Somniloquy: Augmenting Network Interfaces to Reduce PC Energy Usage

Yuvraj Agarwal[‡], *Steve Hodges*[†], *Ranveer Chandra*[†], *James Scott*[†], *Paramvir Bahl*[†], *Rajesh Gupta*[‡]

[†]*Microsoft Research,* [‡]*University of California, San Diego*
[‡]*yuvraj@cs.ucsd.edu,* [†]{*shodges, ranveer, jws, bahl*}*@microsoft.com,* [‡]*gupta@cs.ucsd.edu*

Abstract

Reducing the energy consumption of PCs is becoming increasingly important with rising energy costs and environmental concerns. Sleep states such as S3 (suspend to RAM) save energy, but are often not appropriate because ongoing networking tasks, such as accepting remote desktop logins or performing background file transfers, must be supported. In this paper we present *Somniloquy*, an architecture that augments network interfaces to allow PCs in S3 to be responsive to network traffic. We show that many applications, such as remote desktop and VoIP, can be supported without application-specific code in the augmented network interface by using application-level wakeup triggers. A further class of applications, such as instant messaging and peer-to-peer file sharing, can be supported with modest processing and memory resources in the network interface. Experiments using our prototype Somniloquy implementation, a USB-based network interface, demonstrates energy savings of 60% to 80% in most commonly occuring scenarios. This translates to significant cost savings for PC users.

1 Introduction

Many personal computers (PCs) remain switched on for much or all of the time, even when a user is not present [23], despite the existence of low power modes, such as sleep or suspend-to-RAM (ACPI state S3) and hibernate (ACPI state S4) [1]. The resulting electricity usage wastes money and has a negative impact on the environment.

PCs are left on for a variety of reasons (see Section 2), including ensuring remote access to local files, maintaining the reachability of users via incoming email, instant messaging (IM) or voice-over-IP (VoIP) clients, file sharing and content distribution, and so on. Unfortunately, these are all incompatible with current power-saving schemes such as S3 and S4, in which the PC does not respond to remote network events. Existing solutions for sleep-mode responsiveness such as Wake-On-LAN (WoL) [18] have not proven successful "in the wild" for a number of reasons, such as the need to modify application servers or configure network hardware. A few initial proposals suggest the use of network proxies [4, 7, 11] to perform lightweight protocol functionality, such as responding to ARPs. However, such a system too requires significant modifications to the network infrastructure, and to the best of our knowledge such a prototype has not been described in published form (see Section 6 for a full discussion).

In this paper, we present a system, called Somniloquy[1], that supports continuous operation of many network-facing applications, even while a PC is asleep. Somniloquy provides functionality that is not present in existing wake-up systems. In particular, it allows a PC to sleep while continuing to run some applications, such as BitTorrent and large web downloads, in the background. In existing systems, these applications would stop when the PC sleeps.

Somniloquy achieves the above functionality by embedding a low power secondary processor in the PC's network interface. This processor runs an embedded operating system and impersonates the sleeping PC to other hosts on the network. Many applications can be supported, either with or without application-specific code "stubs" on the secondary processor. Applications simply requiring the PC to be woken up on an event can be supported without stubs, while other applications require stubs but in return support greater levels of functionality during the sleep state.

We have prototyped Somniloquy using a USB-based low power network interface. Our system works for desktops and laptops, over wired and wireless networks, and is incrementally deployable on systems with an existing network interface. It does not require any changes to the operating system, to network hardware (e.g. routers), or to remote application servers. We have implemented support for applications including remote desktop access, SSH, telnet, VoIP, IM, web downloads

[1] somniloquy: the act or habit of talking in one's sleep.

and BitTorrent. Our system can also be extended to support other applications. We have evaluated Somniloquy in various settings, and in our testbed (Section 5) a PC in Somniloquy mode consumes 11x to 24x less power than a PC in idle state. For commonly occurring scenarios this translates to energy savings of 60% to 80%.

We make the following contributions in this paper:

- We present a new architecture to significantly reduce the energy consumption of a PC while maintaining network presence. This is accomplished without changes in the network infrastructure.
- We show that several applications — BitTorrent, web downloads, IM, remote desktop, etc. — can consume much less energy. This is achieved without modifying the remote application servers.
- We present and empirically validate a model to predict the energy savings of Somniloquy for various applications.
- We demonstrate the feasibility of Somniloquy via a prototype using commodity hardware. This prototype is incrementally deployable, and saves significant energy in a number of scenarios.

2 Motivation

Prior studies have shown that that users often leave their computer powered on, even when they are largely idle [4]. A study by Roberson et. al. [23] shows that in offices, 67% of desktop PCs remain powered on outside work hours, and only 4% use sleep mode. In home environments, Roth et. al. [24] show that average residential computer is on 34% of the time, but is not being actively used for more than half the time.

To uncover the reasons why people do not use sleep mode, we conducted an informal survey. We passed it among our contacts who in turn circulated it further. We had 107 respondents from various parts of the world, of which 58 worked in the IT sector. 30% of the respondents left at least one machine at home on all of the time, and 75% of the respondents left at least one work machine on even when no one was using it.

Among the people who left their home machine powered on, 29% did so for remote access, 45% for quick availability and 57% for applications running in the background, of which file sharing/downloading (40%) and IM/e-mail (37%) were most popular. In the office environment, 52% of respondents left their machines on for remote access, and 35% did so to support applications running in the background, of which e-mail and IM were most popular (47%).

Although this survey should not be regarded as representative of all users, and is not statistically significant, it does highlight two important points. First, a number of PCs don't go to sleep even when they are unused. Second, significant energy savings can be achieved if only a few applications — remote reachability, file sharing, file downloads, instant messaging, e-mail — can be handled when the PC is asleep.

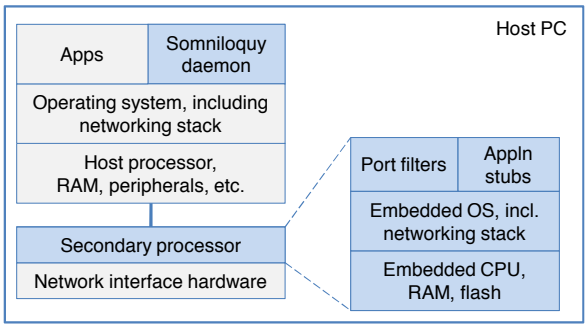

Figure 1: *Somniloquy augments the PC network interface with a low power secondary processor that runs an embedded OS and networking stack, network port filters and lightweight versions of certain applications (stubs). Shading indicates elements introduced by Somniloquy.*

3 The Somniloquy Architecture

Our primary aims during the development of Somniloquy were:

- to allow an unattended PC to be in low power S3 state while still being available and active for network-facing applications as if the PC were fully on;
- to do so without changing the user experience of the PC or requiring modification to the network infrastructure or remote application servers.

We accomplish these goals by augmenting the PC's network interface hardware with an always-on, low power embedded CPU, as shown in Figure 1. This *secondary processor* has a relatively small amount of memory and flash storage [2] which consumes much less power than if it were sharing the larger disk and memory of the host processor. It runs an embedded operating system with a full TCP/IP networking stack, such as embedded Linux or Windows CE. The flash storage is used as a temporary buffer to store data before the data is transferred in a larger chunk to the PC. A larger flash on the secondary processor allows the PC to sleep longer (Section 3.2. This architecture has a couple of useful properties. First, it does not require any changes to the host operating system, and second, it can be incrementally deployed on existing PCs using a peripheral network interface (Section 4).

[2] Our prototype had 64 MB DRAM and 2 GB of flash.

The software components of Somniloquy and their interactions are illustrated in Figure 2. The high-level operation of Somniloquy is as follows: When the host PC is powered on, the secondary processor does nothing; the network stack on the host processor communicates directly with the network interface hardware. When the PC initiates sleep, the Somniloquy daemon on the host processor captures the sleep event, and transfers the network state to the secondary processor. This state includes the ARP table entries, IP address, DHCP lease details, and associated SSID for wireless networks i.e. MAC- and IP-layer information. It also includes details of what events the host should be woken on, and application-specific details such as ongoing file downloads that should continue during sleep. Following the transfer of this information to the secondary processor, the host PC enters sleep.

Although the host processor is asleep, power to the network interface and the secondary processor is maintained [1]. To maintain transparent reachability to the host while it is asleep, the secondary processor impersonates the host by using the same MAC and IP addresses, host name, DHCP details, and for wireless, the same SSID. It also handles traffic at the link and network layers, such as ARP requests and pings – thereby maintaining basic presence on the network. New incoming connection requests for the host processor are now received and handled by the network stack running on the secondary processor. In this way the PC's transition into sleep is transparent to remote hosts on the network.

To ensure that the host PC is reachable by various applications, a process on the secondary processor monitors incoming packets. This process watches for patterns, such as requests on specific port numbers, which should trigger wake-up of the host processor. Although, this simple architecture [4, 7, 11] supports several applications with minimal complexity, Somniloquy can get much greater energy savings for some applications by not waking up the host processor for simple tasks, for example, to send instant messenger presence updates. To perform these tasks on the secondary processor, we require the application writer to add a small amount of application specific code ("stubs") on the host and secondary processor. In the rest of this section we describe in more detail how we handle various applications – with and without application stubs.

3.1 Somniloquy without Application Stubs

The Somniloquy daemon on the host processor specifies packet filters, i.e. patterns on incoming packets, on which the secondary processor should wake up the host processor from sleep state. The Somniloquy daemon creates filters at various layers of the network stack. At the link layer and network layer, the secondary processor can

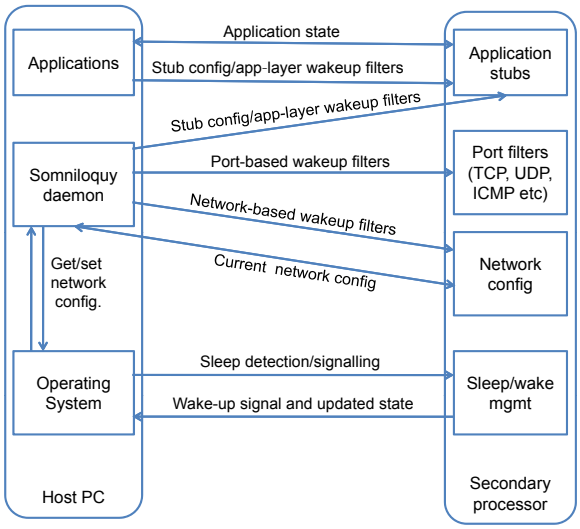

Figure 2: *Somniloquy software components on the host PC and the secondary processor, and their interactions.*

be told to wake the computer when it detects a particular packet, analogously to the magic packets used by Wake on LAN, though not requiring the MAC address to be known by the remote host (see further discussion in Section 6). Trigger conditions at the transport layer may also be specified, for example, wake on TCP port 23 for telnet requests. Similarly, Somniloquy also supports wake-ups on patterns in the application payload.

Although the host PC will wake up within a few seconds, it will not receive the packet(s) that triggered the wake-up. One way to solve this problem is to buffer the packet on the secondary processor and replay it on the network stack of the host processor once it has woken up. However, since the time to wake up is just a few seconds, most sources can be relied upon to retry the connection request. For example, any protocol using TCP as the transport layer will automatically retransmit the initial SYN packet. Even UDP-based applications that are designed for Internet use are designed to cope with packet loss using automatic retransmissions.

This simple packet filter based approach to triggering wake-ups has the advantage that application-specific code does not need to be executed on the secondary processor. Nonetheless, it is sufficient to support many applications that get triggered on remote connection requests, such as remote file access, remote desktop access, telnet and ssh requests to name a few.

3.2 Application-specific Extensions

Several applications maintain active state on the PC even when it is idle, and hence prevent a PC from going to sleep. For example, a movie download client on a home

PC (e.g. from Netflix) will require the host PC to be awake for a few hours while downloading the movie. An instant messenger (IM) client will require the PC to be on in order for the user to stay "online" (reachable) to their contacts.

Somniloquy provides a way for these applications to consume significantly less power. By performing lightweight operations on the secondary processor, it can opportunistically put the host processor to sleep. For example, the secondary processor can send and receive presence updates to/from the IM server while the host processor is asleep. During a large download, the secondary processor can download portions of the file, putting the host processor to sleep in the meantime.

The key to supporting these applications is the use of stubs that run on the host and the secondary processor. We have implemented stubs for three popular applications – IM (MSN, AOL, ICQ), BitTorrent, and web download. Here, we will describe the general guidelines for writing these stubs, and describe the specific implementations for the three applications in Section 4.

Writing application stubs: When designing an application stub, the first step is to understand the subset of the application's functionality that needs to run when the PC is asleep. This is implemented as a stub on the secondary processor. For example, for an IM stub, the functionality to send and receive presence updates is essential to maintain IM reachability. However, the stub need not include any UI-related code – such as opening a chat window.

We note that it is not feasible for the stub to reuse the entire original application code from the host PC. The application code might depend on drivers (display, disk, etc.) that are absent on the secondary processor. Furthermore, running the entire application might overload the secondary processor. Therefore, only the essential components of the application are implemented as part of the application stub.

Another step in designing application stubs is to decide when to wake up the host processor. Triggers can be user-defined, for example waking up on an incoming call from a specific IM contact. Triggers may also occur when the secondary's processor's resources are insufficient, for example when the flash is full or more CPU resources are needed. In all of these cases, the stub wakes up the host processor.

To interface with the application on the host PC and the Somniloquy daemon, the application stub needs to have a component on the host processor. This component registers two callback functions with the Somniloquy daemon — one that is called just before the PC goes to sleep and the other just after it has woken up. The first function transfers the application state to the stub on the secondary processor, and also sets the trigger conditions on which to wake the host processor. These values depend on the application being handled by the stub. The second callback function, which is called when the host resumes from sleep, checks the event that caused the wakeup — whether it was caused by a trigger condition on the secondary processor or due to user activity. It handles these events differently. If the wakeup was caused by user activity, the stub transfers state from the secondary processor, and disables it. However, if the wakeup was caused by a trigger condition on the secondary processor, the application stub handles it as defined by the user. For example, for an incoming VoIP call, the stub engages the incoming call functionality of the VoIP application.

Having determined what functionality needs to be supported by the application stub and host-based callbacks, and what state must pass between them, the final step is to implement this. We have used two manual approaches to doing this. For the download stub, we built all the functionality ourselves based on detailed knowledge of the application protocols, and for the BitTorrent and IM stubs, we trimmed down existing application code to reduce memory and CPU footprint. An alternative could be to automatically learn protocol behavior to build these application stubs. However, we believe that this is an extremely difficult problem. There are parts of the application that are difficult to infer, and any inaccuracy in the application stub will make it unusable. For example, knowledge of how BitTorrent hashes the file blocks is necessary for the stub to successfully share a file with peers. We are unaware of any automatic tool that can learn such application behavior. Therefore, we believe that the best (although perhaps not the most elegant) approach to building these stubs is to modify application source code and remove functionality that is not required by the secondary processor. In the future, with a greater incentive to save energy, we expect that application developers will compete for energy consumption, and hence provide stubs for their applications using the guidelines described in this section.

We realize that partial application stubs might be created using tools such as the Generic Application-Level Protocol Analyzer [6] and Discoverer [8], which automatically learn the behavior and message formats for a range of protocols. As part of future work, we plan to explore how the knowledge of the protocol can be augmented with application-specific behavior to ease the development of application stubs.

When to use application stubs? Not all applications are conducive to low-power operation via application stubs. A CPU intensive application, such as a compilation job, will be very slow on the secondary processor since it has a less powerful CPU and low memory. Similarly, an I/O intensive application, such as a disk indexer, will need to read the disk very often and will therefore

need the PC to be awake. Download and file sharing applications are an interesting exception, because portions of a file can be transferred by the secondary processor whilst the host sleeps. We will discuss this approach in more detail in Section 4.4.

Even for an application stub that saves energy for a given application, it is not always useful to offload the application to the secondary processor when the host PC is going to sleep. Several other applications may also want to run their application stubs on the secondary processor. This might overload the CPU of the (weaker, low power) secondary processor. In this case, it might be beneficial to keep the host PC awake.

One way to solve this problem is to modify the Somniloquy daemon to predict the CPU utilization of the stubs for all applications that are willing to be offloaded to the secondary processor. However, making this prediction is extremely difficult. There might be little correlation between the CPU utilization of the application on the host PC, and the stub on the secondary processor, because of different processor architectures, and varying application demands. Instead, we take a systems approach. We monitor the CPU utilization of the secondary processor; if it remains at more than 90% continuously(>30 seconds), we wake up the PC, and resume all applications on the host processor. If the CPU utilization of these applications decreases by more than 10% on the host processor, we repeat the same procedure — offload to the secondary processor and stay there if CPU utilization is less than 90%. In our Somniloquy deployment the need to move applications arose when running multiple application stubs on the secondary processor, such as two concurrent 8 Mbps web downloads and two concurrent BitTorrent downloads of Section 5.3.2.

Incremental Deployment: We realize that Somniloquy may never be universally deployed, and that getting software vendors to try for incremental deployment requires a low-effort mechanism to ensure that their Somniloquy-enhanced software is compatible with machines and platforms that do not have Somniloquy support. The Somniloquy daemon queries the OS to determine the presence of a secondary processor, and the supported application stubs. Applications then need to query the Somniloquy daemon, and invoke the application stubs only if the OS supports Somniloquy, and the corresponding stubs are implemented on the secondary processor.

3.3 Quantifying Energy Savings

The amount of energy saved through adoption of Somniloquy is quite easy to predict; it depends on the relative power consumption of the awake and sleep states, and the proportion of time that a machine can be kept asleep when it would previously have been awake. For applications without stubs, this proportion is largely dependent on the actions of a remote user - how frequently a remote ssh session is initiated for example, and for how long. On the other hand, for applications with stubs the secondary processor may regularly wake up the host to perform some task or other. We quantify the energy savings for an application with different wake-up intervals in Section 5.4.4.

More formally, suppose the host is woken up once every T_{sleep} seconds, whereupon it stays awake for T_{awake} seconds. T_{awake} includes the time it takes to transfer data between the PC and the secondary processor. Also assume that d is sum of the time to wake up the host plus the time to transition to sleep. Suppose:

- P_a is the power consumption of the PC when it is awake (in W)
- P_s is power consumed in sleep mode (in W), and
- P_e is power consumed by the secondary (embedded) processor (in W)

The energy (E) consumed during Somniloquy operation is given by:

$$\begin{aligned} E_{somniloquy} &= E_{PCinSleepMode} + E_{PCinAwakeMode} \\ &\quad + E_{SecondaryProcessor} \\ &= T_{sleep} * P_s + (T_{awake} + d) * P_a \\ &\quad + (T_{awake} + d + T_{sleep}) * P_e \; Joules \end{aligned}$$

In the absence of Somniloquy, the amount of energy consumed by the host PC in the same time is $E_{host} = P_a * (T_{awake} + T_{sleep})$ Joules. Therefore, the ratio of energy consumed by Somniloquy compared to the host PC being always on is given by:

$$\frac{E_{somniloquy}}{E_{host}} = \frac{T_{sleep}*(P_e+P_s)+T_{awake}*(P_a+P_e)+d*(P_a+P_s)}{P_a*(T_{awake}+T_{sleep})}$$

Typically, as we show in Section 5, P_e and P_s are two orders of magnitude less than P_a for a desktop computer, and d is around 10 seconds (to wake up the host, and put it back to sleep). Therefore, for most energy savings, we would want T_{awake} to be much less than T_{sleep}, i.e. if $T_{awake} \ll T_{sleep}$, then the ratio $E_{somniloquy}/E_{host}$ is approximately $(P_e + P_s)/P_a$. We will present the approximate energy savings for different applications in Section 4.

Of course, Somniloquy could save more energy by disabling the secondary processor when the PC is awake. This would require the PC to enable the secondary processor before going to sleep, and disable it when the PC has woken up. We were unable to fully implement this functionality in our prototype, but we expect this to be a minor fix in a production system.

3.4 Discussion

Security: A common requirement of corporate IT departments is that all PCs should be up to date with the latest OS and application patches. Somniloquy can ensure that this constraint is met even when PCs are asleep. This is achieved using a port-based trigger to wake up the host PC when the SMS (Systems Management Server) contacts the host PC to install updates.

Somniloquy ensures that the secondary processor is secure by patching its OS whenever security updates become available. Also, it prevents attackers from replacing the secondary processor by requiring that it be a physically part of the PC (as part of the network interface). In some cases however, the functionality that Somniloquy provides could be misused to conduct attacks that spuriously wake up the PC and waste energy. This kind of denial-of-service attack would be particularly effective for mobile devices where a drained battery might result. One way to address this issue is to disable port triggers, and instead exclusively use application stubs which ensure that only authenticated remote hosts are allowed to trigger wakeup.

Another concern is that application stubs, and hence the use of extra code, increases the PC's attack surface. To mitigate the impact of this vulnerability we use a few techniques. First, the secondary processor only listens on ports that have been opened by applications on the host PC. Second, we require the PC and the secondary processor to be on the same administrative domain.

We also note that modern processors have additional security features built in, for example an execute-disable bit, used by some applications to prevent executing arbitrary code and preventing buffer overflows. We realize that a low power processor may not currently support this advanced functionality, although we expect that in the future low-power chips will also be available with these features.

Alternative Design: With the increasing prevalence of multi-core PCs, one idea to alleviate the need for the additional secondary processor introduced by Somniloquy would be to use one of the cores of the host CPU instead. Running just one core at the lowest possible clock frequency would minimize energy consumption and obviate the need for a separate low power processor in the NIC.

However, it turns out that such an approach is not useful without significant modification to today's PC architecture. Our measurements (see Section 5.1) show that the power consumption of a multi-core PC with only one core active, running at the lowest permissible clock speed is still approximately 50 times that of our low power secondary processor, even with all other peripherals in their lowest power modes – e.g. disk spun down. This is because of the lack of truly fine-grained power control of PC components such as the Northbridge, Southbridge, memory buses, parts of the storage hierarchy and various peripherals. Even if fine-grained control were available, the base power consumption of individual components (NIC, hard drive) is significant (see Table 2). One way to reduce this base power draw would be to have a separate and relatively simple core with a small amount of associated memory running from a separate power domain so that it can function without powering on other components. Such an architecture is very similar to Somniloquy, and most of our design principles can easily be adopted.

4 Prototype Implementation

We have prototyped Somniloquy using *gumstix*, a low power modular embedded processor platform manufactured by Gumstix Inc that support a wide variety of peripherals.

4.1 Hardware and Software Overview

An important goal when prototyping Somniloquy was to have it work with existing unmodified desktops and laptops, and for both wired and wireless networks. Furthermore, we required the platform to be low power, have a small form factor, and be well supported for development. The gumstix platform served all these design requirements well. The specific components we use for Somniloquy include a connex-200xm processor board, an etherstix network interface card (NIC) (for wired Ethernet), a wifistix NIC (for Wi-Fi), and a thumbstix combined USB interface/breakout board. The connex-200xm employs a low power 200 MHz PXA255 XScale processor, with 16 MB of non-volatile flash and 64 MB of RAM. The etherstix provides a 10/100BaseT wired Ethernet interface plus an SD memory slot to which we have attached a 2GB SD card. The thumbstix provides a USB connector, serial connections and general purpose input and output (GPIO) connections from the XScale.

To enable Somniloquy we needed mechanisms to wake-up the host PC, and also to detect its state (awake or in S3). To achieve this we added a custom designed circuit board that incorporates a single chip — the FT232RL from FTDI. The FT232RL is a USB-to-Serial converter chip supporting functionality such as sending a resume signal to the host and detecting the state of the host, both over the USB bus. This board is attached to the computer via a second USB port and to the thumbstix module (and thence to the XScale processor) via a two-wire serial (RS232) interface plus two GPIO lines. One GPIO line is connected to the FT232RL's 'ring indicator' input to wake up the computer. The second GPIO

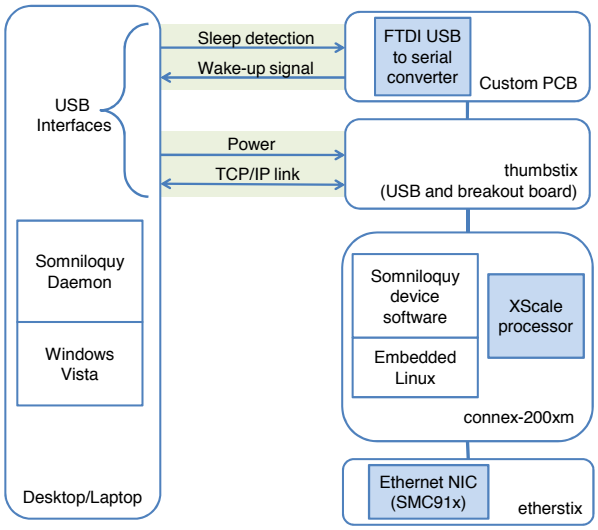

Figure 3: *Block diagram of the Somniloquy prototype system - Wired-1NIC version. The figure shows various components of the gumstix and the USB interfaces to the host laptop.*

line is connected to the FT232RL's 'sleep' output which can be polled by the gumstix to detect whether the host PC is active or in S3.

As mentioned above (and shown in Figure 3), the computer is connected to the secondary processor via two USB connections. One of these provides power and two-way communications between the two processors. It is configured to appear as a point-to-point network interface ("USBNet"), over which the gumstix and the host computer communicate using TCP/IP. The second USB interface provides sleep and wake-up signaling, and a serial port for debugging purposes. The use of two USB interfaces is not a fundamental requirement, it is simply for ease of prototyping.

Since we use standard USB ports for interfacing with the host and for sleep signaling, our prototype works on any recent desktop or laptop that supports USB. We run an embedded distribution of Linux on the gumstix that supports a full TCP/IP stack, DHCP, configurable routing tables, a configurable firewall, SSH and serial port communication. This provides a flexible prototyping platform for Somniloquy with very low power operation.

We have implemented the Somniloquy host software on Windows Vista. The Somniloquy daemon detects transition to S3 sleep state, and before this is allowed to occur we transfer the network state (MAC address, IP address, and in the case of the wireless prototype, the SSID of the AP) and other information about the wakeup triggers as discussed in Section 3.

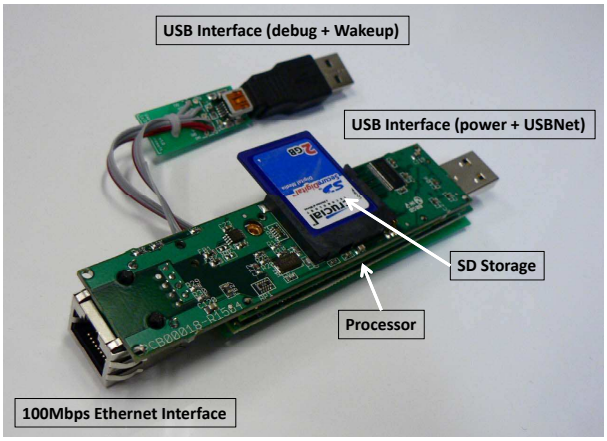

Figure 4: *Photograph of the gumstix-based Somniloquy prototype - Wired-1NIC version.*

4.2 Three different prototypes

We have prototyped three different Somniloquy designs to explore different aspects of operation. The first uses the gumstix as an augmented Ethernet interface, as described in Section 3. However, in our prototype this has some performance limitations so we have also implemented a second design which uses the gumstix in cooperation with an existing high-speed Ethernet interface. Finally, we have a Wi-Fi version. All three prototypes are described in further detail below:

Augmented Network Interface: We call this implementation the *Wired-1NIC* version. The architecture is shown in Figure 3, with a photograph of the prototype shown in Figure 4. In this prototype, we disable the NIC of the host, and configure the PC to use the USBNet interface (USB connection between the gumstix and the host) as its only NIC. The gumstix is connected to the network using its Ethernet connection. To enable the host PC to be on the network, we set up a transparent layer-2 software bridge between the USBnet interface to the host and the Ethernet interface of the gumstix. This bridge is active when the host is awake. When the host transitions to sleep, the gumstix disables the bridge, and resets the MAC address of its Ethernet interface to that of the USBNet interface of the host. The gumstix thus appears to the rest of the network as the host itself, since it has the same network parameters (IP, MAC address). When the host wakes up, the gumstix resets its MAC address to its original value and starts bridging traffic to the host again.

Although our *Wired-1NIC* prototype hardware supports a 100 Mbps Ethernet interface, we are limited to a throughput of 5 Mbps due to the bandwidth supported by the USBNet interface driver. There is also a slight overhead of bridging traffic on the gumstix. Although this limits bandwidth to the host significantly in our prototype, we note that in a final integrated version, this over-

head of bridging can be avoided by allowing both the host and the low power secondary processor to access the NIC directly.

Using Existing Network Interface: Somniloquy can coexist with an existing NIC. On such systems, the overhead of bridging is avoided by using the existing Ethernet interface on the host PC for data transfer when it is awake, with the gumstix using its own Ethernet interface (while still impersonating the host PC) when the host is asleep. We have built this version where the gumstix does not perform Layer-2 bridging, and call it the *Wired-2NIC* prototype.

Using Wi-Fi: We have also implemented a wireless version of Somniloquy. We were unable to implement a one-NIC version since the Marvell 88W8385 802.11 b/g chipset present on the wifistix does not currently support layer 2 bridging. We have however implemented a *Wireless-2NIC* version.

4.3 Applications Without Stubs

We have implemented a flexible packet filter on the gumstix using the BSD raw socket interface to support applications that do not require stubs, e.g. RDP, SSH, telnet and SMB connections. Every application in this class provides a regular expression matched against incoming packets to decide whether to trigger host wakeup. For example, handling incoming remote desktop requests requires the host to be woken up when the gumstix receives a TCP packet with destination port 3389.

We note that waking up the host computer is not enough; the incoming connection request must somehow be conveyed to the host. We accomplish this by using the `iptables` firewall on the gumstix to filter any response to TCP or UDP packets that the gumstix does not handle itself. Thus trigger packets are not acknowledged by the gumstix and the remote client sends retries. After the host has resumed, one of the retries will reach it (since it is still using the same IP and MAC addresses) and it will respond directly. Using port-based filtering, we have implemented wake-up triggers for four applications: remote desktop requests (RDP), remote secure shell (SSH), file access requests (SMB), and Voice over IP calls (SIP/VoIP).

4.4 Applications Using Stubs

To demonstrate how modest application stubs can enable significant sleep-mode operation in Somniloquy, we have also implemented application stubs for three applications that were popular in our informal survey: background web download, peer to peer content distribution using BitTorrent, and instant messaging. For all these applications, we did not have to modify the operating system or the existing applications on the PC, which were only available to us in binaries. To capture the state of the application for the respective stub, we wrote wrappers around the binaries.

Background Web Downloads: We developed the web download stub for `wget` which works as follows: When the host PC transitions to sleep, the status of active downloads is sent to the stub running on the gumstix. The status includes the download URL, the offset of how much download has taken place, the buffer space available, and the credentials (if required for the download). Most popular web servers (e.g. IIS and Apache) allow these byte ranges to be specified using the HTTP 'Accept-Ranges' primitives [22]. The web download stub then resumes the downloads from the respective offsets of the files, and stores the data on the flash storage of the gumstix. If the flash memory fills up before the downloads complete, the stub wakes up the host PC and transfers the downloaded files from flash storage to the host PC, thereby freeing up space. The host PC then goes back to sleep while the stub continues the downloads. At the end of a download, the gumstix wakes up the host PC, and transfers the remaining part of the file.

The download stub consumes significantly less energy to download a file than keeping the PC awake to download it. The overhead is a slight increase in latency. We can quantify the savings and overhead using the model described in Section 3.3. If flash storage is F MB and the download bandwidth is B MBps, then the host PC is woken up every F/B seconds, and it is awake for F/T seconds, where T is the transfer rate between the host and the gumstix. Therefore, using the formula in Section 3.3, Somniloquy gives most energy savings at low B and high T. We empirically validate this observation in Section 5.4.4. When T is of the same order as B, Somniloquy might not save much energy. This can happen if the NIC supports very high rates (e.g. 1 Gbps), while the secondary processor can only support lower data rates (up to 100 Mbps) or if the transfer rate T is limited. However, we anticipate the download stub to be primarily used in scenarios where the download speeds are limited by the last mile connection of at most a few tens of Mbps – here, this stub is nearly always beneficial.

BitTorrent: For the BitTorrent stub we customized a console-based client, *ctorrent*, to run on the gumstix with a low CPU utilization and memory footprint. Prior to suspending to S3, the host computer transfers the '.torrent' file and the portion of the file that has already been downloaded to the gumstix. The BitTorrent stub on the gumstix then resumes download of the torrent file and stores it temporarily on the SD flash memory of the gumstix. When the download completes, the stub wakes up the host and transfers the file.

When only downloading content, the energy saved by

using this stub is similar to that of the web download stub, i.e., frequency of waking up the PC and the duration for which it is woken up depends on the download bandwidth B, the transfer speed T and the flash size F. However, when uploading/sharing (which is key to altruistic P2P applications), the energy savings are much more. The same file chunk can be uploaded to many peers, and hence the PC can sleep for much longer – implying more energy savings using the formula in Section 3.3.

Instant Messaging: For the IM stub, we used a console-only IM client called *finch* that supports many IM protocols such as MSN, AOL, ICQ, etc. On the PC, we used the corresponding GUI version of the IM client. To ensure our goal of a low memory and CPU footprint we customized finch to include only the features salient to our aim of waking up the host processor when an incoming chat message arrives. This only requires authentication, presence updates and notifications; we disabled other functionality. The host processor transfers over the authentication credentials for relevant IM accounts before going to S3. The gumstix then logs into the relevant IM servers, and when an incoming message arrives it triggers wakeup. The energy saved by the IM stub is thus similar to applications that are handled using packet filters (e.g. SSH/RDP), where the duration for which a host can sleep depends on the frequency of occurrence of wake-up triggers.

5 System Evaluation

We present the benefits of Somniloquy in four steps. First, we show that gumstix consumes much less power than a PC by profiling standalone desktops, laptops and the gumstix in different power states. Second, we measure the energy saved (and latency introduced) by Somniloquy when used on an "idle" host processor. Third, we show how Somniloquy affects the performance of various applications, with and without application stubs. Finally, we quantify Somniloquy's energy savings — monetary and environmental cost for an enterprise and battery lifetime increase for laptops.

Methodology: To measure the power consumption of laptops and desktop PCs, we used a commercially available mains power meter, *Watts-Up* [3]. To measure the power consumption of the standalone gumstix, we built a USB extension cable with a 100 mΩ 0.1% sense resistor, which was inserted in series with the +5 V supply line, and we used this cable to connect the gumstix to the computer. We calculated the power draw of the gumstix by measuring the voltage drop across the sense resistor. All power numbers presented in this section are averaged across at least five runs.

[3] `http://www.wattsupmeters.com/`

Condition	Optiplex 745	Dimension 4600
Normal idle state	102.1 W	72.7 W
Lowest CPU frequency	97.4 W	N/A
Disable multiple cores	93.1 W	N/A
'Base power'	93.1 W	72.7 W
Suspend state (S3)	1.2 W	3.6 W
Time to enter S3	9.4 s	5.8 s
Time to resume from S3	4.4 s	6.2 s

Table 1: *Power consumption and S3 suspend/resume time for two desktops under various operating conditions. In all cases the processor is idle and the hard disk is spun down. The power consumed by other peripherals such as displays is not included.*

Condition	Lenovo X60	Toshiba M400	Lenovo T60
Normal idle state	16.0 W	27.4 W	29.7 W
Backlight minimum	13.8 W	22.4 W	24.7 W
Screen turned off	11 W	18.3 W	21.3 W
'Base power'	11 W	18.3 W	21.3 W
Suspend state (S3)	0.74 W	1.15 W	0.55 W
Battery capacity	65 Wh	50 Wh	85 Wh
Base lifetime	5.9 h	2.7 h	4.0 h
Suspend lifetime	88 h	43 h	155 h
Time to enter S3	8.7 s	5.5 s	4.9 s
Time to resume from S3	3.0 s	3.6 s	4.8 s

Table 2: *Power consumption and battery lifetime of three laptops under various operating conditions, and the time to change power states.*

5.1 Microbenchmarks – Power, Latency

Desktops: Table 1 presents the average power consumption for two Dell desktop machines: an Intel dual core (2.4 GHz Core2Duo) OptiPlex 745 with 2 GB RAM running Windows Vista, and a 2.4 GHz Pentium 4 Dimension 4600 with 512 MB RAM running Windows XP. The display is turned off in these experiments, and only the essential system processes are left running. The power consumption of the desktop in S3 is two orders of magnitude less than when it is awake. This is consistent with prior published data on the power consumption of modern PCs [7]. We use the term 'base power' to indicate the lowest power mode that a PC can be in and still be responsive to network traffic (without using Somniloquy). To get this number, we further scaled down the CPU to the lowest permissible frequency on these desktops. Furthermore, we disabled the multi-core functionality using the system BIOS to effectively use only one core and verified that the system was actually doing so by using a processor ID utility supplied by Intel. The time taken for the desktops to resume from S3 and reconnect to the network is of the order of a few seconds (Table 1).

	Gumstix state	Power
	Wired version	
1	gumstix only - no Ethernet	210 mW
2	gumstix + Ethernet idle	1073 mW
3	gumstix + Ethernet bridging	1131 mW
4	gumstix + Ethernet + write to flash	1675 mW
5	gumstix + Ethernet broadcast storm	1695 mW
6	gumstix + Ethernet unicast storm	1162 mW
	Wireless version	
7	gumstix only – no Wi-Fi	210 mW
8	gumstix + Wi-Fi associated (PSM)	290 mW
9	gumstix + Wi-Fi associated (CAM)	1300 mW
10	gumstix + Wi-Fi broadcast storm	1350 mW
11	gumstix + Wi-Fi unicast storm	1600 mW

Table 3: *Power consumption for the gumstix platform in various states of operation.*

Laptops: Table 2 presents the average power consumption of three popular laptops: a Lenovo X60 tablet PC with 2 GB RAM running Windows Vista, a Toshiba laptop with 1 GB RAM running Windows XP, and a Lenovo T60 laptop with 1 GB RAM running Windows Vista. For all power measurements, the processor is set to the lowest speed and is idle, the hard disk is spun down and the wireless network interface is powered on. The base power is between 11 W and 22 W, resulting in a battery lifetime of around 4 to 5 hours with the batteries that are present on these laptops. Using the sleep/S3 state can dramatically extend the battery lifetime, to between 40 and 150 hours for the laptops we tested, although the laptop is unreachable in this state.

Gumstix: Table 3 shows the average power consumed by the gumstix (with both etherstix and wifistix) in various states of operation. The gumstix has a base power of approximately 210 mW when no network interface is present (row 1). A gumstix with an active network interface typically consumes approximately 1070-1300 mW (rows 2 and 9), however with an associated Wi-Fi interface in power save mode it consumes only 290 mW (row 8). The power consumption of the gumstix when its network interface is active and the downloaded data is being written to flash is around 1675 mW (row 4). Broadcast and unicast 'storms' (continuous traffic) increase the power consumption by a few hundred milliwatts[4]. Importantly, the power consumption of the gumstix is approximately one tenth that of an awake laptop in the lowest power state, and approximately 50 times less than an idle desktop.

[4]Wi-Fi broadcasts are sent at 6 Mbps while unicasts are sent at 54 Mbps in our setup. Consequently a unicast storm consumes more power than a broadcast storm.

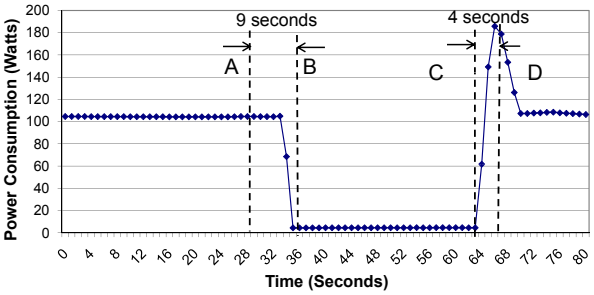

Figure 5: *Power consumption and state transitions for our desktop testbed.*

5.2 Somniloquy in Operation

We now report the power consumption of Somniloquy in operation. For these measurements we use two testbed systems: a desktop (Dell OptiPlex 745 with 2 GB RAM running Windows Vista) with the Wired-1NIC prototype of Somniloquy, and a laptop (Lenovo X60 tablet PC running Windows Vista) with the Wireless-2NIC version of Somniloquy. Thus, our tests span both Ethernet and Wi-Fi networks, and both the integrated single network interface, and the higher performance versions which uses the existing internal network interface. The test traffic is generated using a standard desktop machine running on the same (wireless or wired) LAN subnet as the testbed machine.

Figure 5 shows the power consumption of our desktop testbed. Initially the desktop's host processor is awake and uses the gumstix for bridging, and the whole system draws 104 W of power. At time 'A' a state change to S3 is initiated by the user. This request completes at time 'B' after which the power draw of the system is approximately 4.4 W, i.e. 24x less. This power is split between the gumstix, the DRAM of the PC, and other power chain elements in the PC. Subsequently at time 'C' the gumstix, which has been actively monitoring the network interface, wakes up the host in response to a network event. This request completes at time 'D' when the host system has fully resumed. As the figures illustrate this resume event takes about 4 seconds. We do not show the laptop figure for space reasons; the trace looks very similar with a starting power of 16 W with the screen on (which drops to 11 W if the screen is turned off), a power draw of 1 W when using Somniloquy (11x less than the screen-off case) and a resume time of 3 seconds.

5.3 Application Performance

As described earlier there are two classes of applications that are supported by Somniloquy: first, a large class of applications that do not require application stubs, and second a smaller class of applications that can be sup-

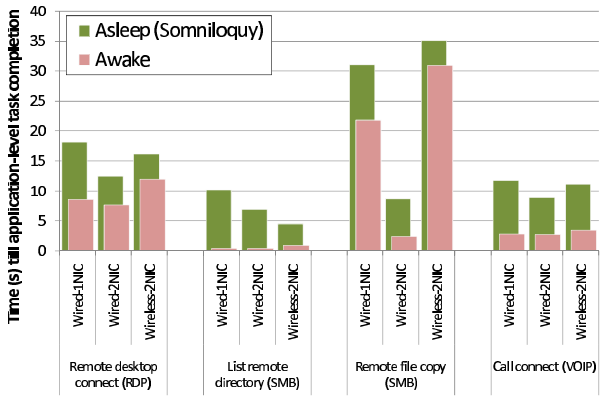

Figure 6: *Application-layer latency for three Somniloquy testbeds and four application types.*

ported using application stubs running on the gumstix. We performed a number of experiments to evaluate the performance of both these classes of applications.

5.3.1 Applications without stubs

We now quantify the end-to-end latency (as perceived by users) incurred by the applications that are handled by Somniloquy without using application stubs. For these experiments, we use the same two testbeds as above, with the addition of a third testbed based on the Wired-2NIC prototype (using same desktop machine as the Wired-1NIC case), providing a direct comparison between the 1NIC and 2NIC cases. In each case the latency reported is the mean over five test runs.

Figure 6 reports the time taken to satisfy an incoming application-layer request for four sample applications. For each application, we show the latency for "awake" operation (i.e. when the host is on and directly responds to the request) and when the host is in S3 and Somniloquy prototype receives the incoming packet and triggers wake-up of the host.

The four applications we tested were:

Remote desktop access (RDP): Here we used a stopwatch to measure the latency between initiating a remote desktop session to the host and the remote desktop being displayed. A stopwatch was used to ensure that true user-perceived latency was measured. The gumstix was configured to wakeup the main processor on detecting TCP traffic on port 3389 (the RDP port).

Remote directory listing (SMB): A directory listing from the Somniloquy testbed was requested by the tester machine (via Windows file sharing, which is based on the SMB protocol). The time between the request being initiated and the listing being returned was measured using a simple script. The secondary processor was configured to initiate wake-up on detection of traffic on either of the TCP ports used by SMB,i.e. ports 137 and 445.

Remote file copy (SMB): The SMB protocol was used again, but this time to transfer a 17 MB file from the Somniloquy testbed to the tester machine.

VoIP call (SIP): A Voice-over-IP call was placed to a user who had been running a SIP client on the Somniloquy laptop before it had entered S3. On receipt of the incoming call the SIP server responded with a TCP connection to the testbed, causing the gumstix to trigger wakeup. A similar procedure was used in [2]. Once again, the latencies were measured using a stopwatch to measure true user-perceived delay.

As Figure 6 shows, Somniloquy adds between 4-10 s latency in all cases. As described in Section 5.2 earlier, part of this latency is attributed to resuming from S3, i.e. 4-5 s for the desktop and 2-3 s for the laptop, and is independent of Somniloquy. Further latency is due to the delay for TCP to retransmit the request, and for the host to respond to the request (which may take longer since it has just resumed). Note that the Wired-1NIC prototype shows higher latency than the Wired-2NIC prototype. This is purely an artifact of our prototype caused by the overhead of MAC bridging and largely the slower speed of the USBNet IP link between the gumstix and the host. The latter is particularly obvious in the file copy test, where the file copy time with the Wired-2NIC case is much faster than for Wired-1NIC (although the Wired-1NIC speed is still faster than Wireless-2NIC). While Somniloquy does result in 4-10 s additional application-layer latency, these delays are acceptable for real usage (including VoIP [2]) in exchange for the substantial benefit of 20x-50x power savings.

5.3.2 Applications Requiring Stubs

In this section we present evaluations for applications that require stub support on the gumstix, primarily looking at the overhead in terms of memory consumption and processing capabilities that they impose on the gumstix. We have implemented application stubs for three common applications — background downloads using the http protocol, P2P file sharing using BitTorrent, and maintaining presence on IM networks — as described in Section 4.

To study the overhead of IM clients, we run the corresponding application stub using up to three different IM protocols simultaneously — MSN Messenger, AOL Messenger and ICQ Chat. Table 4 shows the processor utilization and memory footprint of the Wired-1NIC prototype when running these IM clients. Since the behavior of the IM stub is such that it maintains presence of the user on various networks and on receipt of an appropriate trigger (IM from someone) wakes up the host, the latency values are similar to those of the VoIP application

Accounts	Processor 95th percentile	Memory 95th percentile
None	0.0%	5.9 MB
MSN only	10.0%	6.5 MB
MSN+AOL	21.6%	6.7 MB
MSN+AOL+ICQ	26.0%	6.9 MB

Table 4: *Processor and memory utilization for the IM stub for various configurations. Total memory for the gumstix is 64 MB.*

Configuration	Processor 95th percentile	Memory 95th percentile
Single download		
4MB cache	16.0%	6.5 MB
8MB cache	16.0%	10.6 MB
16MB cache	16.1%	18.9 MB
Two simultaneous downloads (4 MB cache)		
1st download	16%	6.5 MB
2nd download	24%	7.0 MB

Table 5: *Processor and memory utilization for the BitTorent stub for various configurations. Total memory for the gumstix is 64 MB.*

as reported in Figure 6. For our Wired-1NIC prototype the additional latency for the IM stub when using Somniloquy is around seven seconds.

To evaluate the overhead of P2P file sharing using the BitTorrent stub on the gumstix, we initiated downloads using a torrent from a remote website[5] into the 2 GB SD card of the Wired-1NIC gumstix. We varied the memory cache available to the stub while conducting a single download, and then tested two simultaneous downloads. The results in Table 5 show that the memory footprint of the stub increases proportionally to the cache size as expected, while the processor utilization remains constant. When there are two simultaneous downloads, each instance of the stub uses memory proportional to its specified 4 MB cache.

Finally, to evaluate the web-download stub on the gumstix we initiate download of a large (300 MB) file from a local web server. We varied the throughput of the downloads and measured the processor utilization and the memory consumption of the gumstix, and experimented with two simultaneous downloads. As shown in Table 6, the processor utilization increases as the download rate increases although the memory footprint for each download remains constant.

The above results show that using application stubs, we can support fairly complex tasks and applications, including background web downloads and P2P file sharing using relatively modest resources on the gumstix. It is important to note that the power consumption of the gumstix did not exceed 2 W in all of these experiments.

[5] http://www.legaltorrents.com/

Configuration	Processor 95th percentile	Memory 95th percentile
Single download		
2Mbps	9.2%	1.8 MB
4Mbps	21%	1.8 MB
8Mbps	50%	1.8 MB
Two simultaneous downloads (4 Mbps each)		
1st download	31%	1.8 MB
2nd download	26.3%	1.8 MB

Table 6: *Processor and memory utilization for the web download stub for various configurations. Total memory for the gumstix is 64 MB.*

5.4 Energy Savings using Somniloquy

In addition to evaluating the operating performance of our Somniloquy prototypes, it's also important to assess the higher level goal of this work, namely the impact on PC energy consumption. In this section we present some data which demonstrates the potential of Somniloquy to reduce both desktop and laptop energy usage in general terms. We also verify the energy saving model presented in Section 3.3, which allows the specific savings in a given application scenario to be calculated. Unless otherwise noted, we are using the Wired-1NIC version of our prototype for the desktop energy measurements and the Wireless-2NIC version for the laptop energy measurements.

5.4.1 Reducing Desktop Energy Consumption

Our testbed desktop PC consumes 102 W in normal operation and <5 W in S3 with Somniloquy. Somniloquy therefore saves around 97 W. On this basis, if Somniloquy were to be deployed in an environment where a PC is actively used for an average of 45 hours each week (i.e. 27% of the time), this would result in 620 kWh of savings per computer in a year. Assuming 0.61 kg CO_2/kWH[6] and US$ 0.09/kWH[7], this means an annual saving of 378 kg of CO_2 (to put it in perspective, the average US residents annual CO_2 emissions are 20 metric tonnes as compared to a worldwide average of 4 metric tonnes per person[8]) and US$ 56 per computer. We

[6] http://www.eia.doe.gov/cneaf/electricity/page/co2_report/co2report.html

[7] http://www.eia.doe.gov/cneaf/electricity/epa/epa_sum.html

[8] http://www.sciencedaily.com/releases/2008/04/080428120658.htm

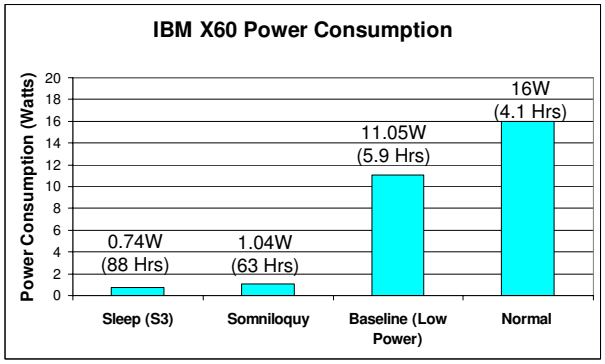

Figure 7: *Power consumption and the resulting estimated battery lifetime of a Lenovo X60 using Somniloquy. The lifetime is calculated using the standard 65 Watt hour battery of the laptop.*

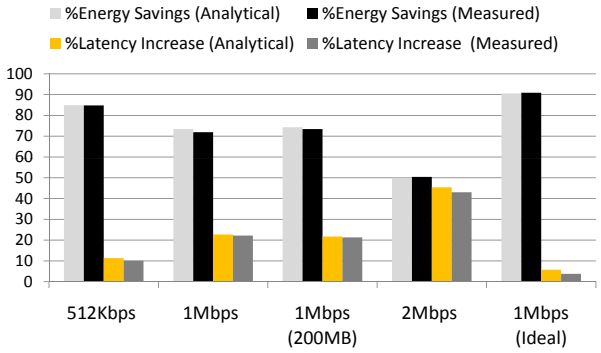

Figure 8: *Comparing the analytical results with the measured values for the web-download stub. The flash storage available on the gumstix is set to 100 MB, unless stated otherwise.*

believe this is significantly higher than the bill of materials cost of the components required to implement a commoditized Somniloquy-enabled network card. In this case, deployments of Somniloquy-enabled devices would pay for themselves within a year.

5.4.2 Desktop Energy Savings for Real Workloads

We now estimate the energy savings enabled by Somniloquy under realistic workloads. We use the data provided by [20], relating to the use patterns of twenty two distinct desktop PCs; each of which is classified as being either idle, active, sleep or turned off. We then compute the energy consumed by each of the PCs with and without Somniloquy using the formula of Section 3.3. For ease of exposition, we bin the data into three different categories: PCs that are idle for $<$25% of the time (7 machines), idle for 25%-75% of the time (6 machines) and finally those that are idle for $>$75% of the time (9 machines). The average energy savings for these twenty two PCs when using Somniloquy is 65%, as compared to normal operation without Somniloquy. The average energy savings for the PCs in the individual categories are 38%, 68% and 85% respectively. As expected, the most energy savings are for the PCs with larger idle times since they have more opportunity to use Somniloquy.

5.4.3 Increasing Laptop Battery Lifetime

Figure 7 shows the average power consumption of the laptop testbed when operating normally (i.e. no power saving mechanisms), with standard power saving mechanisms in place (the baseline power), when Somniloquy (Wireless-2NIC) is operational, and in the standard S3 mode (without the gumstix attached). Somniloquy adds a relatively low overhead of 300 mW to S3 mode, resulting in a total power consumption which is close to just 1 W, as compared to the 11 W of the idle laptop. This means that when the laptop needs to be attached to the network and available for remote applications but is otherwise idle, it can be put into Somniloquy mode to enable an order of magnitude decrease in power consumption and a resulting increase in battery lifetime from 5.9 hours to 63 hours (using the standard 65 Watt-Hour battery).

5.4.4 Energy Savings for Specific Applications

The basic analysis of energy consumption and battery lifetime presented above is very generic; for a given usage scenario it should be possible to use the energy saving model presented in Section 3.3 to predict savings much more accurately. In order to validate this model we ran experiments downloading content from a remote web server, and measured both energy consumption and latency so as to compare them with their corresponding analytical values. Note that we only measure the energy consumption for the duration of the application.

The web download stub was chosen since it was relatively easy to change the duty-cycle of the host, i.e. the duration for which the host can sleep (T_{sleep}) after which it needs to be woken up to transfer data from the gumstix (T_{awake}). As discussed in Section 3.3, T_{sleep} depends on the download bandwidth and the amount of flash storage on the gumstix, while T_{awake} depends on the amount of flash storage on the gumstix and the transfer rate between the gumstix and the host. We downloaded a 300 MB file at various link bandwidths ranging from 512 Kbps to 2 Mbps, and used two different flash storage sizes at the gumstix - 100 MB and 200 MB, effectively varying T_{sleep} from approximately 1600 seconds down to 400 seconds. We measured the power consumed during the download using the methodology described in the beginning of this section. In Figure 8, we present the measured energy savings and the corresponding predicted values

using our model for four different data points. As we can see from the figure, the predicted energy savings and the increased latency closely match the measured values (within 1.5%). The values do not exactly match since the actual measured power values vary over time, and the time taken to suspend and resume also varies across runs. We used a fixed value for these in the formula.

Figure 8 also illustrates that increasing the bandwidth from 512 Kbps to 2 Mbps reduces the energy savings from 85% to 50%, and increases the latency from 11% to 43%, although a larger amount of flash storage improves the energy saving and latency. As explained earlier this is due to the limited transfer speed of the USBnet interface in our prototype (<5 Mbps), because of which the PC is awake for longer periods of time while transferring the data from the gumstix (T_{awake}= 181 seconds to transfer 100 MB of data). In Figure 8 we have also plotted an ideal case (1 Mbps-ideal) where the host can read the flash storage of the gumstix directly. For the ideal case the duration for which the host needs to stay awake to transfer data from the gumstix reduces considerably (T_{awake}= 23 seconds). This improves energy savings to 91% and limits the increase in latency when using Somniloquy to less than 5%.

6 Related Work

There have been several proposals to reduce the energy consumption of desktop PCs and laptops. Prior work can largely be grouped in three categories: reducing the active power consumption of devices (when awake) [3, 5, 9, 10, 16, 17], reducing the power consumption of the network infrastructure (e.g. routers and switches) [11, 12, 21], and opportunistically putting the devices to sleep. Somniloquy falls in the third category. Since a machine in sleep state consumes significantly less power than in lowest power active state [11, 27] (verified by us in Section 5), significant energy savings are possible by putting the machine to sleep whenever possible.

For opportunistic-sleep systems, the biggest challenge is to ensure connectivity when the host is asleep. Prior techniques to solve this problem either use advanced functionality in the NIC [18] or use extra network interfaces [26, 27]. We now compare and contrast Somniloquy to both these classes of work.

Among schemes that do not use an extra network interface, the most well-known are Wake-on-LAN (WoL) [18] and its wireless equivalent, Wake-on-WLAN (WoWLAN). In both these schemes, the NIC parses incoming packets when the host is asleep. It wakes up the host PC whenever an incoming "magic" packet is received. According to the specification [18], the magic packet payload must include 6 characters of a wakeup pattern that is set by the host PC, followed by 8 copies of the NIC's MAC address. In WoWLAN, the only difference is that this packet is sent over the Wireless LAN. Although most modern NICs implement WoL functionality, few deployed systems actually use this functionality, due to four main reasons. First, the remote host must know that the PC is asleep and that it must wake it up before pursuing application functionality. Second, the remote host must have a way of sending a packet to the sleeping PC through any firewalls/NAT boxes, which typically do not allow incoming connections without special configuration. Third, the remote host must know the MAC address of the sleeping PC. Fourth, WoWLAN does not work when laptops change their subnet because of mobility. In contrast, Somniloquy does not require the extra configuration of firewalls/NAT boxes, and is transparent to remote application servers. It can handle mobility across subnets since the secondary processor can re-associate with services such as Dynamic DNS (to redirect a permanent host name to the PC's new IP address), and re-log-in to servers such as IM servers. In addition to these differences, Somniloquy also allows applications to be offloaded to the low power processor. There is no such concept in WoL, which instead wakes up the host when any pattern is matched.

Intel recently announced its Remote-Wake' [14] chipset technology (RWT) that claims to extend WoL on new motherboards by allowing VoIP calls to wake up a system, although its general applicability to other applications is not known. The details of this technology are not published. In contrast, Somniloquy goes beyond just WoL or RWT. It allows low power operation for various applications other than VoIP. Furthermore, Somniloquy does not require modifications to application end points or servers. RWT requires applications to first contact a server, which then sends a special packet to the PC to signal a wake up.

Another approach is to use additional "low-power" network interfaces to maintain connectivity to the PC that is asleep. This approach has been proposed for use with mobile devices. For example, Wake-on-Wireless [26] wakes up the host PC on receiving a special packet on the low power network interface. Turducken [27] uses several tiers of network interfaces and processors with different power characteristics, and wakes up the upper tier when the lower tier cannot handle a task. In contrast to these schemes, Somniloquy requires only a single network interface, and presents the paradigm of a single PC to users rather than a multi-tiered system, preserving the current user experience and therefore requiring less training to use. Somniloquy also gives the impression to remote application servers that a device remains awake all the time even though it is actually asleep, since the same MAC and IP addresses are used. This level of

transparency is not provided either by Wake-on-Wireless or Turducken. Finally, we have gone into more detail than previous work on ways of supporting applications that require interactions among the secondary and the host processor to perform offload – such as IM, BitTorrent and web downloads.

To reduce the power consumed by desktop PCs, some early proposals have suggested the use of proxies on the subnet that function on behalf of the desktop PC when it is asleep [4, 7, 11]. The proxy monitors incoming packets for the PC, and wakes it up using WoL when the PC needs to handle the packet. We are not aware of any published prototype implementations of such systems. Recently, Sabhanatarajan et. al. [25] propose a smart NIC that can act as proxy for a host to save power. However, the authors focus primarily on the design of a high speed packet classifier for such an interface. In comparison, Somniloquy has much wider applicability than the above schemes. It can be used in homes and small offices where it might be infeasible to deploy a dedicated server to handle processing for another PC.

A contemporaneous effort to Somniloquy is the idea of a Network Connection Proxy (NCP) [15, 20], which is a network entity that maintains the presence of a sleeping PC. In [15], the authors define the requirements of an NCP and propose modifications to the socket layer (similar to Split TCP) for keeping TCP connections alive through a PC's sleep transitions. In [20], the authors extend these APIs to support other protocols as well. Somniloquy is similar in spirit to NCP, and NCP's socket APIs can reduce Somniloquy's overhead when waking up from sleep (Section 3.1). Furthermore, to the best of our knowledge, Somniloquy is the first published prototype of any proxying system.

We note that the concept of adding more processing to the network interface is not new. Existing products offload processing to the NIC to improve performance (TCP offload [19]) and remote manageability (Intel AMT [13]). Somniloquy uses a similar offloading paradigm, but to conserve energy instead of improving performance or manageability.

7 Conclusions

We have presented Somniloquy, a system that augments network interfaces to allow PCs to be put into low-power sleep states opportunistically, without sacrificing functionality. Somniloquy enables several new energy saving opportunities. First, PCs can be put to sleep while maintaining network reachability, without special network infrastructure as needed by previous solutions (e.g. WoL). Second, some applications can be run in sleep mode thereby requiring much less power. In this paper, we have shown the feasibility for three such applications to be run in sleep mode: BitTorrent, instant messaging, and web downloads.

Somniloquy achieves these energy savings without requiring any modifications to network, to remote application servers, or to the user experience of the PC. Furthermore, Somniloquy can be incrementally deployed on legacy network interfaces, and does not rely on changes to the CPU scheduler or the memory manager to implement this functionality, thus it is compatible with a wide class of machines and operating systems.

Our prototype implementation, based on a USB peripheral, includes support for waking up the PC on network events such as incoming file copy requests, VoIP calls, instant messages and remote desktop connections, and we have also demonstrated that file sharing/content distribution systems (e.g. BitTorrent, web downloads) can run in the augmented network interface, allowing for file downloads to progress without the PC being awake. Our tests show power savings of 24x are possible for desktop PCs left on when idle, or 11x for laptops. For PCs that are left idle most of the time, this translates to energy savings of 60% to 80%. The electricity savings made are such that deploying a productized version of Somniloquy could pay for itself within a year.

Acknowledgements

We would like to thank John Dunagan, Gunjan Gupta, Srikanth Kandula, Jitu Padhye, Patrick Verkaik, Kashi Vishwanath and the anonymous reviewers for their comments on various versions of this paper. We would also like to acknowledge the feedback received from Alex Snoeren, Stefan Savage and Geoff Voelker. Finally, we are grateful to our shepherd, Jeffrey Mogul for meticulously guiding us towards the final version of the paper. His help was invaluable.

References

[1] ACPI. Advanced Configuration and Power Interface Specification, Revision 3.0b. `http://www.acpi.info`.

[2] Y. Agarwal, R. Chandra, A. Wolman, P. Bahl, K. Chin, and R. Gupta. Wireless Wakeups Revisited: Energy Management for VoIP over Wi-Fi Smartphones. In *MobiSys '07: Proceedings of the 5th international conference on Mobile systems, applications and services*, pages 179–191, New York, NY, USA, 2007. ACM.

[3] Y. Agarwal, T. Pering, R. Want, and R. Gupta. "SwitchR: Reducing System Power Consumption in a Multi-Clients, Multi-Radio Environment". In *Proceedings of IEEE International Symposium on Wearable Computing (ISWC)*, 2008.

[4] M. Allman, K. Christensen, B. Nordman, and V. Paxon. Enabling an Energy-Efficient Future Internet Through Selectively Connected End Systems. In *6th ACM Workshop on Hot Topics in Networks (HotNets)*. ACM, November 2007.

[5] M. Anand, E. B. Nightingale, and J. Flinn. Self-tuning Wireless Network Power Management. In *MobiCom '03: Proceedings of the 9th annual international conference on Mobile computing and networking*, pages 176–189, New York, NY, USA, 2003. ACM Press.

[6] N. Borisov, D. Brumley, H. J. Wang, J. Dunagan, P. Joshi, and C. Guo. A generic application-level protocol analyzer and its language. In *Proceedings of the 14th Annual Network and Distributed System Security Symposium (NDSS)*, 2007.

[7] K. Christensen, C. Gunaratne, and B. Nordman. The Next Frontier for Communication Networks: Power Management. *Computer Communications*, 27(18):1758–1770, 2004.

[8] W. Cui, J. Kannan, and H. J. Wang. Discoverer : Automatic Protocol Reverse Engineering from Network Traces. In *Proceedings of the USENIX Security Symposium*, 2007.

[9] K. Flautner, S. K. Reinhardt, and T. N. Mudge. Automatic Performance Setting for Dynamic Voltage Scaling. In *MobiCom '01: Proceedings of the 6th annual international conference on Mobile computing and networking*, pages 260–271, 2001.

[10] J. Flinn and M. Satyanarayanan. Managing Battery Lifetime with Energy-Aware Adaptation. *ACM Trans. Comput. Syst.*, 22(2):137–179, 2004.

[11] C. Gunaratne, K. Christensen, and B. Nordman. Managing Energy Consumption Costs in Desktop PCs and LAN Switches with Proxying, Split TCP Connections, and Scaling of Link Speed. *Int. J. Netw. Manag.*, 15(5):297–310, 2005.

[12] M. Gupta and S. Singh. Greening of the Internet. In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 19–26, New York, NY, USA, 2003. ACM.

[13] Intel. Intel Active Management Technology (AMT). `http://www.intel.com/technology/platform-technology/intel-amt/`.

[14] Intel. Intel Remote Wake Technology. `http://www.intel.com/support/chipsets/rwt/`.

[15] M. Jimeno, K. Christensen, and B. Nordman. A Network Connection Proxy to Enable Hosts to Sleep and Save Energy. In *IEEE International Performance Computing and Communications Conference*, 2008.

[16] R. Kravets and P. Krishnan. Application-driven Power Management for Mobile Communication. *Wireless Networks*, 6(4):263–277, 2000.

[17] X. Li, R. Gupta, S. V. Adve, and Y. Zhou. Cross-Component Energy Management: Joint Adaptation of Processor and Memory. *ACM Trans. Archit. Code Optim.*, 4(3):14, 2007.

[18] P. Lieberman. Wake-on-LAN technology. `http://www.liebsoft.com/index.cfm/whitepapers/Wake_On_LAN`.

[19] J. C. Mogul. TCP Offload Is a Dumb Idea Whose Time Has Come. In *HotOS*, pages 25–30, 2003.

[20] S. Nedevschi, J. Chandrashekar, B. Nordman, S. Ratnasamy, and N. Taft. Skilled in the art of being idle: reducing energy waste in networked systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[21] S. Nedevschi, L. Popa, G. Iannaccone, S. Ratnasamy, and D. Wetherall. Reducing Network Energy Consumption via Sleeping and Rate-Adaptation. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 323–336. USENIX Association Berkeley, CA, USA, 2008.

[22] R.Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, and T. Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, June 1999.

[23] J. Roberson, C. Webber, M. McWhinney, R. Brown, M. Pinckard, and J. Busch. After-hours Power Status of Office Equipment and Energy use of Miscellaneous Plug-load Equipment. *Lawrence Berkeley National Laboratory, Berkeley, California. Report# LBNL-53729-Revised*, 2004.

[24] K. Roth and K. McKenney. Energy Consumption by Consumer Electronics in US Residences. *Final Report to the Consumer Electronics Association (CEA)*, 2007.

[25] K. Sabhanatarajan, A. G.-R. M. Oden, M. Navada, and A. George. Smart-NICs: Power Proxying for Reduced Power Consumption in Network Edge Devices. In *ISVLSI '08*, 2008.

[26] E. Shih, P. Bahl, and M. J. Sinclair. Wake on Wireless: An Event Driven Energy Saving Strategy for Battery Operated Devices. In *MobiCom '02: Proceedings of the 8th annual international conference on Mobile computing and networking*, pages 160–171, New York, NY, USA, 2002. ACM Press.

[27] J. Sorber, N. Banerjee, M. D. Corner, and S. Rollins. Turducken: Hierarchical Power Management for Mobile Devices. In *MobiSys '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services*, 2005.