Turducken: Hierarchical Power Management for Mobile Devices

Jacob Sorber Nilanjan Banerjee Mark D. Corner Sami Rollins†

Department of Computer Science
University of Massachusetts, Amherst, MA
{sorber, nilanb, mcorner}@cs.umass.edu

†Department of Computer Science
Mt. Holyoke College, South Hadley, MA
srollins@mtholyoke.edu

Abstract—

Maintaining optimal consistency in a distributed system requires that nodes be always-on to synchronize information. Unfortunately, mobile devices such as laptops do not have adequate battery capacity for constant processing and communication. Even by powering off unnecessary components, such as the screen and disk, current laptops only have a lifetime of a few hours. Although PDAs and sensors are similarly limited in lifetime, a PDA's power requirement is an order-of-magnitude smaller than a laptop's, and a sensor's is an order-of-magnitude smaller than a PDA's. By combining these diverse platforms into a single integrated laptop, we can reduce the power cost of always-on operation. This paper presents the design, implementation, and evaluation of Turducken, a Hierarchical Power Management architecture for mobile systems. We focus on a particular instantiation of HPM, which provides high levels of consistency in a laptop by integrating two additional low power processors. We demonstrate that a Turducken system can provide battery lifetimes of up to *ten times* that of a standard laptop for always-on operation and *three times* for a system that periodically sleeps.

I. Introduction

The performance and utility of any distributed system is impacted by the availability of the participating nodes. In order to execute tasks remotely and maintain consistency of distributed data stores, nodes must be powered on and connected to one another. These requirements are difficult to support in a wired environment; if the participating nodes are mobile, it becomes even more of a challenge. It is particularly difficult to ensure that a mobile node remains *always-on* to participate in the system.

Mobile devices are unique in that they have finite lifetimes. In larger mobile devices, such as laptops, aggressive power management is often used to extend device lifetime by reducing the amount of time the device remains on. Although PDAs and sensors are similarly limited in lifetime, a PDA's power requirement is an order-of-magnitude smaller than a laptop's and a sensor's is another order-of-magnitude smaller than a PDA's. However, these reduced power draws come at the price of reduced functionality and computational power.

This paper presents the design, implementation, and evaluation of Turducken, a mobile device architecture that enables full device functionality, always-on availability, and extended device lifetime. Turducken integrates several mobile computing platforms that operate at different power levels into a single device that can operate at the power level of any one of its subsystems. While the system supports all of the functionality of its highest power subsystem, it can utilize lower power subsystems to execute simpler tasks, thus reducing the system-wide power consumption and extending the system lifetime. Moreover, by integrating an always-on subsystem such as a sensor, we can achieve always-on availability.

Because maintaining consistency of distributed data stores is one of the most integral tasks for mobile distributed systems, we focus our attention on Turducken's ability to maintain high levels of consistency. Our evaluation compares several Turducken configurations running three common, data-driven applications: time synchronization, web caching, and email. Our results indicate that a Turducken system that integrates an x86-based laptop with a StrongARM and a sensor provides the same level of consistency as a standard laptop computer; however, it lasts up to *ten times* as long for always-on operation and *three times* for a system that periodically sleeps. Additionally, we present a theoretical analysis of the lifetime gain of using a Turducken system to execute any task. This analysis demonstrates that Turducken is useful for a broad set of distributed services.

In Section II we provide further motivation and introduce the Turducken approach. Section III describes the design of the hardware components as well as the software architecture. Section IV presents a prototype implementation, which we evaluate in Section V. Section VI presents related work, and we conclude in Section VII.

II. Motivation

A. Consistency in Mobile Systems

A fundamental goal in mobile distributed systems is providing consistency between data stores. Distributed file systems, databases, and applications such as email and the web demand that a user's local view of data be consistent with the view at other nodes in the system. This consistency is ensured through frequent synchronization between nodes. For two end-points to maintain optimal consistency, they must both be *always connected* and *always powered on*. Unfortunately, if either node is mobile, the system cannot make this guarantee and consistency is sacrificed.

The lack of a network connection between two nodes is primarily attributable to physical proximity and wireless network coverage. Network partitions can also be the result of several other factors, including: firewalls; integration of inexpensive short-range wireless connections in consumer devices; or location-based services that intentionally make services only available in specific physical locales. While an end system can attempt to mask these disconnections, it can do little to affect the infrastructure that provides connectivity.

Even if a network path does exist between two endpoints, in a mobile system there is no guarantee that both nodes will be powered on. Mobile nodes have a finite energy supply and thus a finite lifetime. A node may be off because it has exhausted its battery supply, because it has intentionally powered down to conserve energy, or because the user has turned off the device. In any case, if the node performing synchronization or the node with the most recent version of the file is not on, then synchronization cannot occur. Traditionally, mobile systems address these problems by attempting to mitigate their effects. For example, many systems cache and buffer updates and opportunistically perform synchronization when nodes are powered on and connected. Similarly, many systems support weak-consistency models. This ensures that the system can be used locally and remotely even if nodes are disconnected or off. In essence, these techniques allow the system to function even if data stores are not consistent; however, as data stores become increasingly inconsistent, they also become less valuable.

B. Energy Management Approaches

To achieve high levels of consistency, mobile nodes must be powered on as much as possible so they may take advantage of network connectivity when it exists and may perform synchronization as frequently as possible. This requires that a device be on and consuming energy, even when no useful tasks can be accomplished. For instance, ensuring that a user's mail is immediately delivered to a mobile device requires that the device to be powered on, even when no new mail is arriving. This approach can be very energy inefficient, thus negatively impacting the lifetime of the system.

One approach to reduce energy consumption is to leave the mobile device in an *always-on* mode, but turn off the screen, aggressively spin down the disk [4, 3, 8], scale the CPU voltage and frequency [30, 7, 5, 15, 5], manage wireless interface usage [1], turn off banks of RAM [12, 16, 9, 18], and recompile programs for low power operation [28]. Unfortunately, we observe that a sample laptop using many of these methods only has a lifetime of approximately 8 hours and a standard PDA only lasts for 14 hours. To keep a device in an always-on state requires the user to charge it several times a day, even if it is not actively used. These low-power modes were designed to save power while providing interactivity, not to enable always-on functionality.

Another approach is to *suspend* the device, refreshing only the RAM, and wake up at periodic intervals to perform synchronization (e.g., download web updates). For instance, to extend the lifetime of an IBM Thinkpad to 3 days, we can wake the laptop for approximately 2 minutes of every hour. However, there is a trade-off between the frequency with which we wake the device and the level of consistency maintained: waking up more often costs more energy, but provides higher consistency. Additionally, there is no guarantee that a device will be in range of a network and able to perform synchronization when it wakes. An approach such as Wake-on-Wireless [24] can reduce the amount of energy spent waking a device if no network connectivity exists. However, a significant amount of energy is still wasted if a high-power device, such as a laptop, wakes to discover that a network connection exists but no updates are ready (e.g., no new mail has arrived or a cached web page has not changed).

While these approaches provide considerable energy savings, they are inappropriate for extending the maximum lifetime of the device while providing high consistency. This is because they fail to address the *non-reducible* power of mobile devices [14], which dominates the lifetime of the battery. The reducible power is the amount

of power that can be eliminated from a running system while maintaining the ability to do computation. Common sources of non-reducible power include the power supply, the on-board oscillators, the memory and I/O buses, and the limited range of frequency and voltage scaling [2]. Some small embedded systems have proposed using multiple processor cores that can be shut off [19, 11]. However, such a system only reduces the power draw of the processor, which constitutes less than 10% of the power consumed by a laptop [2]. Even the most highly-optimized laptop computer incurs a significant energy cost to wake up and download a piece of data.

C. A New Approach: Turducken

The amount of non-reducible power varies for different devices. For example, the non-reducible power of a StrongARM-based PDA is on the order of twenty-times smaller than the non-reducible power of an x86-based laptop. As another example, the non-reducible power of a small sensor is significantly smaller than that of a device such as a wireless music player. Typically, devices are carefully optimized to provide their promised functionality at the lowest possible energy cost, and devices that provide less functionality have smaller non-reducible power. Fortunately, there is significant overlap in the functionality provided by high-power and low-power devices. For example, maintaining a consistent view of a file requires only the ability to connect to a network and download data; a variety of devices can provide this functionality.

The goal of our approach, Turducken, is to reduce the energy cost of maintaining high levels of consistency on mobile devices by combining several optimized mobile platforms into one integrated system. By combining a very low-power platform such as sensor with a very high-power platform such as a laptop, we can produce a system that can be always-on and still have all of the functionality of a laptop computer.

The system is composed of a set of *subsystems*, each with a set of capabilities and a power mode. The system as a whole executes tasks (e.g., downloads data updates) by waking the subsystem that has the capabilities to execute the task in the most efficient manner. For example, we can integrate a StrongARM processor, along with its memory and storage, and a standard x86 laptop. We then suspend the x86 subsystem and rely upon the StrongARM subsystem to wakeup and perform periodic tasks.

For instance, if the StrongARM subsystem wakes up periodically to cache a copy of frequently-used web pages, when the user opens the laptop, those pages will be available and highly consistent. If the laptop alone were to frequently wake itself up and cache those same pages, it would attain the same level of consistency; however, the overall *lifetime* of the system would be greatly diminished.

Note that in this integrated system all of the subsystems use a common battery and are connected by a common bus, but they effectively form a tightly coupled distributed system. However, from the user's perspective it appears to be a normal laptop. The addition of extra components does increase the weight and cost of a mobile system. For instance, adding a StrongARM mobile processor and memory to the inside of a laptop may add $100 and a few ounces. However, the extra capabilities the system provides outweigh these costs. Another observation is that this system could be commercially built using commodity components. The architecture is fully composable: any set of subsystems can be used together to give a wide variety of power modes and can be applied to many mobile devices.

D. Lifetime Gains

We can demonstrate Turducken's effectiveness using a simplified analysis of the expected gains in the lifetime of the system. Here we analyze the expected lifetimes of two different systems. The first is a normal laptop that wakes up periodically to synchronize and goes back to sleep. The second is a simplified Turducken system that consists of an x86 subsystem integrated with a StrongARM subsystem; the x86 subsystem remains suspended, while the StrongARM subsystem periodically wakes up and performs the same synchronization task.

Our analysis shows that there are two circumstances in which Turducken provides gains in the system lifetime: 1) if the fraction of time the laptop spends awake is large enough to overcome the extra burden of the StrongARM's suspension power, and 2) if the time in which the StrongARM can perform the synchronization task is a reasonable multiple of the time the x86 takes.

The first equation details the lifetime of a laptop that wakes at periodic intervals:

$$L_L = \frac{C}{f_A^L \cdot P_A^L + (1 - f_A^L) \cdot P_S^L}, \tag{1}$$

where C is the capacity of the battery, f_A^L is the fraction of time the laptop spends awake, P_A^L is the power it expends while awake, $1 - f_A^L$ is the fraction of time the laptop spends asleep, and P_S^L is the power it expends while suspended.

The lifetime of a Turducken system, consisting of an x86 subsystem paired with a StrongARM subsystem, can be represented as:

$$L_T = \frac{C}{f_A^P \cdot P_A^P + (1 - f_A^P) \cdot P_S^P + P_S^L}, \quad (2)$$

where f_A^P and P_A^P are the fraction of time and power the StrongARM subsystem spends awake, and $1 - f_A^P$ and P_S^P are the fraction of time and power the StrongARM subsystem spends suspended respectively. C is the battery capacity. The x86 remains suspended while the StrongARM wakes up. Thus the x86 expends P_S^L all of time.

Using these two equations we can express the gain of the Turducken system as the ratio of the lifetime of the Turducken system to that of a standard laptop:

$$\frac{L_T}{L_L} = \frac{f_A^L \cdot P_A^L + (1 - f_A^L) \cdot P_S^L}{f_A^P \cdot P_A^P + (1 - f_A^P) \cdot P_S^P + P_S^L}. \quad (3)$$

As long as this ratio is greater than one, Turducken has a positive impact on the lifetime of the system. Using the proof found in the appendix and a set of measurements taken from a prototype system running the web caching application, described in Section IV, we find that if the web caching application only runs 17 seconds of every hour, then the StrongARM can perform the synchronization task up to 5 times slower than the x86. In fact, because web caching is network bound, the ratio of execution time is actually one-to-one, and for reasonable levels of consistency the web caching application runs much more than 17 seconds of every hour. Because of these two factors, Turducken typically provides an increase in lifetime substantially greater than this lower bound.

The remainder of this paper describes the design and prototype implementation of the hardware and software components of our system. Additionally, we present a set of experiments which quantify the benefits of using the Turducken system.

III. System Design

The design of a Turducken system is composed of three parts: the hardware, the underlying system architecture, and the model for distributing applications across the subsystems. In general, the design is similar to many distributed systems; each subsystem is under autonomous control while decisions are made in a distributed manner. Client applications reside at the most powerful subsystem, and tasks that support those applications are distributed among the various subsystems.

A. Hardware Design

A Turducken system is designed in a strictly hierarchical manner, and each subsystem is more powerful than any subsystem below it. Each subsystem can communicate with a *superior* subsystem and an *inferior* subsystem—the two exceptions being the top and bottom of the hierarchy. Communication occurs via a local communication network and the subsystems are connected to a common power source. Moreover, each subsystem has the ability to draw its superior subsystem out of a suspended mode. It is fully composable; the system will still operate correctly if subsystems are added, removed, or changed. This provides a flexible architecture that can accommodate the evolving number of hardware platforms available in low-power computing. An overview diagram of our design is shown in Figure 1.

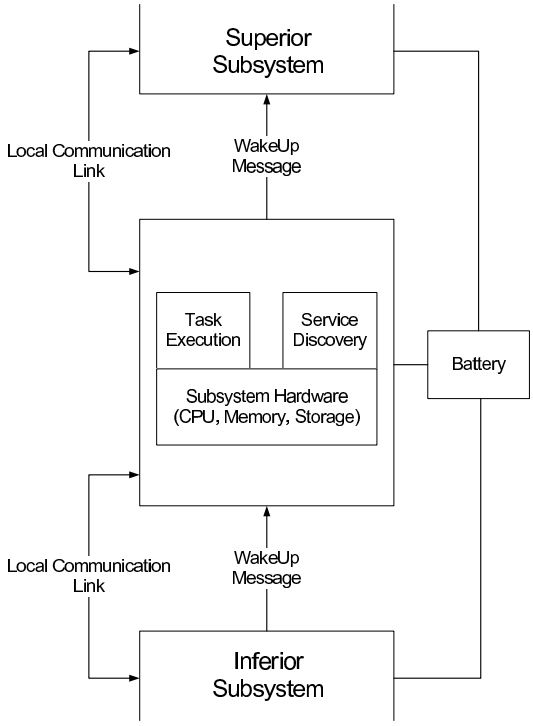

Fig. 1. System Design

Each subsystem contains an independent processor, memory, internal bus, and persistent storage system. Each may also have an independent external wireless network interface, although these can be shared by routing through the inter-subsystem communication network. The set of subsystems can be architecturally homogeneous and span a range of power requirements. By limiting the interface between subsystems, we achieve composability. Integrating new subsystems with differing instruction sets, capabilities, operating systems, and power requirements is straightforward.

A Turducken system is also fully autonomous and does not depend on any special hardware from the external network. For instance, Turducken does not require external networks to be equipped with hardware wakeup signals,

such as those used in the Wake-On-Wireless project [24]. This ensures that the system will work with high-powered access points, as well as low-power, peer-to-peer, wireless devices. Because there is no dependence on the external, wireless networking hardware, Turducken will work with any radio interface, as well as accommodate multiple radios in the same system.

B. System Architecture

The system as a whole is responsible for accepting tasks from the user and executing them in a way that extends the lifetime of the system. Tasks can be anything from keeping the time synchronized to ensuring that the local copy of the user's email is current. The user, or a service executing on behalf of the user, introduces tasks at the highest level and the system distributes these tasks among the different subsystems in a way that extends the lifetime of the overall system. Each subsystem is capable of several operations: perform tasks or discover services; inform other subsystems when necessary; and manage its local consumption of power. We discuss each responsibility in more detail below.

Perform a task. A subsystem can perform a task if the required service is reachable and ready to be used. Ideally, a task will be executed by the most efficient subsystem capable of performing that task. For example, the highest-power subsystem would be required to synchronize a very large media file while a StrongARM subsystem can perform the task of synchronizing a cache of web pages. For some applications, a subsystem will also need to pass the results of task execution to its superior subsystem. For example, a web page cached by a StrongARM subsystem will ultimately be delivered to the highest subsystem in response to a user request.

Perform service discovery. A subsystem can also monitor the availability of a service required by a higher-power subsystem in order to perform a task. Service discovery may simply discover the existence of a service, or it may determine whether a particular service needs to be used (e.g., whether or not a user has new email that needs to be fetched). Again, service discovery should be performed by the lowest-power subsystem that is capable of discovering the service. In many cases, it is also possible to further decompose service discovery. For example, to determine if a large media file is available to be synchronized, a sensor subsystem can monitor the network for connectivity, a StrongARM subsystem can determine if the file has changed, and the x86 subsystem can actually perform the task.

Enter a suspension state. If a subsystem is not needed to perform a task or service discovery, it may put itself to sleep in order to conserve energy. In some cases, this may require that the subsystem delegate tasks or service discovery jobs to its inferior subsystem. For example, a StrongARM subsystem may notify a sensor subsystem that it is going to sleep and needs to be woken when a network connection is available.

Wake its superior subsystem. Once a subsystem has discovered an appropriate service, it may need to wake its superior subsystem so that it can perform the task. Each subsystem is capable of waking its superior subsystem. In this way, a subsystem can rely on its inferior subsystem to tell it when there is something to do rather than requiring the system to wake periodically and check.

C. Distributing Applications

There are several methods of distributing application responsibilities over the subsystems. We describe each of these options here:

System-Aware Architecture. The first option is to build an application that is customized for the system. Such an application requires designers to create application components for each subsystem. In addition, the application must define the messaging protocol used to communicate between components. This hand-coded option is useful for new applications and also for applications, such as time synchronization, which are fairly simple to implement.

Proxy-Based Architecture. A second option is to use a proxy-based architecture that can take advantage of existing distributed application components. Using this architecture, a subsystem that executes tasks appears as a proxy service provider or a replicated server to superior subsystems. Many distributed applications, such as distributed file systems, email, and web caching, already support this design. Therefore, the advantage is simplicity and deployability—proxies only require recompiling and reconfiguring the application rather than rewriting the application. Unfortunately, not all applications will tolerate a proxy that queues responses, requiring some modification of applications. One possibility is to use queued RPC as found in the Rover toolkit [10].

Transparent Architecture. A final option is to develop a Turducken system component that is capable of transparently migrating application processes. One way to support this is by using traditional process migration [13, 21, 26, 27]. We have eliminated this as a possibility due to the complications arising from different architectures, operating systems, and memory capacities. Another possibility is to use virtual machines, either programming-language virtual machines such as those used for Java, or a lightweight, OS-level virtual machine such as De-

nali [31]. Unfortunately, the current lack of a virtual machine that runs on all of these platforms, and the vastly differing capabilities of the subsystems makes this difficult.

IV. Prototype Implementation

To demonstrate the efficacy of our approach, and to provide a test platform for our work, we have built a prototype Turducken system. The prototype currently consists of a hardware implementation and three applications: time synchronization, web caching, and IMAP synchronization.

A. Hardware Implementation

The hardware prototype, shown in Figure 2, consists of three subsystems: an x86-based IBM Thinkpad X31, a Compaq iPAQ 3870 StrongARM-based PDA, and a CrossBow Mica2Dot Mote. The Mote and PDA are directly connected via a serial interface and the PDA and the laptop are directly connected via a USB interface. The Mote can wake the PDA through the use of the serial DCD line, and the PDA can wake the laptop by sending a request to the Mote, which wakes the laptop by triggering a relay connected to the keyboard. Our prototype can currently be reconfigured as: x86, x86+sensor, or x86+StrongARM+sensor. Each subsystem also contains a real-time clock (RTC) that can generate a wake interrupt. If we reconfigure the system as x86 only, it can suspend itself and use its RTC to wake it at set intervals.

This prototype differs from our design in three significant ways. First, the hardware components are all physically separate—a deployed system would integrate all of the components into a laptop form-factor. The connections shown in the picture would all be internal to the system. Second, there is a plethora of extra parts in our prototype. An integrated implementation would eliminate much of the PDA, including its screen, sleeve, and buttons. Third, each subsystem is run from its own battery. The Turducken design assumes that there is only a single, shared battery. This has implications for how we evaluated the system, as we explain in the evaluation section.

In our implementation, there are two types of wireless interfaces: WiFi and the Mote's custom radio interface. There are both advantages and disadvantages to having access to multiple wireless standards. It does allow the system to take advantage of a broader range of services by allowing it to communicate with more devices; however, it makes system design more challenging since certain tasks may require a particular interface and cannot be accomplished by all subsystems. To mitigate this disadvantage, we have attached a WiFi detector to the Mote. The detector can determine if WiFi signals are present, though it cannot communicate using WiFi or discover if an access point is open or closed.

Even though the x86 and StrongARM each have WiFi interfaces, there is no reason to use them both in the Turducken system. In a configuration that includes both, we turn off the x86's interface and route all traffic through the StrongARM. This saves power, thus extending the battery lifetime of the system.

	Execution Subsystem	Incoming or Outgoing
Time Sync	$\geq$Sensor	Incoming
Web Cache	$\geq$StrongARM	Incoming
IMAP Sync	$\geq$StrongARM	Both

Table I. This table shows a summary of the application characteristics. The execution subsystem denotes where the application is carried out, and Incoming or Outgoing describes the direction of updates.

B. Applications

We have developed and deployed three applications that are representative of commonly-used mobile distributed services: time synchronization, web caching, and IMAP synchronization. Time synchronization is necessary for timestamping distributed updates and determining timeouts in soft-state protocols. Web caching on mobile devices allows the mobile node to serve pages during periods of disconnection and improves response time when connected. IMAP synchronization maintains a local mail cache that can serve mail during periods of disconnection and improves response time. In addition, a local IMAP store can buffer outgoing mail and send it when the node is connected.

These applications also represent three broader classes of applications. These classes are defined by the traits listed in Table I. Time synchronization represents applications that require limited processing and limited transmission of incoming data updates. Web caching represents applications that require more significant processing and larger amounts of incoming data. This is similar to a variety of publish-subscribe systems. IMAP synchronization represents applications that require fairly significant processing and support for outgoing as well as incoming updates. This is similar to the requirements of distributed file and database systems, though the consistency requirements are not as strict.

B.1 Time Synchronization

The time synchronization application follows the system-aware programming model. The sensor runs a cus-

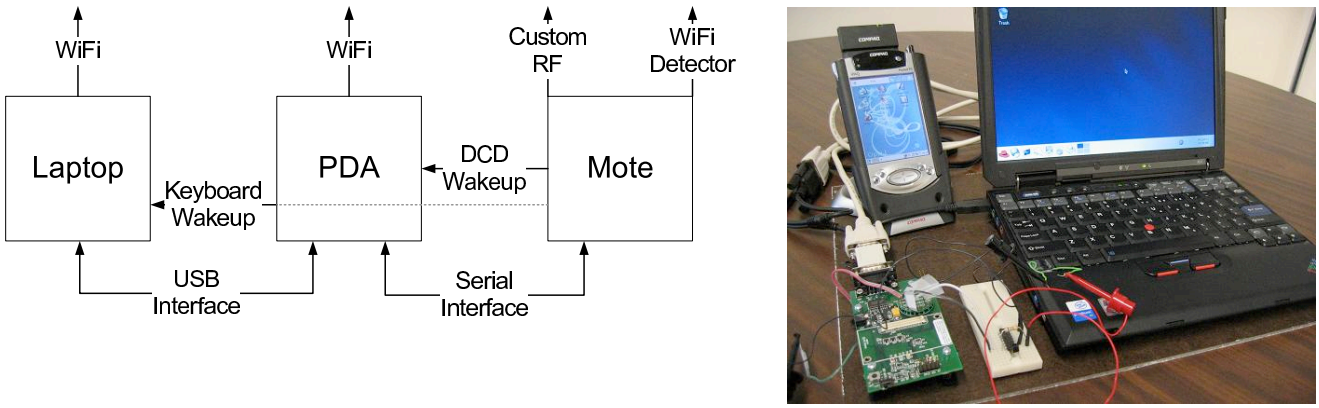

Fig. 2. These figures show the prototype implementation of the Turducken System. The diagram on the left shows the logical connections between components and the photo on the right shows the current prototype.

tom built Network Time Protocol (NTP) client that synchronizes its local clock with a known time server every t seconds. The StrongARM and x86 can then request the current time from the sensor and update their local clocks. We define an explict API for this communication. When the sensor is not present, the the x86 uses its RTC to wake every t seconds and synchronize with the remote time server using the UNIX utility `ntpdate`.

B.2 Web Cache

The web cache application follows a proxy-based programming model. The sensor detects the presence of a WiFi signal; the StrongARM runs a Squid proxy cache; and the x86 runs a web browser. Every t seconds, the sensor determines whether a WiFi connection is available and, if so, wakes the StrongARM. The StrongARM remains awake for 30 seconds while the proxy continuously fetches expired cache items. Web requests originating from the web browser running on the x86 are routed through the StrongARM. These requests can be transparently serviced by the proxy when no network connection is available.

When the StrongARM is not present, the Squid proxy runs on the x86 and the cache is stored on the system's hard disk. The sensor or RTC wakes the x86 every t seconds. If a connection is present, it remains awake for 30 seconds while the Squid proxy fetches expired cache items. Again, the Squid proxy can transparently fulfill requests from a web browser.

B.3 IMAP Synchronization

The IMAP synchronization application also follows a proxy-based programming model. The sensor detects the presence of a WiFi signal and the StrongARM runs a UNIX utility named `mailsync`, which performs synchronization between an IMAP server and a *secondary* mail store. The x86 maintains the *primary* mail store and uses `mailsync` to synchronize with the StrongARM's secondary mail store. The x86 also runs the user's mail client. Every t seconds, the sensor determines whether a WiFi connection is available and, if so, wakes the StrongARM. The StrongARM uses `mailsync` to retrieve incoming mail from and send outgoing updates to the user's mail server. Incoming mail is stored in the secondary mail store hosted on the StrongARM.

When the user turns on the x86, it synchronizes its primary store with the secondary store on the StrongARM. The user accesses mail by configuring the mail client to point to the primary mail store on the x86. When the user suspends the x86, any changes the user has made will be synchronized with the StrongARM which will synchronize with the remote mail server when connected.

In some cases, the user may receive pieces of mail that are too large to be stored in the StrongARM's flash memory. To accomodate this scenario, the primary mail store also synchronizes with the remote mail server when possible. In addition, we would like to modify the StrongARM to wake the x86 when it detects this situation, though we have not yet implemented this feature.

If the StrongARM is not present, the x86 synchronizes directly with the remote mail server when connected. Similar to the web cache, the sensor or RTC wakes the x86 every t seconds. If the x86 discovers that no connection is present, it goes back into a suspended mode without performing synchronization.

Both the IMAP synchronization and the web caching applications were implemented using standard components. Due to the distributed nature of these applications, recoding is not necessary in order to deploy them on our prototype Turducken system. Each component can simply be recompiled for both the x86 and StrongARM architectures.

V. Evaluation

The primary goal of Turducken is to extend the lifetime of a mobile computing device while allowing it to remain aware of its environment when not actively in use. In our evaluation of the Turducken system, we measure the lifetime of several Turducken configurations running the following three sample applications: time synchronization, web caching, and IMAP synchronization. For each application, we compare the system lifetimes of different configurations with respect to data consistency. Finally, we focus on the web caching application and compare system performance with respect to variable network and service availability.

A. Methodology

Our evaluation measures the lifetime of several system configurations running varied workloads. Measuring the lifetime of a Turducken system presents a number of interesting challenges. Explicitly measuring the lifetime of a single configuration running a single workload can take longer than a week. Collecting even a small number of data points using this method is impractical with only a single prototype system. Instead, for our experiments we measure the energy consumed by the system while performing tasks for the given workload and while in a suspended state. We use those values to calculate the total system lifetime.

Because our prototype is powered by three individual batteries, we were unable to derive detailed power traces for the system as a whole. Fortunately, we can calculate the lifetime of the system using only the average energy that is required to performing a given task. We make the assumption that the power draw of a full system will be no greater than the sum of the power draw of each subsystem. This estimate is conservative since an integrated system can use more power-efficient communication links between subsystems.

For the experiments presented here, we measure the amount of energy consumed by each subsystem over a fixed period of time, and calculate the amount of time it takes the entire system to drain a full battery. This calculation depends on several factors: the average power draw of each subsystem while active; the average power draw of each subsystem while suspended; the amount of time each subsystem spends active; and the amount of time each subsystem is suspended.

To determine the power draw of both the x86 and StrongARM in suspended mode, we use the native power management interface on each device to measure the percentage of the battery available, suspend the device for 10 hours, wake the device, and again measure the percentage of the battery available. Using the number of Joules available in a full battery and the percentage of the battery used during the experiment, we determine the total number of Joules used. We divide this value by the total experiment time to obtain the power draw of each device in suspended mode. For the StrongARM, we obtain the full battery capacity from the manufacturer's specification. For the x86 we use the estimated capacity specified by the device's power management system. We have confirmed the suspension power draw of the x86 using a power meter.

To determine the power draw of the x86 and StrongARM subsystems in active mode we run each application on each system configuration for a 24-hour period. For each device, we measure the amount of time it is active, t_A, the amount of time it is suspended, t_S, and the total energy, E, consumed by the subsystem. Using the total amount of time suspended and the suspended power draw, P_S, we calculate the energy consumed while suspended, E_S over the 24 hour period:

$$E_S = P_S t_S. \tag{4}$$

We then use the total energy, E, and the energy used while suspended, E_S, to compute the energy used while active:

$$E_A = E - E_S. \tag{5}$$

By dividing the energy used while active by the amount of time the system is active, we obtain the power draw, P_A, of each subsystem in the active state:

$$P_A = \frac{E_A}{t_A}. \tag{6}$$

The resulting power draws are shown in Tables II, III, and IV.

For the sensor subsystem, we assume it will be always on and establish a generous upper bound on the power draw from the Crossbow datasheets. Even using this upper bound, the power draw of the sensor has very little impact on the lifetime of the system.

Using these individual measurements, we calculate the power draw of the full system as the sum of the power draw of each subsystem in the appropriate state. Using this value, we calculate the amount of time it takes the entire system to drain the entire battery of the x86 subsystem.

B. Consistency

The goal of our first set of experiments is to vary the level of consistency required and observe the consequent lifetimes of several system configurations. To accomplish

Mode	x86	Sensor
Active (mW)	11,600	26.4
Suspended (mW)	180	0.056

Table II. The active and suspended power consumption of each subsystem running the time application. The active power consumption for the StrongARM was not measured since it never synchronizes with the time server.

Mode	x86	StrongArm	Sensor
Active (mW)	10,955	740	26.4
Suspended (mW)	180	40	0.056

Table III. The active and suspended power consumption of each subsystem running the web caching application.

this, we vary the interval at which the system wakes to perform synchronization from 0 (always on) to 0.5 hours. A wake interval of *i* minutes ensures that data is inconsistent for no longer than *i* minutes.

For each of these experiments, a wireless network is always present, the remote service is available, and new data updates are ready. For the time synchronization application, we assume that the time is synchronized whenever the system wakes. For the web caching application, the system maintains a 5 MB cache consisting of 15 web sites. For the IMAP synchronization application, the Turducken system fetches data updates and sends any queued, local updates upon waking. For this experiment, the x86 subsystem wakes for 2 minutes of every hour to simulate a user creating modifications to the local mail store. This store initially contains 4MB of mail in four separate folders. The queued updates to the local store are sent to the remote IMAP server when the StrongARM wakes to synchronize. In addition, new mail is sent to the inbox at a rate of 120KB per hour. During synchronization, the Turducken client fetches this mail.

The results of the time synchronization experiment are shown in Figure 3. When the system synchronizes frequently, the lifetime of the x86-only system degrades drastically while both the x86+StrongARM+sensor and x86+sensor configurations maintain nearly constant lifetimes. This is a consequence of the fact that when using a Turducken system, the x86 and StrongARM never need to

Mode	x86	StrongArm	Sensor
Active (mW)	11,720	810	26.4
Suspended (mW)	180	40	0.056

Table IV. The active and suspended power consumption of each subsystem running the IMAP synchronization application.

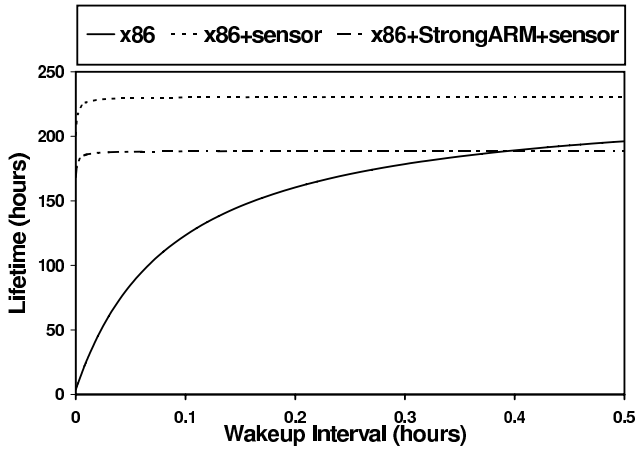

Fig. 3. The lifetime of three system configurations running the time synchronization application. As the system wakes more frequently, Turducken provides a more significant gain in lifetime.

come out of a suspended state. In this case, the x86+sensor configuration has a lifetime of about 225 hours and the x86+StrongARM+sensor has a lifetime of approximately 180 hours. The difference between these two configurations is a result of the energy draw of the StrongARM in suspended mode.

Figure 4 shows the results of the web caching experiment. We observe that the x86+StrongARM+sensor consistently performs better than the other configurations, providing a ten times improvement for always on operation and a three times improvement for less-stringent levels of consistency. Additionally, we observe that as the wake interval grows, the lifetime gain lessens. This is a result of the energy required to power the StrongARM subsystem in suspended mode. Similarly, the x86+sensor performs worse than the x86-only configuration for larger wake intervals because of the additional energy required to power the sensor device. Again, we can conclude from these observations that the higher the level of consistency required, the better the performance of Turducken.

Figure 5 shows the results of the IMAP synchronization experiment. The relative performance for IMAP synchronization is very similar to the web caching application, however, we observe that the absolute system lifetimes are significantly smaller. This is a result of the workload of IMAP synchronization. This particular experiment requires that the x86 subsystem wake periodically to simulate a user updating the local mail store, which costs additional energy. This application also introduces addi-

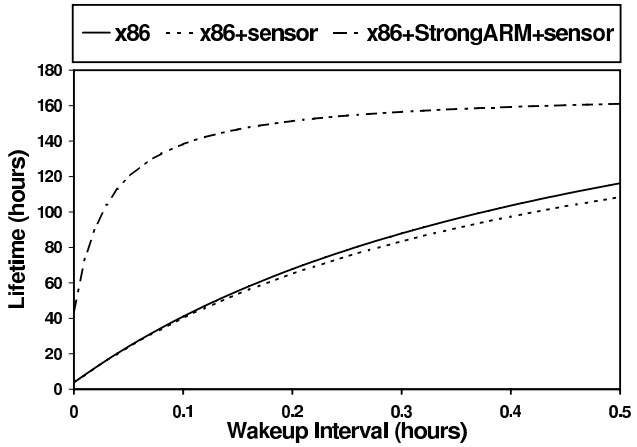

Fig. 4. The lifetime of three system configurations running the web caching application. For this application, the full Turducken system offers up to a 4 times longer lifetime and consistently performs better than the x86 only configuration.

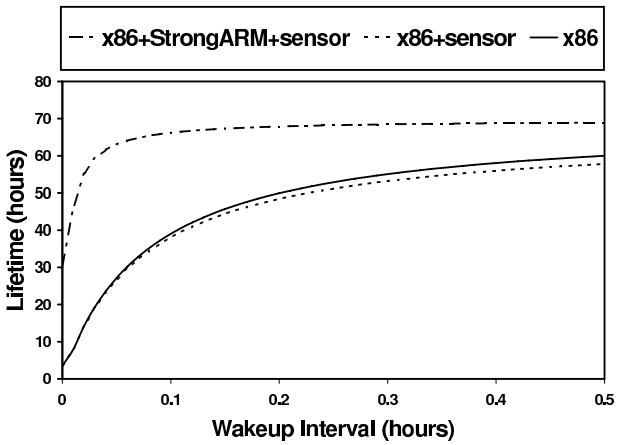

Fig. 5. The lifetime of three system configurations running the IMAP synchronization application. For this application, the full Turducken system offers a 1.5 times longer lifetime and consistently performs better than the x86 only configuration.

tional outgoing network traffic which impacts energy usage. However, we still observe that Turducken enjoys at least a 150% improvement in system lifetime for wakeup intervals less than six minutes. If the x86 subsystem does not perform periodic synchronization and only wakes up once an hour to send and receive updates its average lifetime is found to be 75 hours. However, the cost of this gain in system lifetime is that the expected time to get an update is $\frac{1}{p}$ hours, where p is the probability of a network connection being available. Since this latency can be large for small values of p, it is reasonable to sacrifice 13% of the system's lifetime in exchange for one-tenth the expected latency.

Figure 6 shows the average power draw for each subsystem. Each bar represents the total average power consumed by a particular configuration running a particular application. A bar is composed of several components that show each subsystem's contribution to the average power draw of the entire system. We further decompose each subsystem's contribution into its active and suspended modes. For example, for the x86-only configuration running the time application, the graph shows that the x86 spends most of its time suspended and a small amount of time in its active mode. Similarly, when it is augmented with a sensor, it spends all of its time suspended and the sensor expends a negligible amount of power. In the web caching experiment, the x86+StrongARM+sensor configuration is able to replace the active power of the x86 with the StrongARM. The mail experiment sees a similar gain; however, because the x86 spends more time in active mode, the resulting active power draw is larger. We observe that Turducken systems achieve lower average power consumption by replacing active power consumption in less efficient subsystems with more efficient ones.

C. Network and Service Availability

The goal of our second set of experiments is to vary the availability of a wireless network and the availability of the required service, and observe the consequent lifetimes of several system configurations. For this set of experiments, we look exclusively at the web caching application and fix the wake interval at 12 minutes. In the first experiment, we vary the probability that a wireless network is available from 0 (network never available) to 1 (network always available). In the second experiment, we fix the probability of wireless network availability at 1 and vary the probability that a set of web servers is reachable from 0 (web servers never reachable) to 1 (web servers always reachable). For this experiment, we assume that either all web servers are reachable or no web servers are reachable and we assume that it takes a trivial amount of time to determine reachability for all servers.

The results of varying the network availability are shown in Figure 7. When the probability of WiFi is low, the x86+sensor system performs best. This is because it can avoid waking the x86 if no signal is present. The x86+StrongARM+sensor enjoys the same benefit, but incurs the cost of powering the StrongARM in suspended

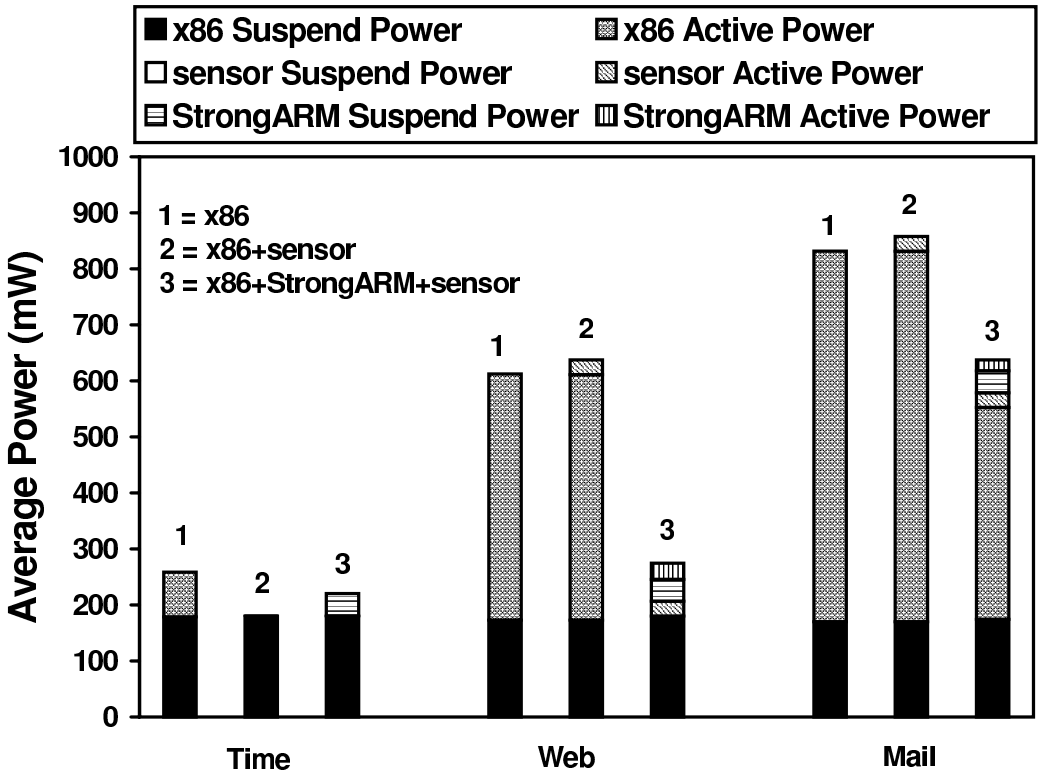

Fig. 6. This figure shows how each subsystem, in different states, contributes to the average power draw of the system as a whole. We observe that Turducken systems achieve battery lifetime gains by replacing active power consumption in less efficient subsystems with more efficient ones.

mode. Interestingly, the x86-only configuration performs similar to the x86+StrongARM+sensor for low probabilities. This implies that the cost to periodically wake the x86 to discover that no network is present is roughly equivalent to the cost of powering the StrongARM and sensor in suspended mode. As the probability of a network connection increases, the x86+StrongARM+sensor remains nearly constant, outperforming the other configurations by up to a factor of 2. This is a result of the energy saved fetching web pages using the StrongARM without waking the x86. We can conclude that Turducken provides a greater benefit as network coverage increases, and performs no worse than an x86 alone as coverage decreases.

The results of varying the availability of web servers is shown in Figure 8. The results for this experiment are similar to the previous experiment with the exception of the x86+sensor configuration. While the sensor can determine the presence of WiFi, it cannot determine the reachability of a web server. Therefore, the sensor must always wake the x86 to determine if the web servers are reachable. This costs the x86+sensor configuration up to 40 hours of lifetime. However, as the probability of service increases, the benefit of Turducken increases.

D. Observations

Our primary observation is simple: for many common distributed applications, a Turducken system can maintain a high level of consistency at a fraction of the power cost of a conventional laptop. This allows system behavior which has traditionally been ruled out in favor of conserving battery power. Naturally, there is a cost incurred when powering additional devices. This cost becomes noticeable when the system wakes up less frequently, reducing the benefit and retaining the cost of the additional hard-

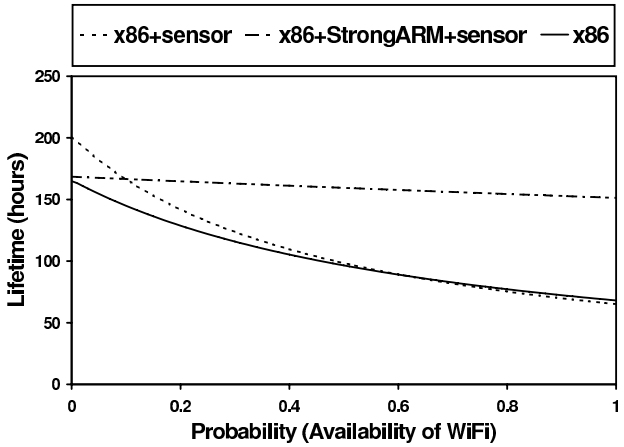

Fig. 7. This figure shows the battery lifetime of different configurations with respect to varying the probability of availability of WiFi. As network coverage increases, Turducken provides a greater benefit.

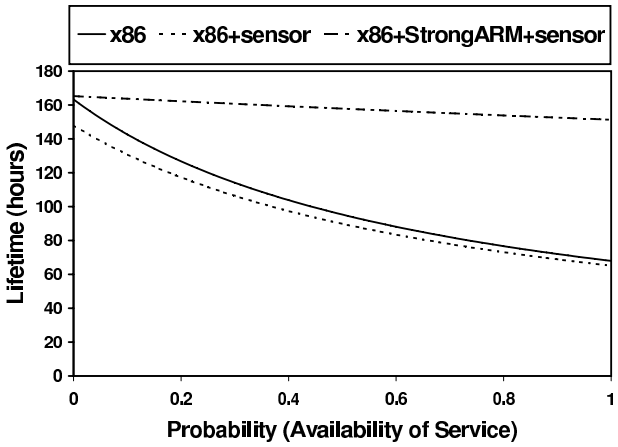

Fig. 8. This figure shows the battery lifetime of different configurations with respect to varying probability that a set of web servers is reachable. The benefit of Turducken is evident as the probability that the servers are available increases.

ware. Fortunately, even if the system never wakes up, the x86+StrongARM+sensor configuration will last 82% as long as the x86-only system.

Our experiments have also shown that the main limiting factor of the system's battery lifetime is the suspended power draw of the x86 subsystem. Our proposed solution to this is to use hibernation, which involves saving the machine's state to disk and powering it down. When the system is restored, it boots to the previously saved state. Clearly, it will cost more in both energy and latency to wake a device out of hibernation; however, during times of little or no activity (e.g. at night), using hibernation could result in significant power savings, potentially extending the system's lifetime to *over a month* on a single charge.

Additionally, it is clear that the benefit achieved is highly application dependent. For example, in the case of very simple applications, like time synchronization, the x86+sensor subsystem achieves the best performance. The best set of subsystems for a particular Turducken system depends on the target applications that the system will host.

VI. Related Work

A number of related research projects have explored strategies for reducing energy consumption of mobile devices. The Wake-on-Wireless project (WoW) [24] proposes augmenting a PDA with a wireless sensor. An in-network server notifies the sensor when it should wake the PDA such that it can serve incoming requests. The goal of WoW is similar to the goal of Turducken; low-power operating modes in mobile devices. However, this paper has shown the value in augmenting laptops with multiple subsystems that can *execute* synchronization jobs: subsystems perform many operations without waking up superior subsystems. Also, Turducken is a completely standalone system, not requiring any support from the wireless network. Some work has also looked at integrating multiple radios into a mobile platform [22, 20], and we use this idea in Hierarchical Power Management; however, we are focused on integrating entire subsystems. Mayo and Ranganathan proposed energy scale-down as a technique for saving power in mobile devices [17]. They make a similar observation that different mobile devices are optimized for different power points. They specifically reiterate the ideas of using wireless LAN energy management and multiple processor cores, as well as possibly using multiple displays in a mobile device.

Several projects have looked at managing energy from a whole-system standpoint. The Odyssey System [6] trades off resources, such as energy, for application fidelity. The ECOsystem [32] manages energy as any other operating system resource, enforcing fairness between applications, as well as setting global energy constraints. Simunic, et al. [25] propose a general method to manage energy consumption in across several system components. These systems are primarily designed for making short-term decisions and do not directly address non-reducible power in mobile devices.

An alternative to reducing the energy consumed while utilizing remote services is to ensure that the services are available locally, on the user's personal devices. A number of research projects have focused on ensuring availability of a user's personal data. The Personal Server [29] is a compact storage device which can provide reliable access to a user's personal data. Because the device does not have any kind of display, it operates at a low power point. However, unlike Turducken, the Personal Server provides a specific set of services and does not provide the same level of composability or flexibility in managing energy usage. Another approach is to ensure personal data availability by monitoring devices in a Personal Area Network (PAN), and migrating data from a device when its energy supply becomes critically low [23]. Again, this does not ensure that a device can use services provided outside of the PAN. Additionally, the focus of Turducken is to increase availability for a single, integrated system. However, we expect that the techniques developed for Turducken could also be useful managing energy and availability in a disconnected mobile distributed system.

VII. Conclusions

In this paper, we have presented the design and prototype implementation of Turducken, an approach that integrates, into a single system, a series of components that operate at various power levels. Turducken provides both a hardware and software infrastructure that can intelligently use available energy while maximizing device utility. We have demonstrated a prototype implementation and evaluated of its performance. We found that by using additional low-power subsystems for synchronization tasks, we can enable greater levels of consistency in distributed services. These techniques give the Turducken system a lifetime that exceeds that of a standard laptop by as much as ten times for always-on operation and three times for less stringent consistency requirements. Until there is a significant improvement in battery technology, strategies like Turducken are imperative for intelligently managing energy.

References

[1] M. Anand, E. B. Nightingale, and J. Flinn. Self-tuning wireless network power management. In *Proceedings of the 9th ACM International Conference on Mobile Computing and Networking (MobiCom'03)*, San Diego, CA, September 2003.

[2] G. Chinn, S. Desai, Eric DiStefano, K. Ravichandran, and S. Thakkar. Mobile PC platforms enabled with Intel Centrino mobile technology. *Intel Technology Journal*, 7(2), May 2003.

[3] F. Douglis, P. Krishnan, and B. N. Bershad. Adaptive disk spin-down policies for mobile computers. In *Proceedings of the 2nd Symposium on Mobile and Location-Independent Computing*, April 1995.

[4] F. Douglis, P. Krishnan, and B. Marsh. Thwarting the power-hungry disk. In *Proceedings of The USENIX Winter 1994 Technical Conference*, San Francisco, CA, 1994.

[5] K. Flautner, S. Reinhardt, and T. Mudge. Automatic performance-setting for dynamic voltage scaling. In *Proceedings of the Seventh ACM International Conference on Mobile Computing and Networking (MobiCom'01)*, Rome, Italy, July 2001.

[6] J. Flinn and M. Satyanarayanan. Managing battery lifetime with energy-aware adaptation. *ACM Transactions on Computer Systems (TOCS)*, 22(2), May 2004.

[7] K. Govil, E. Chan, and H. Wasserman. Comparing algorithms for dynamic speed-setting of a low-power CPU. In *Proceedings of the First ACM International Conference on Mobile Computing and Networking (MobiCom'95)*, Berkeley, CA, November 1995.

[8] D. P. Helmbold, D. D. E. Long, and B. Sherrod. A dynamic disk spin-down technique for mobile computing. In *Proceedings of the Second ACM International Conference on Mobile Computing and Networking (MobiCom'96)*, Rye, NY, November 1996.

[9] H. Huang, P. Pillai, and K. G. Shin. Design and implementation of power-aware virtual memory. In *Proceedings of USENIX Technical Conference*, San Antonio, TX, June 2003.

[10] A. D. Joseph and M. F. Kaashoek. Building reliable mobile-aware applications using the Rover toolkit. In *Proceedings of The Second ACM International Conference on Mobile Computing and Networking (MobiCom'96)*, White Plains, NY, November 1996.

[11] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th Annual International Symposium on Microarchitecture*, San Diego, CA, December 2003.

[12] A. R. Lebeck, X. Fan, H. Zeng, and C. S. Ellis. Power aware page allocation. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, November 2000.

[13] M. Litzkow, M. Livny, and M. Mutka. Condor: A hunter of idle workstations. In *Proceedings of the Eighth International Conference on Distributed Computing Systems*, June 1988.

[14] J. Lorch. A complete picture of the energy consumption of a portable computer. Master's thesis, University of California at Berkeley, December 1995.

[15] J. R. Lorch and A. J. Smith. Reducing processor power consumption by improving processor time management in a single-user operating system. In *Proceedings of the Second ACM International Conference on Mobile Computing and Networking (MobiCom'96)*, Rye, NY, Novemeber 1996.

[16] V. De La Luz, M. Kandemir, and I. Kolcu. Automatic data migration for reducing energy consumption in multi-bank memory systems. In *Proceedings of the 39th conference on Design automation*, New Orleans, LA, June 2002.

[17] R. Mayo and P. Ranganathan. Energy consumption in mobile devices: Why future systems need requirements-aware energy scale-down. *Lecture Notes in Computer Science*, 2003. Special Issue on Power Management.

[18] E. Musoll, T. Lang, and L. Cortadella. Exploiting the locality of memory references to reduce the address bus energy. In *Proceedings of the 1997 International Symposium on Low power electronics and design*, Monterey, CA, August 1997.

[19] C. M. Olsen and L. Alex Morrow. Multi-processor computer system having low power consumption. In *Proceedings of the Second International Workshop on Power-Aware Computer Systems*, Cambridge, MA, February 2002.

[20] T. Pering, V. Raghunathan, and R. Want. Exploiting radio hierarchies for power-efficient wireless device discovery and connec-

tion setup. In *Proceedings of the IEEE International Conference on VLSI Design*, January 2005.

[21] J. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under Unix. In *Proceedings of the USENIX Winter 1995 Technical Conference*, January 1995.

[22] P. Rodriguez, R. Chakravorty, J. Chesterfield, I. Pratt, and S. Banerjee. Mar: A commuter router infrastructure for the mobile internet. In *Proceedings of the Second International Conference on Mobile Systems, Applications, and Services*, Boston, MA, June 2004.

[23] S. Rollins, K. Almeroth, D. Milojicic, and K. Nagaraja. Power-aware data management for small devices. In *Proceedings of the Fifth ACM international workshop on Wireless mobile multimedia*, Atlanta, GA, September 2002.

[24] E. Shih, P. Bahl, and M. J. Sinclair. Wake on Wireless: An event driven energy saving strategy for battery operated devices. In *Proceedings of the Eighth ACM Conference on Mobile Computing and Networking*, Atlanta, GA, September 2002.

[25] T. Simunic, L. Benini, P. Glynn, and G. De Micheli. Dynamic power management for portable systems. In *Proceedings of the Sixth ACM International Conference on Mobile Computing and Networking (MobiCom'00)*, Boston, MA, August 2000.

[26] G. Stellner. CoCheck: Checkpointing and process migration for MPI. In *Proceedings of the Tenth International Parallel Processing Symposium*, April 1996.

[27] M. M. Theimer, K. A. Lantz, and D. R. Cheriton. Preemptable remote execution facilities for the V system. In *Proceedings of the 10th Symposium on Operating Systems Principles (SOSP'85)*, Orcas Island, WA, December 1985.

[28] V. Tiwari, S. Malik, and A. Wolfe. Compilation techniques for low energy: An overview. In *Proceedings of the 1994 IEEE Symposium on Low Power Electronics*, October 1994.

[29] R. Want, T. Pering, G. Danneels, M. Kumar, M. Sundar, and J. Light. The personal server - changing the way we think about ubiquitous computing. In *Proceedings of Ubicomp 2002: 4th International Conference on Ubiquitous Computing*, Goteborg, Sweden, September 2002.

[30] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *Proceedings of The First Symposium on Operating Systems Design and Implementation (OSDI'94)*, Monterey, CA, November 1994.

[31] A. Whitaker, R. S. Cox, M. Shaw, and S. D. Gribble. Constructing services with interposable virtual hardware. In *Proceedings of the First Symposium on Networked Systems Design and Implementation (NSDI '04)*, San Francisco, CA, March 2004.

[32] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat. Ecosystem: Managing energy as a first class operating system resource. In *Proceedings of the Tenth international conference on architectural support for programming languages and operating systems*, San Jose, CA, October 2002.

APPENDIX

The ratio of lifetimes is given by:

$$\frac{L_T}{L_L} = \frac{f_A^L \cdot P_A^L + (1 - f_A^L) \cdot P_S^L}{f_A^P \cdot P_A^P + (1 - f_A^P) \cdot P_S^P + P_S^L}. \quad (7)$$

Using the set of measurements found in Table III, we find that $P_A^L = 14.8 \cdot P_A^P$, $P_S^L = 0.24 \cdot P_A^P$, and $P_S^P = 0.05 \cdot P_A^P$. Substituting these into equation 7, we find that

$$\frac{L_T}{L_L} = \frac{0.26 + 14.8 \cdot f_A^L}{0.95 \cdot f_A^P + 0.31}. \quad (8)$$

As long as this ratio is greater than 1, Turducken provides a gain in system lifetime. If we define a factor $x = \frac{f_A^P}{f_A^L}$, then the condition becomes:

$$14.8 \cdot f_A^L > 0.95 \cdot x \cdot f_A^L + 0.05 \quad (9)$$

or

$$x < 15.57 - \frac{0.05}{f_A^L}. \quad (10)$$

So, for example if $x = 5$ and the laptop is on for a fraction of time greater than $f_A^L = 0.05/10.57$, or 17 seconds of every hour, Turducken provides a benefit. In other words if the web cache only runs 17 seconds of every hour the StrongARM can be up to 5 times slower at refreshing the cache.