Offloading to Improve the Battery Life of Mobile Devices

Ranveer Chandra, Steve Hodges, and Anirudh Badam, *Microsoft Research*
Jian Huang, *Georgia Tech*

Mobile devices continue to become more capable; over the last 15 years, we've seen both processor and network speeds increase one thousand fold. However, the batteries upon which most mobile devices depend have not advanced as quickly. In fact, despite significant research into new battery chemistries and construction techniques, typical rechargeable battery energy density has only doubled in that same 15-year period. The designers of mobile devices inevitably have to incorporate batteries that are larger or heavier than they'd like to address the practicalities of usage. By way of example, the size and weight of a mobile phone is constrained by the need for a battery with enough capacity to reliably operate for a day; conversely, tablets and laptops that meet the expectations of today's consumers in terms of form factor and weight can rarely operate for a full day.

Increasing battery energy density is not the only way to improve battery lifetime; a large body of research has explored various ways to reduce mobile device power consumption as an alternative. Many different approaches have been taken, from designing more power-efficient hardware components to developing power-aware software applications. One paradigm that we've been exploring at Microsoft Research for many years is the intelligent offloading of computation, communication, and storage. By passing responsibility for some of these operations away from a mobile device's main processor, it's possible to save power on the mobile device. It's often possible to simultaneously increase performance. This article presents an overview of three such offload techniques that we've developed and concludes with some reflections on the progression of this research.

OFFLOADING TO A LOW-POWER PROCESSOR

Mobile devices, such as laptops, tablets, and smartphones, feature a powerful CPU that's closely coupled with system memory, network interfaces, graphics, and other components. This set of components might be implemented across several discrete packaged components or as a system-on-a-chip (SoC). However, even the latter, which is designed to be as compact and energy efficient as possible, typically only provides coarse granular control of power consumption. As a result, the system often ends up consuming more power than strictly necessary when running relatively lightweight tasks.

Increasing battery energy density is not the only way to improve battery lifetime... [reducing] mobile device power consumption [is] an alternative.

The first offloading technique we developed as part of our aim to reduce the power consumption of mobile devices is called *Somniloquy*.[1] We initially developed it with laptop-class devices in mind. Although the power consumption of these devices can be controlled using techniques such as voltage and frequency scaling, at a coarse power-control level, there are only a handful of different operating states, as defined by the advanced configuration and power interface (ACPI) specification (see Table 1).

Somniloquy is designed to let a range of tasks with basic computation, memory, and peripheral requirements execute while the rest of the device is in one of the four ACPI G1 states—typically S3 or S4 (see Table 1). This is achieved by adding a low-power secondary processor that can be operated independently of the main CPU, such that it can continue to execute lightweight tasks when the CPU and associated subsystems are in a G1 state. In some cases, the secondary processor can include its own RAM and flash memory, a network interface, and other peripherals such as sensors—all operating in a separate power domain from the main CPU and its associated system components.

The secondary processor is designed to run lightweight appli-

SPOTLIGHT

TABLE 1
The advanced configuration and power interface (ACPI) power states.

Category	Subcategory	Power state
G0, Working	S0	The system is under normal operation, although the monitor might be powered down if the user is "away."
G1, Sleeping	S1, Power on suspend	The CPU has been suspended and the caches flushed, but power is maintained; some peripherals might be powered down, but RAM and CPU are powered.
	S2, CPU off	The state here is the same as with S1, but the CPU has powered down.
	S3, Standby/ sleep/suspend-to-RAM	Most components have powered down, but not the RAM.
	S4, Hibernate/suspend-to-disk	The system has powered down but can resume without a reboot.
G2, Soft off	S5	Various components, such as USB peripherals, the clock, and the network interface, run in a very low-power mode and are able to wake the system, which must undergo a full reboot.
G3, Mechanical off	S6	There is absolutely no power to any component, apart from a battery-backed real-time clock.

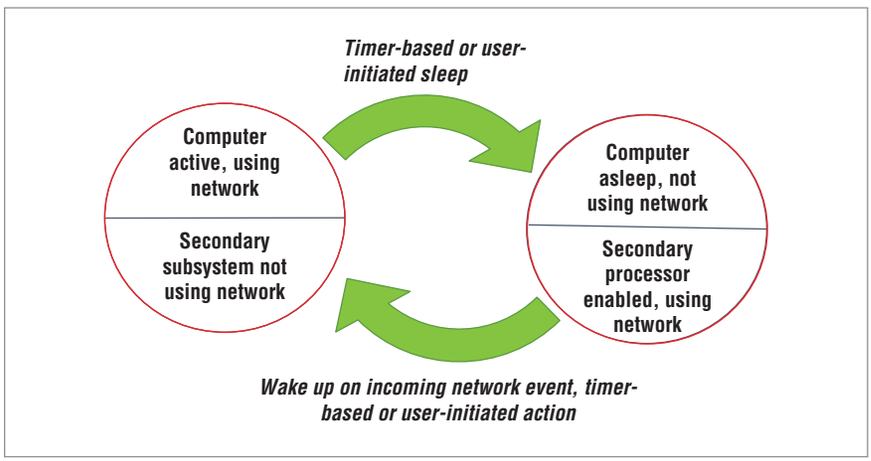

Figure 1. Somniloquy has two main operating states—in one, the main CPU is active; in the second, control resides with the secondary processor while the main CPU sleeps. The transitions between them are initiated by the events shown in the figure.

cations autonomously on behalf of the main CPU. It can often do this for long periods of time, but if resources only available to the CPU are required, then the secondary processor can wake the CPU, moving it into the G0 ACPI state. Eventually, when the CPU-bound operations are complete, the main system can return to a G1 state, and the secondary processor resumes operation. Figure 1 shows the two main operating modes of a Somniloquy-enabled device.

Applications simply requiring the main CPU to be woken from G1 on a network event—for example, an incoming WhatsApp message—can be implemented using "wakeup filters" on the secondary processor. These simply monitor incoming network traffic for certain types of packets—for example, based on the TCP/IP port number—and wake up the main CPU when a match is detected. That packet can then be passed up to the main CPU's networking stack to be processed transparently to the remote host that initiated the communication. This is similar to wake-on-network functionality, but the additional level of control afforded by wakeup filters results in a much more useful system. Figure 2 illustrates the level of power saving possible.

More sophisticated tasks can be offloaded from the main CPU to the secondary processor with the introduction of "application stubs." These implement a simplified subset of the full functionality of an application. For example, an instant messenger (IM) application stub running on the secondary processor would periodically indicate presence to the IM server, even when the main CPU is in a G1 sleep state. Should an incoming IM message be received, the application stub running on the secondary processor would wake the main CPU thereby handing over control. Figure 3 shows the Somniloquy architecture.

OFFLOADING TO A NEARBY DEVICE

While Somniloquy requires additional hardware to support computational offloading, another approach we've explored is WearDrive.[2] Here, we reduce the power consumption of a mobile device that needs to periodically store data (such as a wearable fitness tracker) by offloading storage operations to a nearby, less resource-constrained device. In particular, WearDrive can improve the battery

lifetime by offloading storage from a wearable device to a nearby phone.

The key insight behind WearDrive is that the battery-powered RAM in a mobile device can be considered persistent as long as the battery capacity is monitored to ensure that the RAM remains reliably powered. In essence, this "battery-backed" RAM (BB-RAM) obviates the need to write to truly persistent storage, such as flash memory—which is a slow and thus relatively power-hungry operation. WearDrive guarantees data integrity by tracking the amount of BB-RAM in use, which might span noncontiguous physical memory pages. When the battery level reaches a threshold that indicates that power is running out, WearDrive stops using BB-RAM and treats all RAM as volatile. At this point, data is written to local flash memory to ensure durability. Removing the battery while the system is running would of course lead to data loss, but most wearables' batteries aren't removable.

Many of today's wearable applications span both the wearable and a companion phone app, often leveraging the phone's large display, cellular connectivity, or data-processing capabilities to complement the capabilities of the wearable itself. We leverage this symbiotic relationship to provide further energy savings for the wearable (see Figure 4). The power required by the wearable to perform wireless transfer data is less than would be required to store the data in flash memory. Essentially, WearDrive exploits the battery in a nearby phone to provide durability for the data in the wearable's BB-RAM storage. Data is asynchronously transmitted to the phone, which ultimately performs the energy-intensive operations of encrypting and storing that data within its local flash, or uploading it to the cloud.

Connectivity between the wearable and the phone can be provided using Bluetooth low energy (BLE) or Wi-Fi. In WearDrive, we use the former for signaling and for short, infrequent data transfers. However, BLE isn't particularly energy efficient for large data transfers, so we complement its use with Wi-Fi direct; in this case, data transmission is initiated over BLE, with both devices switching over to Wi-Fi.

To further improve energy efficiency, WearDrive uses the BB-RAM approach on the phone as well as the wearable. Data in the wearable's BB-RAM is

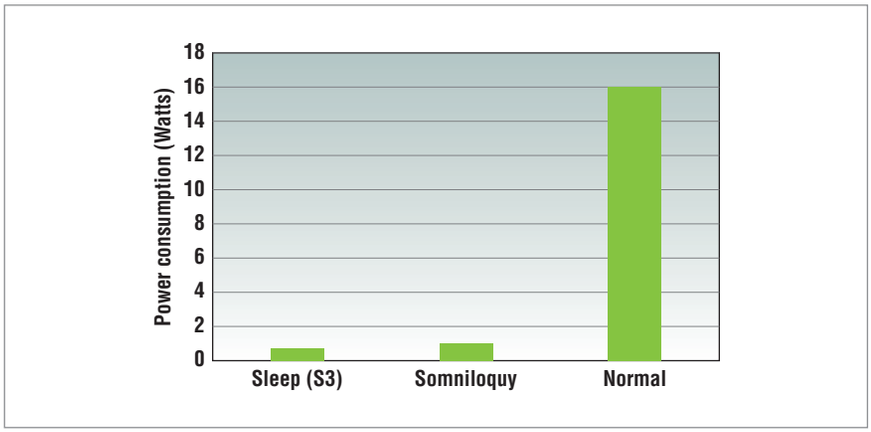

Figure 2. Power consumption of a Lenovo X60 using Somniloquy. Somniloquy provides increased functionality of filtering network packets that is not possible in the S3 sleep state, even when wake-on-network is available. The power saving compared with supporting the same functionality on the main CPU is significant.

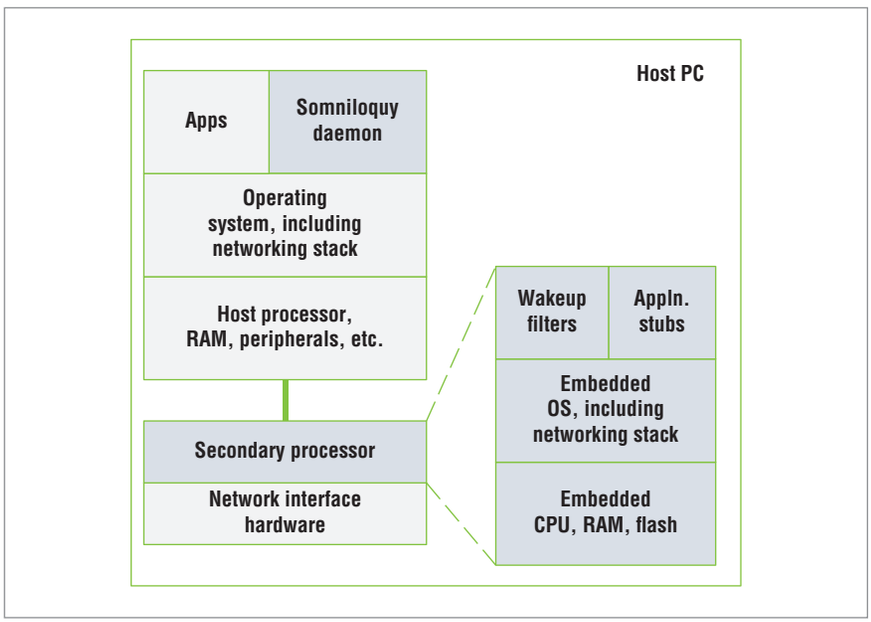

Figure 3. New components, shown in blue, must be added to a mobile device to implement Somniloquy. In addition to the low-power secondary processor and associated embedded operating system, wakeup filters and application stubs interact with the main processor via a Somniloquy daemon.

SPOTLIGHT

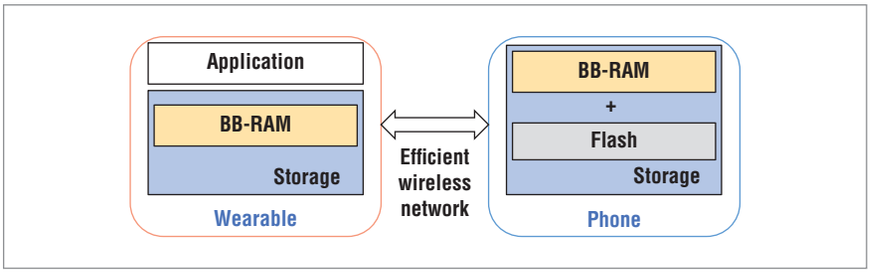

Figure 4. The WearDrive architecture. Essentially, WearDrive exploits the battery in a nearby phone to provide durability for the data in the wearable's "battery-backed" RAM (BB-RAM) storage.

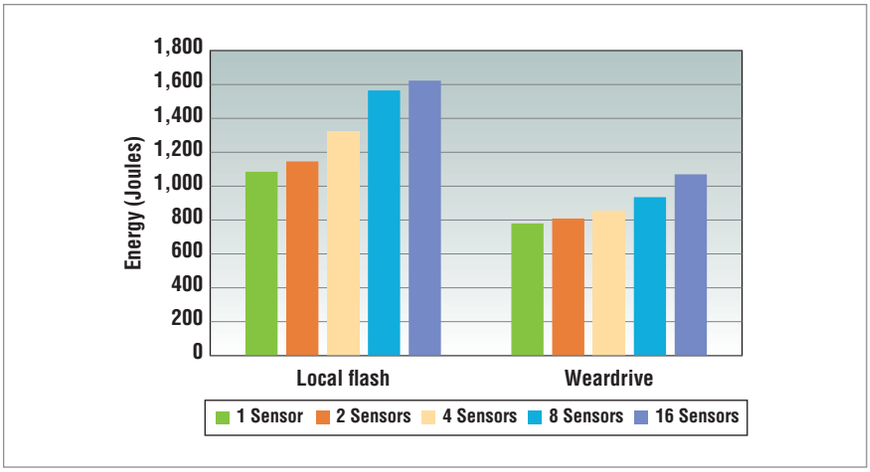

Figure 5. Energy used when storing to local flash memory vs. using WearDrive, based on between 1 and 16 sensors generating data continuously at 1 Hz for 24 hours. A typical smartwatch battery stores between 3000—6000 Joules of energy.

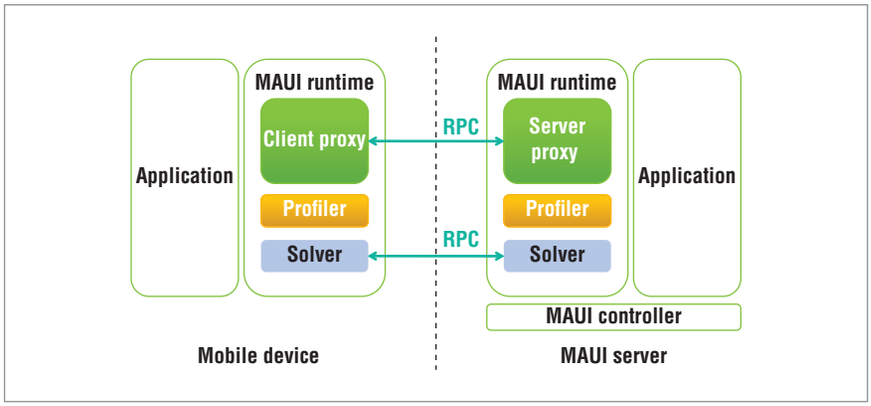

Figure 6. The MAUI (Mobile Assistance Using Infrastructure) architecture. The runtime on the phone dynamically determines if it is more power efficient to offload a function call to the server. The server could be a nearby device or in the cloud.

periodically migrated to the phone's BB-RAM and flushed when the wearable needs to shut down or the battery is depleted. Likewise, the phone uses its flash memory as the secondary storage for its data in BB-RAM.

Our experiments with WearDrive show that the BB-RAM approach—where writing to flash memory is delayed until battery capacity falls to a certain threshold—is 1.2–1.6 times more energy efficient than existing solutions that write to flash memory independent of battery capacity. When used in conjunction with offloading to a nearby phone, WearDrive has even greater benefits, reducing energy consumption of the wearable by a factor of up to 15, with minimal impact on power consumption on the phone for realistic workloads. Figure 5 highlights the energy savings for one specific scenario where the wearable device is continuously capturing sensor data which needs to be securely stored in persistent memory. WearDrive also lets the wearable be designed with less RAM and slower and smaller flash memory, thereby reducing its cost.

OFFLOADING TO A CLOUD OR CLOUDLET

A third approach we've explored to improve the battery life of mobile devices is to opportunistically offload computation, either to a nearby device or to the cloud. We built a Mobile Assistance Using Infrastructure (MAUI)[3] system based on two key insights. First, executing a task on a more powerful device takes less time, thereby consuming less energy. Second, the mobile device can go to a low power state while the task is being executed at a remote server. Both of these result in a lower-latency user experience and an improved battery life.

However, it might not always be energy efficient to offload computation to the cloud. For example, when connectivity is poor, it might consume more energy to offload an operation with its associated data. To determine when it's more energy efficient to offload versus compute locally, MAUI profiles the code contained in functions that are used by the application and the corresponding data requirements using

a component called the MAUI Profiler. The Profiler also monitors the network conditions and feeds this information to a Solver, which dynamically determines if a particular function should be run locally or remotely based on current conditions. Finally, the MAUI Proxy serializes function calls as necessary so that they can run at the appropriate location. The system architecture of MAUI is shown in Figure 6.

A key question addressed by MAUI concerns managing this computational offload in such a way that it is lightweight enough for the application developer to manage while remaining seamless to the user. To address this, we let a developer tag a function as "remote-able." This ensures that functions that access local peripherals, such as a GPS, cannot be run remotely. Then the Solver in the MAUI runtime dynamically decides if it's more energy efficient to offload a method to a remote server.

A key benefit of the MAUI approach is that it adapts to the operating environment. For example, if the user is playing a game and has a good network connection, then the intensive computation to determine the opponent's move might be offloaded to a remote server to minimize overall latency and power consumption and provide the best gaming experience. However, if the user has a slow network connection, then the computation might run locally at the expense of power and less effective gameplay.

Increasingly, techniques such as the ones we've discussed here are being used in commercial solutions. For example, many modern smartphone SoCs have a low-power core[4] that can act as a secondary "sensor" processor that wakes the main CPU from a sleep state only when necessary, in a manner analogous to that of Somniloquy. This enables a variety of applications ranging from activity monitoring to spoken keyword detection without significantly compromising battery life. Various forms of offloading computation to cloud processing services are also emerging, and we expect the use of cloud processing to become more dynamic, as envisaged in the MAUI system. Techniques such as WearDrive, which offload storage to a local device to improve battery life, are still predominately in the research stage.

A challenge that we hope to address in future work is the creation and adoption of easy-to-use primitives and APIs that let developers leverage the benefits of these systems reliably and seamlessly. Also challenging is the fact that power consumption in the network is still significant. For a fixed, wired network infrastructure, this is less of an issue, because power is more readily available. For wireless communications powered by the mobile device, we're looking forward to the roll-out of the impending 5G standard (see www.3gpp.org), which promises to offer a low-latency, high-bandwidth, and power-efficient last-mile link to the base station. This will make it extremely attractive to offload computation, storage, and potentially other tasks to the cellular base station and the cloud and is likely to make the offloading technologies increasingly relevant for improving the battery life of mobile devices. P

REFERENCES

1. Y. Agarwal et al., "Somniloquy: Augmenting Network Interfaces to Reduce PC Energy Usage," *Proc. 6th USENIX Symp. Networked Systems Design and Implementation* (NSDI), 2009, pp. 365–380.
2. J. Huang et al., "WearDrive: Fast and Energy-Efficient Storage for Wearables," *Proc. 2015 USENIX Conf. on Usenix Annual Technical Conference* (ATC), 2015, pp. 613–625.
3. E. Cuervo et al., "MAUI: Making Smartphones Last Longer with Code Offload," *Proc. 8th Int'l Conf. Mobile Systems, Applications, and Services* (MobiSys), 2010, pp. 49–62.
4. R.C. Johnson, "iPhone 5-Like Motion Processor for Any Mobile Device," *EE Times*, 12 Sept. 2013; www.eetimes.com/document.asp?doc_id=1319462.

Ranveer Chandra is a principal researcher at Microsoft Research. Contact him at ranveer@microsoft.com.

Steve Hodges is a principal researcher at Microsoft Research, where he leads the Sensors and Devices research group. He is also a Visiting Professor at the School of Computing Science, Newcastle University. Contact him at steve.hodges@microsoft.com.

Anirudh Badam is a researcher at Microsoft Research. Contact him at anirudh.badam@microsoft.com.

Jian Huang is a graduate student at Georgia Tech. Contact him at jian.huang@gatech.edu.