop translates to multiple host instructions updating the stack array and the emulated SP. This is costly, as suspend/resume frequently makes function calls and operates stack heavily.

ARK avoids such expensive stack emulation by emitting host push/pop instructions to directly operate the guest stack *in place*. This is possible because ARK emulates the Linux kernel's virtual address space (§4). ARK also ensures the host code generate the same stack frames as the guest would do by making amendment instructions avoid using stack, which would introduce extra stack contents. In addition, this further facilitates the migration in abort (§6).

Caveats fixed i) As the host saves on the guest stack the code cache addresses, which are meaningless to the guest CPU, upon migrating from the peripheral core (host) to the CPU (guest), the DBT rewrites all code cache addresses on stack with their corresponding guest addresses. ii) guest push/pop instruction may involve emulated registers (i.e., scratch register). ARK must emit multiple host instructions to correctly synchronize the emulated registers in memory.

6 Translated $\longrightarrow$ Native Fallback

As described in Section 3, when going off the hot paths, ARK migrates the kernel phase back to the CPU and continues as native execution, analogous to virtual-to-physical migration of VMs [85]. Migrating one DBT context is natural, as ARK passes through most CPU registers and uses the kernel stack in place (§5.3). Yet, to migrate *all* active DBT contexts, ARK address the following unique challenges.

Migrate DBT contexts for deferred work After fallback, all blocked workitems should continue their execution on the CPU. Unfortunately, their enclosing DBT contexts do not have counterparts in the Linux kernel. To solve this issue, we again exploit the insight that the workitems are oblivious to their execution contexts. Upon migration, the Linux kernel creates temporary kernel threads as "receivers" for blocked workitems to execute in. Once the migrated workitems complete, the receiver threads terminate.

Migrate DBT context for interrupt If fallback happens inside an ISA-neutral interrupt handler (translated), the remainder of the handler should migrate to the CPU. This challenge, again, is that ARK's interrupt context has no counterpart on the CPU: the interrupt never occurs to the CPU. ARK addresses this by *rethrowing* the interrupt as an IPI (interprocessor interrupt) from the peripheral core to the CPU; the Linux kernel uses the IPI context as the receiver for the migrated interrupt handler to finish execution.

Section 7 will evaluate the fallback frequency and cost.

7 Evaluation

We seek to answer the following questions:

1. Does ARK incur tractable engineering efforts? (§7.2)
2. Is ARK correct and low-overhead? (§7.3)
3. Does ARK yield energy efficiency benefit? What are the major factors impacting the benefit? (§7.4)

7.1 Methodology

Test Platform We evaluate ARK on OMAP4460, an ARM-based SoC [83] as summarized in Table 6. We chose this SoC mainly for its good documentation and longtime kernel support (since 2.6.11), which allows our study of kernel ABI over a long timespan in Section 2. As Cortex-M3 on the platform is incapable of DVFS, for fair comparison, we run both cores at their highest clock rates. Note that OMAP4460 is not completely aligned with our hardware model, for which we apply workarounds as will be discussed in Section 7.5.

Benchmark setup We benchmark ARK on the whole suspend/resume kernel phases. We run a user program as the test harness that periodically kicks ARK for suspend/resume; the generated kernel workloads are the same as in all ephemeral tasks. Our benchmark is macro: it exercise extensive drivers and services, during which ARK translates and executes over 200 million instructions.

The benchmark operates nine devices for suspend/resume. 1. **SD card**: SanDisk Ultra 16GB SDHC1 Class 10 card; 2. **Flash drive**: a generic drive connected via USB; 3. **MMC controller**: on-chip OMAP HSMMC host controller; 4. **USB controller**: on-chip OMAP HS multiport USB host controller; 5. **Regulator**: TWL6030 power management IC connected via I2C; 6. **Keyboard**: Dell KB212-B keyboard connected via USB; 7. **Camera**: Logitech c270 connected via USB; 8. **Bluetooth NIC**: an adapter with Broadcom BCM20702 chipset connected via USB; 9. **WiFi NIC**: TI WL1251 module. The kernel invokes sophisticated drivers, thoroughly exercising various services including deferred work (2–4,6–8), slab/buddy allocator (1–4,6–9), softirq (9), DMA (2,6–9), threaded IRQ (1,5,9), and firmware upload (9).

We measure device suspend/resume executed by ARK on Cortex-M3 and report the measured results. We compare ARK to native Linux execution on Cortex-A9. We further compare to a baseline ARK version: its DBT is a straightforward v7m port of QEMU that misses optimizations described in Section 5. We report measurements taken with warm DBT code cache, as this reflects the real-world scenario where device suspend/resume is frequently exercised.

7.2 Analysis of engineering efforts

ARK eliminates source refactoring of the Linux kernel (§2.3). As shown in Table 5, ARK transparently reuses substantial kernel code (15K SLoC in our test), most of which are drivers

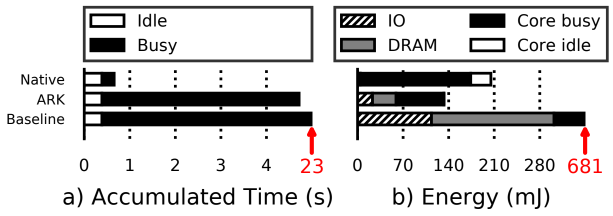


Figure 5: Execution time and energy in device suspend/resume. ARK substantially reduces the energy.

and their libraries. We stress that ARK, as a driver-agnostic effort, not only enables reuse of the drivers under test but also other drivers in the ARMv7 Linux kernel.

Table 5 also shows that ARK requires modest efforts in developing new software for the peripheral core. The 9K new SLoC for DBT is low as compared to commodity DBT (e.g., QEMU has 2M SLoC). ARK implements emulation in as low as 1K SLoC and in return avoids translating generic, sophisticated Linux kernel services [34, 21]. The result validates our principle of specializing these emulated services.

Existing code (unchanged)	
Translated	15K SLoC
Substituted w/ emu	25K SLoC
New implementation	
DBT	9K SLoC
Emulation	1K SLoC

Table 5: Source code

ARK meets our goal of "build once, run with many". We verify that the ARK binary works with a variety of kernel configuration variants (including `defconfig-omap4` and `yes-to-all`) of Linux 4.4. We also verify that ARK works with a wide range of Linux versions, from version 3.16 (2014) to 4.20 (most recent at the time of writing). This is because ARK only depends on a narrow ABI shown in Table 2, which has not changed since Linux 3.16.

7.3 Measured execution characteristics

ARK's correctness Formally, we derive translation rules from the specification of ARM ISA [72]; experimentally, we validate ARK by comparing its execution results side-by-side with native execution and examining the translated code with the native kernel binary. Over 200 million executed instructions, we verify that ARK's translation preserves kernel's semantics and presents consistent execution results.

Core activity We trace core states during ARK execution. Figure 5 (a) shows the breakdown of execution time. Compared to the native execution on CPU, ARK shows the same amount of accumulated idle time but much longer (16×) busy time. The reasons are Cortex-M3's much lower clock rate (1/6 of the A9's clock rate) and ARK's execution overhead. Despite the extended busy time, ARK still yields energy benefit, as we will show below.

Memory activity We collect DRAM activities by sampling the hardware counters of the SoC's DDR controller. We observed that ARK on Cortex-M3 generates much higher average DRAM utilization (32 MB/s read and 2MB/s write) than the native execution on A9 (only 8MB/s read and 4MB/s write). We attribute such thrashing to M3's small (32KB) last-level cache (LLC). Throughout the test, the ARK emitted and executed around 230KB host instructions, which far exceeds the LLC capacity and likely causes thrashing. By contrast, Cortex-A9 has a much larger LLC (1MB), which absorbs most of the kernel memory access. The memory activity has a strong energy impact, as will be shown below.

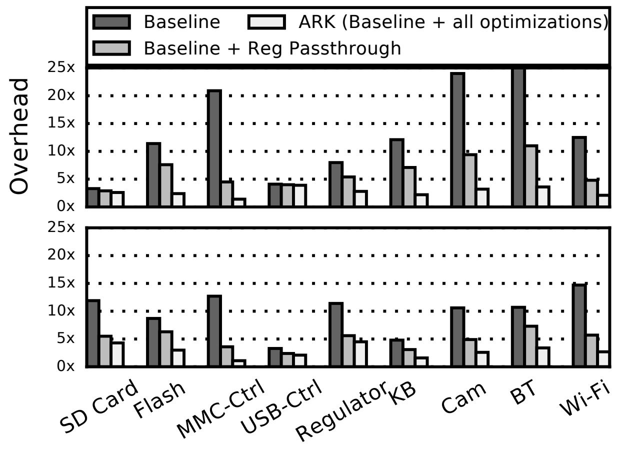


Figure 6: Busy execution overhead for devices under test (top: suspend; bottom: resume). Our DBT optimizations reduce the overhead by up to one order of magnitude

Busy execution overhead Our measurement shows that ARK incurs low overhead in busy kernel execution, which includes both DBT and emulation. We report the overhead as the ratio between ARK's cycle count on Cortex-M3 to the Linux's cycle count on A9. Note that an M3 cycle is 6× longer than A9 due to different clock rates.

Overall, the execution overhead is 2.7× on average (suspend: 2.9×; resume: 2.6×). Of individual drivers, the execution overhead ranges from 1.1× to 4.5× as shown in Figure 6. Our DBT optimizations (§5) have strong impact on lowering the overhead. Lacking them, our baseline design incurs a 13.9× overhead on average, 5.2× higher than ARK. We examined how our optimizations contribute to the gap: register passthrough (§5.2) reduces the baseline's overhead by 2.5× to 5.5×. Remaining optimizations (e.g., control transfer) collectively reduce the overhead by additional 2×. Our optimizations are less effective on drivers with very dense control transfer (e.g., USB) due to high DBT cost.

Emulated services Our profiling shows that ARK's emulated services incur low overhead. Overall, the emulated services only contribute 1% of total busy execution. i) The early, core-specific interrupt handling (§4.2) takes 3.9K Cortex-M3 cycles, only 1.5–2× more cycles than the native execution on

	CPU	Peripheral core
Core	Cortex A9@1.2GHz	Cortex M3@200MHz
Cache	L1:64KB + L2:1MB	L1:32KB
Typical busy/idle power	630mW/80mW	17mW/1mW

Table 6: The test platform - OMAP4460 on a Pandaboard

A9. ii) Emulated workqueues (§4.3) incurs a typical queueing delay of tens of thousands M3 cycles. The delay is longer than the native execution but does not break the deferred execution semantics.

Fallback frequency & cost We stress test ARK by repeating the benchmark for 1000 runs. Throughout the 1000 runs, ARK encounters only four cases when the execution goes off the hot path, all of which caused by the WiFi hardware failing to respond to resume commands; it is likely due to an existing glitch in WiFi firmware. In such a case, ARK migrates execution by spending around 20 us on rewriting code cache addresses on stack (§5.3), 17 us to flush Cortex-M3's cache, and 2 us to wake up the CPU through an IPI.

7.4 Energy benefits

Methodology We study system-level energy and in particular how it is affected by ARK's its extended execution time. We include energy of both cores, DRAM, and IO.

We measure power of cores by sampling the inductors on the power rails for the CPU and the peripheral core. As the board lacks measurement points for DRAM power [19], we model DRAM power as a function of DRAM power state and read/write activities, with Micron's official power model for LPDDR2 [55]. The system energy of ARK is given by:

$$E_{ARK} = \underbrace{E_{core}}_{\text{Measured}} + T_{idle} \cdot \underbrace{(P_{mem\_sr} + P_{io})}_{\textit{Modeled}} + T_{busy} \cdot \underbrace{(P_{mem} + P_{io})}_{\textit{Modeled}}$$

Here, E_{core} is the measured core energy. All Ts are measured execution time. Ps are power consumptions for DRAM and IO: P_{mem} is DRAM's active power derived from measured DRAM activities as described in Section 7.3; P_{mem_sr} is DRAM's self-refresh power, 1.3mW according to the Micron model; P_{io} is the average IO power which we estimate as 5mW based on prior work [90]. Note that during suspend/resume, IO devices no longer actively perform work, thus consuming much less power.

Energy saving ARK consumes 66% energy (a reduction of 34%) of the native execution, despite its longer execution time. The energy breakdown in Figure 5(b) shows the benefit comes from two portions: i) in busy execution, ARK's energy efficiency is 23% higher than the native execution due to low overhead (on average 2.7×); ii) during system idle, ARK reduces system energy to a negligible portion, as the peripheral core's idle power is only 1.25% of the CPU's. Figure 5(b) highlights the significance of our DBT optimizations: the baseline, like ARK, benefits from lower idle power as well; however its high execution overhead ultimately leads to 5.1× energy compared to the native execution. Interestingly, ARK

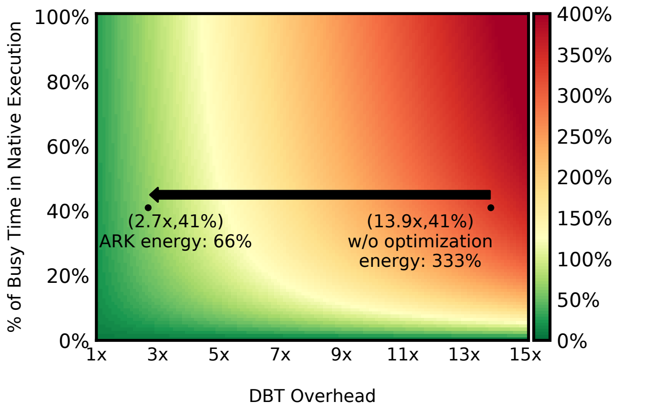


Figure 7: System energy consumption (inc. cores, DRAM, and IO) of ARK relative to native execution (100%), under different DBT overheads (x-axis) and processor core usage (y-axis). ARK's low energy hinges on low DBT overhead.

consumes *more* DRAM energy than the native execution. We deem the cause as Cortex-M3's tiny LLC (32KB) as describe earlier. Our result suggests that the current size is suboptimal for the offloaded kernel execution.

What-if analysis How sensitive is ARK's energy saving to two major factors: the DBT overhead (ARK's behavior) and the processor core usage (Linux's behavior)? To answer the question, we estimate the *what-if* energy consumption by using the power model as described above. The analysis results in Figure 7 show two findings. i) ARK's energy benefit will be more pronounced with lower core usage (i.e., longer core idle), because ARK's efficiency advantage over native execution is higher during core idle. ii) ARK's energy benefit critically depends on DBT. When the DBT overhead (on x-axis) drops to below 3.5×, ARK saves energy even for 100% busy execution; when the overhead exceeds 5.2×, ARK wastes energy even for 20% busy execution, the lowest core usage observed on embedded platforms in prior work [92].

Qualitative comparison with big.LITTLE We estimate ARK saves tangible energy compared to a LITTLE core. We use parameters based on recent big.LITTLE characterizations [66, 25]: compared to the big (i.e., CPU on our platform), a LITTLE core has 40 mW idle power [69] and offers 1.3× energy efficiency at 70% clock rate [47]. We favorably assume LITTLE's DRAM utilization is as low as the big, while in reality the utilization should be higher due to LITTLE's smaller LLC. Even with this favorable assumption for LITTLE and unfavorable, tiny LLC for ARK, LITTLE consumes 77% energy of native execution, more than ARK (51%–66%), mainly because LITTLE's idle power is 40× of Cortex-M3. Furthermore, ARK's advantage will be even more pronounced with a proper LLC as discussed earlier.

Battery life extension Based on ARK's energy reduction in device suspend/resume, we project the battery life extension for ephemeral tasks reported in prior work [38]. When the ephemeral tasks are executed at 5-second intervals and the native device suspend/resume consumes 90% system energy in a wakeup cycle, ARK extends the battery life by 18% (4.3 hours per day); with 30-second task intervals and a 50% energy consumption percentage, ARK extends the battery life by 7% (1.6 hours per day). This extension is tangible compared to complementary approaches in prior work [38, 90].

7.5 Discussions

Workarounds for OMAP4460 While OMAP4460 mostly matches our hardware model as summarized in Table 1, for minor mismatch we apply the following workarounds. ***Memory mapping*** Our hardware model (§2.2) mandates that the peripheral core should address the entire kernel memory. Yet, Cortex-M3, according to ARM's hardware specification [82], is only able to address memory in certain range (up to `0xE0000000`), which unfortunately does not encompass the Linux kernel's default address range. As a workaround, we configure the Linux kernel source, shifting its address range to be addressable by Cortex-M3. ***Interrupt handling*** While our hardware model mandates that both processors should receive all interrupts, OMAP4460 only routes a subset of them (39/102) to Cortex-M3, leaving out IO devices such as certain GPIO pins. These IO devices hence are unsupported by the ARK prototype and are not tested in our evaluation.

Recommendation to SoC architects To make SoCs friendly to a transkernel, architects may consider: i) routing all interrupts to CPU and the peripheral core, ideally with the identical interrupt line numbers; ii) making the peripheral core capable of addressing the whole memory address space; iii) enlarging the peripheral core's LLC size modestly. We expect a careful increase (e.g., to 64 KB or 128 KB) will significantly reduce DRAM power at a moderate overhead in the core power.

Applicability to a pair of 64-bit/32-bit ISAs While today's smart devices often use ARMv7 CPUs, emerging ARM SoCs start to combine 64-bit CPUs (ARMv8) with 32-bit peripheral core (ARMv7m), as listed in Table 1. On one hand, transkernel's idea of exploiting ISA similarity still applies, as exemplified by G2→H2 in Table 7; on the other hand, its DBT overhead may increase significantly for the following reasons. Compared to the 32-bit ISA, the 64-bit ISA has richer instruction semantics, more general purpose registers, and a much larger address space. As a result, ARK cannot pass through 64-bit CPU registers but instead have to emulate them in memory; ARK must translate the guest's 64-bit memory addresses to 32-bit addresses supported by the host (Table 7 G1→H1), e.g., by keeping consistent two sets of page tables, for 64-bit and 32-bit virtual address spaces, respectively; with large physical memory (>4GB), even this technique will not work because the peripheral core's page tables are incapable of mapping the extra physical memory.

ARMv8	ARMv7m (by ARK, ideally)

G1:
`ldrb w2, [x22, #1059]`
`ldrb w1, [x0, #160]`
	H1:
	`(emulate x22+#1059 in addr1)`
	`ldrb r2, [addr1]`
	`(emulate x0+#160 in addr2)`
	`ldrb r1, [addr2]`

G2: `cmp w2, w1`	**H2:** `cmp r2, r1`
G3: `beq mmc_select_bus_width+0x160`	**H3:** `beq mmc_select_bus_width+0x160`

Table 7: Ideal AARCH64 translation by ARK for `mmc_compare_ext_csds()` in Linux v4.4. While identity mapping still exists (G2→H2), software emulation can diminish ARK's benefits (G1→H1).

8 Related Work

OS for heterogeneous cores A multikernel OS runs its kernels on individual processors. A number of such OSes are designed anew with a strong distributed system flavor. They define explicit message interfaces among kernels [6, 88, 2]; some additionally exploit managed languages/compilers to generate such interfaces [61]. Unlike them, transkernel targets spanning an *existing* monolithic kernel and therefore adopts DBT to address the resultant interface challenge.

OSes like Popcorn [5] and K2 [43] seek to present a single Linux image over heterogeneous processors. For sharing kernel state across ISAs, they rely on manual source tweaks or hand-crafted communication. They face the interface difficulty as described in §2.3.

Prior systems distribute OS functions over CPU and accelerators [57, 77]. The accelerators cannot operate autonomously, which is however required by device suspend/resume. Prior systems offload apps from a smartphone (weak) to cloud servers (strong) for efficiency [15, 14]. Unlike them, transkernel offloads kernel workloads from a strong processor to a weak peripheral core on the same chip.

DBT DBT has been used for system emulation [7] and binary instrumentation [34, 27, 49, 21]; DeVuyst et al. [18] uses DBT to speed up process migration. Related to transkernel, prior systems run translated user programs atop an emulated syscall interface [7, 29, 87]. Unlike them, transkernel translates kernel code and emulates a narrow interface *inside* the kernel. Prior systems use DBT to run binaries in commodity ISAs (e.g., x86) on specialized VLIW cores and hence gain efficiency [9, 35, 73, 74]. None runs on microcontrollers to our knowledge. transkernel demonstrates that DBT can gain efficiency even on off-the-shelf cores. Existing DBT engines leverage ISA similarities, e.g., between aarch32 and aarch64 [17, 16]. They still fall into the classic DBT paradigm, where the host ISA is brawny and the guest ISA is wimpy (i.e., lower register pressure). With an inverse DBT paradigm, ARK addresses very different challenges. Much work is done on optimizing DBT translation rules, using optimizers [28, 3] or machine learning[86]. Compared to them, ARK leverages ISA similarities and hence reuses code optimization already in guest code by guest compilers.

Kernels and drivers The transkernel is inspired by the POSIX emulator [30] however is different as it emulates kernel ABIs. Prior kernel studies show rapid evolution of the Linux kernel and the interfaces between kernel layers are unstable [68, 67]. This observation motivates transkernel. Extensive work transplants device drivers to a separate core [23], user space [22], or a separate VM [39]. However, the transplant code cannot operate independent of the kernel, whereas transkernel must execute autonomously.

Encapsulating the NetBSD kernel subsystems (e.g., drivers) behind stable interfaces respected by developers, rump kernel [33] seeks to enable their reuse in foreign environments, e.g., hypervisors. The transkernel targets a different goal: spanning a live Linux kernel instance over heterogeneous processors. Applying Rump kernel's approach to Linux is difficult, as Linux intentionally rejects interface stability for drivers [36].

Suspend/resume's inefficiency raises attention for cloud servers [53, 89] and mobile [38]. Drowsy [38] mitigates inefficiency by reducing the devices involved in suspend/resume through user/kernel co-design; Xi *et al.* propose to reorder devices to resume [89]. While acknowledging the value of such kernel optimizations, we believe ARK is a key complement that works on unmodified binaries. ARK can co-exist with the mentioned optimizations in the same kernel. PowerNap [53] takes a hardware approach to speed up suspend/resume for servers. It does not treat kernel execution for operating diverse IO on embedded platforms. Kernels may put idle devices to low power at runtime [90], complementary to suspend/resume that ensures all devices are off.

9 Conclusions

We present transkernel, a new executor model for a peripheral core to execute a commodity kernel's phases, notably device suspend/resume. The transkernel executes the kernel binary through cross-ISA DBT. It translates stateful code while emulating stateless services; it picks a stable ABI for emulation; it specializes for hot paths; it exploits ISA similarities for DBT. We experimentally demonstrate that the approach is feasible and beneficial. The transkernel represents a new OS design point for harnessing heterogeneous SoCs.

Acknowledgments

The authors were supported in part by NSF Award #1619075, #1718702, and a Google Faculty Award. The authors thank the paper shepherd, Prof. Timothy Roscoe, and the anonymous reviewers for their insightful feedback. The authors thank Prof. Lin Zhong for providing a JTAG debugger. The authors are grateful to numerous video game emulators that inspired this project.

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), 2009.

[2] N. Asmussen, M. Völp, B. Nöthen, H. Härtig, and G. P. Fettweis. M3: A Hardware/Operating-System Co-Design to Tame Heterogeneous Manycores. In Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS), 2016.

[3] S. Bansal and A. Aiken. Binary translation using peephole superoptimizers. In Proc. USENIX Conf. Operating Systems Design and Implementation (OSDI), 2008.

[4] A. Barbalace, R. Lyerly, C. Jelesnianski, A. Carno, H.-R. Chuang, V. Legout, and B. Ravindran. Breaking the boundaries in heterogeneous-ISA datacenters. In Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS), 2017.

[5] A. Barbalace, M. Sadini, S. Ansary, C. Jelesnianski, A. Ravichandran, C. Kendir, A. Murray, and B. Ravindran. Popcorn: Bridging the Programmability Gap in heterogeneous-ISA Platforms. In Proc. The European Conf. Computer Systems (EuroSys), 2015.

[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), 2009.

[7] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In Proc. USENIX Annual Technical Conference (ATC), 2005.

[8] E. Blem, J. Menon, T. Vijayaraghavan, and K. Sankaralingam. ISA wars: Understanding the relevance of ISA being RISC or CISC to performance, power, and energy on modern architectures. ACM Transactions on Computer Systems (TOCS), 33(1):3, 2015.

[9] D. Boggs, G. Brown, N. Tuck, and K. S. Venkatraman. Denver: Nvidia's First 64-bit ARM Processor. IEEE Micro, 35(2):46–55, 2015.

[10] S. Boyd-Wickizer and N. Zeldovich. Tolerating Malicious Device Drivers in Linux. In Proc. USENIX Annual Technical Conference (ATC), 2010.

[11] A. L. Brown and R. J. Wysocki. Suspend-to-RAM in Linux. In Ottawa Linux Symposium, 2008.

[12] X. Chen, N. Ding, A. Jindal, Y. C. Hu, M. Gupta, and R. Vannithamby. Smartphone Energy Drain in the Wild: Analysis and Implications. In Proc. ACM SIGMETRICS (SIGMETRICS), 2015.

[13] X. Chen, A. Jindal, N. Ding, Y. C. Hu, M. Gupta, and R. Vannithamby. Smartphone Background Activities in the Wild: Origin, Energy Drain, and Optimization. In Proc. Ann. Int. Conf. Mobile Computing & Networking (MobiCom), 2015.

[14] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. CloneCloud: Elastic Execution Between Mobile Device and Cloud. In Proc. The European Conf. Computer Systems (EuroSys), 2011.

[15] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. MAUI: making smartphones last longer with code offload. In Proc. ACM Int. Conf. Mobile Systems, Applications, & Services (MobiSys), 2010.

[16] A. d'Antras, C. Gorgovan, J. Garside, J. Goodacre, and M. Luján. HyperMAMBO-X64: Using Virtualization to Support High-Performance Transparent Binary Translation. In Proc. Int. Conf. Virtual Execution Environments (VEE), 2017.

[17] A. d'Antras, C. Gorgovan, J. Garside, and M. Luján. Low Overhead Dynamic Binary Translation on ARM. In Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI), 2017.

[18] 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), 2012.

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

[20] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An Operating System Architecture for Application-level Resource Management. In Proc. ACM Symp. Operating Systems Principles (SOSP), 1995.

[21] P. Feiner, A. D. Brown, and A. Goel. Comprehensive kernel instrumentation via dynamic binary translation. In ACM SIGARCH Computer Architecture News, 2012.

[22] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha. The Design and Implementation of Microdrivers. In Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS), 2008.

[23] B. Gerofi, A. Santogidis, D. Martinet, and Y. Ishikawa. PicoDriver: Fast-path Device Drivers for Multi-kernel Operating Systems. In Proc. Int. Symp. on High-Performance Parallel and Distributed Computing (HPDC), 2018.

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

[25] M. Hähnel and H. Härtig. Heterogeneity by the numbers: A study of the ODROID XU+E big.little platform. In Y. Agarwal and K. Rajamani, editors, Proc. Workshp. Power-Aware Computing and Systems (HotPower), 2014.

[26] U. Hansson. SDIO power on/off time impacts system suspend/resume time! http://connect.linaro.org/resource/sfo17/sfo17-402/, 2017.

[27] B. Hawkins, B. Demsky, D. Bruening, and Q. Zhao. Optimizing Binary Translation of Dynamically Generated Code. In Proc. Int. Symp. on Code Generation and Optimization (CGO), 2015.

[28] D. Hong, C. Hsu, P. Yew, J. Wu, W. Hsu, P. Liu, C. Wang, and Y. Chung. HQEMU: a multi-threaded and retargetable dynamic binary translator on multicores. In Proc. Int. Symp. on Code Generation and Optimization (CGO), 2012.

[29] R. J. Hookway and M. A. Herdeg. Digital FX! 32: Combining emulation and binary translation. Digital Technical Journal, 9:3–12, 1997.

[30] J. Howell, B. Parno, and J. R. Douceur. How to Run POSIX Apps in a Minimal Picoprocess. In Proc. USENIX Annual Technical Conference (ATC), 2013.

[31] Intel. Intel SuspendResume Project. https://01.org/suspendresume, 2015.

[32] A. Kadav and M. M. Swift. Understanding Modern Device Drivers. In Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS), 2012.

[33] A. Kantee and J. Cormack. Rump Kernels No OS? No Problem! Login: USENIX Magazine, 39(5), 2014.

[34] P. Kedia and S. Bansal. Fast Dynamic Binary Translation for the Kernel. In Proc. ACM Symp. Operating Systems Principles (SOSP), 2013.

[35] A. Klaiber. The technology behind Crusoe processors. Transmeta Technical Brief, 2000.

[36] G. Kroah-Hartman. The Linux Kernel Driver Interface – Stable API Nonsense. https://www.kernel.org/doc/Documentation/

process/stable-api-nonsense.rst. (Accessed on 05/04/2019).

[37] M. Larabel. A Stable Linux Kernel API/ABI? "The Most Insane Proposal" For Linux Development. https://www.phoronix.com/scan.php?page=news_item&px=Linux-Kernel-Stable-API-ABI, 2016.

[38] M. Lentz, J. Litton, and B. Bhattacharjee. Drowsy Power Management. In Proc. ACM Symp. Operating Systems Principles (SOSP), 2015.

[39] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In Proc. USENIX Conf. Operating Systems Design and Implementation (OSDI), 2004.

[40] T. Li, P. Brett, R. Knauerhase, D. Koufaty, D. Reddy, and S. Hahn. Operating system support for overlapping-ISA heterogeneous multi-core architectures. In Proc. IEEE Int. Symp. on High Performance Computer Architecture (HPCA), 2010.

[41] Y. Li, B. Dolan-Gavitt, S. Weber, and J. Cappos. Lock-in-Pop: securing privileged operating system kernels by keeping on the beaten path. In Proc. USENIX Annual Technical Conference (ATC), 2017.

[42] 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), 2012.

[43] F. X. Lin, Z. Wang, and L. Zhong. K2: A mobile operating system for heterogeneous coherence domains. In Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS), 2014.

[44] R. Liu and F. X. Lin. Understanding the Characteristics of Android Wear OS. In Proc. ACM Int. Conf. Mobile Systems, Applications, & Services (MobiSys), 2016.

[45] X. Liu, T. Chen, F. Qian, Z. Guo, F. X. Lin, X. Wang, and K. Chen. Characterizing Smartwatch Usage in the Wild. In Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services, 2017.

[46] LKML. [GIT PULL] PM updates for 2.6.33, 2009.

[47] D. Loghin, B. M. Tudor, H. Zhang, B. C. Ooi, and Y. M. Teo. A Performance Study of Big Data on Small Nodes. Proc. VLDB Endow., 8(7):762–773, 2015.

[48] G. Lu, J. Zhan, X. Lin, C. Tan, and L. Wang. On Horizontal Decomposition of the Operating System. CoRR, abs/1604.01378, 2016.

[49] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI), 2005.

[50] LWN. Redesigning asynchronous suspend/resume. https://lwn.net/Articles/366915/, 2009.

[51] 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), 2013.

[52] MediaTek. Microsoft Azure Sphere MCU with extensive I/O peripheral subsystem for diverse IoT applications. https://www.mediatek.com/products/azureSphere/mt3620, 2018.

[53] D. Meisner, B. T. Gold, and T. F. Wenisch. PowerNap: Eliminating Server Idle Power. In Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS), 2009.

[54] D. Meisner and T. F. Wenisch. DreamWeaver: architectural support for deep sleep. In Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS), 2012.

[55] Micron Technology, Inc. TN4201 LPDDR2 System Power Calculator. https://www.micron.com/support/tools-and-utilities/power-calc, 2013.

[56] Mike Turquette. The Common Clk Framework. https://www.kernel.org/doc/Documentation/clk.txt.

[57] C. Min, W. Kang, M. Kumar, S. Kashyap, S. Maass, H. Jo, and T. Kim. Solros: a data-centric operating system architecture for heterogeneous computing. In Proc. The European Conf. Computer Systems (EuroSys), 2018.

[58] J. Mogul, J. Mudigonda, N. Binkert, P. Ranganathan, and V. Talwar. Using Asymmetric Single-ISA CMPs to Save Energy on Operating Systems. IEEE Micro, 28(3):26–41, 2008.

[59] J. Morrison, D. Yang, and C. Davis. Apple watch: teardown. https://www.techinsights.com/about-techinsights/overview/blog/apple-watch-teardown/. (Accessed on 01/10/2019).

[60] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI), 2007.

[61] 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), 2009.

[62] NXP Semiconductors. i.MX 6SoloX - fact sheet. https://www.nxp.com/docs/en/fact-sheet/IMX6SOLOXFS.pdf. (Accessed on 05/14/2019).

[63] NXP Semiconductors. i.MX 8M Family of Applications Processors Fact Sheet. https://www.nxp.com/docs/en/fact-sheet/i.MX8M-FS.pdf. (Accessed on 05/14/2019).

[64] NXP Semiconductors. i.MX 7DS power consumption measurement. https://www.nxp.com/docs/en/application-note/AN5383.pdf, 2016.

[65] NXP Semiconductors. i.MX 7 Series Applications Processors | Arm® Cortex®-A7, Cortex-M4 | NXP. https://www.nxp.com/products/processors-and-microcontrollers/arm-based-processors-and-mcus/i.mx-applications-processors/i.mx-7-processors:IMX7-SERIES, 2017. (Accessed on 05/14/2019).

[66] H. Oi. A Case Study of Energy Efficiency on a Heterogeneous Multi-Processor. SIGMETRICS Perform. Eval. Rev., 45(2):70–72, 2017.

[67] Y. Padioleau, J. L. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in Linux device drivers. In J. S. Sventek and S. Hand, editors, Proc. The European Conf. Computer Systems (EuroSys), 2008.

[68] Y. Padioleau, J. L. Lawall, and G. Muller. Understanding collateral evolution in Linux device drivers. In ACM SIGOPS Operating Systems Review, 2006.

[69] N. Peters, S. Park, S. Chakraborty, B. Meurer, H. Payer, and D. Clifford. Web browser workload characterization for power management on HMP platforms. In Proc. IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES), 2016.

[70] A. Ponomarenko. ABI Compliance Checker. https://lvc.github.io/abi-compliance-checker/, 2018.

[71] 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), 2011.

[72] A. Reid. Trustworthy Specifications of ARM v8-A and v8-M System Level Architecture. In Proc. Formal Methods in Computer-Aided Design (FMCAD), 2016.

[73] S. Rokicki, E. Rohou, and S. Derrien. Hardware-accelerated dynamic binary translation. In Proc. ACM/IEEE Design Automation & Test in Europe Conf. (DATE), 2017.

[74] S. Rokicki, E. Rohou, and S. Derrien. Supporting runtime reconfigurable VLIWs cores through dynamic binary translation. In 2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19-23, 2018, 2018.

[75] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang. LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation. In Proc. USENIX Conf. Operating Systems Design and Implementation (OSDI), 2018.

[76] H. Shen, A. Balasubramanian, A. LaMarca, and D. Wetherall. Enhancing Mobile Apps to Use Sensor Hubs Without Programmer Effort. In Proc. Int. Conf. Ubiquitous Computing (UbiComp), 2015.

[77] M. Silberstein, B. Ford, I. Keidar, and E. Witchel. GPUfs: Integrating a File System with GPUs. In Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS), 2013.

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

[79] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering Device Drivers. In Proc. USENIX Conf. Operating Systems Design and Implementation (OSDI), 2004.

[80] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the Reliability of Commodity Operating Systems. In Proc. ACM Symp. Operating Systems Principles (SOSP), 2003.

[81] Texas Instruments. AM5728 Sitara Processor: Dual Arm Cortex-A15 & Dual DSP, Multimedia | TI.com. http://www.ti.com/product/AM5728. (Accessed on 05/14/2019).

[82] Texas Instruments. Cortex-M3: Processor technical reference manual. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0337h/index.html. (Accessed on 05/07/2019).

[83] Texas Instruments. OMAP4 Applications Processor: Technical Reference Manual. http://www.ti.com/lit/ug/swpu235ab/swpu235ab.pdf, 2010. (Accessed on 05/14/2019).

[84] D. Vasisht, Z. Kapetanovic, J. Won, X. Jin, R. Chandra, S. Sinha, A. Kapoor, M. Sudarshan, and S. Stratman. FarmBeats: An IoT Platform for Data-Driven Agriculture. In Proc. USENIX Symp. Networked Systems Design and Implementation (NSDI), 2017.

[85] VMWARE. Virtual Machine to Physical Machine Migration. https://www.vmware.com/support/v2p/doc/V2P_TechNote.pdf, 2004.

[86] W. Wang, S. McCamant, A. Zhai, and P.-C. Yew. Enhancing Cross-ISA DBT Through Automatically Learned Translation Rules. In Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS), 2018.

[87] W. Wang, P.-C. Yew, A. Zhai, S. McCamant, Y. Wu, and J. Bobba. Enabling Cross-ISA Offloading for COTS Binaries. In Proc. ACM Int. Conf. Mobile Systems, Applications, & Services (MobiSys), 2017.

[88] 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.

[89] S. L. Xi, M. Guevara, J. Nelson, P. Pensabene, and B. C. Lee. Understanding the Critical Path in Power State Transition Latencies. In Proc. ACM/IEEE Int. Symp. Low Power Electronics & Design (ISLPED), 2013.

[90] C. Xu, F. X. Lin, Y. Wang, and L. Zhong. Automated OS-level Device Power Management for SoCs. In Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS), 2015.

[91] F. Xu, Y. Liu, T. Moscibroda, R. Chandra, L. Jin, Y. Zhang, and Q. Li. Optimizing Background Email Sync on Smartphones. In Proc. ACM Int. Conf. Mobile Systems, Applications, & Services (MobiSys), 2013.

[92] S. Zhai, L. Guo, X. Li, and F. X. Lin. Decelerating Suspend and Resume in Operating Systems. In Proc. ACM Workshp. Mobile Computing Systems & Applications (HotMobile), 2017.

[93] Q. Zhu, M. Zhu, B. Wu, X. Shen, K. Shen, and Z. Wang. Software Engagement with Sleeping CPUs. In Proc. Workshp. Hot Topics in Operating Systems (HotOS), 2015.