Decelerating Suspend and Resume in Operating Systems

Shuang Zhai, Liwei Guo, Xiangyu Li, and Felix Xiaozhu Lin
Purdue ECE

ABSTRACT

Short-lived tasks have a large impact on mobile computer's battery life. In executing such tasks, the whole system transitions in and out of the deep sleep mode. This suspend/resume procedure is controlled by the operating system (OS), which consumes a dominating portion of energy. Through characterizing the Linux kernel on a variety of modern system-on-chips (SoCs), we show that the OS suspend/resume mechanism is fundamentally slowed down by various IO devices, which frequently keep CPU waiting.

To minimize energy consumption, we advocate offloading the OS suspend/resume to a miniature processor that waits more efficiently. To this end, we propose a new virtual executor that runs on a miniature core and directly executes the unmodified kernel binary of the main CPU. We construct the virtual executor centering on software-only, cross-ISA binary translation, an approach previously considered prohibitively expensive. Through novel designs and optimizations, we reduce the translation overhead by 5×. The preliminary benchmarks show promising energy efficiency.

CCS Concepts

•**Software and its engineering** → **Operating systems;**

1. INTRODUCTION

Today's mobile and wearable computers see a large number of intermittent, short-lived tasks such as push notification [3], email sync [21], and "always-on" UI [7]. The short-lived tasks, as shown in recent work, drain a large portion of battery, e.g. 29% on smartphones [3]. To execute a short-lived task, the whole system exits from a deep-sleep state, runs user code, and re-enters the deep-sleep state. The power state transitions are performed by suspend/resume, a core power management (PM) function in OS.

Ironically, despite critical to system energy efficiency, the suspend/resume procedure itself is expensive. In running a short-lived task, suspend/resume often dominates the energy consumption, sometimes incurring 10× higher energy than the task's user code execution [5] . Most of the energy is consumed by CPU, since short-lived tasks are often driven by background activities and are executed with screen off.

To pinpoint the bottlenecks, we profile the Linux suspend/resume on a variety of mobile SoCs. Our findings indicate that suspend/resume is fundamentally slowed down by IO: the CPU spends much of its time waiting (being not only idle but also busy) for hundreds of IO devices to complete their power state transitions. Unfortunately, shortening such transitions is difficult: the transition delay is often bound by physical factors or interface standards; the slow IO devices are diverse and platform-dependent; transitions of multiple IO devices can hardly be parallelized.

The profiling suggests that suspend/resume poorly fits today's high-frequency, complex processors that are intended for rich mobile applications. Instead, the OS suspend/resume should be offloaded to miniature, low-power processors which we dubbed "PM cores". This goal, however, is challenged by the high complexity of the OS suspend/resume code, the PM core's instruction set architecture (ISA), and the demand for remaining compatible with commodity OSes.

To this end, we present a novel virtual executor: running on a PM core, it completely takes charge of suspend/resume by directly executing the main CPU's unmodified kernel binary. This greatly reduces energy cost by keeping the main CPU powered off for OS suspend/resume.

To fit a weak PM core, our virtual executor is purely software-based, an approach previously believed too expensive. To make it practical, we *specialize* the virtual executor for the kernel suspend/resume path, rather than supporting generic kernel code. The virtual executor consists of two key components: a translator that dynamically translates the main CPU's kernel binary and executes it; a function pruner that replaces generic, expensive OS functions with specialized, simplified versions.

Of the two components, the binary translator's overhead is critical to the entire system's efficiency – its software-based cross-ISA translation, according to conventional wisdom, incurs prohibitive overhead. To tackle this problem, our insight is to exploit the *similarity* between heterogeneous ISAs, which are commonly seen among co-located modern processors, as exemplified by the ARM A-series and M-series cores. Through a set of key optimizations, we reduce the translation overhead by 5× as compared to the state of the art. Based on the measured power and overhead, we estimate to reduce the total energy cost of suspend/resume by up to 80% and extend the battery life by 30%.

We have made the following contributions:

• We quantify the Linux suspend/resume procedure on a variety of SoCs and examine the causes of high energy consumption and long delay;

• We offload the OS suspend/resume to an extremely low-power core, which runs a novel virtual executor for executing the unmodified kernel binary with low overhead;

• We describe a first-of-its-kind working prototype of the virtual executor; at the time of writing, the prototype contains 50.5K SLoC, of which 4.5K are new[1]. The preliminary

[1]Generated using David A. Wheeler's 'SLOCCount'.

HotMobile '17, February 21-22, 2017, Sonoma, CA, USA

DOI: http://dx.doi.org/10.1145/3032970.3032975

results show great promise.

2. BACKGROUND & MOTIVATION

System suspend/resume Suspend/resume is a core functionaltiy of OS power management (PM). In a nutshell, a suspend procedure is often initiated by userspace. The OS synchronizes file systems, freezes all user processes, turns off individual IO devices, and finally powers off CPU cores. Resume is a mirror procedure. Suspend/resume is complicated and platform-specific; kernel invokes various subsystems including those managing clocks, buses, power regulators, and IO devices. The procedure happens in multiple iterations, each taking multiple sequential stages.

As a rough estimation based on Linux 4.4, the suspend/resume execution involves more than 30K SLoC: 10K in the framework itself, 10K in clock, 10K in platform PM code, and several thousand lines scattered in ∼100 device drivers that are rapidly evolving.

OS must be responsible Suspend/resume has heavy functional dependency on the OS infrastructure. First, the procedure relies on the OS knowledge, e.g., to properly save/restore the IO device state. Second, it invokes other OS services through a wide, internal interface. For instance, suspend/resume may request memory pages from the kernel page allocator or schedule a tasklet to be executed in the future. As a result, the suspend/resume code is tightly woven into the whole kernel source.

PM "co-processor" A great portion of low-level PM functionalities, e.g. clock management, used to be implemented in hardware. As modern PM becomes increasingly complex, there has been a trend moving them to software atop a dedicated PM co-processor. Examples include the `wkup` core in TI AM335x [18] and the power management controller in NVIDIA Tegra [12].

Existing PM co-processors, however, cannot fulfil the OS's role of performing suspend/resume. As mentioned above, suspend/resume heavily depends on the OS knowledge and services, which are unavailable to the co-processors.

Heterogeneous and incoherent cores In pursuit of energy proportionality, modern SoCs often embrace heterogeneous cores. For programming ease, ARM big.LITTLE enforces same ISA and coherent caches across these cores, which, however, limits the core heterogeneity. After removing a unified ISA and hardware cache coherence [16, 17, 13], the hardware vendor has great flexibility in incorporating disparate cores aggressively optimized for different objectives, e.g., performance or efficiency. To harness the resultant architecture for general-purpose programming, prior work proposed novel OS structures [11, 6, 1]. Much work focused on providing a convenient single system image, however at the cost of engineering and compatiblity difficulty. We will discuss them in Section 4.

3. A CASE FOR OFFLOADING

We next show why it is compelling to "decelerate" suspend/resume by executing it on a less performant yet much more efficient processor. To do so, we present our characterization of the current suspend/resume on a variety of SoCs from major vendors, as listed in Figure 1. For each platform, we study the latest vendor-released Linux kernel which is often thoroughly tuned and has production quality.

3.1 Energy "hotspots" in short-lived tasks

A short-lived task in modern mobile systems often lasts from hundreds of ms [7, 5] to a few seconds [21]. Of the total energy consumed, the OS suspend/resume constitutes a significant, sometimes dominating, portion, up to 90% [5]. The combined delay of suspend/resume is also long: as shown in Figure 1, it ranges from a few hundred ms to nearly one sec.

Fortunately, in most scenarios, the delay in suspend/resume does not directly affect user experience: as prior work discovered [3], most short-lived tasks are initiated by background activities when user is not paying attention; even when the user initiates a short-lived task (e.g. checking the smartwatch face), she may only perceive the resume delay but not the suspend delay.

Implication For common suspend/resume scenarios where user experience is unaffected, minimizing energy consumption is the primary concern; performance may be traded for lower energy.

3.2 IO power state transition is slow

A defining feature of the suspend/resume path is that CPU waits – being either idle or busy – for a large number of IO devices to finish their power state transitions. Because of the IO wait, the suspend/resume bogs down, as shown in Figure 1. It is worth noting that the wait happens in numerous small episodes; across all platforms, hundreds of IO devices (of which Figure 1 only itemizes top ones) contribute to the delay.

These transitions are fundamentally slow, bound by slow peripherals (e.g. flash storage), low-speed interconnects (e.g. I2C), wimpy microcontrollers embedded in the IO devices, or physical factors (e.g. voltage ramp up). This is exemplified by the following cases that we have manually discovered.

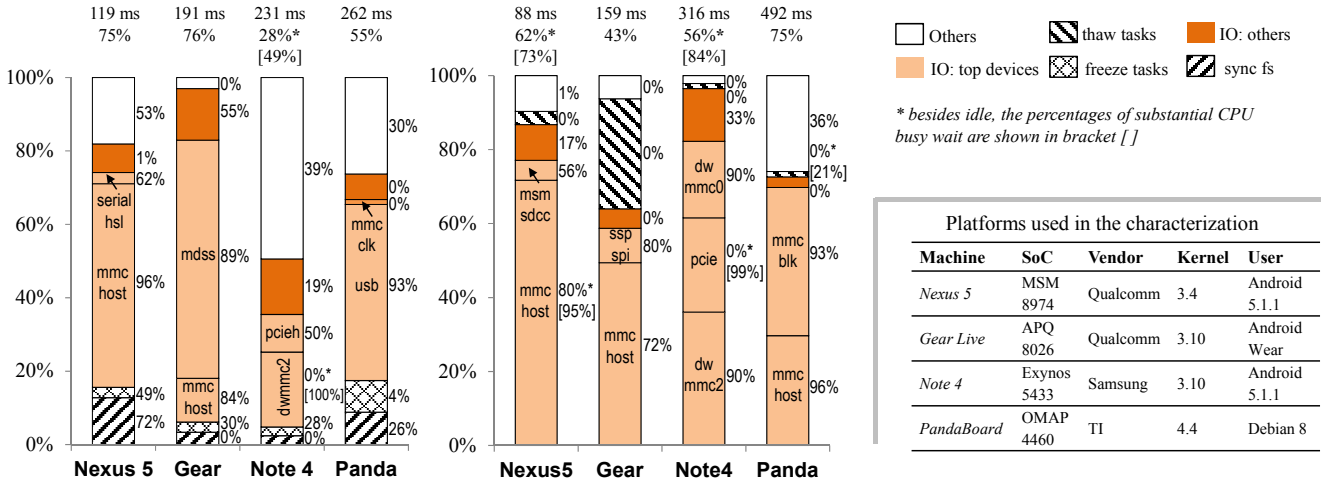
• *MMC host.* MMC host implements hardware protocols for interacting with storage devices such as eMMC. In powering on/off the storage devices, CPU waits as mandated by the MMC protocol. For example, when the host commands eMMC to sleep, the latter often needs extra time to ensure data become persistent; the kernel must wait synchronously for either a completion notification or timeout.

• *Display controller (MDSS).* Before powering off the display controller, the kernel must ensure no image frame is being transmitted to the display panel in order to avoid persistent visual artifacts. In order to do so, the kernel must wait for the completion of the current frame transmission, which often takes tens of ms.

• *SD card.* To resume an SD card, the kernel switches on the voltage supply and waits for the voltage to ramp up and stabilize – before any further operations can be done to the card. The voltage ramp-up may take up to tens of ms.

• *Battery fuel gauge.* Prior to powering off the gauge, the kernel needs to retrieve the last temperature and current readings over I2C. The gauge, likely due to its low-power nature, often takes tens of ms to respond. When the system runs normally, the sensors are read in the background and the latency is hidden; however, suspend/resume requires the reading to happen synchronously, exposing the latency.

Slow IO devices are diverse As reflected in our measurement, multiple IO device are slowing down the entire procedure. For each SoC, there exist a couple of dominating IO devices, and the remaining (hundreds of) devices form a



(a) Suspend (b) Resume

Platforms used in the characterization

Machine	SoC	Vendor	Kernel	User
Nexus 5	MSM 8974	Qualcomm	3.4	Android 5.1.1
Gear Live	APQ 8026	Qualcomm	3.10	Android Wear
Note 4	Exynos 5433	Samsung	3.10	Android 5.1.1
PandaBoard	OMAP 4460	TI	4.4	Debian 8

Figure 1: A characterization of suspend/resume delays, normalized to 100%. The absolute delays and the percentage of the time when all CPU cores are idle are on top. For each component, the idle percentage is shown to its right.

	Busy @ 1200MHz	Busy @ 350MHz	Idle
Cortex-A9	672	79.8	25.2
	Busy @ 200MHz	**Busy @ 58.3MHz**	**Idle**
Cortex-M3	21.1	2.5*	3.8

Both cores consume less than 0.1 mW when inactive.

Table 1: Power of the heterogeneous cores on OMAP4, in mW. All numbers are measured except that the one with (*) is estimated based on the power scaling ratio of A9.

long tail. Across SoCs, the IO devices that contribute most delays are not the same. In short, there is no silver bullet for addressing slow IO power state transition.

Asynchronous PM is not panacea One may consider overlapping the power state transitions of multiple IO devices. The major hurdle, however, is the *implicit* hardware dependency among IO devices, which requires certain transitions to happen sequentially.

Because of this, the Linux kernel community has been very conservative after a long debate [9, 8]. So far, asynchrony exists mostly among same-type devices, e.g. multiple SATA drives, for which a single driver has sufficient knowledge to orchestrate asynchronous PM. Such "intra-driver" asynchrony, however, is less useful to an IO-rich or accelerator-rich system which can have hundreds of different devices.

Implication The suspend/resume procedure is slow because CPU has to wait for various IO. Since the lengths of IO wait are indifferent to CPU performance, a simpler, less-performant core will consume much less energy due to its lower idle and active power. See Table 1 for quantitative evidence.

3.3 Kernel execution favors weak cores

Beyond IO wait periods, it has been known that kernel execution is more energy efficient on simpler and weaker cores [10]. This is because the kernel's unique characteristics, such as small code working set, less predictable control flow, and frequent IO register access, can hardly benefit from advanced, energy-hungry microarchitectures. Our kernel microbenchmark partially confirms this argument: in executing a CPU-intensive benchmark on a big core (Cortex-A9) and a weak core (Cortex-M3), their respective cycle counts often differ by less than 2× while their energy consumptions differ by more than 5×.

Implication In the suspend/resume procedure, the kernel execution outside of IO wait periods is likely to gain energy efficiency from weak cores. To us, the gained efficiency can be used as headroom to enable virtual execution that bridges heterogeneous ISAs, as will be discussed below.

4. DESIGN OVERVIEW

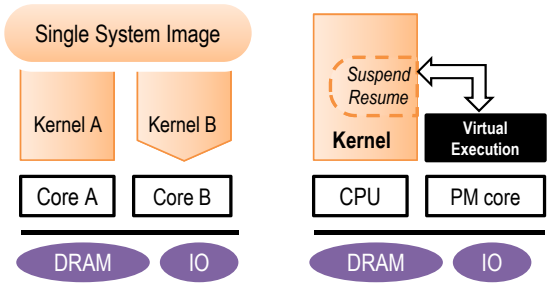
We have shown that suspend/resume is an energy-critical procedure where weak cores naturally fit – in both idling and busy execution. Taking this insight to its extreme, we advocate removing the main CPU from the kernel suspend/resume path, and offloading the procedure to a miniature processor which we dubbed "PM core".

PM core Conceptually, the PM core should be i) aggressively optimized for energy and ii) loosely coupled with the main CPU. We thus impose minimum hardware requirements: the PM core should reside on the same platform as the CPU and share access to DRAM and IO devices; its ISA should be *similar*, but does not have to be identical, to the main CPU. Other than that, the PM core may have no MMU or cache that is coherent with the main CPU.

Such minimum requirements qualify many existing low-power processors, such as ARM Cortex-M and Intel Quark. Their power-optimized ISAs are similar, although not identical, to their performance-optimized counterparts, such as ARM Cortex-A and Intel i7. They are already conveniently incorporated in commodity hardware platforms [16, 17, 13].

Challenges & Opportunities The above goal raises a great deal of challenges, including i) ISA heterogeneity, ii) the complexity of kernel's suspend/resume code, and iii) that the kernel itself is rapidly evolving and may easily obsolete our engineering effort. No prior system has addressed these challenges altogether, to the best of our knowledge.

On the bright side, we set to exploit two key opportunities. First, we leverage the aforementioned ISA similarity to reduce execution overhead. Second, we will specialize our system for supporting suspend/resume, rather than arbitrary kernel code. For example, since the main CPU and PM core are never intended to run concurrently and they each flush own caches when offloading starts and ends, we



(a) A multikernel OS (prior art) **(b) Offloading suspend/resume via virtualization (this work)**

Figure 2: A comparison of two alternative OS structures for harnessing heterogeneous, incoherent cores

can avoid software cache coherence between the cores. We will present more details in Section 5.

4.1 Design choice exploration

Manual code partitioning is infeasible To enable suspend/resume offloading, one may be tempted to manually "carve out" suspend/resume code from an existing kernel. This is difficult, if not infeasible: first, there is no clear boundary around the suspend/resume logic: it shares extensive state with the rest of the kernel; second, much of the suspend/resume code is also executed in other kernel contexts, e.g. runtime power management [20] or user-initiated resume, which are not suitable for offloading.

A multikernel OS breaks compatibility Targeting spanning one OS over heterogeneous processors, prior work [6, 1, 11] advocated running multiple kernels – one for each core type – under a single system image, as shown in Figure 2(a). While the resultant OS eases user-level programming, it requires deep customization of kernel structures and thus gives up compatibility with commodity OSes.

In such a multikernel OS, kernel instances coordinate by either message passing or shared memory. Both suffer from compatibility difficulty. While message passing significantly departs from commodity OS designs, the shared-memory structure too requires tedious and repetitive efforts: kernel images for different core types must be built from a common source tree; across the resultant images, any given memory object must be placed at the same virtual address. This requires manual tweaks of kernel configurations, linker commands, and the internal memory layout of various data structures. What is worse, such efforts have to be repeated for each release of a commodity OS.

4.2 A specialized virtual executor

Different from the aforementioned choices, we take a radical design point: using the PM core to execute the main CPU's *unmodified* kernel binary, as shown in Figure 2(b). For idling, this new design enjoys the low idle power of PM core; for busy execution, this design uses the PM core's superior efficiency to offset the overhead of virtualization, and in return gains compatibility with commodity kernels.

The virtual executor consists of two key components.

Dynamic binary translator To bridge the heterogeneity gap, we retrofit the technique of dynamic binary translation. In a nutshell, the dynamic binary translator reads in the kernel code (in the main CPU's ISA), converts it to blocks of instructions in the PM core's ISA (called "translation blocks"), and executes the translation blocks on-the-fly. For efficiency, the translator caches recent translation blocks.

Minimizing the translation overhead is critical to our system; we will discuss our key optimizations in Section 5.

Kernel function pruner As the virtual executor translates unmodified kernel binary, it judiciously prunes the execution of kernel functions that i) can be greatly simplified for the purpose of suspend/resume and ii) have stable kernel API which is not likely to change in future versions. For example, since suspend/resume is mostly single-threaded, the virtual executor can redirect invocations of the CPU scheduler to a minimalist implementation.

Our kernel function pruner is reminiscent of the concept of para-virtualization. Overall, we slightly increase the virtual executor's dependency on the kernel; in return, the virtual executor is able to exploit the extra knowledge about kernel internals for higher execution efficiency.

How much kernel source will be modified? Our design requires a one-time, minor change to the kernel: change the order between the power state transition of the main CPU itself and the rest of the suspend/resume procedure. With the change, as soon as suspend is initiated, the kernel turns off CPU and shifts the remaining responsibility to the PM core; in resume, the CPU remains off until all other steps, such as powering IO devices, are completed.

5. BAREMETAL BINARY TRANSLATION

We will next focus on dynamic binary translator, the heart of virtual executor. Compared to prior work [2, 4], our translator is novel on two aspects. First, it is the first working prototype running on a simple core and translates binaries for a much more complex core, which is enabled through painstaking engineering efforts as will be discussed below. Second, it exploits the *similarity* between heterogeneous ISAs to aggressively reduce the execution overhead.

Test platform We choose our test platform to be the TI OMAP4 [16] that features loosely coupled ARM Cortex-A9 and M3. Compared to other SoCs, the OMAP4 hardware is well documented; it receives great support from the mainline Linux at the time of writing. Other viable heterogeneous SoC testbeds include the Freescale i.MX6 SoloX (Cortex-A9 + M4) and the TI Sitara AM5728 (Cortex-A15 + M4).

Note that our design and optimizations below are not specific to OMAP4 but applicable to the ARMv7a and v7m ISAs, which are widely used in today's mobile systems.

5.1 Baseline design

On the Cortex-A9 of TI OMAP4, we run a recent mainline Linux kernel (4.4), in which the suspend/resume function is considered mature. In building virtual executor on Cortex-M3, we retrofit the opensource QEMU [2] which provides a complete framework for cross-ISA binary translation.

Restructuring QEMU. To fit the weak M3 core, we have refactored the QEMU codebase that has over 2.6 million SLoC. First, we have extracted its dynamic binary translation core and packaged it as a library. The refactoring cuts the size of translator code by 87%, from 6.5 MB to around 800 KB.

Second, we have aggressively narrowed the interface between the translator and its underlying software. Ideally, the translator should run baremetal on the PM core. Yet, the original QEMU depends on a POSIX interface, which is

often made available through a full-grown OS. To fix this, we reduced the interface to threading, timer, memory mapping, and heap management, which can be supported by a mini runtime with a few simple libraries. For quickly prototyping, we currently bring up uCLinux and uClibc (both forked from the EmCraft release [15]) as the runtime. The final image size of whole virtual executor is around 2MB.

A new translation target Intended for desktop or server, QEMU lacks support for embedded ISA, e.g. ARMv7m for Cortex-M3, as its translation targets or "backend". Initially, we intended to use QEMU's architecture-independent IR interpreter as the backend for M3. However, benchmarking kernel functions (e.g. `kallsyms_lookup_name`) shows that interpretation leads to more than a 100× slowdown as compared to A9 native execution. To avoid such substantial efficiency loss, we have built a new backend for ARMv7m under the QEMU framework in around 3K SLoC. Our initial backend design incurs near 20× overhead, which is on a par with other backends shipped with QEMU. This design serves as the baseline of our optimizations below.

5.2 Key optimizations

We next sketch our optimizations that make cross-ISA translation practical. Using the virtualization lingo, we refer to the main CPU that runs the kernel natively as *guest*, and the PM core that runs the translator as *host*.

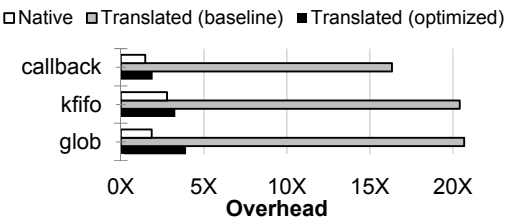
Direct register mapping QEMU, aiming binary translation between arbitrary ISA pairs, features a generic intermediate representation (IR). The IR drops much of the architecture-specific information.

Observing that ARMv7a and ARMv7m have the same number of general-purpose CPU registers, our translator reduces register emulation cost by breaking the generic IR and backs the guest registers with the same host registers. The caveat, however, is that the translator must reserve at least one host register for its own use, e.g. storing the pointer to the emulated CPU state; any guest access to reserved registers must be emulated by memory operations, which is more expensive. Thus, we carefully choose the reserved register to be the least-used one in all kernel instructions; the usage of this register is 8× lower than the most-used one.

Observing that the CPU status flags in the guest and the host share the format, our translator directly passes through flags to avoid emulation as necessitated by the QEMU IR. The emulation is costly: one flag bit is backed by a dedicated software variable; each guest instruction affecting CPU flags is often translated into 10+ host instructions; this is exacerbated by the pervasive conditional executions and loops in the kernel code.

Overall, register mapping reduces total overhead by ∼24%.

Baremetal stacks The stock QEMU emulates the guest stack: it stores a stack pointer register (`SP`) and manipulates it in software, according to guest push/pop. Through profiling, we notice this is expensive as the suspend/resume path invokes numerous driver functions; each function is short, but does push/pop often more than 10 times. Therefore, our translator employs a dedicated guest stack that is directly operated by the host hardware `SP`. Upon entering the translated code for execution, the translator switches to the guest stack by replacing `SP`; upon exiting from the translated code, the translator switches back to its own stack. This reduces the overhead by at least ∼4%.



(a) Measured execution overhead on M3 for kernel benchmarks, normalized to the A9 native cycles

Platform	*Nexus5*	*Gear*	*Note4*	*Panda*
Reduction%	67.1	47.9	79.2	48.6

(b) Estimated energy reduction for suspend/resume

Figure 3: Benchmark results

Relaxed handling of interrupts and exceptions Since suspend/resume does not handle latency-sensitive interrupts, our translator reduces its rate of checking interrupts. In stock QEMU, pending interrupts are checked upon entering each translation block. In executing the branch-heavy kernel code, this incurs extra 7 host instructions per 20 – 30 instructions. By checking interrupts every dozens of blocks, our translator reduces the total overhead by ∼27%.

Our translator also maximizes the execution period within the translated code by leveraging kernel invariants. In the stock QEMU, a branch that goes across page boundary triggers an exit from the translated code and a costly page lookup, since the translator must check whether the destination page is mapped or not. According to our profiling, 93% branches in kernel code are across pages. Observing that the kernel memory is never paged out, our translator continues executing translated code after following a cross-page branch. This reduces the overhead by at least 25%.

Kernel virtual memory As mentioned before, we do not assume that a weak PM core has an MMU that is coherent with the main CPU. Therefore, the virtual executor needs to emulate the kernel virtual memory in software. Conceptually, in translating each memory access the virtual executor needs to walk the main CPU's page table that resides in the shared DRAM. We exploit a Linux kernel invariant: most of the kernel memory is *linearly mapped*; a virtual and its corresponding physical address only differ by a constant offset. Therefore, for any access in the linear-mapping memory, the translator simply does one bound check and one addition; only for translating temporary mappings, e.g., for IO memory, the virtual executor walks the CPU's page table.

5.3 Kernel microbenchmarks

We demonstrate the efficacy of our design through a set of benchmarks, all excerpted from the real Linux kernel source: `glob` is a compute-intensive routine that matches string patterns, which is widely used, e.g., in kernel symbol lookup; `kfifo` exercises the virtual executor by manipulating one of the most common kernel data structures; `callback` mimics the complex control flow of suspend/resume that invokes numerous callback functions provided by drivers.

For each benchmark, we measure the cycles in executing i) the native binary for A9, ii) the native binary for M3, iii) our baseline translation on M3, and iv) our optimized translation on M3. We define the execution overhead as the measured cycles normalized to A9's native execution cycles.

As shown in Figure 3(a), our optimizations use less than

5× cycles to finish the same benchmarks compared to baseline. More importantly, our execution overhead is within 2× of the M3 native execution overhead, which is the theoretical lower bound for offloaded execution.

5.4 Benefit of energy efficiency

We estimate the total energy reduction as follows.

• Without offloading, the energy consumption is given by:
$E_{cpu} = (T_{busy\_exec} + T_{busy\_wait}) \cdot P_{busy} + T_{idle} \cdot P_{idle}$
Here, Ps are the power of main CPU and Ts are the busy/idle durations measured from profiling in Figure 1.

• With offloading, the energy consumption is given by:
$E_{pm} = X \cdot F \cdot T_{busy\_exec} \cdot P'_{busy} + T_{busy\_wait} \cdot P'_{busy} + T_{idle} \cdot P'_{idle}$
Here, Ts are identical as above and P's are the power of PM core. $X \cdot F$ captures the kernel execution slowdown due to offloading: X is the execution overhead in cycle count ratio as defined and measured in Section 5.3; F is the clock ratio between the main CPU and the PM core.

To calculate the energy saving, i.e. $(E_{cpu} - E_{pm})/E_{cpu}$, we plug in the measured power of OMAP4 listed in Table 1, which was also used in our prior work [6]. Note that we use the power measured at both cores' full clockrates; if we consider that both cores can exploit DVFS to run at their lowest clock rates, our energy saving will be even higher.

As summarized in Figure 3(b), our system is able to reduce energy consumption in suspend/resume by up to 80%. Given the significance of suspend/resume (§2), the reduction will substantially extend battery life: for example, in the background sensing scenario evaluated in prior work [5], our system will extend the battery life by up to 30%.

6. RELATED WORK

Suspend/resume The significance of efficient suspend/resume has been recognized for mobile devices [5]. Prior work has already shown some evidences that IO devices are one key bottleneck [5, 19]. Inspired by it, we present quantitative results, and provide in-depth examination of individual IOs. Prior work also seeks to speed up suspend/resume through reordering power operations [5, 19]; complementary to it, our approach focuses on reducing the energy cost. Our own work advocates automatic *runtime* IO power management [20]; orthogonal to it, this work focuses on power management when the platform is turned on or off.

OS for heterogeneous processors To harness a set of heterogeneous, incoherent cores, much work [1, 6, 11] proposes per-core kernels under a single system image, at the cost of compatibility with commodity OSes.

K2 [6] enables asymmetric CPUs to independently execute the same OS. Popcorn [1] bridges heterogeneous CPUs by running multiple message-passing OS kernels. GPUfs [14] bring file abstraction to GPU code. Compared to it, we reduce the dependency among the software stacks on different cores, offering better compatibility with commodity kernels.

Dynamic binary translation (DBT) DBT is a classic technique widely proven useful [2, 4]. While prior work runs DBT on powerful, full-fledged hosts, we are the first building DBT for a simple core that even lacks MMU. Recent work applies DBT to whole OS kernels [4] and exploits kernel invariants for performance, which inspires our proposal. Compared to them, we use DBT to bridge heterogeneity gap and optimize it for specific subsystem – suspend/resume.

7. CONCLUSIONS

Suspend/resume is seriously inefficient due to slow power state transitions of IO devices. To address this problem, we set to offload the OS execution of suspend/resume to a weak, heterogeneous core. To this end, we propose a specialized virtual executor running on the weak core that directly executes the kernel binary. Through a novel design and optimizations, we aggressively reduce the virtualization overhead and therefore harvest energy efficiency and compatibility. A full implementation is in progress.

Acknowledgement The work was supported in part by Purdue's Summer Undergraduate Research Fellowship (SURF), a Google faculty award, and NSF Award #1464357.

8. REFERENCES

[1] 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*, 2015.

[2] F. Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, 2005.

[3] 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*, 2015.

[4] P. Kedia and S. Bansal. Fast dynamic binary translation for the kernel. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013.

[5] M. Lentz, J. Litton, and B. Bhattacharjee. Drowsy power management. In *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015.

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

[7] 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*, 2016.

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

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

[10] J. Mogul, J. Mudigonda, N. Binkert, P. Ranganathan, and V. Talwar. Using asymmetric single-isa cmps to save energy on operating systems. *Micro, IEEE*, 28(3):26–41, 2008.

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

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

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

[14] M. Silberstein, B. Ford, I. Keidar, and E. Witchel. Gpufs: Integrating a file system with gpus. In *Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, 2013.

[15] E. Systems. uclinux for cortex-m3 and cortex-m4. https://github.com/EmcraftSystems/linux-emcraft, 2015.

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

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

[18] Texas Instruments. AM335x Sitara Processors: Technical reference manual, 2015.

[19] 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*, 2013.

[20] C. Xu, F. X. Lin, Y. Wang, and L. Zhong. Automated os-level device power management for socs. In *Proc. ACM Int. Conf. Architectural Support for Programming Languages & Operating Systems (ASPLOS)*, 2015.

[21] 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.