TRANSKERNEL: AN EXECUTOR FOR COMMODITY KERNELS ON PERIPHERAL CORES

A Thesis

Submitted to the Faculty

of

Purdue University

by

Shuang Zhai

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

August 2019

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF THESIS APPROVAL

Dr. Felix Xiaozhu Lin, Chair

School of Electrical and Computer Engineering

Dr. Vijay Raghunathan

School of Electrical and Computer Engineering

Dr. T.N. Vijaykumar

School of Electrical and Computer Engineering

Approved by:

Dr. Dimitrios Peroulis

Head of the School Graduate Program

ACKNOWLEDGMENTS

First, of all, I would like to show my sincere thanks to my advisor, Prof. Felix Xiaozhu Lin, for his guidance and mentorship during my senior year and my master's study. He always encourages me to explore problems deeper while having a big picture in mind.

I want to express my appreciation to my committee members, Prof. Vijay Raghunathan and Prof. T.N. Vijaykumar, for their constructive feedbacks on my thesis.

For transkernel project, I would like to express thanks to Renju Liu, who was engaged in the early stage of this project. I am grateful for the collaboration with Liwei Guo and Yi Qiao. This project cannot be finished without you.

In addition, it is my pleasure to meet and work with other members in XSEL Lab, including Hongyu Miao, Heejin Park and Tiantu Xu.

Last but not least, I want to thank my parents for their unconditional support.

TABLE OF CONTENTS

Page

LIST OF TABLES vi

LIST OF FIGURES vii

ABSTRACT viii

1 INTRODUCTION 1

2 BACKGROUND AND MOTIVATIONS 5

2.1 Kernel execution in device suspend/resume 5

2.2 A peripheral core in heterogeneous SoC 8

2.3 Design space exploration 9

2.4 Design objectives for software on a peripheral core 13

3 THE TRANSKERNEL MODEL 14

4 A TRANSKERNEL IMPLEMENTATION 17

4.1 A Scheduler of DBT Context 18

4.2 Interrupt and Exception Handling 19

4.3 Deferred Work 20

4.4 Locking 21

4.5 Memory Allocation 22

4.6 Delays & Timekeeping 22

5 THE CROSS-ISA DBT ENGINE 24

5.1 Exploiting Similar Instruction Semantics 25

5.2 Passthrough of CPU registers 27

5.3 Control Transfer and Stack Manipulation 28

6 TRANSLATED-TO-NATIVE FALLBACK 30

7 EVALUATION 31

7.1 Methodology 31

Page

7.2 Analysis of engineering efforts 33
7.3 Measured execution characteristics 33
7.4 Energy benefits 36
8 RELATED WORK 41
9 CONCLUSIONS 43
REFERENCES 44

LIST OF TABLES

Table Page

2.1 Linux kernel functions and data types referenced in device suspend/resume. 11

4.1 Top: Major kernel services supported by ARK. Bottom: Linux kernel ABI (12 funcs+1 var) that ARK depends on . . . 19

5.1 The Linux kernel binary characterization. Column 3: the number of v7m instructions emitted for one v7a instruction . . . 25

5.2 Sample translation by ARK. By contrast, our baseline QEMU port translates G1–G3 to **27** v7m instructions . . . 26

7.1 I/O devices description and kernel services used . . . 32

7.2 Source code . . . 32

7.3 Test platform and power models in use . . . 36

7.4 Battery life extension under different suspend/resume intervals and energy ratio . . . 39

LIST OF FIGURES

Figure Page

1.1 An overview of this work 2

2.1 Alternative approaches for offloading kernel phases 10

3.1 The ARK structure on a peripheral core 16

7.1 Measured execution time and modeled energy in device suspend/resume. ARK substantially reduces the energy. 34

7.2 Busy execution overhead for devices under test. Our DBT optimizations reduce the overhead by up to one order of magnitude 34

7.3 System energy consumption of ARK relative to the native execution (100%), under different DBT overheads (x-axis) and busy execution fractions (y-axis). ARK's low energy hinges on low DBT overhead. 38

ABSTRACT

Zhai, Shuang MS, Purdue University, August 2019. Transkernel: An Executor for Commodity Kernels on Peripheral Cores. Major Professor: Felix Xiaozhu Lin.

Modern mobile devices have numerous ephemeral tasks. These tasks are driven by background activities, such as push notifications and sensor readings. In order to execute these tasks, the whole platform has to periodically wake up beforehand, and go to sleep afterwards. During this process, the OS kernel operates on power state of various IO devices, which has been identified as the bottleneck for energy efficiency. To this end, we want to offload this kernel phase to a more energy efficient, microcontroller level core, named peripheral core.

To execute commodity OS on a peripheral core, existing approaches either require much engineering effort or incur high execution cost. Therefore, we proposed a new OS model called transkernel. By utilizing cross-ISA dynamic binary translation (DBT) technique, transkernel creates a virtualized environment on the peripheral core. It relies on a small set of stable interfaces. It is specialized for frequently executed kernel path. It exploits ISA similarities to reduce DBT overhead.

We implement a transkernel model on ARM platform. With novel design and optimization, we demonstrate that a transkernel can gain energy efficiency. Moreover, it provides a new OS design to harness heterogeneous SoCs.

1. INTRODUCTION

Modern mobile and embedded platforms observe a large number of ephemeral tasks. These tasks are often driven by periodic or background activities. Such tasks include acquiring sensor readings, smart watch display updates [1], push notification services [2], and periodic data sync [3]. They drain up to 30% of battery life on smartphones [4, 5] and smart watches [6], and almost entire battery life on IoT devices for surveillance [7]. In order to execute an ephemeral task, the platform has to be waken up from deep sleep mode. After finishing such a task, it will go back to sleep again. This procedure, called system suspend/resume, has been introduced into OS kernel mainline for two decades and is intended for energy saving. Ironically, recent studies [2, 8] show that this mechanism becomes a heavy burden for today's mobile platforms. Sometimes system suspend/resume can consume much more energy than ephemeral tasks.

Why is the system suspend/resume so inefficient? Recent studies [8–10] show the energy bottlenecks are device suspend/resume, as shown in Figure 1.1. In this process, the kernel operates a variety of IO devices (or simply *devices* in this thesis) by calling device drivers suspend/resume callbacks. Those callbacks will drain all pending tasks (either finish them or abort them), and make sure the IO devices go to the target power states. The device suspend/resume is complex, incur numerous CPU idle periods, and is proven difficult to optimize as shown in prior works [10–12].

In order to execute device suspend/resume more efficiently, we suggest that it should be executed on a more energy efficient, microcontroller-level core. By design, this type of cores has much simpler microarchitectures (e.g., fewer pipeline stages, smaller cache size) and trimmed-down ISAs. Modern SoC designers already incorporate such cores (called peripheral cores) in their products [13–15], alongside with the normal CPU. For workloads with heavy IO and low performance demands, a peripheral core delivers much higher efficiency than the CPU [16–19].

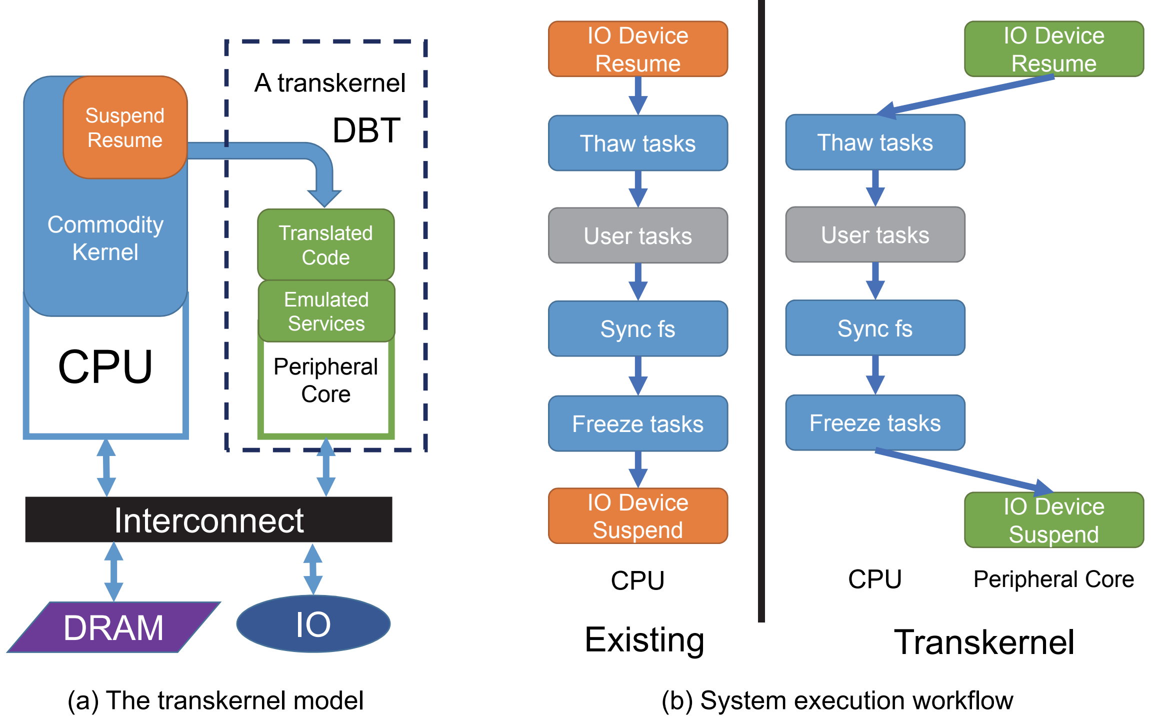


Fig. 1.1. An overview of this work

In order to offload a *commodity* kernel's phases to a peripheral core, we face the following challenges:

- The kernel code is complex and fast-evolving.
- The peripheral core has different ISA and wimpy hardware.

To be more specific, diverse device drivers implement suspend/resume and invoke multiple kernel layers, which makes code transplant to the peripheral core difficult. The multikernel designs [20–22] are intended for maintaining a single OS image atop the CPU and the peripheral core. They require to craft a wide interface for synchronizing kernel state between two type of cores in different ISAs. Unfortunately, such interface is fragile due to changes in the kernel's build configurations, the kernel's compilation flags, and the kernel source code updates.

We want to enable the offloading to meet the following goals: i) Being able to reuse most of the commodity kernel code with tractable engineering effort; ii) developing

and compiling the software for a peripheral core *once* that can work with *many* builds of the kernel from different configurations and release versions; iii) low runtime overhead to achieve energy efficiency gain.

We take an approach which seems to be infeasible, shown in Figure 1.1: By applying dynamic binary translation (DBT), we enable the peripheral core to execute unmodified binary of a commodity kernel, although this technique is previously believed to be expensive [20].

Therefore, we propose a new executor model applied for a peripheral core, named transkernel. A transkernel hosts a DBT engine. It is able to translate the original kernel binary executed in the offloaded kernel phases. Underneath the translated code, to bridge the ISA gap, the transkernel implements a small set of emulated services. The role of those emulated services is to provide lightweight alternatives for their counterparts in the commodity kernel. The transkernel follows four principles: i) translating stateful code while emulating stateless kernel services; ii) choosing narrow and stable interfaces for emulation; iii) specializing for frequently executed paths; iv) exploiting similarities between heterogeneous ISAs to reduce DBT overhead.

By applying the model, we have built a transkernel prototype called ARK. Targeting on an ARM-based heterogeneous SoC, ARK runs on a Cortex-M3 peripheral core alongside Linux running on the Cortex-A9 CPU. ARK demonstrates that the idea of transkernel is feasible. ARK transparently executes unmodified Linux kernel drivers and libraries. It depends on a narrow, stable binary interface (ABI), including only 12 kernel functions and one variable. ARK is able to execute kernel phases that invoke multiple layers of kernel, including diverse drivers (e.g. USB host controller and Bluetooth NIC) and sophisticated kernel services (e.g. deferred work and interrupt handling). ARK only incurs 2.7× busy execution overhead as compared to native kernel execution on the CPU. This is crucial to ARK's benefit. It reduces device suspend/resume energy consumption by 34%. This is a tangible battery life extension in real world scenarios.

In this thesis, we make the following contributions on OS and DBT:

- We present a new OS model, the transkernel, that enables a peripheral core to execute commodity kernel phases of the CPU. Targeting on heterogeneous multiprocessors, the transkernel provides a new design point which combines DBT for bridging ISA gaps and emulation for catering hardware heterogeneity.

- We contribute a new paradigm of cross-ISA DBT. It runs on a microcontroller-like core, and is able to dynamically translate and execute unmodified kernel binary of the CPU. We contribute new DBT optimizations that exploit ISA similarities. We demonstrate that while traditional usage of cross-ISA DBT trades in efficiency for compatibility, it can actually gain efficiency on existing hardware.

- We implement a transkernel model, ARK, on top of a heterogeneous ARM SoC. ARK contributes concrete designs of kernel service emulation. ARK meets our goal of tractable engineering efforts and "build once run with many". ARK offers tangible benefits of energy efficiency.

2. BACKGROUND AND MOTIVATIONS

In this chapter, we will discuss device suspend/resume, which is the major kernel bottleneck in ephemeral tasks. To alleviate its energy inefficiency, we argue that it should be offloaded to a peripheral core. We will first discuss challenges in exiting approaches, and then motivate our design objectives.

2.1 Kernel execution in device suspend/resume

The OS kernel will put the whole platform into a deep sleep state after a long period of system idle. During suspend, the kernel synchronizes file systems with persistent storage, and then freezes all user tasks. After that, the kernel puts each individual IO devices to low power modes (i.e., device suspend). Finally, the CPU is turned off. Resume follows a mirrored procedure. A detailed description on suspend/resume can be found in Linux documentation [23]. To execute an ephemeral task, usually more time is spent on the kernel execution (hundreds of milliseconds [24]) than user applications (tens of milliseconds [8]). And the kernel execution consumes several times more energy compared to the user code [2].

Problem: device suspend/resume Recent work [10] profiles Linux suspend/resume on various mobile devices, and demonstrates that device suspend/resume is the bottleneck in kernel execution. These kernel phases (device suspend/resume) happen right before powering off the CPU and right after powering on the CPU. During the phases, the Linux kernel drains all pending IO tasks and puts the devices into target power states. We will show a brief summarize of findings in prior work below.

1. ***Device suspend/resume is inefficient.*** It consumes 54% on average and up to 66% of the total energy spent during the kernel execution. The transitions

of device power states take long. CPU idle periods are frequently observed. They show up in the form of many short epochs, and each of them takes several milliseconds.

2. ***Devices are diverse.*** For each platform, the kernel suspend/resume operates on tens of different devices. And various devices incur long kernel execution delay across different platforms, which makes optimizations on specific devices impractical.

3. ***Optimization is difficult.*** Device power state transitions are limited by slow hardware, low-speed buses (e.g., I2C bus), and physical factors (e.g., capacitor voltage ramp-up). Devices have *implicit* power, voltage, and clock dependencies, requiring certain power transitions to happen in a specific order. For example, if the processor communicate with the voltage regulator through I2C bus, it is impossible to shut down the I2C controller before turning off the voltage regulator. Modern Linux already overlaps the transitions of different devices with best efforts [11, 12]. Yet, as shown in the prior work, CPU idle still constitutes up to 68% of the duration of device suspend/resume.

Challenge: Widespread, complex code invoked Device suspend/resume involves multiple kernel layers [25, 26], including callbacks in individual drivers (e.g., for the USB controller), driver libraries (e.g., for the generic clock framework), kernel libraries (e.g., for manipulating queue in kernel), and kernel services (e.g., memory allocator). The execution is control-heavy, with numerous branches and function calls. In a recent Linux source tree (4.4), we find that suspend/resume callbacks are implemented in over 1000 device drivers, which covers almost all driver classes. Those callbacks invoke over 43K SLoC in driver libraries, 8K SLoC in kernel libraries, and 43K SLoC in kernel services.

Opportunities We identify the following kernel behaviors in device suspend/resume as opportunities.

1. ***Beaten kernel paths*** Every successful attempt of suspend/resume phases usually follows the same execution path (i.e., beaten paths [27]). Under this scenario, the kernel is able to acquires all necessary resources without any errors. When going off the beaten path, the kernel needs to handle rare cases, such as races between IO events (e.g., improper use in drivers sharing the same bus), resource shortage (e.g., low physical memory), and hardware failures. These branches typically cease the current suspend/resume attempt, perform diagnostics and fix-ups, and schedule a later retry. Compared to the beaten paths, they invoke very different and complicate kernel services, e.g., syslog and file systems.

2. ***Simple concurrency*** The kernel suspend/resume follows a simple concurrency model. There are very few contexts exist, including syscall path (which initiates the platform suspend/resume), interrupt handlers, and deferred kernel work. The purpose of concurrency is for leveraging hardware asynchrony and kernel modularity rather than exploiting multicore parallelism.

3. ***Low sensitivity to execution delay*** On embedded platforms, most ephemeral tasks are driven by background activities [2, 4, 28]. They are insensitive to the execution latency. This contrasts to many servers for interactive user requests [28, 29], which require fast response.

Summary: design implications To address the inefficiency in suspend/resume, prior works and our observations suggest that we should systematically treating the device phase. We face challenges that the invoked kernel code is diverse, complex, and cross-layer; we see opportunities that allow focusing on beaten kernel paths, specializing for simple concurrency, and trading in execution delay for higher efficiency.

2.2 A peripheral core in heterogeneous SoC

We believe that the efficiency problem of device suspend/resume can be substantially mitigated by introducing a peripheral core on a modern heterogeneous SoC, which has the following characteristics.

Hardware model

1. ***Asymmetric processors***: The CPU and the peripheral core provides different trade-offs between performance and energy efficiency. The peripheral core does not have MMU. It only has memory protection unit (MPU), which can map at most tens memory regions without virtual address translation capability. It cannot run commodity OSes.

2. ***Heterogeneous, yet similar ISAs***: The two processors have different ISAs. Many instructions from the two ISAs have similar *semantics*, as will be discussed in detail below.

3. ***Loose coupling***: The two processors are located in separate power domains. One of them can be turned on while the other one is off.

4. ***Shared platform resources***: Both processors share system DRAM and can access to IO devices registers. IO interrupts are physically routed to both processors.

Many SoC designers incorporate the peripheral core into their products (e.g. iPad Pro and Azure Sphere), which fit this hardware model [30–33].

Promise of high efficiency Prior studies [16, 18, 19, 34, 35] have shown that a peripheral core is an ideal place to execute IO-heavy workloads with low performance demand efficiently. It benefits the kernel's device suspend/resume in the following ways:

1. The peripheral core can operate independently while leaving the CPU offline.

2. The idle power of a peripheral core is often significantly lower than CPU [17,36], favoring device suspend/resume workloads with numerous CPU idle epochs.

3. Its simple microarchitecture suits kernel execution, whose irregular behaviors often see very little benefits from higher power, advanced microarchitectures (e.g., speculative execution) [37]. Note that a peripheral core offers much higher energy efficiency than a LITTLE core in ARM big.LITTLE [38]. This is because both big cores and LITTLE cores require an agreement on ISA and tight core coupling. We will examine big.LITTLE in Chapter 7.

ISA similarity On an heterogeneous SoC we target, the CPU and the peripheral core's ISAs are often from the same family, e.g. ARM or MIPS. Compared to the CPU, the peripheral core often implements a subset of instructions with same or similar *semantics*, but in different *encoding*. The common examples are SoCs integrating ARMv7-A ISA and ARMv7-M ISA [13, 14, 32, 39, 40]. Other ISA families offer their ISA options that can be integrated on the same SoC, e.g. NanoMIPS and MIPS32, IA-32 and x86-64. We believe the existence of ISA similarities are by choice. i) For ISA designers, it is feasible to explore performance-efficiency trade-offs within one ISA family, since the family choice is merely about instruction *syntax* rather than *semantics* [41]. The designers likely start from common instruction semantics and instantiate them differently to better cater for workloads of specific processor profiles. ii) For SoC vendors, incorporating ISAs from the same family on one chip reduces the efforts in building software tools and libraries [42] as well as facilitating silicon design and ISA licensing.

2.3 Design space exploration

Next, we will explore OS designs that enable a peripheral core to execute a commodity kernel's phases. Our challenges are listed as below:

- The kernel code is complex and fast-evolving.

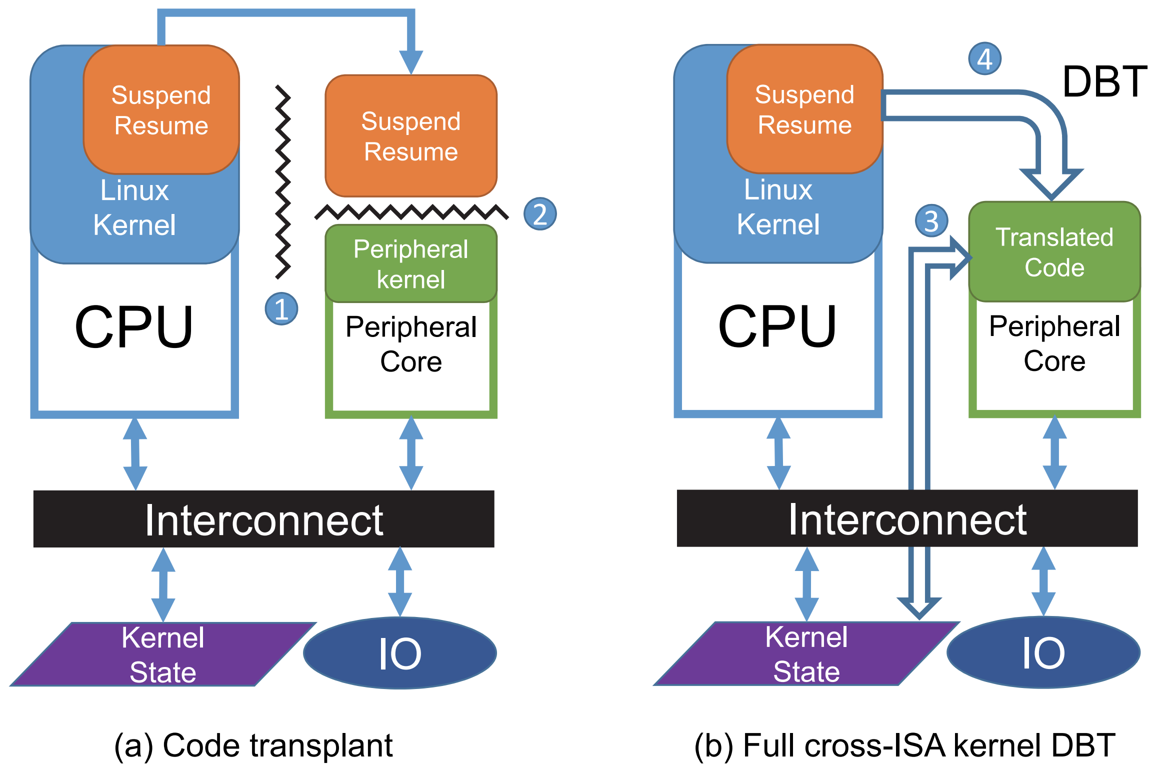


Fig. 2.1. Alternative approaches for offloading kernel phases

- The peripheral core's different ISA and wimpy hardware compare with the CPU.

Code transplant creates fragile interfaces Because a commodity kernel (e.g., Linux) cannot manage heterogeneous processors with different ISAs out-of-box [17, 20], one possible solution is to identify the boundary of device suspend/resume source code from the Linux kernel, cross-compile and run it on top of a "peripheral kernel" on the peripheral core. The peripheral kernel is vital to the offload kernel phases autonomously. This approach results in a multikernel OS [22] shown in Figure 2.1(a). However, it relies on two interfaces (shown as ∿∿ in the figure) which are difficult to maintain stability.

① The interface between the two kernels, needed for their data dependency. Before and after offloading, the CPU's kernel and peripheral kernel synchronize the Linux kernel state through this interface. Examples of kernel state include device descrip-

Table 2.1.
Linux kernel functions and data types referenced in device suspend/resume.

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*	

Legend: **# funcs** / *# types*; columns broken down into: device specific, driver libraries, kernel library, kernel services

(A) # of functions & types with changed ABI across versions

Layer	# of functions
device specific	359
driver libraries	845
kernel library	217
kernel services	858

(B) # of functions

Columns: breakdown of funcs by layers. Exported symbols only. Build config: omap2defconfig. ABI changes detected with ABI compliance checker [43]

tions and configurations, device request blocks, and pending IO tasks. Whether the interface is based on message passing [21, 22] or software shared memory [17, 20, 44], it is essentially based on an agreement on the definitions of shared Linux kernel data types, including their semantics and/or memory layout. The kernel data types are easily changed by choices of ISA, kernel configurations, and kernel releases. Therefore, the agreement is highly fragile. Table 2.1(a) summarizes numerous changes to the data types referenced in device suspend/resume across different versions. It shows that the efforts in building this interface is not only tedious (for maintaining the agreement on shared memory data types across heterogeneous ISAs) but repetitive: any data type change would break the interface and require to revise and rebuild the peripheral kernel [44, 45].

② The interface between the transplant code and the peripheral kernel. The transplant code has functional dependency which is resolved by the peripheral kernel's interface. The interface is determined by the choice of transplant boundary. Com-

mon transplant boundaries include the bottom of device-specific code [44–46], that of driver classes [47,48], and that of driver libraries [17]. All these choices expose at least hundreds of Linux kernel functions on this interface, as summarized in Table 2.1(b). This is due to diverse drivers on embedded platforms and Linux's sophisticated internals. Implementing such an interface on the peripheral core requires huge engineering effort. As the Linux kernel is rapidly evolving, maintaining the interface is even difficult. As shown in Table 2.1(a), the ABI of these functions significantly changes as Linux evolves [49]. The peripheral kernel has to be revised and rebuilt to comply to the updated ABI for every kernel release.

Current cross-ISA DBT is unaffordable Alternatively, another solution is to run DBT on the peripheral core for translating the Linux kernel for the *entire* offloaded phase, shown in Figure 2.1(b). DBT is a well-know technique to allow a host processor (in this case, the peripheral core) to execution instructions in a guest ISA (the CPU's ISA). DBT does not have difficulties mentioned above, because the translated code follows the same behavior as the kernel's binaries, and directly operates the kernel state (3). The DBT only relies on a stable interface, the CPU's ISA (4), to interact with the Linux kernel. However, prior work [50] shows that existing cross-ISA DBT incurs high overhead. The overhead is furthur magnified by our inverse DBT paradigm. Whereas existing cross-ISA DBT is designed and optimized for a powerful host (e.g., an x86 desktop) serving a weak guest (e.g., an emulated ARM platform) [51, 52], our DBT host, a peripheral core with simpler architecture and microarchitecture, emulates a fully-fledged CPU. A straightforward port of a popular DBT engine exhibits up to 25× slowdown as will be shown in Chapter 7. The overhead would kill the advantage of hardware efficiency and lead to overall efficiency loss. Furthermore, whole Linux kernel cross-ISA DBT is complex and expensive [53]. A peripheral core lacks necessary environment, e.g. multiple address spaces and POSIX support, for developing and debugging such complex software.

2.4 Design objectives for software on a peripheral core

To overcome the difficulties, we set the following three objectives.

G1. Tractable engineering effort. To avoid tedious code transplant, we want to reuse much of the commodity kernel source, in particular the rich, fast-evolving drivers that are impractical to build anew. We target a simple structure for the peripheral core's software.

G2. Build once, work with many. Every build of peripheral core's software should work with a wide variety of configurations and releases of the commodity kernel. This requires the peripheral core's software to interact with the commodity kernel through a narrow and stable ABI.

G3. Low overhead. The offloaded kernel phases should incur low execution overhead in order to yield a tangible efficiency gain.

3. THE TRANSKERNEL MODEL

We propose a new executor model called transkernel. Running on a peripheral core, a transkernel consists of two key components. It has a DBT engine to transparently translate and execute unmodified commodity kernel binary, which includes device drivers, device libraries, and stateful kernel services. There is a set of emulated stateless kernel services to serve under the translated code. A concrete transkernel implementation targets a specific commodity kernel, e.g. Linux. In order to achieve the goals mentioned in Chapter 2, the transkernel follows four principles:

Translating stateful code; emulating stateless services By *stateful code*, we refer to the offloaded code that must operate on the same kernel state shared with the CPU. We believe that the stateful code include device drivers, driver libraries, and a small set of kernel services. They cover the most diverse and widespread code invoked in device suspend/resume (§2). Through translation, the transkernel reuses commodity kernel code transparently without transplanting kernel source to a peripheral core (G1); the translated code operates kernel state without relying on a fragile ABI (G2).

We do not strictly preserve the semantics of all kernel services. Instead, we relax the semantics of the emulated services to be stateless, so that the state of these services only lives within one device suspend/resume phase. Because the emulated services are stateless, they do not have to synchronize the kernel state with the CPU's kernel.

Choosing a narrow, stable interface for emulation We determine the translation/emulation boundary to be a small set of kernel functions and kernel variables. We ensure that the ABI of the chosen kernel functions are unaffected by kernel config-

urations and has not changed since long in the kernel evolution history. (G2) It also increases the chance that the build of transkernel to work with future kernel releases.

Specializing for the beaten path The transkernel only executes the beaten path of device suspend/resume; in the rare events of the execution goes off the beaten path, it has a mechanism to transparently falls back to CPU. Instead of strictly following kernel's behaviors, the emulated services seek *functional equivalence*; under the same interfaces, they only implement features needed by the beaten path. This is in the spirit of kernel specialization as in library OSes [54–56]. It entails tractable implementation (G1).

Exploiting ISA similarities for DBT We do not follow generic cross-DBT design principles which optimize for translation between arbitrary ISA pairs. Instead, we exploit similarities in instructions semantics, register usage, and control flow transfer. This reduces the overhead of DBT on peripheral core significantly and ultimately makes transkernel practical. (G3)

Chapter 4 below describes how we apply the model and implement transkernel, in particular our translation/emulation decisions for major kernel services, and our choices of the emulation interface. Chapter 5 will describe our DBT design and optimizations.

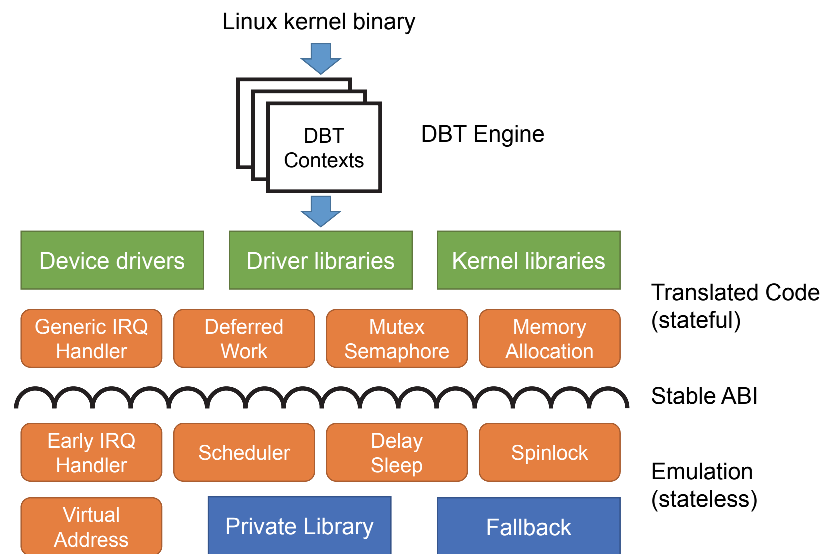


Fig. 3.1. The ARK structure on a peripheral core

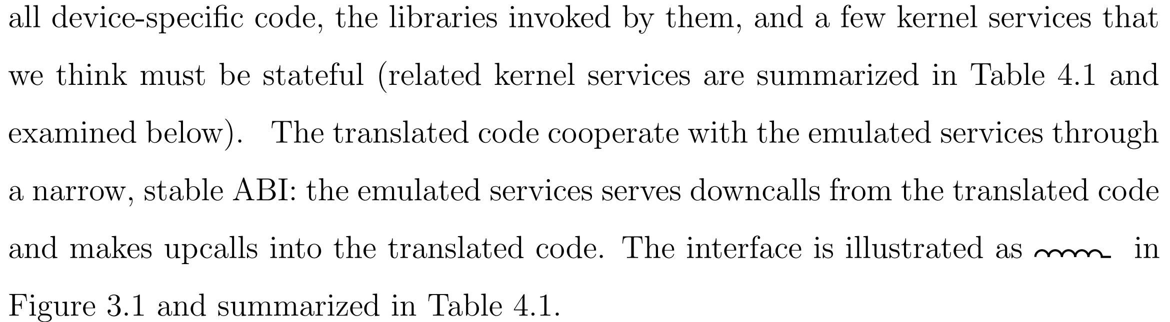
4. A TRANSKERNEL IMPLEMENTATION

We implement ARK, a transkernel targeting on ARM SoC. The CPU has the ARMv7-A ISA and the peripheral core has the ARMv7-M ISA. This is a popular combination which fits our hardware model described in Section 2.2. The CPU runs Linux v4.4.

The offloading workflow ARK is packed as a standalone executable for the peripheral core, accompanied by a small Linux kernel module for the CPU to replace the kernel entry points for device suspend/resume, allowing control transfer between the peripheral core and the CPU. We refer to control transfer as *handoff*. Prior to a device suspend phase, the kernel leaves only one core on and shuts down all other cores, passes control to the peripheral core, and shuts down the last CPU core. Then, the ARK executes the device phase to suspend the entire platform. Normally, device resume is executed by the ARK on the peripheral core. However, in case of urgent wakeup events which require immediate response (e.g. unlocking a smartphone screen by a user), the ARK wakes up the CPU and the kernel resumes with native execution.

System structure The ARK consists of a DBT engine, a set of emulated kernel services. Besides, the ARK also implements a small library to manage the peripheral core's private hardware resource, e.g., cache controller and interrupt controller.

As shown in Figure 3.1, ARK runs a DBT engine, a set of emulated kernel services, and a small library for managing the peripheral core's private hardware, e.g. controllers of cache and interrupt. Upon booting, ARK replicates the linear memory mappings of the Linux kernel, so that it can address the Linux kernel's memory objects, which is similar to prior systems [17, 44]. To access I/O regions, ARK allocates MPU entries for them and set the corresponding attributes; in case of limited number of entries, ARK allocates entries for the regions on demand. ARK translates



all device-specific code, the libraries invoked by them, and a few kernel services that we think must be stateful (related kernel services are summarized in Table 4.1 and examined below). The translated code cooperate with the emulated services through a narrow, stable ABI: the emulated services serves downcalls from the translated code and makes upcalls into the translated code. The interface is illustrated as 〰 in Figure 3.1 and summarized in Table 4.1.

To support concurrency in the kernel phases, ARK runs multiple DBT contexts. Each context has its own DBT state (e.g., virtual CPU registers and a stack) and is able to independently executes DBT and the emulated services. Switch among DBT contexts is as cheap as updating the pointer of current DBT state to the next runnable DBT context.

ARK is specialized for the beaten paths. If the execution reaches unbeaten branches pre-defined by us, e.g. blacklisted kernel functions, ARK migrates all the DBT contexts of *translated* code back to the CPU and continues as *native* execution there (§6).

4.1 A Scheduler of DBT Context

Instead of translating Linux scheduler, ARK emulates a simple scheduler, which does not share state with Linux scheduler. Because of simple concurrency model of suspend/resume (§2), ARK does not preserve Linux's preemptive multithreading but instead cooperatively select the DBT context to execute: 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). ARK uses a simple, round-robin scheduling strategy to manage tens of contexts. 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 a hardware interrupt occurs, ARK saves the current DBT context state and switches to the IRQ context to translate and execute the kernel interrupt handler (§4.2).

Table 4.1.
Top: Major kernel services supported by ARK. Bottom: Linux kernel ABI (12 funcs+1 var) that ARK depends on

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); all others unchanged since 2011 (v2.6).

4.2 Interrupt and Exception Handling

During the offloaded phase, all interrupts are redirected to the peripheral core and handled by ARK. The CPU is relieved from interrupts and stays in deep sleep mode.

Kernel interrupt handlers ARK emulates the early stage of interrupt handling while translating the later stage of kernel interrupt handler. This is because the early stage behavior varies depending on ISA and hardware environment, e.g., stack layout during interrupt and IRQ line number. 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 4.1), 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. Since the CPU controller's registers are private and not accessible from the peripheral core by physical design, upon accessing them (e.g. for masking interrupt sources) the translated code triggers faults. When identifying an access to the controller register, ARK operates the peripheral core's controller accordingly and skips to the next instruction.

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

In the device phase, device drivers frequently submit works which are expected to be done in the future. ARK translates the Linux services that schedule the deferred work; it also translates the actual execution of the deferred work. The reason to translate them is that such services must be *stateful*: the peripheral core may need to execute deferred work created on the CPU prior to the offloading, e.g. draining unfinished USB requests; it may defer new work until after the completion of resume, e.g., reclaiming memory for finished USB request blocks.

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 go to *sleep* with separate DBT contexts so that if one of them go to sleep it will not block other deferred works. 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 (bottom halves) to a kernel thread which executes the work after the hardware IRQ is handled. A threaded IRQ handler may go to 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 4.1).

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 sleep in 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 To protect short critical sections, spinlocks are implemented as disabling interrupt of current core. ARK emulates spinlocks, because their implementation is core-specific and that ARK can safely assume all spinlocks are free at handoff points: handoff happens between one CPU core and one peripheral core as described in Chapter 4. They do not hold any spinlock. All other CPU cores are offline and cannot hold spinlocks. Hence, ARK emulates spinlock acquire/release by enabling/disabling interrupt of the peripheral core. This is because ARK runs on one peripheral core, and the only possible concurrent execution comes from hardware 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 mu-

tex preventing suspend/resume from concurrently changing clock configuration [57]. Furthermore, mutex's seemingly simple interface (i.e., compare and 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 it fails to acquire the lock and therefore needs to update reference count and puts the caller to sleep. for which the translated execution makes downcalls to the emulated services.

4.5 Memory Allocation

The device driver frequently allocates memory dynamically. For example, USB driver allocates memory to store the response of URB (USB Request Block). Such requests are served by Linux kernel slab allocator. In the extreme cases when no physical pages available, the kernel will swap out user pages or kill user processes.

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 rare case that the allocation enters the slow path (e.g. due to low physical memory), ARK aborts offloading. With a stateful allocator, the offloaded execution can free dynamic memory allocated by during the kernel execution on CPU, and vice versa. Compare to prior work that instantiates per-kernel allocators with split physical memory [17], ARK reduces memory fragmentation and avoids tracking *which* processor should free *what* dynamic memory pieces. Our experience in Chapter 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 In the Linux kernel, `udelay()` and `msleep()` are used for busy waiting and idle sleeping. ARK emulates them by converting wait time to hardware timer cycles on

the peripheral core. For msleep, ARK also marks the caller context as unschedulable. After timer is fired, the corresponding context may continue execution.

jiffies Jiffies is a global integer to record the elapsed time since the system booted. It is used as a normal precision timekeeper, and is updated periodically by the Linux kernel. By properly configuring the hardware timer of the peripheral core, ARK directly updates jiffies on behalf of CPU's kernel. All timekeeping related functions can be translated since they only depend on this variable. Also jiffies is the only shared variable on the kernel ABI which ARK depends upon.

5. THE CROSS-ISA DBT ENGINE

A Cross-ISA DBT Primer DBT is a well known technique allowing host processor to execute instructions in guest ISA. To do so, a special program, called DBT engine, runs on host processor to emulate the hardware environment of guest processor. The engine loads guest instructions and translates them to host instructions based on pre-defined translation rules at run time. The translation is done in unit of translation block, a sequence of instructions with one entry point and one or more exit targets. To avoid the translation effort for the same translation block, the translated instructions are stored in a dedicate piece of memory, called code cache. Therefore, all substantial execution of the same translation block does not require translation. A dispatcher manages all the translated blocks and decides the control flow.

Design overview We build ARK atop QEMU [53], 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. This is confirmed by static analysis of v7a's Linux binary. (§5.1)

2. Both ISAs have the same number of general purpose registers. The condition flags in both ISAs have the same semantics. (§5.2)

3. Both ISAs use program counter (PC), link register (LR), and stack pointer (SP) in the same way. (§5.3)

Table 5.1.
The Linux kernel binary characterization. Column 3: the number of v7m instructions emitted for one v7a instruction

Category		Count	v7m
w/ counterparts	Identity	741k	1
	Side effect	42	3-5
	Constraints on constant	3291	2-5
	Shift modes	2395	2
w/o counterparts		2	2-5

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

We build translation rules by carefully examine the semantics of instructions described in ARM architecture reference manual [58,59]. Our overall guideline is to map as many v7a instructions to single v7m instructions that have identical semantics as possible. We call them *counterpart* instructions. For instructions without identical counterparts, 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 building translation rules by looking for semantic equivalence of a sequence of instructions in cross-ISA DBT [60]. This is because ISA similarity allows identity translation for most guest instructions.

To verify the coverage and efficiency of such translation rules, we statically analyze the Linux kernel binaries and categorize them. Note that we only take load/store and data-processing instructions into consideration, since other instructions (e.g., cache maintenance) are managed by ARK's library as described in Chapter 4. Table 5.1 summarizes over 747k instructions from the Linux kernel which fall into our scope. Among them, 99% can be translated with identity rules, for which ARK only needs

Table 5.2.
Sample translation by ARK. By contrast, our baseline QEMU port translates G1–G3 to **27** v7m instructions

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`

to convert instruction encoding at run time. Less than 1% of v7a instructions in the Linux kernel have v7m counterparts but may require amendment instructions, which fortunately fall into a few categories: i) ***Side effects.*** After load/store, some v7a instructions support special addressing modes and may additionally update register values (shown in Table 5.2, G1). ARK emits amendment instruction to emulate the extra side effect (H3). ii) ***Constraints on constants.*** The range of immediate value that can be encoded in a single v7m instruction is often narrower (Table 5.2, G2). In such cases, the amendment instructions load the immediate to a scratch register, perform shift/rotation, 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 shown as Table 5.2 G1, where a v7m instruction cannot perform LSR (logic shift right) within the instruction as its v7a counterpart. Similar to above, the amendment instructions load the operand to a scratch register and perform shift on the register.

Beyond the above, only 2 instances of 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 5.2,

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 emitting a host instruction, ARK follows register allocation in the guest counterpart with best efforts (e.g., one-to-one mapping in best case, as in Table 5.2, G1). The choice of register is decided by the compiler generating the guest binary, which has the entire source code to make optimal decision. ARK emits much fewer host instructions than QEMU, which emulates all guest registers in memory.

Caveats fixed To bridge the gap between two ISAs, the amendment host instructions may need extra scratch registers, as exemplified by `t0` in Table 5.2, H2-H6. However, since both the host and guest have same number of general purpose registers, the need for scratch registers put a higher pressure on register allocation. QEMU allocates the register at the scope of each translation block and do not maintain the same mapping, which requires expensive register load/stores. 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; we experimentally determined it as R10 by analyzing the ARM Linux kernel. We find that most amendment instructions are satisfied by *one* scratch register; in rare cases when extra scratch registers are needed, ARK follows a common register allocation design to firstly allocate dead registers, or spill unused ones to memory if there is no dead register.

Condition flags Both the guest and the host ISAs involve five condition flags (e.g. zero and carry) with identical semantics. QEMU emulates guest CPU condition flags as individual variables in host memory. Therefore, to translate each instructions that

may affects the flags, QEMU emits as many as 7 host instructions to manipulate emulated flags. To relieve expensive flags emulation, the host instructions emitted by ARK directly set and test the host's hardware condition flags. Fortunately, most guest (v7a) instructions and their host (v7m) counterparts have identical behaviors in testing/setting flags, verified by comparing instructions semantics from both ISAs. Such flag passthrough hence incurs much lower overhead than QEMU. Such optimization provides substantial benefit for the control-heavy suspend/resume, which contains extensive conditional branches (§2).

Caveats fixed Amendment host instructions may affect the hardware condition flags unexpectedly, e.g., checking address of memory access. 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, storing function return address) to facilitate function call/return. QEMU emulates guest PC and LR in host memory. For each guest function calls, it loads the emulated PC (return address) to emulated LR and pushes to stack. However, this is expensive since the return address, loaded from stack or the emulated LR, points to a guest address. Each function return hence pops the return address from stack, causes the DBT to jump back to dispatcher 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 PC or LR. For each guest function calls, the host code pushes the host PC (return-to addresses in *code cache*) to stack and to LR; for each guest function returns, the host code loads the hardware PC with the return address (which points to code cache) popped from

the stack or from the hardware LR. By doing so, ARK no longer seek for help from dispatcher in all function returns. Our optimization is inspired by same-ISA DBT [61].

Stack and SP QEMU emulates the guest stack with an array and the guest SP in the host memory. Each guest push/pop translates to multiple memory store/load and emulated SP update.This is expensive, especially for device suspend/resume which frequently makes function calls, read/write local variables and operates stack heavily.

The 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). In addition, hostly translating SP operations and preserving stack frames further makes the migration in abort (§6) feasible.

Caveats fixed i) When there is a function call, the return address pointing to the code cache is pushed to stack. However, the instructions in the code cache is not executable for 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). If a push/pop instruction involve one of the emulated registers, ARK must emit multiple host instructions to correctly synchronize the emulated registers in memory.

6. TRANSLATED-TO-NATIVE FALLBACK

When ARK goes off the beaten paths, it migrates device suspend/resume back to the CPU to continue execution, analogous to virtual-to-physical migration of VMs [62]. Migrating one DBT context is straightforward: it passes emulated CPU states (registers, condition flags) and stack contents with fixed return address (§5.3) to CPU, flushes the cache, and wake up the CPU through IPI. However, because ARK implements multiple DBT contexts, there are more challenges to address.

Migrate DBT contexts for deferred work After fallback, all unfinished deferred work should migrate to the CPU and continue execution. Since deferred work is emulated as described in Section 4.3, the workitems are stored in ARK which cannot migrate to CPU's existing kernel structure directly. To address this issue, upon fallback, ARK initiates kernel worker threads to receive and execute unfinished workitems. Those worker threads are reclaimed once all workitems are drained.

Migrate DBT context for interrupt If fallback happens in the execution of the ISA-neutral interrupt handler (translated), the remainder of the handler should migrate to the CPU. The challenge is that the interrupt happens when the CPU is off; there is no corresponding interrupt context on the CPU. ARK addresses this by initiate an IPI from the peripheral core to the CPU by using hardware mailbox; the Linux kernel uses the IPI context as the receiver for the migrated interrupt handler, and starts executing from the fallback point in the interrupt handler. ARK gives up complete transparency: if Linux checks the CPU's interrupt controller, it will find that the interrupt comes from IPI instead of an IO device. However, we do not see an ISA-neutral interrupt handler check interrupt source in our evaluation.

7. EVALUATION

In this chapter, we seek to answer the following questions:

1. Can ARK be implemented and maintained with manageable engineering efforts? (§7.2)

2. Can ARK achieve low execution overhead? (§7.3)

3. Can ARK yield energy efficiency benefit compared with alternatives? How does major factors impact energy savings? (§7.4)

7.1 Methodology

Test Platform We evaluate ARK on Pandaboard-ES with TI OMAP4460 SoC [30]. As summarized in Table 7.3, it has dual ARM Cortex-A9 cores and dual Cortex-M3 cores. The commodity Linux kernel manages dual A9, whereas the ARK runs on one M3. For all platforms that satisfied our hardware requirement (§2.2), this board has detailed documentation from vendor which ease our development. Also, its community has long-time kernel support since 2011, which allows us to study kernel ABI evolution. As Cortex-M3 on the platform is incapable of DVFS, for fair comparison, we run both cores at their highest clock rates.

Test setup We test ARK with Linux kernel v4.4. We tune the kernel configuration so that suspend/resume operates nine devices, as shown in Table 7.1. The corresponding drivers exercise various kernel services to verify the functionality of ARK design.

We measure device suspend/resume executed by ARK on Cortex-M3 and report the results. We compare ARK to native Linux execution on Cortex-A9. We fur-

Table 7.1.
I/O devices description and kernel services used

Device Name	Decription	Interface	Services*
SD Card	SanDisk Ultra 16GB SDHC Class 10	SDIO	2,5
Flash drive	Generic 256MB thumb drive	USB	1-2,4
MMC controller	OMAP HSMMC host controller	On chip	1-2
USB controller	OMAP HS multiport USB host controller	On chip	1-2
Regulator	TI TWL6030 PMIC	I2C	5
Keyboard	Dell KB212B keyboard	USB	1-2,4
Camera	Logitech C270	USB	1-2,4
Bluetooth	Broadcom BCM20702	USB	1-2,4
WiFi	TI WL1251	SDIO	2-6

*1. deferred work 2. memory allocator 3. softirq 4. DMA 5. threaded IRQ 6. firmware upload

Table 7.2.
Source code

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

ther compare to a baseline version, which inherits infrastructure from QEMU with a straightforward implementation of translation rules from ARMv7-A to ARMv7-M. The baseline version lacks the optimizations described in Chapter 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 the tedious Linux kernel transplant (§2.3). In our evaluation, ARK transparently translate and execute 15K SloC kernel code, mostly from drivers and driver libraries, as shown in Table 7.2. In addition, ARK is able to execute other drivers in ARMv7-A Linux kernel without engineering effort to porting them to the peripheral core.

Table 7.2 also shows that ARK does not require too much engineering efforts in developing new software for a microcontroller-like core. Compared to a commodity DBT codebase (2M SLoC in QEMU), ARK's 9K new SLoC for new translation rules and optimizations is only a small fraction. By adding 1K new SLoC to implement emulated services, ARK avoids make DBT support sophisticated Linux kernel services (25K SLoC) which has been shown challenging in prior works [61, 63] The result validates our principle of specializing these emulated services.

Our code analysis shows that 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 `omap2plus_defconfig` 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 4.17 (most recent at the time of writing). This is because ARK only depends on a narrow ABI shown in Table 4.1; the ABI has not changed since Linux 3.16.

7.3 Measured execution characteristics

Core activity We trace core activities during ARK execution and native execution by instrumentation. Figure 7.1 (a) shows the breakdown of execution time. ARK shows the same amount of idle time but consumes longer time on busy execution compared with native execution. This is due to clock frequency ratio (M3's clock frequency is 1/6 of A9's)and ARK's execution overhead (both software and architecture difference). Baseline design incurs longer busy execution time due to DBT overhead. Although the extended busy time, ARK still demonstrate energy benefit.

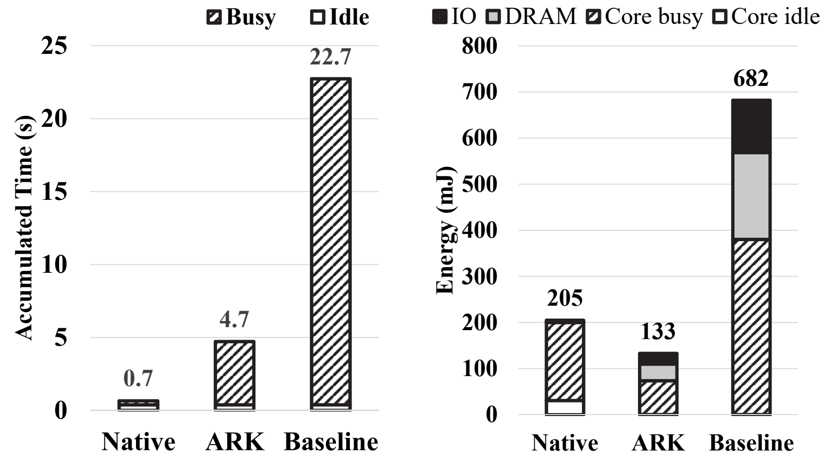


Fig. 7.1. Measured execution time and modeled energy in device suspend/resume. ARK substantially reduces the energy.

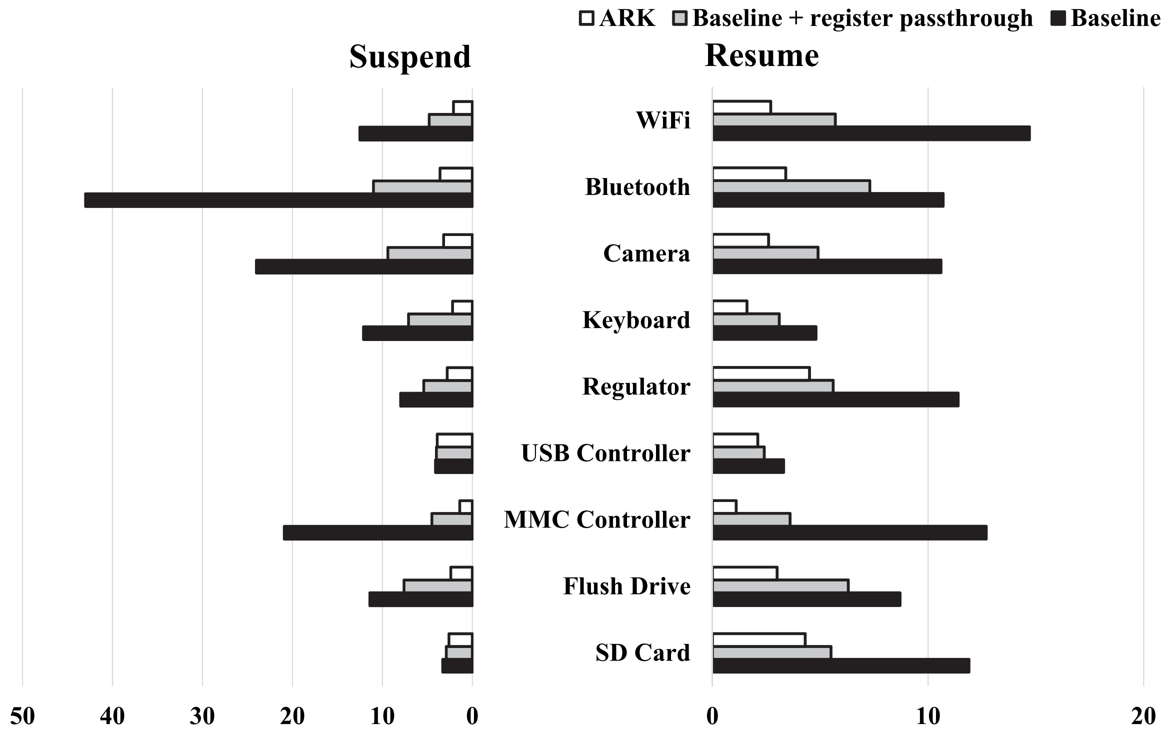


Fig. 7.2. Busy execution overhead for devices under test. Our DBT optimizations reduce the overhead by up to one order of magnitude

Memory activity We measure memory utilization by reading hardware counters of LPDDR controller. It reports memory requests for both read and write from Cortex-A9 or Cortex-M3 separately. By multiplying corresponding LLC size of two cores, we are able to calculate average memory utilization. As shown in Table 7.3, ARK on Cortex M3 surprisingly generates more memory traffic compared to the native execution on A9, even with longer execution time. We attribute this phenomenon to different LLC sizes of two cores. Throughout the experiment, the ARK emitted over 230KB instructions, which is far more than LLC capacity of M3. Furthermore, although touched kernel data and ARK emulated CPU structure are small, such temporal locality with small memory footprint does not benefit from M3's unified cache design. On the contrary, Cortex A9's larger LLC and split L1 cache absorbs more memory access. As shown below, the difference in memory utilization has a huge impact on energy consumption.

Busy execution overhead Our measurement shows that ARK incurs low overhead in busy kernel execution, as the result of DBT optimization and kernel service emulation. We calculate the overhead as the ratio between ARK's cycle count on Cortex-M3 to the Linux's cycle count on A9. Due to clockrate difference, a cycle on M3 is equivalent to 6 cycles on A9 in terms of time.

Overall, the execution overhead of ARK is 2.7× on average, with suspend overhead as 2.9× and resume overhead as 2.6×. Figure 7.2 shows for individual drivers the execution overhead, which ranges from 1.1× to 4.5×. Through comparing the execution overhead of different version, our DBT optimizations described in Chapter 5 demonstrate substantial performance improvement. For our baseline design, the average execution overhead is 13.9×, 5.2× higher than ARK. The overhead is further reduced by 2.5×, to 5.5× after applying register passthrough (§5.2) to the baseline. The remaining optimizations (e.g. control transfer) collectively reduce the overhead by additional 2×. We notice that our optimizations are less effective on drivers with

Table 7.3.
Test platform and power models in use

		CPU	Peripheral core
Processor	Core	2 * Cortex A9 @ 1.2GHz	2 * Cortex M3 @ 200MHz
	Arch	ARMv7-A	ARMv7-M
	Cache	L1:64KB + L2:1MB	L1:32KB
	Power	$P_{cpu_busy/idle}$:630mW/80mW [64]	$P_{pc_busy/idle}$:17mW/1mW [65]
DRAM	Model	Micron LPDDR2 [66]	
	Self Refresh	P_{mem_sr}:1.3mW	
	Utilization	8MB/s read, 4MB/s write	32MB/s read, 2MB/s write
	Active	P_{mem}:3.8mW	P'_{mem}8.4mW
IO	Power	P_{io}: 5mW [67]	

very dense control transfer (e.g. USB host controller) due to high DBT cost. As we will show below, the low overhead has a direct impact on energy saving.

Emulated services Our profiling shows that ARK's emulated services incur low overhead. Overall, the emulated services only contribute 1% of total busy execution. We evaluate the overhead of individual emulated services and summarize as below. i) The early, ISA-specific interrupt handling (§4.2) takes 3.9K Cortex-M3 cycles, only 1.5–2× more cycles than the native execution. ii) Emulated workqueues (§4.3) incurs a delay of tens of thousands M3 cycles to manipulate queue structure. The delay is longer than the native execution but does not break the deferred execution semantics. iii) For migrating one DBT context to the CPU in fallback, ARK spends around 20 us on rewriting return addresses on stack to pointing to CPU kernel's addresses(§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 model the system power based on measured hardware activities as summarized in Section 7.3. We choose modeling because i) our test platform is a

development board, which is not optimized for energy efficiency at production level; ii) the board lacks separated power trace for DRAM [68]. We consider the platform power consumption as a function of core activities, DRAM utilization, and IO. As shown in Table 7.3, we use TI PET (power estimation tool) [64, 65] to model core power as a function of core activities; we use Micron's LPDDR2 power modeling spreadsheet [66] to model DRAM power as a function of DRAM self refresh ratio and read/write activities. These power tools are gathered from vendor's official websites. Based on prior work [67] we assume 5mW of average IO power over device suspend/resume, which reasonably approximation during kernel device suspend/resume phases.

- The system energy with native execution is given by:

  $E_{cpu} = T_{idle} \cdot (P_{cpu_idle} + P_{mem_sr} + P_{io}) + T_{busy} \cdot (P_{cpu_busy} + P_{mem} + P_{io})$

- The system energy of ARK is given by:

  $E_{ARK} = T_{idle} \cdot (P_{pc_idle} + P_{mem_sr} + P_{io}) + T_{busy} \cdot F \cdot C \cdot (P_{pc_busy} + P'_{mem} + P_{io})$

Here, all Ts are elapsed time measured in native execution on CPU. Ps are power consumption for cores and DRAM in different power states. The DRAM's active power P_{mem} is derived from measured memory utilization and power modeling tool. F captures the clockrate ratio between CPU and the peripheral core. C is the average measured overhead in busy execution, obtained by ratio of cycle count between ARK and native execution.

Energy saving ARK saves 34% of evergy compared with native execution on A9, even at the cost of longer execution. The energy breakdown in Figure 7.1(b) shows that the benefit comes from two portions. i) energy reduction in busy execution: due to its low overhead (on average 2.7×), the ARK's energy efficiency in busy execution is 23% higher than the native execution. ii) excellent idle energy efficiency of peripheral core: ARK reduces the system idle energy to a negligible portion, since the peripheral core's idle power is 1.25% of the CPU's (1mW vs 80mW). Figure 7.1 also shows that our DBT optimizations are crucial to energy benefit. Although the baseline also

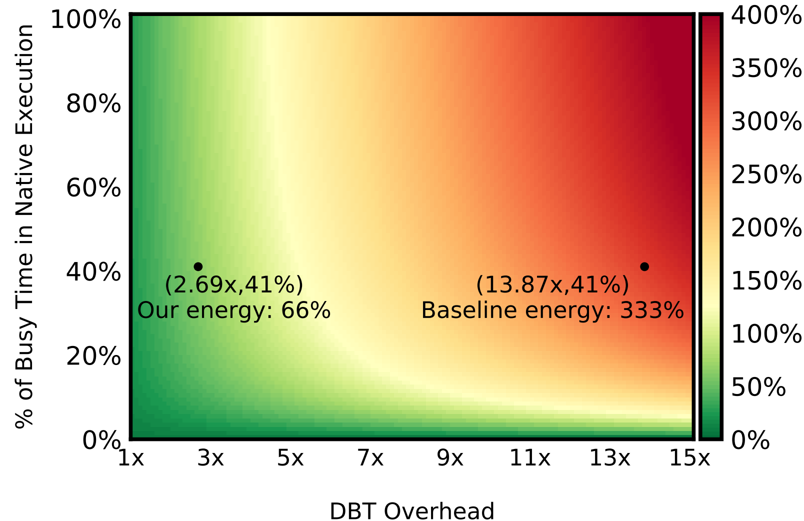


Fig. 7.3. System energy consumption of ARK relative to the native execution (100%), under different DBT overheads (x-axis) and busy execution fractions (y-axis). ARK's low energy hinges on low DBT overhead.

benefits from lower idle power, its high execution overhead ultimately leads to 5.1× energy compared to the native execution.

Another interesting fact that we find is ARK consumes *more* DRAM energy than the native execution, due to Cortex-M3's tiny LLC (32KB) as describe earlier. The LLC size trades off between the core power and the DRAM power. Our result suggests that the current size is suboptimal for the offloaded kernel execution. We recommend future hardware designers to carefully increase the LLC size (to 64 KB or 128 KB). It will significantly reduce DRAM power at the cost of moderate increase in core power.

What-if analysis We further study ARK's energy benefit by tuning a few parameters in previous model:

- The fraction of busy execution time in native execution on CPU (denoted by $T_{busy}/(T_{idle} + T_{busy})$), which is determined by workload's characteristic.
- The DBT overhead (denoted by C), which is affected by ARK.

Table 7.4.
Battery life extension under different suspend/resume intervals and energy ratio

	1:9	1:3	1:1
5s	18 %	15.5%	11%
30s	8.75 %	8.1%	6.7%

We estimate energy consumption by using the above power model and plugging in different values for the two factors. Our analysis results in in Figure 7.3 show two findings. i) ARK's energy benefit will be more noticeable when the kernel incurs less CPU busy time (i.e. more CPU idle). This is because ARK's energy efficiency advantage compared to CPU is higher during idle periods than during busy execution. ii) DBT's overhead has a huge impact on ARK's energy benefit. When the overhead drops to below 3.5×, ARK saves energy even for 100% kernel execution; when the overhead exceeds 5.2× which is the break-even point, ARK wastes energy even for 20% busy execution time. This is the lowest fraction observed on embedded platforms in prior work [10].

Qualitative comparison with big.LITTLE We compare ARK's energy saving with executing device suspend/resume only on the LITTLE core in big.LITTLE architecture. For fair comparison, we study recent papers and model LITTLE core's power consumption and execution efficiency as follows: It provides 1.3× energy efficiency at 70% of clock frequency compared with CPU [69]. The idle power consumption is 40mW [70]. We assume that the LITTLE core has the same memory utilization as CPU, considering LITTLE core's smaller LLC but longer execution period. We assume the same IO power. Under this model, we estimate that LITTLE core is only capable of save 23% of energy, which is less than ARK with 34%. We attribute this to huge idle power gap between LITTLE and the peripheral core (40mW vs 1mW). The busy execution efficiency is similar.

Battery life extension Based on ARK's energy savings in device suspend/resume, we estimate the battery life extension for executing ephemeral tasks using the similar settings as prior work [2]. As shown in Figure 7.4, ARK extends the battery life by up to 18% when a ephemeral task is executed every 5 seconds and device suspend/resume consumes 90 percent of total energy. The saving converts to extra 4.3 hours battery life per day, which is tangle compared with other approaches [2, 67].

8. RELATED WORK

OS for heterogeneous processors To harness heterogeneous processors, some multikernel OS designs [17,20,21,21,71] launch one kernel for each coherence domain, and coordinate through IPC or shared memory. However, they rely on agreement on kernel ABI which is fragile as described in Section 2.3. Unlike them, transkernel bridges the heterogeneity gap by using DBT. Some systems offload CPU kernel functionalities to accelerators [72,73]. But those accelerators cannot work while the CPU is off, which is the prerequisite for ARK to save energy.

DBT DBT has been used for system emulation [53] and binary instrumentation [61, 63, 74, 75]; DeVuyst et al. [76] uses DBT to accelerate process migration across heterogeneous cores. Related to transkernel, prior systems run translated user programs atop an emulated syscall interface [52,53,77]. 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 to exploit instruction level parallelism, and hence gain efficiency [78–81]. transkernel demonstrates that DBT can gain efficiency without introducing new hardware. Existing DBT engines leverage ISA similarities, e.g. between aarch32 and aarch64 [51, 82]. They still fall into the classic DBT paradigm, where the host ISA is brawny and the guest ISA is wimpy. With an inverse DBT paradigm, ARK addresses very different challenges. Much work is done on optimizing cross-ISA DBT translation by building translation rules using automatic machine learning [83], or by applying compiler techniques on resultant instructions, such as LLVM optimizer [84] and peephole optimization [85]. Compared to them, ARK leverages ISA similarities and therefore reuses code optimization already present in the guest code by the guest compilers.

Kernel and drivers Prior kernel studies show rapid evolution of the Linux kernel and the interfaces between kernel layers are unstable [25, 86]. This observation motivates transkernel. Extensive work transplants device drivers to a separate core [44], user space [45], or a separate VM [87]. However, the transplant code cannot operate independent of the kernel, whereas transkernel must execute autonomously.

Suspend/resume Energy inefficiency in suspend/resume raises attention for cloud servers [28, 88] and mobile [2]. Drowsy [2] mitigates the problem by introducing a new power state to minimize the number of devices involved during suspend/resume. It requires recompilation of the Linux kernel. Xi et al. [88] propose to reorder the resume sequence of devices. Our approach does not require changes to original kernel, and is complementary to them. PowerNap [28] expedites suspend/resume by exploiting hardware power state for servers. However, its model only operates on limited number of devices, which contrast to embedded platform with diverse IO. A kernel may put idle devices to low power modes at run time [67], which is complementary to suspend/resume that ensures all devices are off.

9. CONCLUSIONS

In this thesis, we present transkernel, a new model to execute kernel device suspend/resume on a peripheral core. It adopts cross-ISA DBT which creates a virtualized environment on a microcontroller-like core. To overcome hardware heterogeneity at an affordable cost, transkernel follows four principles: it translates stateful code while emulating stateless services; it chooses a narrow and stable interface for emulation; it specialized for frequently executed path; it exploits ISA similarities for DBT overhead reduction. Through experiment and analysis, we demonstrate that this approach is feasible and has tangible energy savings. Moreover, the transkernel provides a new OS design for heterogeneous SoCs.

REFERENCES

REFERENCES

[1] R. Liu and F. X. Lin, "Understanding the characteristics of android wear os," in *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '16. New York, NY, USA: ACM, 2016, pp. 151–164. [Online]. Available: http://doi.acm.org/10.1145/2906388.2906398

[2] M. Lentz, J. Litton, and B. Bhattacharjee, "Drowsy power management," in *Proceedings of the 25th Symposium on Operating Systems Principles*, ser. SOSP '15. New York, NY, USA: ACM, 2015, pp. 230–244. [Online]. Available: http://doi.acm.org/10.1145/2815400.2815414

[3] 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, pp. 55–68.

[4] 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 *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '15. New York, NY, USA: ACM, 2015, pp. 40–52. [Online]. Available: http://doi.acm.org/10.1145/2789168.2790107

[5] X. Chen, N. Ding, A. Jindal, Y. C. Hu, M. Gupta, and R. Vannithamby, "Smartphone energy drain in the wild: Analysis and implications," *ACM SIGMETRICS Performance Evaluation Review*, vol. 43, no. 1, pp. 151–164, 2015.

[6] 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*, ser. MobiSys '17. New York, NY, USA: ACM, 2017, pp. 385–398. [Online]. Available: http://doi.acm.org/10.1145/3081333.3081351

[7] 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 *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017, pp. 515–529. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/vasisht

[8] R. Liu and F. X. Lin, "Understanding the characteristics of android wear os," in *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '16. New York, NY, USA: ACM, 2016, pp. 151–164. [Online]. Available: http://doi.acm.org/10.1145/2906388.2906398

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

[10] S. Zhai, L. Guo, X. Li, and F. X. Lin, "Decelerating suspend and resume in operating systems," in *Proceedings of the 18th International Workshop on Mobile Computing Systems and Applications*, ser. HotMobile '17. New York, NY, USA: ACM, 2017, pp. 31–36. [Online]. Available: http://doi.acm.org/10.1145/3032970.3032975

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

[12] LKML, "[git pull] pm updates for 2.6.33," 2009.

[13] MediaTek, "Microsoft azure sphere mcu with extensive i/o peripheral subsystem for diverse iot applications," https://www.mediatek.com/products/azureSphere/mt3620, 2018.

[14] NXP Semiconductors, "i.mx 7dual family of applications processors datasheet," https://www.nxp.com/docs/en/data-sheet/IMX7DCEC.pdf, 2017.

[15] Apple, "Apple motion coprocessor," https://en.wikipedia.org/wiki/Apple_motion_coprocessors.

[16] 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)*. New York, NY, USA: ACM, 2012, pp. 13–24. [Online]. Available: http://doi.acm.org/10.1145/2150976.2150979

[17] 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)*. ACM, 2014, pp. 285–300.

[18] D. Meisner and T. F. Wenisch, "Dreamweaver: architectural support for deep sleep," in *Proceedings of the 17th international conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '12. New York, NY, USA: ACM, 2012, pp. 313–324. [Online]. Available: http://doi.acm.org/10.1145/2150976.2151009

[19] 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)*. Berkeley, CA, USA: USENIX Association, 2009, pp. 365–380.

[20] 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 *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys '15. New York, NY, USA: ACM, 2015, pp. 29:1–29:16. [Online]. Available: http://doi.acm.org/10.1145/2741948.2741962

[21] 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)*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 221–234. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629597

[22] 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)*. ACM, 2009, pp. 29–44.

[23] A. L. Brown and R. J. Wysocki, "Suspend-to-ram in linux," in *Ottawa Linux Symposium*, 2008, pp. 39–52.

[24] Intel, "Intel suspendresume project," https://01.org/suspendresume, 2015.

[25] Y. Padioleau, J. L. Lawall, and G. Muller, "Understanding collateral evolution in linux device drivers," in *ACM SIGOPS Operating Systems Review*, vol. 40, no. 4. ACM, 2006, pp. 59–71.

[26] A. Kadav and M. M. Swift, "Understanding modern device drivers," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII. New York, NY, USA: ACM, 2012, pp. 87–98. [Online]. Available: http://doi.acm.org/10.1145/2150976.2150987

[27] 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 *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*. USENIX Association, 2017, pp. 1–13.

[28] D. Meisner, B. T. Gold, and T. F. Wenisch, "Powernap: Eliminating server idle power," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIV. New York, NY, USA: ACM, 2009, pp. 205–216. [Online]. Available: http://doi.acm.org/10.1145/1508244.1508269

[29] Q. Zhu, M. Zhu, B. Wu, X. Shen, K. Shen, and Z. Wang, "Software engagement with sleeping cpus," in *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. Kartause Ittingen, Switzerland: USENIX Association, May 2015. [Online]. Available: https://www.usenix.org/conference/hotos15/workshop-program/presentation/zhu

[30] Texas Instruments, "OMAP4460 technical reference manual," http://www.ti.com/lit/ug/swpu235ab/swpu235ab.pdf, 2014.

[31] ——, "OMAP572x: Technical reference manual," http://www.ti.com/lit/ug/spruhz6k/spruhz6k.pdf, 2018.

[32] NXP Semiconductors, "Vybrid vf6xx family," https://www.nxp.com/docs/en/fact-sheet/VYBRIDVF6FS.pdf, 2014.

[33] Samsung, "Exynos 4210 application processor," http://www.samsung.com/global/business/semiconductor/product/application/detail?productId=7644&iaId=844, 2012.

[34] 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)*. ACM, 2005, pp. 261–274.

[35] H. Shen, A. Balasubramanian, A. LaMarca, and D. Wetherall, "Enhancing mobile apps to use sensor hubs without programmer effort," in *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, ser. UbiComp '15. New York, NY, USA: ACM, 2015, pp. 227–238. [Online]. Available: http://doi.acm.org/10.1145/2750858.2804260

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

[37] J. Mogul, J. Mudigonda, N. Binkert, P. Ranganathan, and V. Talwar, "Using asymmetric single-isa cmps to save energy on operating systems," *Micro, IEEE*, vol. 28, no. 3, pp. 26–41, 2008.

[38] P. Greenhalgh, "Big.LITTLE processing with ARM Cortex-A15 and Cortex-A7," Tech. Rep., 2011.

[39] NXP Semiconductors, "i.mx 6dual/6quad applications processors for industrial products," https://www.nxp.com/docs/en/data-sheet/IMX7DCEC.pdf, 2017.

[40] Renesas, "R-car h3," https://www.renesas.com/en-us/solutions/automotive/products/rcar-h3.html, 2018.

[41] 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)*, vol. 33, no. 1, p. 3, 2015.

[42] T. Li, P. Brett, R. Knauerhase, D. Koufaty, D. Reddy, and S. Hahn, "Operating system support for overlapping-isa heterogeneous multi-core architectures," in *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*. IEEE, 2010, pp. 1–12.

[43] A. Ponomarenko, "Abi compliance checker," https://lvc.github.io/abi-compliance-checker/, 2018.

[44] B. Gerofi, A. Santogidis, D. Martinet, and Y. Ishikawa, "Picodriver: Fast-path device drivers for multi-kernel operating systems," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '18. New York, NY, USA: ACM, 2018, pp. 2–13. [Online]. Available: http://doi.acm.org/10.1145/3208040.3208060

[45] V. Ganapathy, M. J. Renzelmann, A. Balakrishnan, M. M. Swift, and S. Jha, "The design and implementation of microdrivers," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIII. New York, NY, USA: ACM, 2008, pp. 168–178. [Online]. Available: http://doi.acm.org/10.1145/1346281.1346303

[46] M. M. Swift, B. N. Bershad, and H. M. Levy, "Improving the Reliability of Commodity Operating Systems," in *Proc. ACM SOSP*, 2003.

[47] S. Boyd-Wickizer and N. Zeldovich, "Tolerating malicious device drivers in linux." in *USENIX Annual Technical Conference*. Boston, 2010.

[48] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy, "Recovering Device Drivers," in *Proc. USENIX OSDI*, 2004.

[49] 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.

[50] A. Barbalace, R. Lyerly, C. Jelesnianski, A. Carno, H.-R. Chuang, V. Legout, and B. Ravindran, "Breaking the boundaries in heterogeneous-isa datacenters," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2017, pp. 645–659.

[51] A. d'Antras, C. Gorgovan, J. Garside, and M. Luján, "Low overhead dynamic binary translation on arm," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017. New York, NY, USA: ACM, 2017, pp. 333–346. [Online]. Available: http://doi.acm.org/10.1145/3062341.3062371

[52] W. Wang, P.-C. Yew, A. Zhai, S. McCamant, Y. Wu, and J. Bobba, "Enabling cross-isa offloading for cots binaries," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2017, pp. 319–331.

[53] F. Bellard, "Qemu, a fast and portable dynamic translator." in *USENIX Annual Technical Conference, FREENIX Track*, 2005, pp. 41–46.

[54] 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)*, ser. SOSP '95. New York, NY, USA: ACM, 1995, pp. 251–266. [Online]. Available: http://doi.acm.org/10.1145/224056.224076

[55] 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)*, ser. ASPLOS XVI. New York, NY, USA: ACM, 2011, pp. 291–304. [Online]. Available: http://doi.acm.org/10.1145/1950365.1950399

[56] 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)*. ACM, 2013, pp. 461–472.

[57] Mike Turquette, "The common clk framework," https://www.kernel.org/doc/Documentation/clk.txt.

[58] ARM, "ARM architecture reference manual: Armv7-a and armv7-r edition," https://static.docs.arm.com/ddi0406/c/DDI0406C_C_arm_architecture_reference_manual.pdf, 2014.

[59] ——, "Armv7-m architecture reference manual," https://static.docs.arm.com/ddi0403/eb/DDI0403E_B_armv7m_arm.pdf, 2014.

[60] W. Wang, S. McCamant, A. Zhai, and P.-C. Yew, "Enhancing cross-isa dbt through automatically learned translation rules," in *Proceedings of the Twenty-Third International Conference on Architectural Support*

for Programming Languages and Operating Systems, ser. ASPLOS '18. New York, NY, USA: ACM, 2018, pp. 84–97. [Online]. Available: http://doi.acm.org/10.1145/3173162.3177160

[61] P. Kedia and S. Bansal, "Fast dynamic binary translation for the kernel," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: ACM, 2013, pp. 101–115. [Online]. Available: http://doi.acm.org/10.1145/2517349.2522718

[62] VMWARE, "Virtual machine to physical machine migration," https://www.vmware.com/support/v2p/doc/V2P_TechNote.pdf, 2004.

[63] P. Feiner, A. D. Brown, and A. Goel, "Comprehensive kernel instrumentation via dynamic binary translation," in *ACM SIGARCH Computer Architecture News*, vol. 40, no. 1. ACM, 2012, pp. 135–146.

[64] Texas Instruments, "Am43xx power estimation tool," http://www.ti.com/lit/an/spraca3/spraca3.pdf, 2017.

[65] ——, "Am572x power estimation tool," http://www.ti.com/lit/an/spraca0/spraca0.pdf, 2018.

[66] Micron Technology, Inc., "Tn4201 lpddr2 system power calculator," https://www.micron.com/support/tools-and-utilities/power-calc, 2013.

[67] 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)*. New York, NY, USA: ACM, 2015.

[68] eLinux.org, "PandaBoard Power Measurements," http://elinux.org/PandaBoard_Power_Measurements.

[69] 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.*, vol. 8, no. 7, pp. 762–773, Feb. 2015. [Online]. Available: http://dx.doi.org/10.14778/2752939.2752945

[70] N. Peters, S. Park, S. Chakraborty, B. Meurer, H. Payer, and D. Clifford, "Web browser workload characterization for power management on hmp platforms," in *2016 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Oct 2016, pp. 1–10.

[71] N. Asmussen, M. Völp, B. Nöthen, H. Härtig, and G. P. Fettweis, "M3: A hardware/operating-system co-design to tame heterogeneous manycores," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016*, 2016, pp. 189–203. [Online]. Available: http://doi.acm.org/10.1145/2872362.2872371

[72] 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 *Proceedings of the Thirteenth EuroSys Conference*. ACM, 2018, p. 36.

[73] 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)*, ser. ASPLOS '13. New York, NY, USA: ACM, 2013, pp. 485–498. [Online]. Available: http://doi.acm.org/10.1145/2451116.2451169

[74] B. Hawkins, B. Demsky, D. Bruening, and Q. Zhao, "Optimizing binary translation of dynamically generated code," in *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 68–78. [Online]. Available: http://dl.acm.org/citation.cfm?id=2738600.2738610

[75] 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. of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation.* New York, NY, USA: ACM Press, 2005, pp. 190–200.

[76] 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).* New York, NY, USA: ACM, 2012, pp. 261–272. [Online]. Available: http://doi.acm.org/10.1145/2150976.2151004

[77] R. J. Hookway and M. A. Herdeg, "Digital fx! 32: Combining emulation and binary translation," *Digital Technical Journal*, vol. 9, pp. 3–12, 1997.

[78] D. Boggs, G. Brown, N. Tuck, and K. S. Venkatraman, "Denver: Nvidia's first 64-bit ARM processor," *IEEE Micro*, vol. 35, no. 2, pp. 46–55, 2015. [Online]. Available: https://doi.org/10.1109/MM.2015.12

[79] A. Klaiber, "The technology behind crusoe processors," *Transmeta Technical Brief*, 2000.

[80] S. Rokicki, E. Rohou, and S. Derrien, "Hardware-accelerated dynamic binary translation," in *Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017*, 2017, pp. 1062–1067. [Online]. Available: https://doi.org/10.23919/DATE.2017.7927147

[81] ——, "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, pp. 1009–1014. [Online]. Available: https://doi.org/10.23919/DATE.2018.8342160

[82] A. d'Antras, C. Gorgovan, J. Garside, J. Goodacre, and M. Luján, "Hypermambo-x64: Using virtualization to support high-performance transparent binary translation," in *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '17. New York, NY, USA: ACM, 2017, pp. 228–241. [Online]. Available: http://doi.acm.org/10.1145/3050748.3050756

[83] W. Wang, S. McCamant, A. Zhai, and P. Yew, "Enhancing cross-isa DBT through automatically learned translation rules," in *Proceedings of the Twenty-Third International Conference on Architectural Support for*

Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018, 2018, pp. 84–97. [Online]. Available: http://doi.acm.org/10.1145/3173162.3177160

[84] D.-Y. Hong, C.-C. Hsu, P.-C. Yew, J.-J. Wu, W.-C. Hsu, P. Liu, C.-M. Wang, and Y.-C. Chung, "Hqemu: A multi-threaded and retargetable dynamic binary translator on multicores," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, ser. CGO '12. New York, NY, USA: ACM, 2012, pp. 104–113. [Online]. Available: http://doi.acm.org/10.1145/2259016.2259030

[85] S. Bansal and A. Aiken, "Binary translation using peephole superoptimizers," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*. USENIX Association, 2008, pp. 177–192.

[86] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller, "Documenting and automating collateral evolutions in linux device drivers," in *Acm sigops operating systems review*, vol. 42, no. 4. ACM, 2008, pp. 247–260.

[87] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz, "Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines," in *Proc. USENIX OSDI*, 2004.

[88] S. L. Xi, M. Guevara, J. Nelson, P. Pensabene, and B. C. Lee, "Understanding the critical path in power state transition latencies," in *Proceedings of the 2013 International Symposium on Low Power Electronics and Design*, ser. ISLPED '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 317–322. [Online]. Available: http://dl.acm.org/citation.cfm?id=2648668.2648746