Energy Discounted Computing on Multicore Smartphones

Meng Zhu and Kai Shen, ***University of Rochester***

https://www.usenix.org/conference/atc16/technical-sessions/presentation/zhu

This paper is included in the Proceedings of the 2016 USENIX Annual Technical Conference (USENIC ATC '16).

June 22–24, 2016 • Denver, CO, USA

978-1-931971-30-0

Open access to the Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC '16) is sponsored by USENIX.

Energy Discounted Computing on Multicore Smartphones

Meng Zhu Kai Shen
University of Rochester

Abstract

Multicore processors are not energy proportional: the first running CPU core that activates shared resources incurs much higher power cost than each additional core does. On the other hand, typical smartphone applications exhibit little parallelism and therefore when one core is activated by an interactive application, computing resources at other cores are available at a deep energy discount. By non-work-conserving scheduling, we exploit energy-discounted co-run opportunities to process best-effort smartphone tasks that involve no direct user interaction (e.g., data compression / encryption for cloud backup, background sensing, and offline bytecode compilation). We show that, for optimal co-run energy discount, the best-effort processing must not elevate the overall system power state (specifically, no reduction of the multicore CPU idle state, no increase of the core frequency, and no impact on the system suspension period). In addition, we use available ARM performance counters to identify co-run resource contention on the multicore processor and throttle best-effort task when it interferes with interactivity. Experimental results on a multicore smartphone show that we can reach up to 63% energy discount in the best-effort task processing with little performance impact on the interactive applications.

1 Introduction

Energy remains the critical resource bottleneck for typical smartphone usage. Due to the slow progress on battery technologies and size restrictions of hand-held devices, battery capacity still limits effective smartphone usage between charges. At the same time, today's popular smartphones are commonly equipped with quad-core or even octa-core processors. Powerful multicore processors put further pressure on the scarce energy resource of a mobile device.

Multicore processors are not energy proportional: the first running CPU incurs much higher power cost than each additional core does. This can be attributed to two reasons. First, modern processors are good at power gating. When the system is completely idle, most parts of the CPU can be shutdown resulting in minimum energy consumption. Second, the sharing of hardware resources on a multicore means that the first running core must activate the bulk of shared resources while additional cores can utilize the already activated resources at much lower cost. This energy disproportionality suggests that a multicore processor is more energy-efficient when more of its cores are utilized at the same time.

Unfortunately, typical smartphone applications are built on event-driven, UI-centric framework and serve only a single user. They do not have sufficient parallelism to utilize multiple CPU cores simultaneously. Recent studies [10, 19] on Android applications show a lack of thread-level parallelism across applications and an over-provisioning of core resources across devices. This implies that smartphone multicore processors often operate at low core utilization resulting in poor energy efficiency. At the same time, when one CPU core is being utilized, computing resources at other cores are available at a deep energy discount.

In this paper, we propose to exploit such energy-discounted co-run opportunities to process best-effort tasks that are useful on a smartphone but do not involve direct user interaction (and thus its time of execution is flexible). One example of best-effort tasks is the file compression and encryption in preparation for backing up the user data to the cloud. Another example is the offline bytecode compilation into native code for optimized application execution in Android. The third example is background sensing and analysis of user's facial expression or eye movements to improve user experience. Work in this paper shows that best-effort tasks may be scheduled to co-run with interactive applications and realize significant energy discount.

The idea of saving smartphone energy by bundling tasks or piggybacking computation on other applications is not new [14, 18]. Unlike previous work, we recognize that optimal energy discount on multicores is only realized when the best-effort task execution does not elevate the overall system power state. Specifically, the best-effort task execution must *not* disrupt the multicore CPU idle state, increase the core frequency, or affect the smartphone's suspension period. In other words, the smartphone's multicore power states should experience no change due to the additional best-effort task execution. We accomplish this objective through careful non-work-conserving CPU scheduling.

While co-execution of applications on multicore processors may improve the energy efficiency, it also risks significant interference on shared hardware resources, memory bandwidth and last-level-cache space in particular, and thereby leads to poor interactive application performance and degraded user experience. To mitigate such contention, we use available processor performance counters to monitor memory bandwidth usage during the co-execution, and throttle the best-effort task when it interferes with the foreground application interactivity.

The rest of this paper is organized as follows. Section 2 elaborates on multicore energy disproportionality and available smartphone best-effort tasks that motivate our work. Section 3 presents our design of energy-discounted computing and resource contention mitigation on multicore smartphones. Section 4 describes our implementation on the Android platform. Section 5 evaluates the energy saving of best-effort task executions and the impact of user interactivity between alternative approaches. We also perform trace-based application analysis to demonstrate the abundance of energy-discounted computing opportunities in various smartphone usage scenarios. We present related work in Section 6 before concluding the paper in Section 7.

2 Motivation

2.1 Multicore Energy Disproportionality

CPUs have traditionally been the biggest energy consumer in the computer system and are not energy proportional. Thanks to many innovations, they are now much improved. Today, multicore processors have very sophisticated power states which can be dynamically adjusted to adapt to different workloads. Specifically, dynamic voltage and frequency scaling (DVFS) is used to achieve a wide range of performance / power settings when the system is active. During the idle period, clock gating and power gating are heavily utilized to power down various parts of the processor in order to achieve low power consumption. These techniques enable the CPU to scale their power consumptions relatively well in relation to their utilizations, making them probably the most energy proportional hardware component in the current computer system.

However, making good energy proportional hardware remains difficult and current CPU's energy proportionality is far from perfect. This is particularly true for multicore processors. Figure 1 shows power consumptions of several multicore smartphone / tablet platforms when different number of cores are active. On all platforms we can observe a disproportionate power jump when activating the first core of each multicore processor. Specifically, we can see that activating each additional cores typically consumes less than half of the power comparing to that of the first core. This is substantial given the small profile of the mobile device.

This energy disproportionality is mostly due to the aggressive hardware sharing. In order to drive down cost, reduce footprint and save power, modern multicore processors share substantial hardware components between cores. CPUs on one socket usually share the oscillator and power rail which forces each CPU to operate at the same frequency. As a result, multicore processors can achieve high energy efficiency during heavy parallel processing. However, if the workload can not scale to take advantage of available cores, the entire socket will have to be kept at certain frequency and voltage level to accommodate a few cores' performance needs resulting in a waste of energy.

Besides limiting the capability of active performance / power scaling, hardware sharing also affects the processor idle state. Table 1 lists the available CPU idle states on the Huawei Mate 7 smartphone. As you can see, individual core idle state (C1) does not have much impact on the overall power consumption. Maximum power saving is only achieved through continuous and simultaneous CPU sleeps (C2) [29]. Again, hardware sharing plays an important role here. For example, L2 cache and related memory subsystem can only be shutdown when the whole CPU cluster is completely idle.

We are aware that various heterogeneous architectures have been proposed to improve the CPU energy proportionality. For example, chips with asymmetric clocking capabilities are able to set different frequencies for different cores, realizing more flexible performance / power scaling. Also, chips equipped with CPUs of different micro-architectures (e.g., ARM big-LITTLE) are used to mitigate the performance vs. power dilemma. However, these techniques do not completely eliminate the energy disproportionality. Hardware sharing is and will continue to be one of the fundamental design principles of multicore processors. Consequently, CPU energy disproportionality will remain a reality that computer systems have to live with in the foreseeable future.

To summarize, modern multicore processors can achieve high energy efficiency when doing heavy parallel processing or in complete idle states. But due to the aggressive hardware sharing, they are very inefficient when dealing with workloads of limited parallelism. Unfortunately, it is well known that typical smartphone applications lack the parallelism to utilize the increasing number of cores available to them. This creates opportunities for the mobile system to make use of the extra computation resources to complete certain tasks at an energy discount.

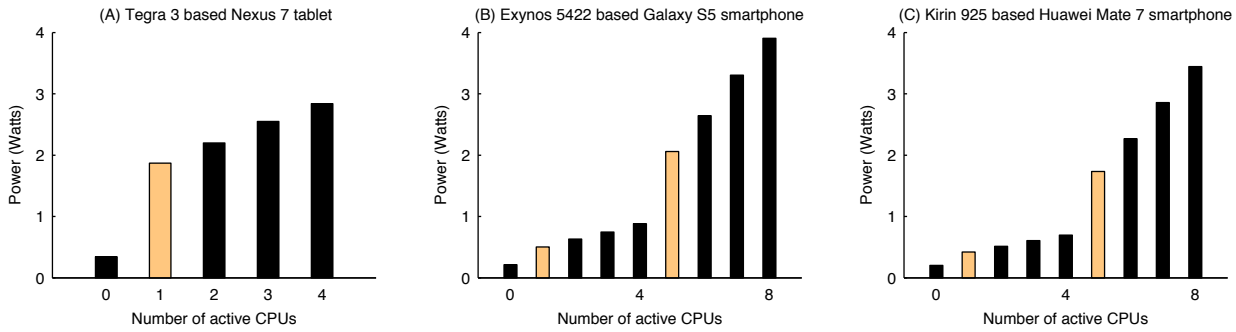

Figure 1: Disproportionate multicore power consumption on the number of active CPUs. The Tegra 3 chipset on Nexus 7 tablet (A) contains a quad-core processor. Both Exynos 5422 (B) and Kirin 925 (C) chipsets contain two heterogeneous quad-core processors. We remove the device battery and use Monsoon Power Meter [4] to measure the whole system power consumption. Devices are put into early-suspend mode where the display and the touchscreen are turned off.

State	Name	Power	Target residency	Description
C0	Wait for interrupt (WFI)	403 mW	1 nSec	Processor is clock gated but can respond to cache / TLB maintenance (e.g., L2 snoop) requests without exiting the WFI state.
C1	Individual powerdown	365 mW	1 mSec	Processor is power gated. All state including L1 cache content is lost and the processor is removed from the coherency protocol.
C2	Cluster powerdown	214 mW	4 mSecs	Can only be entered when all processors are in individual powerdown mode. All state including the L2 cache content is lost.

Table 1: CPU idle states available on the Huawei Mate 7 smartphone. A state's target residency is a value defined in the corresponding *cpuidle* driver in the Linux kernel. It indicates the minimum time period during which the CPU expects to remain idle so that it is worthwhile to enter the state.

2.2 Best-Effort Tasks

We define best-effort tasks as application workloads that are meaningful to the user but do not involve direct interaction and thus have loose quality-of-service requirements. As mobile phones are increasingly used for a variety of purposes, best-effort tasks are becoming common in day-to-day uses. Here are a few examples.

Upload and download operations are common on smartphones. Syncing data with cloud storage services, posting on social websites and software installation / update are typical smartphone usages. Some of them can be delayed to a certain extent. Although significant energy consumption comes from the transmission module, CPUs also consume substantial energy during the process. For example, data compression / decompression requires heavy computation. And encryption / decryption are almost mandatory nowadays which also involve nontrivial CPU processing.

System maintenance work sometimes can also be treated as best-effort tasks. For instance, Android / Linux uses *kswapd* daemon to scan for memory pages that can be swapped out to free up space. Another example is a system daemon called *dhd_dpc* which analyzes network packets and scans for Wi-Fi hotspots. In addition, during application installations, Android would optimize the downloaded packages by recompiling the bytecode for better native performance. All these require substantial CPU processing. Some of them may have timing constraint (e.g., memory management). However, their completions often only matter when the user is also actively using the phone, in which case discounted computation opportunities are likely to be abundant (Section 5.4).

Background sensing is also a suitable best-effort task. Previous work [14] has shown that delaying sensing activities to overlap with other application executions can be more energy efficient. With our technique, the bundled execution can reap even more energy discount. Re-

cent trends also suggest more creative ways of sensing. For example, using camera sensors to analyze user's facial expressions or eye movements [1] may improve user experience. Due to privacy issues, it is beneficial to perform these analysis locally which will put pressure on the device battery life. Since these sensing activities often overlap with user interactive tasks, our technique can be used to substantially lower the energy cost.

Proactive tasks are done predictively to improve user experience. As smartphones getting "smarter", these tasks are becoming increasingly common. For example, Siri can provide recommendations, news and applications "even before you ask" [3]. Previous work [25] also suggests to pre-launch applications to hide user perceived delay. These tasks often do not have hard deadlines and thus can benefit from our technique to save energy.

3 Energy Discounted Computing

Given the energy disproportionality of smartphone multicore processors and the lack of parallelism in typical mobile applications, it is possible to get a deep energy discount by co-scheduling best-effort tasks with the interactive application. However, achieving maximum energy discount without impacting the user experience requires careful system control.

3.1 Power State Preservation

During the execution of interactive applications, the CPU will dynamically adjust its power states to meet the application performance needs. The key principle of reaching the optimal energy efficiency is to utilize the additional (otherwise idle) processor resources without elevating the overall CPU power state.

- *CPU idle state, or ACPI "C" state [24]:* On smartphones, there are often long idle gaps between user interactions during which the user is consuming the content on the screen while all CPUs enter deep sleep state. As simultaneous and continuous sleeps can save a lot of energy [29], it is crucial to keep best-effort tasks from disrupting these idle periods. On the other hand, during active application executions, due to lack of parallelism, idle CPUs will often enter per-core idle states. These shallow sleep states, as we mentioned in Section 2.1, do not save much energy. Thus these idle cores can be utilized to run best-effort tasks at an energy discount. To achieve this, the CPU scheduler needs to schedule best-effort tasks opportunistically in accordance with interactive applications and therefore non-work-conserving CPU scheduling may be necessary. Specifically, the system should schedule best-effort tasks on idle cores only if there is at least one sibling CPU being actively utilized by the interactive application. Otherwise it should enter idle state even when best-effort tasks are ready to run.

- *Core frequency state, or ACPI "P" state:* Modern CPUs use DVFS to quickly adjust power levels to conserve energy and meet performance needs of different workloads. In our co-run scheme, the system should avoid raising the CPU frequency / voltage levels for best-effort tasks. Otherwise, the extra energy consumption will negate the energy discount and the system may well consume more energy than running each task individually combined. At the same time, such caution should not affect the performance of interactive applications. In other words, the CPU frequency adjustment should only focus on the needs of interactive applications and ignore the presence of best-effort tasks.

- *Smartphone suspension state, or ACPI "S" state:* Systems in the suspension state consume very little energy by shutting down most parts of the hardware, including the CPU and memory. On some platforms (notably Android), applications can prevent system suspension by making explicit requests to the operating system. It is important that, in our design, best-effort tasks are not permitted to make such requests. The system should be able to enter the suspension state regardless of best-effort tasks.

To summarize, realizing the maximum energy discount requires judicious control of various aspects of the system to prevent best-effort tasks from elevating the system-wide CPU power states. In other words, best-effort tasks should be *invisible* to the system when making CPU power state adjustments.

3.2 Resource Contention Mitigation

Carefully running best-effort tasks along with interactive applications can bring significant energy savings. However, such savings should not sacrifice user experience. In particular, performance of interactive applications should not be affected.

Co-running tasks on a multicore can potentially slow down each other due to resource contention. This is further exacerbated in our system due to its scheduling strategy—best-effort tasks are intentionally scheduled to run hand in hand with interactive applications.

One easy mitigation is to adjust the CPU scheduling priority. Various parameters are available for this purpose (e.g., nice values and CPU shares on Linux). In our

design, due to the clear importance of interactive applications, we choose to grant absolute priority to them—they are always picked by the scheduler before best-effort tasks. In other words, within one CPU, best-effort tasks can only be scheduled when there is no interactive task waiting.

Absolute priority can eliminate contention on CPU time and mitigate private cache and TLB pollution. However, due to the hardware resource sharing on multi-core processors, contention could also result from shared hardware resources like last-level-cache space and memory bandwidth between cores. Our system uses a simple contention identification approach. Specifically, we monitor the last-level-cache miss rate using the available performance counters. Contention is identified if the miss rate reaches a threshold that suggests memory bandwidth saturation. We acknowledge a limitation of our approach—last-level-cache space contention that does not lead to memory bandwidth saturation will not be identified. Comprehensively identifying cache space contention would be challenging and it generally cannot be accomplished by online monitoring of performance counters alone.

Once the contention is identified, the common approach is to throttle the antagonist (low priority tasks in the contention) executions. This can be done most efficiently on platforms that support certain hardware features such as CPU duty-cycle modulation or asymmetric frequency clocking. Unfortunately, these hardware features are not widely available on today's smartphone processors.

Our system relies on the CPU scheduler to throttle the best-effort tasks. Comparing to the above techniques, however, this is more coarse grained. We can only assert control in the granularity of a CPU quantum (otherwise risk extra scheduling overhead). Fortunately, closely monitoring the contention through performance counters could help time the throttling control more accurate, making this approach quite effective in practice.

4 Implementation

We have implemented our co-run scheme on Huawei Mate 7 smartphone running Android 4.4 and Linux 3.10.30. Our entire modification resides in the Linux kernel.

We use Linux control groups (cgroup) to identify best-effort tasks in the kernel. During system boot, a CPU control group named *best-effort* is created under the cgroup root hierarchy. Best-effort tasks can then be easily added to this group by interacting with the cgroup virtual file system.

Non-work-conserving CPU scheduling We modify the Linux complete fair scheduler to maximize CPU idle state energy saving. Our scheduling policy requires coordination between sibling CPUs. To avoid expensive cross-CPU interrupts and synchronizations, we implemented these communications asynchronously. Specifically, a CPU that wants to schedule best-effort tasks is responsible for checking its siblings' state. We have each CPU maintain a flag indicating its current scheduling state with the following four values:

- *BUSY* indicates the CPU is running normal tasks (e.g., interactive applications),
- *IDLE* indicates the CPU is in idle state (regardless of the level of idle state),
- *BEST-EFFORT* indicates the CPU is running best-effort tasks,
- and *UNDEF* is a transient state (e.g., during context switches).

These per-core flags are cache line aligned to avoid possible false sharing. Although the asynchronous flag read may return stale value under data races, it is likely to be corrected at the next scheduling opportunity. When the scheduler is picking next task to run, normal tasks have absolute priority and are always picked before best-effort ones. If there are only best-effort tasks left in the run queue, the scheduler will first check its siblings' state. If any of them is currently *BUSY*, it will proceed to schedule one of the best-effort tasks. Otherwise, it will enter idle state directly.

Frequency preservation Kernel *cpufreq* governor is responsible for adjusting the CPU frequency. It can come in different flavors but the process usually involves tracking the system load and making adjustments according to some fine-tuned parameters. The load is calculated on a per-core basis by looking at the CPU busy time during the past epoch. Governors typically raise the frequency according to the need of the most heavily loaded CPU (if per-core frequency setting is not available).

To prevent best-effort tasks from affecting the CPU frequency, we track their CPU usage and subtract that from the total CPU busy time when calculating the load. This, along with the absolute priority modification in the scheduler, essentially make best-effort tasks invisible to the CPU governor. While governors completely ignore best-effort tasks, they can still respond to the need of interactive applications just like before.

These modifications reside in the generic governor framework thus individual governor change is not needed.

Suspension management On Android, *wakelock* is used to govern the system suspension state. Applications that want to keep the system awake need to make explicit request to the kernel and grab a wakelock. According to our co-run policy, best-effort tasks should not hold any wakelocks. We modify the wakelock kernel *sysfs* interface to reject any requests made from best-effort tasks.

Contention-triggered throttling Our performance counter based throttling strategy is implemented as a loadable kernel module. We assess the memory bandwidth usage by monitoring the L2 (last-level) cache miss rate. Specifically, we select two events in ARMv7 performance monitoring unit: *ARMV7_A15_PERFCTR_L2_CACHE_REFILL_READ* as L2 cache read miss and *ARMV7_A15_PERFCTR_L2_CACHE_REFILL_WRITE* as L2 cache write miss. Combined they approximate the total access to the main memory. The module triggers periodic interrupt every 20 ms to collect and update the counter statistics. We read the counter value directly from the registers and take care of the overflows. Before picking best-effort task, the CPU scheduler is required to check the latest L2 cache miss rate. If the rate is above certain threshold, the best-effort task will not be scheduled. Similar to our other scheduler modifications, the counter maintenance and lookups are performed in an asynchronous way to avoid the overhead of cross-CPU interrupts and synchronizations.

Our modifications (including the periodic performance counter reading) incurs less than 1% performance overhead for all our benchmarks described in Section 5.1.

5 Evaluation

In this section, we evaluate our techniques on a real device with realistic benchmarks. Section 5.1 introduces our evaluation setup. Section 5.2 evaluates the system energy efficiency. Section 5.3 assesses the effectiveness of our contention mitigation measures. Section 5.4 provides a trace-based application study to demonstrate the abundance of energy-discounted computing opportunities in various smartphone usage scenarios.

5.1 Evaluation Setup

Experimental device We use Huawei Mate 7 smartphone. It was released in October 2014 and is equipped with a Hisilicon Kirin 925 SoC which contains an ARM big.LITTLE octa-core CPU. We use the big cluster in our evaluation. It has four 1.8 GHz ARM Cortex-A15 cores. Each core has its own 32 KB/32 KB L1 instruction and data cache and all cores share a 2 MB L2 cache. It has 2 GB LPDDR3 memory with a bandwidth of 12.8 GB/s.

Power measurement In order to do precise power measurement, we remove the smartphone's battery and connect its power pins to the Monsoon Power Meter [4] which acts as an external power source and measures the phone's overall power consumption. The power meter is able to sample the current at 5 kHz. We turn off hardware components like GPS, cellular and dim the display to the minimum brightness. WiFi is kept on for the purpose of controlling the phone through the host machine without the USB connection (which will disturb the power measurement).

Interactive applications We assemble a suite of benchmarks to represent typical interactive and best-effort application co-run scenarios. Two representative interactive applications are chosen. Bbench [12], a widely used web browsing benchmark which automatically loads and renders locally cached popular websites. It measures the browser performance by tracking the JavaScript *onLoad* event which is triggered once a webpage is fully rendered. We run it using the Android default web browser. Another interactive application is Angry Bird, a popular mobile casual game.

For the co-run experiments, it is important that we are able to measure the *interactivity* of the interactive application. For applications like web browser, the interactivity can be defined as time needed to complete certain tasks. For Bbench we use the aggregated webpage rendering time to measure its interactivity. On the other hand, for games like the Angry Bird, the interactivity is only defined by how responsive the application is. Frames-per-second (FPS) is a more relevant metric. We use GameBench [2] to measure its FPS.

Best-effort applications We select five applications as best-effort tasks.

Spin, a CPU intensive microbenchmark that calculates the n-th triangular number by summation. We choose this microbenchmark to illustrate the optimal co-run scenario—a CPU intensive workload with little memory activity.

Compression compresses a set of files using bzip and *Encryption* encrypts them using the AES encryption. These two are chosen to mimic user download and upload activities.

AppOpt optimizes Android application packages by recompiling them into native code. This is chosen to represent typical deferrable system work.

FaceAnalysis is an in-house developed application that analyzes input faces. It uses Stasm [17], an active shape

model based library, to process images and extract positions of landmark features. This is particularly useful in facial expression analysis. We use it to represent emerging passive sensing applications. To make the experiment reproducible, we use locally cached face images as its input.

Input Workload Application workloads are carefully chosen such that the executions of the interactive application and the best-effort task can mostly overlap with each other when using our co-run strategy. Specifically, Bbench are configured to load 15 websites with two seconds delay (to mimic user think time) between each website. The whole session takes roughly 44 seconds to complete. Angry Bird, on the other hand, is played for 42 seconds. Best-effort tasks are launched in the background shortly after the interactive application starts and the amount of the work is configured such that they can finish right before the interactive application ends under the most strict (throttling-based) best-effort task scheduling policy. To make experiments reproducible, we use *RERUN* [11], a record and replay tool for the Android operating system, to automate the test flow. User interaction sessions are recorded into a sequence of touch and system events. Later, these events are sent back to the phone to replay user interactions with precise timing and accuracy.

5.2 Energy Efficiency

To evaluate the energy efficiency of our system when running best-effort tasks with interactive applications, we run Bbench and Angry Bird with each of the five best-effort workloads. We run each pair under two different scheduling strategies:

- default, where there is no change to the original system behavior;
- power-states-preservation scheduling, where our non-work-conserving scheduling techniques are used.

Figure 2 and Figure 3 show the result. Energy discount (σ) of the best-effort task is calculated as

$$\sigma = \frac{E_{\text{best-effort}} - (E_{\text{co-run}} - E_{\text{interactive}})}{E_{\text{best-effort}}} \qquad (1)$$

where $E_{\text{best-effort}}$ is the amount of energy consumed by the best-effort task running alone under the default system setting, $E_{\text{co-run}}$ is the total system energy consumption of the co-run execution and $E_{\text{interactive}}$ is the total system energy consumption when running the interactive application alone. Each of our energy metrics measures the active energy—those consumed above the system idle power consumption.

The result clearly shows that our system can realize deep energy discount in all co-run scenarios, ranging from 23% to 71%. We attribute this to the fact that the overall CPU power states are preserved—the execution of the best-effort task is completely hidden behind the interactive application power profile.

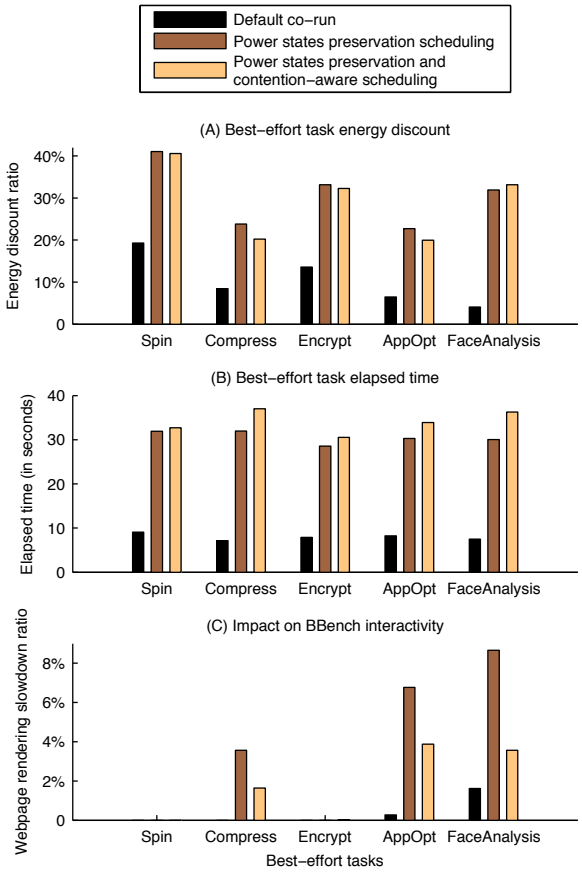

Figure 2: Experimental results of running interactive application BBench web browsing with various best-effort tasks under different scheduling strategies. We show the best-effort task energy discount (A), best-effort task elapsed time (B), and impact on BBench's interactivity (webpage rendering slowdown) (C).

This is further illustrated in Figure 4. When Bbench running alone, the current trace shows the typical burst-then-idle pattern that is common on smartphones due to the long user think time between interactions. During these idle periods, the system is able to enter deep sleep states to conserve energy (trough in the current waveform). However, best-effort tasks, without any control, will disrupt these deep sleep states. In addition, during the burst period, simultaneous executions of both tasks would increase the system load and drive up the CPU fre-

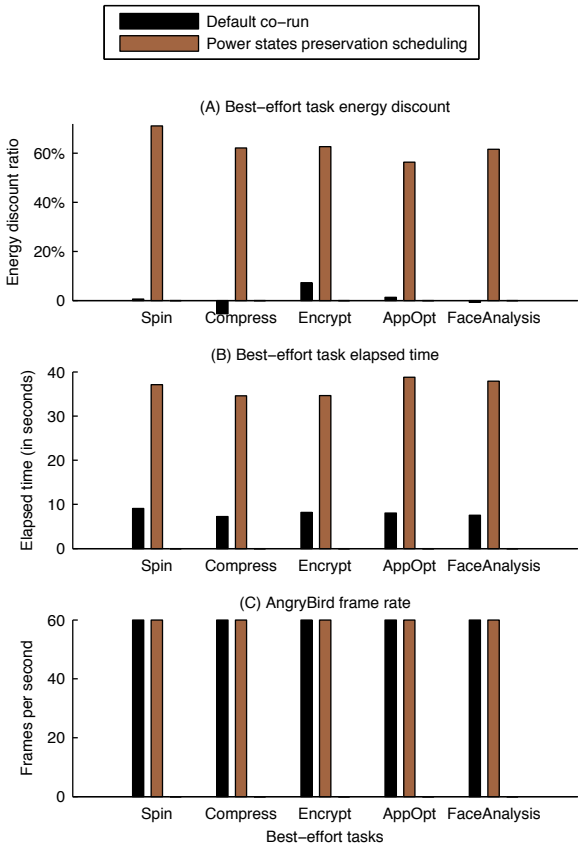

Figure 3: Experimental results of running interactive application Angry Bird with various best-effort tasks under different scheduling strategies. We show the best-effort task energy discount (A), best-effort task elapsed time (B), and Angry Bird's frame rate (C).

quency. In both cases, the system overall power states are elevated, resulting in more energy consumption. These are shown in the current waveform of the default co-run execution. With our power-state-preservation scheduling, both kinds of disruption can be avoided. The best-effort task is piggybacked by the interactive application execution during the burst period, resulting in improved energy efficiency.

It is worth noting that, in Figure 2, the default co-run strategy can also provide some energy discount. This is mostly due to the fact that we intentionally overlap the two application executions by launching them roughly at the same time. Thus a portion of the best-effort task execution happens to be able to utilize some computation resources without elevating the overall CPU power state. In other words, the resulted energy discount is undependable at best. It completely depends on the application characteristics and how the user interacts with them. In fact, in some of our tests, the default co-run strategy can result in more energy consumption than running each task individually combined. Given the typical burst-then-long-idle smartphone usage pattern, the default co-run strategy is likely to perform poorly in most practical scenarios. Our system, on the other hand, always preserves the CPU power states and thus is able to *consistently* provide high energy discount under all circumstances.

Our non-work-conserving scheduling strategy inevitably reduces the system resource utilization and leads to longer execution time of best-effort tasks. Fortunately, best-effort tasks do not involve direct user interaction thus their time of execution is somewhat flexible. The saved energy, on the other hand, could extend the smartphone battery life and let user use their phones more freely which could greatly improve the user experience.

5.3 Throttling-based contention mitigation

As shown in Figure 2, there are non-negligible slowdown on the interactive application when doing power-states-preservation co-run scheduling. In this part of the evaluation, we assess the effectiveness of our throttling-based contention mitigation technique.

We first focus on two application pairs which experience large interactivity slowdown: Bbench+AppOpt and Bbench+FaceAnalysis with 6.77% and 8.66% slowdown respectively. Different L2 miss rate throttling thresholds are used to evaluate their impact on the system performance. Table 2 shows the result. The throttling technique, with properly set threshold, proves to be effective in resource mitigation and minimizing the interactivity slowdown. With L2 miss rate threshold set at 15 misses/μSecs, both the best-effort task elapsed time and the interactivity slowdown remain similar to the non-throttling based scheduling strategy. This means that the throttling mechanism is probably not triggered and a lower threshold is needed. With lower L2 miss rate thresholds, the best-effort task begins to see increased elapsed time while the interactive application performance is improved. This suggests that the system is throttling best-effort tasks while the memory bandwidth is under pressure as it reaches the L2 miss rate threshold. This in turn helps to improve the interactive application performance by reducing the resource contention caused by best-effort task.

However, the benefit is diminished above 10 misses/μSecs. Beyond that, there is very little improvement and even negative impact on the interactive application and the elapsed time of the best-effort task increases dramatically. For a threshold of 6 misses/μSecs, the FaceAnalysis benchmark could not even finish within the 44 seconds Bbench session. This implies that 10 misses/μSecs L2 miss rate is a good indicator of the memory bandwidth saturation and any

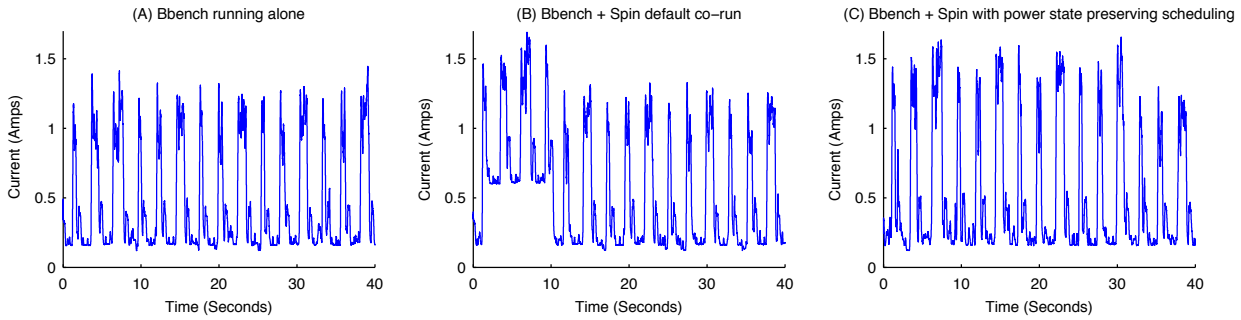

Figure 4: Current trace of Bbench running alone and Bbench+Spin co-run under different scheduling strategies. In the default co-run test (B), the spin task takes 9.07 seconds to finish. Under the power states preserving scheduling strategy (C), the same task takes 31.91 seconds.

lower thresholds can only bring unnecessary slowdown to the best-effort task without benefiting the interactive application.

Next, we apply the contention-aware scheduling technique to all Bbench co-run pairs. The throttling threshold is set at 10 misses/μSecs. Figure 2 shows the result. Besides reducing the Bbench slowdown, we can also see that the throttling technique does not affect the energy savings. In addition, for Spin and Encryption where there is no slowdown of the interactive application, we observe little changes on the best-effort task elapsed time. This suggests that the scheduler is truly contention-aware—it only activates the throttling when there is resource contention in the system.

5.4 Trace-based Application Analysis

In this section we conduct a trace-based application analysis to study the amount of energy discounted computing opportunities in typical smartphone usage scenarios.

Eight popular applications are selected, their detailed descriptions are listed in Table 3. For fair comparisons, each usage scenario lasts for one minute.

We use Linux debugfs-based kernel trace facility to collect CPU frequency and idle state transition events and then calculate the amount of energy discounted computing cycles based on our power-states-preservation scheduling policy. Specifically, if at least one CPU is actively running at certain frequency, other idle CPUs are counted to be able to provide energy discounted computing cycles at that frequency.

For each usage scenario, we measure the abundance of energy discounted computing opportunities in relation to the active CPU usage of the corresponding interactive application. Further, we convert the amount of discounted cycles into meaningful best-effort task workloads by normalizing it to the amount of CPU cycles needed to finish a unit of work (e.g., analyzing one frame of face or encrypt one minute standard resolution video).

(A) Bbench + FaceAnalysis

Throttling threshold (misses/μSecs)	Best-effort task elapsed time	Bbench slowdown ratio
7.5	42.65 Secs	3.94 %
10.0	36.25 Secs	3.57 %
12.5	32.42 Secs	6.58 %
15.0	30.20 Secs	8.73 %

(B) Bbench + AppOpt

Throttling threshold (misses/μSecs)	Best-effort task elapsed time	BBench slowdown ratio
6.0	43.10 Secs	4.20 %
7.5	36.85 Secs	3.72 %
10.0	33.87 Secs	3.88 %
12.5	29.62 Secs	5.80 %
15.0	29.77 Secs	6.81 %

Table 2: The impact of L2 cache miss rate throttling threshold when doing power states preservation and contention-aware scheduling. We show differences in best-effort task elapsed time and Bbench slowdown ratio for two scenarios: (A) Bbench co-running with FaceAnalysis and (B) Bbench co-running with AppOpt.

Table 4 shows the result. As you can see, there are substantial energy discounted computing opportunities in all application categories. This is consistent with the earlier observation that typical smartphone applications lack the parallelism to utilize multicore CPUs. This study demonstrates the potential of exploiting the opportunities enabled by the lack of parallelism in smartphone applications to process useful best-effort tasks at a deep energy

Category	Description
Web Browsing	In Chrome, go to Yahoo.com and browse three top news.
Video Streaming	In YouTube, watch a short HD video for one minute.
Gaming	Play casual game Subway Surf for one minute.
Navigation	In Google Maps, search several nearby attractions and get their directions.
Messaging	In Hangout, open two conversations, type and send two messages.
Social Network	In Facebook, load personal timeline, refresh for friend feeds and browse three posts.
Camera	Use the native camera app to take a minute long video.
Music Streaming	In Google Play Music, stream a song for one minute.

Table 3: Description of eight application scenarios used in the trace-based application study.

Category	Abundance of discounted CPU cycles (multicore)	Abundance of discounted CPU cycles (single-core)	Equivalent work of FaceAnalysis (frames of faces can be analyzed)	Equivalent work of Encryption (minutes of video can be encrypted)
Web Browsing	1.63	0.66	30	21
Video Streaming	2.41	0.85	4	3
Gaming	1.61	0.65	21	15
Navigation	2.42	0.85	13	9
Messaging	2.88	0.97	3	2
Social Network	1.88	0.72	12	9
Camera	2.10	0.77	5	4
Music Streaming	1.63	0.66	7	5

Table 4: Results for the trace-based application study. Each usage scenario lasts for one minute. Abundance of discounted CPU cycles is the ratio of energy discounted CPU cycles to the active CPU cycles used by the corresponding interactive application. In the second column (marked with multicore), energy discounted cycles on all CPUs are counted, assuming the best-effort task has perfect parallelism to utilize all idle CPUs. In the third column (marked with single-core), energy discounted CPU cycles are only counted on one of the eligible CPUs, assuming the best-effort task has no parallelism. We use single-core cycles to calculate the equivalent work of best-effort tasks in column four and five.

discount.

6 Related Work

Smartphone power characterization and energy management has received a great deal of research interests. Using an extensive smartphone power instrumentation platform, Carroll and Heiser [5] developed a power model of various smartphone components and identified promising directions to improve power management. They [6] further suggest that, on a multicore smartphone, CPU cores should be kept online as long as there is work for them. AppScope [26] monitors kernel as well as application activities and correlates them with the smartphone power usage. Song et al. [22] optimized smartphone energy efficiency by lowering CPU frequency when user facing tasks are completed (e.g., display updates finish). Martins et al. [16] monitor and intercept smartphone background activities while the system is in suspension state to extend the battery life. In this paper, we present a new approach to improve smartphone energy efficiency by carefully running best-effort tasks together with interactive applications on a multicore to realize deep energy discounts.

Previous research has recognized the efficiency benefit of piggybacking or co-running background work while the system is active with primary tasks. A classic example [15] is to perform disk work "for free" if such work happens to lie in the disk head rotation and seek path to serve the foreground requests. In the context of mobile systems, Lane et al. [14] showed that performing sensing work while a smartphone is otherwise already active saves substantial wakeup power costs. Nikzad et al. [18] developed an annotation language that demarcates power-hungry executions for delayed execution when the device enters an active state. In this paper,

we make new contributions on energy-efficient multicore piggyback execution by non-work-conserving scheduling that preserves the system power states as if the best-effort task does not run.

Work-conserving schedulers that always utilize available resources when there is work to do is generally desirable for high resource utilization. However, previous research has found the benefits of non-work-conserving scheduling in particular contexts. In disk scheduling, the anticipatory scheduler [13] may keep the disk idling for a short period of time even when there are pending operations. It does so in anticipation of a new I/O operation from the process that issued the just completed operation, which often requires little or no seeking from the current disk head location. In the context of hardware multithreading processors, Fedorova et al. [7] found that running fewer threads than the number of processors may reduce resource contention and improve performance. In this paper, we use non-work-conserving scheduling to run best-effort tasks only when it does not elevate the power states of a smartphone.

There exists a large body of prior work on characterizing smartphone workload behaviors. Gao et al. [10] analyzed a broad range of mobile applications and found that they exhibit little thread-level parallelism and thereby are unable to effectively utilize multiple CPU cores. Seo et al. [19] showed that the lack of thread-level parallelism also prevents mobile applications from effectively utilizing the heterogeneous (big and little) multicore processors. Shingari et al. [21] have identified that co-running multiple mobile applications may yield substantial contention on shared multicore resources. These identified characteristics of smartphone workloads motivate our work of improving execution parallelism and mitigating potentially resulted performance interference.

Multicore performance and power management has been an active area of work for general computer systems. In particular, many techniques were proposed to manage and isolate the shared multicore resources, by partitioning the shared on-chip cache [23, 28], contention-easing scheduling [9, 27], and execution timeslice adjustment [8]. Multicore power disproportionality was also recognized in server power modeling [20]. Mobile systems present new circumstances for multicore performance and power management. First, a lack of thread-level parallelism results in poor multicore energy efficiency on smartphones. Second, the co-existence of interactive and best-effort applications presents differential quality-of-service requirements that complicate resource management.

7 Conclusion

This paper demonstrates the feasibility and benefits in running best-effort tasks on multicore smartphones for an energy discount. The work is motivated by the energy disproportionality of multicore processors and the lack of parallelism in typical smartphone applications. Due to hardware resource sharing, the first running CPU on multicore processors could incur high power cost while each additional core can be used at a much lower energy cost. On the other hand, smartphone applications exhibit little parallelism, leaving room for energy discounted computing on the additional cores.

We propose to exploit these opportunities by running best-effort tasks—tasks that are useful to the user but do not involve direct user interactions. Our contribution lies in the recognition that maximum energy discount can only be realized when overall system power states are preserved. Specifically, the multicore CPU idle state, the processor frequency and the system suspension time should not be affected by the presence of best-effort tasks. We apply careful non-work-conserving CPU scheduling to achieve this goal. In addition, to deal with the interactive application slowdown caused by the co-run activities. We use performance counter based throttling to mitigate the contention on the memory bandwidth. We evaluate our work on Huawei Mate 7 smartphone with realistic workloads. The result shows significant energy discount (up to 63%) for best-effort tasks with minimum impact on the interactive application execution (3.8% slowdown in the worst case).

Acknowledgments This work was supported in part by the National Science Foundation grants CNS-1217372, CNS-1239423, and CCF-1255729, and by a Google Research Award. We thank Handong Ye, Yi Jiang and Jun Wang from FutureWei Technologies, Inc. and Vijayakumar Krishnamurthy, Anthony Mazzola, Chuk Orakwue from Huawei Device for valuable discussions and support. We also thank the anonymous USENIX ATC reviewers and our shepherd Rodrigo Fonseca for comments that helped improve this paper.

References

[1] Fast, affordable eye tracking. `http://www.sticky.ad/`.

[2] Gamebench setup. `https://www.gamebench.net/en/gamebench-setup`.

[3] iOS 9, Siri. `http://www.apple.com/ios/whats-new/#siri`.

[4] Monsoon power meter. https://www.msoon.com/LabEquipment/PowerMonitor/.

[5] CARROLL, A., AND HEISER, G. An analysis of power consumption in a smartphone. In *Proc. of the USENIX Annual Technical Conf.* (Boston, MA, June 2010).

[6] CARROLL, A., AND HEISER, G. Mobile multicores: use them or waste them. In *Proc. of the Workshop on Power-Aware Computing and System (HotPower)* (Nov. 2013).

[7] FEDOROVA, A., SELTZER, M., AND SMITH, M. D. A non-work-conserving operating system scheduler for SMT processors. In *Proc. of the Workshop on the Interaction between Operating Systems and Computer Architecture* (2006).

[8] FEDOROVA, A., SELTZER, M., AND SMITH, M. D. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Proc. of the 16th IEEE Int'l Conf. on Parallel Architecture and Compilation Techniques (PACT)* (Brasov, Romania, Sept. 2007), pp. 25–38.

[9] FEDOROVA, A., SMALL, C., NUSSBAUM, D., AND SELTZER, M. Chip multithreading systems need a new operating system scheduler. In *Proc. of the SIGOPS European Workshop* (Leuven, Belgium, Sept. 2004).

[10] GAO, C., GUTIERREZ, A., RAJAN, M., DRESLINSKI, R. G., MUDGE, T., AND WU, C.-J. A study of mobile device utilization. In *Proc. of the IEEE Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)* (Philadelphia, PA, Mar. 2015).

[11] GOMEZ, L., NEAMTIU, I., AZIM, T., AND MILLSTEIN, T. RERAN: Timing-and touch-sensitive record and replay for Android. In *Proc. of the 35th Int'l Conf. on Software Engineering (ICSE)* (San Francisco, CA, May 2013), pp. 72–81.

[12] GUTIERREZ, A., DRESLINSKI, R. G., WENISCH, T. F., MUDGE, T., SAIDI, A., EMMONS, C., AND PAVER, N. Full-system analysis and characterization of interactive smartphone applications. In *Proc. of the IEEE Int'l Symp. on Workload Characterization (IISWC)* (2011), pp. 81–90.

[13] IYER, S., AND DRUSCHEL, P. Anticipatory Scheduling: A Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O. In *Proc. of the 18th ACM Symp. on Operating Systems Principles (SOSP)* (Banff, Canada, Oct. 2001), pp. 117–130.

[14] LANE, N. D., CHON, Y., ZHOU, L., ZHANG, Y., LI, F., KIM, D., DING, G., ZHAO, F., AND CHA, H. Piggyback crowdsensing (PCS): Energy efficient crowdsourcing of mobile sensor data by exploiting smartphone app opportunities. In *Proc. of the 11th ACM Conf. on Embedded Networked Sensor Systems (SenSys)* (Rome, Italy, Nov. 2013).

[15] LUMB, C. R., SCHINDLER, J., GANGER, G. R., AND NAGLE, D. F. Towards higher disk head utilization: Extracting free bandwidth from busy disk drives. In *Proc. of the 4th USENIX Symp. on Operating Systems Design and Implementation (OSDI)* (San Diego, CA, Oct. 2000).

[16] MARTINS, M., CAPPOS, J., AND FONSECA, R. Selectively taming background android apps to improve battery lifetime. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)* (2015), pp. 563–575.

[17] MILBORROW, S., AND NICOLLS, F. Active shape models with SIFT descriptors and MARS. In *Proc. of the Int'l Conf. on Computer Vision Theory and Applications (VISAPP)* (Lisbon, Portugal, Jan. 2014), pp. 380–387.

[18] NIKZAD, N., CHIPARA, O., AND GRISWOLD, W. G. APE: An annotation language and middleware for energy-efficient mobile application development. In *Proc. of the 36th Int'l Conf. on Software Engineering (ICSE)* (Hyderabad, India, June 2014), pp. 515–526.

[19] SEO, W., IM, D., CHOI, J., AND HUH, J. Big or little: A study of mobile interactive applications on an asymmetric multi-core platform. In *Proc. of the IEEE Int'l Symp. on Workload Characterization (IISWC)* (Atlanta, GA, Oct. 2015).

[20] SHEN, K., SHRIRAMAN, A., DWARKADAS, S., ZHANG, X., AND CHEN, Z. Power containers: An OS facility for fine-grained power and energy management on multicore servers. In *Proc. of the 18th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Houston, TX, Mar. 2013).

[21] SHINGARI, D., ARUNKUMAR, A., AND WU, C.-J. Characterization and throttling-based mitigation of memory interference for heterogeneous smartphones. In *Proc. of the IEEE Int'l Symp. on Workload Characterization (IISWC)* (Atlanta, GA, Oct. 2015), pp. 22–33.

[22] SONG, W., SUNG, N., CHUN, B.-G., AND KIM, J. Reducing energy consumption of smartphones

using user-perceived response time analysis. In *Proc. of the 15th Workshop on Mobile Computing Systems and Applications (HotMobile)* (Santa Barbara, CA, Feb. 2014).

[23] TAM, D., AZIMI, R., SOARES, L., AND STUMM, M. Managing shared L2 caches on multicore systems in software. In *Proc. of the Workshop on the Interaction between Operating Systems and Computer Architecture* (San Diego, CA, June 2007).

[24] UNIFIED EXTENSIBLE FIRMWARE INTERFACE FORUM. Advanced configuration and power interface specification, July 2014. Revision 5.1.

[25] YAN, T., CHU, D., GANESAN, D., KANSAL, A., AND LIU, J. Fast app launching for mobile devices using predictive user context. In *Proc. of the 10th Int'l Conf. on Mobile Systems, Applications, and Services (MobiSys)* (Lake District, United Kingdom, June 2012), pp. 113–126.

[26] YOON, C., KIM, D., JUNG, W., KANG, C., AND CHA, H. AppScope: Application energy metering framework for Android smartphone using kernel activity monitoring. In *Proc. of the USENIX Annual Technical Conf.* (Boston, MA, June 2012).

[27] ZHANG, X., DWARKADAS, S., FOLKMANIS, G., AND SHEN, K. Processor hardware counter statistics as a first-class system resource. In *Proc. of the 11th Workshop on Hot Topics in Operating Systems (HotOS)* (San Diego, CA, May 2007).

[28] ZHANG, X., DWARKADAS, S., AND SHEN, K. Towards practical page coloring-based multi-core cache management. In *Proc. of the 4th EuroSys Conf.* (Nuremberg, Germany, Apr. 2009), pp. 89–102.

[29] ZHU, Q., ZHU, M., WU, B., SHEN, X., SHEN, K., AND WANG, Z. Software engagement with sleeping CPUs. In *15th USENIX Workshop on Hot Topics in Operating Systems (HotOS XV)* (Kartause Ittingen, Switzerland, May 2015).