Balancing Interactive Performance and Budgeted Resources in Mobile Computing

by

Brett D. Higgins

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2014

Doctoral Committee:

Professor Brian D. Noble, Co-Chair
Associate Professor Jason N. Flinn, Co-Chair
Professor Mingyan Liu
Associate Professor Z. Morley Mao

For Hannah and Naomi

ACKNOWLEDGEMENTS

Let me explain.
...no, there is too much.
Let me sum up.

Inigo Montoya

This dissertation owes its existence to the support of many individuals. My co-advisors, Brian Noble and Jason Flinn, provided relentless support, encouragement, and wisdom throughout the entire process. The clarity, impact, and soundness of my work has benefited tremendously from their guidance. Countless times, they have helped pull me out of the weeds of low-level hacking and helped me remember the big picture of my work. I am grateful to them for not giving up on me when it seemed like I might never finish, and I am honored to have put my name alongside theirs in my humble scientific contributions.

I thank my other committee members, Z. Morley Mao and Mingyan Liu, for their valuable feedback and perspective. Their input has helped me see areas in my work that needed more attention; this dissertation is more thorough and complete for their involvement.

Though not official committee members, TJ Guili, David Watson, and Christopher Peplin provided helpful insight and feedback during my research career. They helped me out of ruts when I struggled to make progress, and they provided encouragement during times when the research process became frustrating. Their industry experience provided an important perspective on my academic work.

I didn't interact much with Peter Chen between my prelim exam in 2009 and resuming teaching duties in 2012, but since then, I've gotten to know him as someone technically

brilliant yet extremely humble. I'm grateful to him for getting me started in teaching again and for involving me so deeply in the details of the course. I sometimes found myself enjoying teaching more than research, so his continual encouragement to spend more time on graduating—and his accomodation that allowed me to do so—are greatly appreciated.

For the most part, my CSE colleagues were far-flung across the fourth floor, but Azarias Reda was a stalwart companion (until graduating well before me). He toiled with me on my first big project as we both worked to get our PhDs off the ground. His example of diligence has helped spur me on to finishing. Timur Alperovich was a great friend to me during his years here. I think he paid me more visits during any given week than I made around the CSE building in a year. I was glad to have his encouragement and input when research was especially frustrating. Daniel Fabbri frequently checked in on me, taking time out of his own work to ask how I was doing, in research and in the rest of my life, and celebrating the small victories with me. More recently, Kyungmin Lee has been a great collaborator, helping me finish the last remaining bits of my dissertation while also juggling his own workload. I would still have a lot more left to do if not for his support.

From my very first undergraduate research role in summer of 2006, the students in the Pervasive Computing group welcomed me warmly, freely extending their guidance, knowledge, and friendship. Our group lunches made me feel included and valued, and I enjoyed our conversations on wide-ranging topics which, happily, I eventually grew to understand. Ed Nightingale drew me into the group with his equal depth of knowledge, insight, and kindness, and he was patient with me even when the oddities in my code became his responsibility. Dan Peek was another of my first collaborators, and he is also probably the student I've remained connected with the longest, as he's always sought me out and included me in outings whenever our paths cross. Kaushik Veeraraghavan would often stop by to check on me and always encouraged me to go deeper, think more for myself, and pursue every opportunity for growth and learning. Jodie Su and Mona Attariyan offered great advice and friendly conversation throughout our overlapping time at Michigan, and I could always look to them as examples. Benji Wester and Jake Czyz gave me equal parts warning and excitement about becoming a father while finishing my PhD, so that I was—

well, if not exactly *prepared* per se—at least not as blindsided as I might have been. I've enjoyed my random conversations with folks like Mike Chow, David Devecsery, and Zach Musgrave over the past couple years, as well as the liveliness they brought to my office.

The CSE department staff has been consistently stellar. Thank you all for doing your jobs so well, allowing me to focus on my research while worrying about nary a single administrative concern. Lauri, your friendliness made me smile every time I walked into the building. Steve, you handled my every request without breaking a sweat, and I could listen to you passionately rant about any subject, any day of the week. Dawn, you helped me calm down when I was stressing about failing to meet some deadline or follow some protocol. To these folks and so many others: your dedication helps make PhDs happen.

My family has been a constant source of support and love throughout this arduous process. Mom, Rob, Dad, Danni, Steve, Julianna—thank you for forcing me to explain what I've been doing for all these years in a way that someone who lives outside my head can understand. Thank you for the emotional and financial support that paved my way before grad school, and for your continued encouragement during my highs and lows. Thanks to Hannah's mother Julianna and sister Priscilla, who shared the duty of looking after Naomi many days while I scrambled to finish the dissertation and find a good job.

Hannah, I can scarcely find words to describe the impact of your love and committment throughout this process. You've picked me up when I've fallen flat on my face, you've kicked me into action when I needed it, and you've always found a way to remind me why I started off on this path in the first place. Your fortitude in sticking by me through it all has given me so much that I've needed to see it through. I love you.

Naomi, you are my favorite tiny human in the whole world. Thanks for joining our family at just the right time. You have brought more love out of me than I ever knew I had. I can't wait to explain all of this to you someday.

I feel strange giving a shout out to God as though I just won some sports championship, but I also know that if I was trying to do this on my own strength, I would have flamed out years ago. So—to him who is able to do immeasurably more than all I ask or imagine, according to his power that is at work within me—to him be the glory.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

ABSTRACT

Balancing Interactive Performance and Budgeted Resources in Mobile Computing

by

Brett D. Higgins

Co-Chairs: Brian D. Noble and Jason N. Flinn

In this dissertation, we explore the various limited resources involved in mobile applications — battery energy, cellular data usage, and, critically, user attention — and we devise principled methods for managing the tradeoffs involved in creating a good user experience. Building quality mobile applications requires developers to understand complex interactions between network usage, performance, and resource consumption. Because of this difficulty, developers commonly choose simple but suboptimal approaches that strictly prioritize performance or resource conservation.

These extremes are symptoms of a lack of system-provided abstractions for managing the complexity inherent in managing performance/resource tradeoffs. By providing abstractions that help applications manage these tradeoffs, mobile systems can significantly

improve user-visible performance without exhausting resource budgets. This dissertation explores three such abstractions in detail. We first present *Intentional Networking*, a system that provides synchronization primitives and intelligent scheduling for multi-network traffic. Next, we present *Informed Mobile Prefetching*, a system that helps applications decide when to prefetch data and how aggressively to spend limited battery energy and cellular data resources toward that end. Finally, we present *Meatballs*, a library that helps applications consider the cloudy nature of predictions when making decisions, selectively employing redundancy to mitigate uncertainty and provide more reliable performance. Overall, experiments show that these abstractions can significantly reduce interactive delay without overspending the available energy and data resources.

CHAPTER 1

Introduction

The most precious resource in a computer system is no longer its processor, memory, disk, or network, but rather human attention. [37]

In the past several years, mobile computing has grown remarkably in sophistication and adoption. Today's users are accustomed to now-commonplace computing experiences that would have been difficult to fathom a mere decade ago. Besides technical advances in portability, battery life, and ease of use, the rise of ubiquitous mobile computing has been largely driven by the wide availability of network connectivity—the ability to "carry the Internet in your pocket."

Unfortunately, current mobile networking applications and systems do not use mobile networks as effectively as they could. Though mobile users commonly encounter multiple wireless networks, applications are often designed from the perspective of only using one network at a time. Further, although these networks have different strengths and weaknesses, the default network selection policy on iOS and Android is a simple fixed priority order. WiFi is preferred over cellular, because WiFi is assumed to provide higher bandwidth, lower latency, and lower energy usage. This naive assumption that WiFi will always outperform cellular breaks down under user mobility and with modern cellular technology such as LTE.

State-of-the-art research in multi-networking has likewise produced suboptimal results. For example, there exist systems that will spread connections over multiple networks [47], as well as systems that replace the transport layer with a multi-path variant of TCP [106]. Lacking knowledge about the applications using those networks, these approaches cannot differentiate between, e.g., the user's request to check their email and the background download of cloud-synchronized data. As a result, user-centric data can end up queued behind large background downloads, and the user experiences unnecessary delays. Other systems allow applications to explicitly manage which data is sent on which network, but this places a heavy burden on application developers to manage their connections and decide how to use them, especially as network conditions can change rapidly. Further, fundamental challenges exist in mobile networking that are not as acutely felt in other contexts. For example, wireless radios, especially cellular radios, have an impact on battery life that often far outweighs the impact of the processor, a problem that is exacerbated by the fact that developers often are unaware of the energy impact of network behaviors and protocols [82].

Whereas mobile application developers are typically left on their own to overcome these challenges, we argue that the system can play a major role in improving user-visible performance, simply by providing developers with the right *abstractions* for balancing these tradeoffs. At a high level, we can view the collective concerns of mobile multi-network usage as a *budgeting* problem, wherein the battery energy and cellular data resources are *currency* that can be spent to purchase reductions in user-visible delay.

Three simple, familiar principles will guide our spending:

- Spend resources effectively.
- Live within your means.
- Use it or lose it.

In the context of mobile multi-networking, spending resources effectively is a matter of *selection* and *scheduling*. As described above, mobile applications have varied and rapidly changing behaviors, and the networks they use are likewise diverse and volatile. Thus,

using networks effectively is comprised of selecting the right network for the right traffic and scheduling traffic to prioritize user-centric traffic over background traffic. If the system is to accomplish this task, the application must tell the system which traffic is user-centric and which is not. In this way, we trade a small amount of complexity for a large amount of power; the system is able to make better decisions as a result.

Besides performance, mobile systems must also consider the use of limited resources such as battery energy and cellular data allotments. The consequence of overspending a resource budget is clearly undesirable; either the battery dies, rendering the device unusable until recharge, or the user incurs overage charges or throttling. Typically, mobile operating systems rely on static policies to minimize the usage of these precious resources—for example, preferring WiFi over cellular because WiFi data transfer is usually more energy-efficient. Similarly, many applications will avoid fetching large amounts of background data over cellular. However, these assumptions can often be invalid or irrelevant. For example, sometimes the cellular network has much stronger signal than the WiFi network, and thus it is much more efficient for both throughput and energy.

Though exhausting a resource budget is harmful, it should be considered equally harmful when a resource is conserved unnecessarily. At the time of resource replenishment (battery recharge or the end of the billing cycle) any unused resource supply is essentially wasted if it could have been spent to reduce user-visible delay. That is, a mobile device must use its resources before it "loses" them. Any instance in which user-visible performance is degraded to conserve an overprovisioned resource represents a wasted opportunity.

The immense complexity of this task and the tendency of applications to avoid tackling it suggests that assistance from the system is required. This dissertation describes the design and implementation of three abstractions that shift the burden of balancing resource/performance tradeoffs from the application to the system, managing the above tradeoffs on the application's behalf and spending resources judiciously to purchase improvements in user-visible performance.

It is my thesis that:

By providing programming abstractions that simplify the use of multiple networks, tailoring network use to application needs, and spending limited resources in a principled way to purchase reductions in delay, mobile systems can help applications significantly improve user-visible performance without exhausting those limited resources.

The rest of this document validates the thesis, in three major parts:

System support for application-aware multinetworking. First, recognizing the difficulty of using multiple networks effectively, we propose, implement and evaluate *Intentional Networking*, a system that treats multiple networks in a similar fashion to multiple cores on a multiprocessor and provides abstractions analogous to those found in multithreaded programming. Intentional Networking allows applications to characterize their network traffic with simple labels such as "foreground" and "background", enabling the system to reorder traffic to improve the latency of foreground traffic while maintaining the throughput of background traffic and striping across multiple networks to achieve their combined bandwidth. Experimental results show significant improvement in interactive delay with only minor overhead in background throughput.

Spending budgeted resources to reduce user-visible delay. Second, using application-level data prefetching as a mechanism to spend resources to purchase performance, we propose, implement, and evaluate *Informed Mobile Prefetching* (IMP), a system that helps applications prefetch the right amount of data at the right time. Since energy and cellular data resources are inherently *budgeted* resources (fixed-capacity and replenished in predicable intervals), IMP tracks resource spending and adjusts the amount of prefetching to spend as close as possible to each budget without exceeding it. IMP also tracks how often prefetched data is actually used, prioritizing prefetching for applications where it is most beneficial. Experimental results show that IMP reliably succeeds in meeting its resource budgets while significantly reducing interactive delay.

Incorporating predictor uncertainty in decision-making. Finally, we observe that mobile applications commonly make decisions based on predictors such as network bandwidth and latency—for example, choosing among multiple available wireless networks.

When these predictions are wrong, the delay resulting from the application's incorrect decision can significantly hurt user-visible performance. We therefore propose that mobile applications should explicitly consider the *uncertainty* inherent in their predictors when making decisions. Considering uncertainty introduces another opportunity to purchase performance by spending resources. By starting a *redundant* operation (e.g., transmission of the same data on additional networks) when the predictors are suspected to be unreliable, applications can mask the effects of uncertainty. Just as redundancy is often used in other domains to produce less variable performance in the face of uncertainty, we apply the same principles in the mobile networking domain. We implement three methods for quantifying the uncertainty of predictors and using the uncertainty to decide whether to initiate redundancy. Experimental results show that, when sufficient resources are available and uncertainty exists, our system reduces user-visible delay by up to a factor of two.

This dissertation explores the utility of each of these system services in detail. We begin in Chapter 2 by describing Intentional Networking, a system that provides abstractions for multi-network programming. In Chapter 3, we describe Informed Mobile Prefetching, a system for balancing the costs and benefits of background prefetching. Chapter 4 describes Meatballs, a library for enabling applications to incorporate predictor uncertainty when making decisions, employing redundancy to mitigate the effects of this uncertainty. Chapter 5 gives an overview of prior work related to the topics in this dissertation. The final chapter summarizes the dissertation's contributions and concludes.

CHAPTER 2

System support for application-aware network multinetworking

Our first step towards enabling applications to use multiple networks more effectively lies in managing the diversity of the available networks and harnessing the opportunities they provide. Because these networks have a wide variety of strengths and weaknesses, there is no single "best choice" in all cases, and such diversity of infrastructure is both a challenge and an opportunity. The challenge lies in managing these changing options to best meet each application's needs, which themselves vary with time. However, by doing so, we can provide significant benefits to applications, exploiting multiple networks concurrently and planning future transmissions intelligently. This is particularly valuable for applications with a mix of on-demand and *opportunistic* network activity—messages that still have value even if deferred for a time.

Unfortunately, current approaches to this problem are insufficient. At one extreme, the operating system or a middleware layer makes all routing and interface decisions on behalf of applications, in a one-size-fits-all solution [19, 47]. However, because the entities that make these decisions are ignorant of the intent of the applications that are using the network, they often miss opportunities for optimization. Worse, in an effort to preserve current wired-network semantics, persistent connections generally end up "stuck" on wide-area (but low-performing) networks. At the other extreme, the system makes applications aware of network changes by exposing the low-level details directly to them [20, 72], and

applications must explicitly choose among the available options. This approach is expressive, but neither simple nor elegant; managing multiple wireless networks unnecessarily complicates the task of the application writer.

Intentional Networking occupies the middle ground between these two extremes. In our approach, the system manages most of the messy details of discovering and characterizing available network options. Applications provide hints about traffic semantics using a small number of declarative *labels* to express intent. For instance, a label might differentiate between foreground traffic (e.g., a GUI-initiated request for which a user is waiting) and background traffic (e.g., an opportunistic message that need not happen at any particular time). The system then matches network traffic to available interfaces in an informed way.

Application data sent using different networks may arrive out of order. Constraining data delivery to follow in-order TCP-style semantics could dramatically limit the benefit seen by applications, since short, interactive messages would queue behind all previous opportunistic transfers. Thus, Intentional Networking allows applications to express relaxed ordering constraints for data delivery. The scheduling constraints for mobile network usage are similar to synchronization primitives used by threads running on a multiprocessor. Based on this observation, we provide two primitives: Isolated Reliable Ordered Bytestreams (IROBs), which provide the mutual exclusion synchronization of mutex locks, and ordering constraints, which provide the must-happen-before synchronization of condition variables.

Finally, there are times when none of the currently available network options are appropriate and network traffic is best deferred. For this scenario, Intentional Networking supports a *thunk* model of delayed execution in which the application registers a callback function to be invoked when circumstances change so that it becomes appropriate to transmit data with the specified label. Thunks let applications coalesce, rather than defer, redundant network messages; for instance, an e-mail client that periodically checks for new mail can send only one such request when an appropriate network becomes available.

The contribution of this work comes from defining simple and powerful abstractions for exposing the presence of multiple wireless networks to applications. Our work does

not define a new over-the-wire protocol, but instead provides a portable, user-level implementation that routes traffic over appropriate networks based on application hints. We show that, for many applications, application-aware network selection outperforms even idealized aggregation strategies that lack knowledge of application intent.

We have modified two existing applications to use Intentional Networking: BlueFS [70], a file system for pervasive computing, and the Mozilla Thunderbird [63] open-source email client. We have also created a new automotive participatory sensing application that uses our API. We evaluated the performance of these applications using measurements obtained by driving a vehicle through WiFi and cellular 3G network coverage. Compared to an idealized solution that makes optimal use of the aggregated available networks but lacks knowledge of application intent, our results show that Intentional Networking improves the latency of interactive messages from 48% to 13x for our three applications, while adding no more than 7% throughput overhead.

2.1 Design goals

We next list the major goals that drove the design and implementation of Intentional Networking.

2.1.1 Separate concerns

Our design is guided by the classic principle of separating policy and mechanism. Applications are best situated to determine the actual intent in using the network, e.g., whether a particular message is driven by interactive use or whether it is background traffic. This intent represents the policy for how data should be transmitted.

On the other hand, the operating system or a middleware library is best positioned to provide a common mechanism to implement the specified policies. A common mechanism makes deploying new applications that use multiple mobile networks considerably easier since each application must only provide hints as to its intent. The details of handling

multiple heterogeneous and intermittent mobile networks is encapsulated at lower layers of the system. A common mechanism can also aggregate heterogeneous data transmissions from multiple applications.

Thus, Intentional Networking is designed to have a separation of concerns in which applications disclose policy decisions by labeling the data they transmit and a lower layer of the system implements the mechanism that enacts the policy by mapping data to the networks that best match the labels at the time the data is transmitted.

2.1.2 Be Qualitative

Our design is also guided by the classic principle of keeping the interface as simple as possible, without unduly sacrificing expressiveness. This has resulted in a minimalist, qualitative interface. For instance, we could have required each application to disclose detailed quantitative specifications of the characteristics of the traffic it expects to generate, as well as the quality of service that it requires. However, such a complex interface would place a considerable burden on the application programmer, that of carefully tuning for each possible workload, making it unlikely that the casual developer would use our system.

This principle led to several decisions. Rather than use quantitative specifications, applications express their intentions using only qualitative attributes over the data; i.e., whether a transmission will be small or large, and whether it is interactive or background traffic. We do not mandate what constitutes “small” vs “large”. We allow the application to use these labels as it sees fit. While we may eventually add more attributes to our labels as our experience with the system grows, the current interface is sufficiently expressive to handle several complex applications, as discussed in Section 2.4.

2.1.3 Embrace Concurrency and Failure

Our original goal for Intentional Networking was to provide a single-socket abstraction that assigns labeled traffic to the most appropriate network. However, single-socket semantics require data to be delivered in-order for TCP connections. Unfortunately, this severely

limits the set of optimizations possible when using multiple networks simultaneously.

After several false starts, it became clear to us that going from one to many networks is akin to the transition from single-threaded programming to multi-threaded programming. Some interleavings of execution orders are very useful and desirable, but others lead to incorrect computations.

Just as concurrent systems include mechanisms to allow the programmer to rule out incorrect orderings, we added synchronization abstractions to express both atomicity and happens-before constraints. These mechanisms are both simple and expressive, and are familiar concepts to programmers with training in monitor-style concurrency control.

In addition to expressing such ordering constraints, we also needed mechanisms to deal with partial failure. There are times when some traffic would be ill-served by any available transmission alternative. Therefore, we provided a callback mechanism—similar to exceptions or continuations—to handle delayed transmissions or disconnections.

2.2 Abstractions and interface

In this section, we describe the Intentional Networking application interface. We first describe the basic abstractions in the interface. Applications use *labels* to communicate their intent. These are meaningful in the context of *multi-sockets* and are expressed over message units called *IROBs (Isolated Reliable Ordered Bytestreams)*. IROBs provide atomicity (mutual exclusion); applications may also specify *ordering constraints* among IROBs. When operations must be deferred, applications may register *thunks* to resume them. After describing these fundamental abstractions, we show the Intentional Networking API in Section 2.2.6.

2.2.1 Labels

The label is the principal abstraction available to applications. It is the mechanism by which applications declare the properties of any particular network message. Labels are

Properties	Possible values
Interactivity	Foreground vs. Background
Size	Small vs. Large

Table 2.1: Intentional Networking label properties

system-defined qualitative properties of the message. Our present implementation supports only four labels across two dimensions, interactivity and size, as shown in Table 2.1. A message's label is set to foreground if a user-visible event is waiting for the response. A message is background if its timely delivery is not critical to correct behavior. For example, many hints [99] need not be sent. The small label describes messages that are latency-dominated such as single-packet RPCs, while the large label describes other messages such as those containing multimedia data. We expect to add further dimensions and label values as our experience with applications grows. Yet, the eventual number of possible label values will remain small since interface simplicity is one of our main design goals.

2.2.2 Multi-Sockets

Labels are used in conjunction with label-aware sockets. We call such sockets *multi-sockets*. Intuitively, a multi-socket multiplexes several different labels across a single virtual socket. For the most part, multi-sockets behave exactly as normal ones do. However, multi-socket send calls take a label that is used to assign packets to the best possible interface. Note that the sender is always the entity responsible for assigning labels, and as a consequence, `recv` does not require a label. While we could imagine using one to implement a filtered receive, we have not had to do so for any of our applications so far.

A multi-socket is a single logical connection that dynamically instantiates and uses actual TCP connections over one or more physical interfaces. Multi-sockets provide encapsulation: they hide the presence of multiple network interfaces, routes, and connections from applications. Multi-sockets also encapsulate transient disconnections caused by events such as passing through a wireless dead zone. Applications specify only labels,

which are used by the Intentional Networking traffic manager to choose the right network over which to send data. Applications may optionally be notified about network unavailability on a *per-label*, not per-network basis, through the use of thunks, which are deferred execution environments that execute when an event occurs. Thunks are described in more detail in Section 2.2.5.

Like TCP sockets, multi-sockets support a reliable delivery abstraction. However, multi-sockets relax TCP's ordering constraints by allowing bytes to be reordered subject to application-specified mutual exclusion and ordering constraints, as described in the next two sections.

2.2.3 IROBs

An IROB is the unit of network transmission to which labels are applied. The multi-socket interface guarantees that each IROB is received atomically; i.e., the bytes of the IROB are produced in order without intervening bytes from other network transmissions. However, individual IROBs may be reordered with respect to one another. In other words, an IROB sent after a previously sent IROB may be seen first by the application reading data from the receiving multi-socket. Yet, bytes from the two IROBs will never be intermingled. IROBs thus provide *mutual exclusion* in the same manner that locks provide mutual exclusion for threads in a multithreaded program.

2.2.4 Ordering constraints

Since some applications require *ordering constraints* between IROBs, the multi-socket interface supports the declaration of such constraints. Each multi-socket assigns a unique, monotonically increasing identifier to each IROB. When creating a new IROB, the application may specify the identifiers of any IROB that must be received prior to receiving the one being created. Ordering constraints may only specify IROBs that have a lower unique identifier; this guarantees that such constraints are deadlock free. Applications that desire the sequential byte stream of a TCP socket specify that each IROB must be received after

the one with the next lowest identifier; our API provides default send calls with this behavior for simplicity. However, many of our applications have looser constraints; for instance, the BlueFS file system client requires that asynchronous writes be ordered sequentially with respect to one another, but allows them to be arbitrarily ordered with respect to all other RPC types. The ordering constraints in multi-sockets are similar to those provided by condition variables for threads in a multi-threaded program.

2.2.5 Thunks

It is possible that a labeled IROB may not have any "appropriate" network available at the time it is sent. For example, consider an opportunistic bulk transfer initiated when only a low-bandwidth link is available. Such a transfer would preferably be done at a later time, when a high-bandwidth link is encountered. Alternatively, the mobile computer may be in a wireless dead-zone, with no connectivity.

Naturally, we do not want applications to have to poll for such a link. We also do not want applications to have to establish new connections after short periods of transient disconnection. However, in keeping with our design goals, we want to expose such events to applications when appropriate.

In our interface, the operations that create IROBs take an optional *thunk* argument, which is a function/argument pair that will be used to inform the application about IROBs that cannot be immediately sent due to the lack of an appropriate network. When an IROB is deferred, the call that takes the thunk argument returns a special return code. Later, when data with the specified label can next be transferred, the library notifies the application by calling the thunk function with the specified arguments. The ownership of the argument's resources passes with the thunk, and the handler must take responsibility for them. Thunks may be canceled—for example, if a subsequent `send` would invalidate a prior thunked one.

Thunks are useful for applications that send periodic messages, such as checking for new e-mail. Buffering redundant messages during disconnected periods and sending them all later is clearly undesirable. Instead, such applications register a thunk for the send and

Function	Arguments and return values
ms_socket	(IN family, IN type, IN protocol, OUT multi-socket);
ms_begin_irob	(IN multi-socket, IN label, IN dependencies, IN thunk, IN thunk_data, OUT irob_id);
ms_irob_send	(IN irob_id, IN buf, IN length, IN flags, OUT bytes_sent);
ms_end_irob	(IN irob_id);
ms_send	(IN multi-socket, IN buffer, IN length, IN flags, IN label, IN thunk, IN thunk_data, OUT bytes_sent);
ms_recv	(IN multi-socket, IN buffer, IN length, IN flags, OUT label, OUT bytes_rcvd);

This figure shows the Intentional Networking API for creating and using multi-sockets. Besides the functions shown, multi-sockets also support the traditional socket functions; e.g., `accept`, `select`, and `setsockopt`.

Table 2.2: Intentional Networking API

are notified when an appropriate network is available. The thunk handler sends only one polling request, thereby preserving valuable network bandwidth.

2.2.6 API

Table 2.2 shows the most important functions in the Intentional Networking API. The `ms_socket` call creates a new multi-socket, and the `ms_connect` call connects it to a remote endpoint, which is specified in the same way as for the `connect` system call. Thus, the only difference between `ms_connect` and the standard `connect` system call is that the first argument is a multi-socket.

Typically, we modify an application by replacing the socket, connect, listen, and accept calls with their `ms_*` counterparts. Applications create a new IROB through `ms_begin_irob`, passing a label that describes the atomic message, as well as any ordering constraints. This function also takes an optional thunk and data to be passed to the thunk function. The application then calls `ms_irob_send` to specify the data sent as part of the IROB; typically,

we replace `send` with `ms_irob_send`, one-to-one. The application uses `ms_end_irob` to tell the library that no more data will be sent for the IROB. The `ms_send` call is provided as a convenience; it creates a new IROB that depends on all previous IROBs, specifies the data that comprises the IROB, and ends the IROB. If an application uses just `ms_send` calls, it will provide the behavior of TCP with labels, though no reordering will occur.

The `ms_recv` call returns a label. This is useful for server applications that wish to reply to a client request using the same label provided by the client for the original request. For instance, an IMAP server may wish to reply to client background requests with a background label and reply to foreground requests with a foreground label. Although not shown in Table 2.2, multi-sockets export similar functions to those provided by traditional sockets such as `listen`, `accept`, `select`, and `setsockopt`.

2.2.7 Discussion

It is useful to consider what an application would need to provide on its own to achieve application-aware functionality equivalent to Intentional Networking. First, an application would need to discover new network options, open sockets for each network option, and monitor the connection quality of each network in order to decide which network to use for each transmission. To prioritize on-demand traffic, the application might create multiple sockets per network, then use a platform-specific method to prioritize traffic from one socket over the other. The application would also need to stripe traffic across connections to improve throughput, then manage the inevitable re-ordering of data that arises from such striping. Finally, the application might poll to achieve the functionality of thunks that allows traffic to be altered or dropped if an appropriate network is not currently available.

In contrast, the Intentional Networking abstraction makes this functionality the responsibility of the lower layer of the mobile system, not the application. The application need only annotate its traffic with the simple API in Table 2.2 to achieve the same functionality. While strategies that ignore intent can be implemented without application modification, our evaluation shows that such application-oblivious strategies substantially underperform Intentional Networking.

2.3 Architecture

When we began our work, we faced a decision about whether to implement Intentional Networking at user-level or in the kernel. Good reasons exist for both choices. A kernel implementation can improve performance by integrating tightly with the network stack. However, we decided to implement our initial prototype at user level to provide portability and simplify deployment. Given the wide array of operating systems used by mobile computers and cell phones, a user-level implementation is much easier to port to new platforms. Further, many popular mobile platforms do not allow kernel modifications at all. Even with a user-level implementation, our prototype performs well, as shown in Section 2.5. Our implementation consists of a connection scout daemon that runs on the mobile client, plus a library implementing the API.

2.3.1 Connection scout

The connection scout is a stand-alone user-level process, which we have adapted from the implementation of Virgil [67]. It is responsible for discovering and evaluating the performance of the networking options available at any given time. For each of the mobile computer's wireless network interfaces, the connection scout periodically attempts to establish network connections. After a connection is established, the scout measures the throughput and latency of the connection through active probing. The multi-socket library queries network availability and performance data from the scout using a pipe.

We envision that the connection scout could eventually leverage a lower layer that allows a mobile computer to simultaneously connect to multiple access points via a single physical interface [20, 69] by having the lower layer expose each access point as a separate virtual interface.

2.3.2 Library

The Intentional Networking library exports the interface described in Section 2.2.6. It is responsible for mapping IROBs to interfaces based on their associated labels. For each multi-socket, the library dynamically creates separate TCP sockets for each interface over which it decides to send data. A multi-socket connection persists until no TCP connection can be maintained using any network interface (for example, if the mobile computer moves out of range of a WiFi access point and no other network options are available) or the multi-socket is closed.

We chose to use TCP primarily for simplicity. Since we are not designing a new over-the-wire protocol, TCP's reliability mechanisms limit the amount of effort we must spend implementing ordered delivery of bytes within an IROB or retransmission of bytes lost due to congestion in the network. For the purposes of our prototype and evaluation, we have not found TCP to be a significant source of overhead, but we imagine that a more highly tuned implementation of Intentional Networking would integrate more tightly with the transport layer for optimal performance.

When an initial connection is established over the first TCP socket, a mobile client sends its peer data that includes its available IP addresses and the estimated bandwidth and latency for each one. It piggybacks updates to this information on Intentional Networking headers, as described below. With this information, either peer may establish a new TCP connection when it expects that a new connection would be best suited for data with a specific label.

The library maps labels to TCP connections using active and passive estimates of network bandwidth and latency. The connection scout provides an initial active measurement of connection quality when a new network option is discovered. As the library sends data over the connection, it measures the response time for individual transmissions to generate passive measurements. The connection scout provides periodic active measurements that are used to assess quality during periods where no data are transmitted and passive measurements are unavailable. Active and passive measurements are combined using a flip-flop filter [49] to derive a running estimate of the current connection quality.

The library uses the following strategy to map labels to TCP connections. Foreground data is given the highest priority. IROBs with the {`foreground, small`} label are sent over the lowest latency TCP connection. IROBs with the {`foreground, large`} label are sent over the highest bandwidth connection. These may be the same connection (e.g., if there is only one interface that currently offers connectivity). The actual physical interface used for a specific label may change over time as estimates of link characteristics vary. Background data is given lower priority than foreground data.

Background IROBs are striped over all networks that are not currently sending foreground data. Large, background IROBs are broken into smaller chunks, each of which may be sent over a different network. Our decision to stripe background, but not foreground, IROBs is driven by the different goals of the two labels. A foreground label demands low response time; unfortunately, striping can increase the latency for the last packet to arrive unless the networking layer correctly predicts instantaneous latency for each link. In contrast, the background label specifies data that is not latency-sensitive; thus, a striping strategy that maximizes the utilization of each link is ideal.

The library maintains a collection of IROBs that have been created by the application. Each IROB contains data sent by the application but not yet acknowledged by the peer library on the other side of the multi-socket connection. This means that there is some double-buffering with data contained in the kernel TCP socket buffer; this double-buffering is one performance artifact of a user-level implementation.

Each label has a linked list that indexes all IROBs with that label in FIFO order. Each TCP connection has a list of the labels that it currently is eligible to send; for instance, the lowest latency TCP connection may send either background or foreground data. For each connection, the library sorts the labels in order of preference, i.e., with foreground labels preferred over background ones. When the network is able to send data, the library pulls data from the first IROB on the list associated with the label with highest priority. If no such IROB exists, it moves to the label with next highest priority, and so on. The library encapsulates the IROB data with a 32-byte Intentional-Networking-specific header that includes the IROB identifier and its label, followed by the IROB's ordering constraints.

Additional information may be piggybacked in the header, such as current estimates of network bandwidth and latency. The library is not constrained to send all of an IROB's bytes over a connection at once; it may decide to break an IROB into smaller chunks, each of which is sent with an individual header. As an example, this allows the library to start sending IROB data before the application has called `ms_end_irob` to indicate the end of the IROB. IROB chunks sent over multiple TCP connections are re-assembled by the receiving library so that the bytes of each IROB are delivered atomically and in order.

The receiving library acknowledges each IROB. The acknowledgment is not constrained to travel over the same network over which the chunk was received. This can be useful if, for example, a TCP connection becomes unavailable after data has been received but before the acknowledgment is sent. For efficiency, acknowledgments are piggybacked on outgoing message headers if a message is queued when the acknowledgments are generated. While Intentional Networking generally relies on the underlying TCP acknowledgments and retransmissions to provide reliability, some additional work is required when a TCP connection breaks. In such instances, the sending library polls the receiving library over a different TCP connection to learn the state of any unacknowledged IROBs that were in flight when the connection was broken.

One challenge is that background requests may be sent over the same network as foreground requests. If the library were to send a large amount of background data, it might unnecessarily delay the foreground data. While a kernel implementation could prioritize one over another at the protocol level, a user-level implementation must use other methods. We have chosen to adapt the *anticipatory scheduling* algorithm [42] to solve this dilemma. Since high-priority traffic is likely to exhibit temporal locality, we bound the amount of data buffered in the kernel by a lower-priority IROB to no more than the amount of data that can be sent within 50 ms if a high-priority IROB has recently been sent by the application. This bound is increased to one second as long as no further high-priority IROBs are observed. Anticipatory scheduling therefore optimizes for low latency for foreground IROBs during periods when many such IROBs are sent, and for high throughput for periods with few foreground IROBs.

The library that receives data guarantees that bytes are delivered to the application in a manner that obeys the mutual exclusion and ordering constraints specified by the sender. Once at least one byte from an IROB has been received by an application, no other bytes from another IROB are delivered until all bytes from the first IROB have been delivered. For this reason, the library does not deliver bytes from a low-priority IROB until it has received all of its bytes. Further, the library buffers an IROB until its ordering constraints are satisfied. For instance, if IROB 2 depends on IROB 1, but is received first (because the two IROBs were transmitted over different networks), the library buffers IROB 2 until after IROB 1 has been received by the application. If two IROBs are eligible to be received, the library delivers the higher-priority one first (e.g., a foreground IROB will be received by the application before a background one). Within a label type, FIFO ordering is used to decide which IROB to deliver.

If a TCP connection fails while IROBs are being transmitted, any remaining data for those IROBs will be sent over the next most appropriate connection. The library masks transient disconnections unless all TCP connections fail simultaneously.

When multiple Intentional Networking applications execute concurrently, the activities of all processes are coordinated through shared memory variables and synchronization. We assume that the links closest to the mobile computer are the bottleneck, and that most of these are shared across all paths of interest. Therefore, each library instance updates a shared variable containing the amount of buffered but unsent data on each network that may send foreground data. The total amount of such data across all processes is not allowed to exceed the limit described above for the anticipatory scheduling algorithm, guaranteeing good foreground performance. If an application not modified to use Intentional Networking executes concurrently with one that does use Intentional Networking, the applications use separate connections and do not coordinate with each other. The Intentional Networking application will adjust its estimates of network quality based on passive observations during its execution, and hence will account for the competing traffic in its decisions.

The Intentional Networking library handles connections between two mobile computers with multiple interfaces by potentially establishing a connection per interface-pair. We

do not describe this scenario further as our applications to date have all involved communication between a mobile client and a single-homed server.

2.4 Applications

We have modified three applications to use Intentional Networking: BlueFS, a distributed file system for mobile clients; Thunderbird, the Mozilla e-mail and news client; and a vehicular sensing application of our own creation.

2.4.1 BlueFS

BlueFS [70] is an open-source, server-based distributed file system with support for both traditional mobile computers such as laptops and consumer devices such as cell phones [79]. A BlueFS client interacts with a remote server through remote procedure call, augmented with bulk-transfer capabilities. BlueFS inherits parts of its design from previous mobile computing file systems such as Coda [50]. BlueFS clients fetch file and directory information on demand from a remote file server. Files are cached locally on the client. Modifications to file system data are propagated asynchronously to the remote server in the background, in the same manner as Coda's weakly-connected mode [64]. Clients also prefetch data from the server into their caches to improve performance and support disconnected operation.

We adapted BlueFS to use Intentional Networking by modifying its RPC stub generator to take three optional arguments: an Intentional Networking label, ordering constraints, and a thunk. The RPC package uses one socket to connect a client and server; we changed this to be a multi-socket. We also modified the RPC package to create a new IROB for each RPC request and response message with the label, ordering constraints, and thunk specified by the BlueFS client.

We labeled RPCs that are used to prefetch data and asynchronously write modifications back to the server as {`background, large`}. Other RPCs which fetch data on-demand

from the server were labeled as `foreground`; the vast majority of these are `small` since BlueFS fetches data on a per-file-block basis. While it is true that some demand fetches may be from applications that are not interactive, the Posix API is insufficient to express this to file systems. Therefore, the conservative approach of treating all such requests as `foreground` seemed best.

Since the file server must see modifications in order, we used Intentional Networking ordering constraints to specify that each file modification IROB depends on the previous one of that type (e.g., all such IROBs are delivered sequentially with respect to one another). However, no constraints are expressed with respect to IROBs of other types, so, for example, the library may reorder an on-demand fetch IROB ahead of a modification IROB.

The server RPC library responds to each RPC with the same label used to send the original request. Since the RPC library already uses a unique identifier for each RPC, matching requests and responses was trivial.

In total, we added or modified 400 lines of code in the RPC library to support Intentional Networking, as well as 134 lines of code in BlueFS. For comparison, the original code base has over 44,000 lines of code.

2.4.2 Mozilla Thunderbird

We also used Intentional Networking to improve the interactive performance of Thunderbird [63], Mozilla's mail and news client. For simplicity, we used an IMAP proxy to intercept traffic between Thunderbird and an IMAP server. The proxy running on the mobile computer prefetches e-mail contents and headers from the IMAP server and stores them on the client's local disk. We replaced the proxy's outgoing connection with a multi-socket and labeled the IMAP messages. Prefetch requests and responses are labeled as `background`, while on-demand fetches triggered by the user via the Thunderbird GUI are labeled as `foreground`. Requests are all labeled as `small`, while responses are labeled as `small` or `large`, depending on their actual size. Each response message from the IMAP server is given the same `background` or `foreground` designation as the request that gen-

erated the message. Like the previous application, the IMAP protocol includes a unique identifier for each request/response pair, making it trivial to match requests and responses. Out of 2951 lines of proxy code, we added or changed 124 lines to support Intentional Networking.

2.4.3 Vehicular participatory sensing

Finally, we created a new application targeted at participatory sensing for corporate vehicle fleets. This application is based on specifications for a research/teaching platform developed by Ford Motor Company. The application continuously collects data from a vehicle's internal networks and sensors at a data rate of approximately 25 KB/s. Given ample network bandwidth, the raw data are sent to a cloud server, where they are stored. Raw data can be used for suggesting preventative maintenance, route optimization, improving fuel economy, and other participatory sensing uses.

Since automotive hardware must last a minimum of 10 years and cost reduction is key to profits, the vehicle is expected to have limited storage and computational resources. Therefore, raw data is dropped if sufficient network resources are not available to transmit it immediately. In addition to the raw data, a short 4 KB summary of the data is included. By default, metadata summaries are sent every second, though if bandwidth is insufficient, summaries are generated over longer time periods, e.g., the last 10 seconds. Finally, the vehicle also transmits urgent updates when it encounters anomalous conditions, such as information from the traction control system that indicates slippery road conditions or sudden braking. These updates can be used to warn other vehicles of difficult driving situations such as ice, accidents, or unexpected traffic.

The Intentional Networking version of this application labels metadata summaries as {`background, small`} IROBs and raw data messages as {`background, large`} IROBs. Urgent updates are {`foreground, small`} IROBs. We use ordering constraints to ensure that each raw data IROB is received after the metadata message that summarizes it. The application uses the thunk interface to receive a callback if a background IROB cannot be immediately sent. If the callback is not received before the next raw data message is

collected, the previous raw data message is dropped, and the metadata summary is updated to average values over the current time period and all previous ones since the last metadata summary was sent.

We also created an additional version of the application that does not use Intentional Networking. This version uses `select` to determine when the socket buffer is full. Like the Intentional Networking version, this version omits sending raw data and aggregates metadata when it is unable to transmit for more than a second. Our vehicular sensing application has 2080 lines of code. We added or changed 186 lines to support Intentional Networking.

2.5 Evaluation

We evaluated Intentional Networking by measuring how much it improves network performance for our three applications. Our evaluation uses two different types of network connectivity scenarios: synthetic network conditions that are used as microbenchmarks and traces of actual network connectivity collected from a vehicular testbed. In the latter case, the use of traces provides experimental repeatability and allows a careful comparison among strategies.

2.5.1 Experimental Setup

2.5.1.1 Testbed

We ran all experiments on a testbed in which the client computer is a Dell Precision 350 desktop with a 3.06 GHz Pentium 4 processor and 1 GB DRAM, running a Linux 2.6.22.14 kernel. All servers run on a Dell Optiplex GX270 desktop with a 2.8 GHz Pentium 4 processor and 1 GB DRAM, running a Linux 2.6.18 kernel. These computers are connected via local 100 Mbps Ethernet connections. We emulate wireless network conditions by inserting delays using the netem [56] network emulator and throttling throughput using the Linux Advanced Routing and Traffic Control tools [55].

Scenario	Network Type	Bandwidth (Mbps)	RTT (ms)	Connectivity
Crowded hotspot	Low latency	0.6	20	Continuous
	High bandwidth	2.0	400	Continuous
Intermittent	Wide-area	0.3	400	Continuous
	WiFi hotspots	3.0	60	Intermittent

Table 2.3: Synthetic network scenarios used in the evaluation

For Intentional Networking experiments, we modified the client and server component of each application to use our API as described in the previous section and linked each with the Intentional Networking library. We also ran the connection scout on the client computer. For other experiments, the applications are unmodified. All reported values are the mean of 5 trials; graph error bars show 95% confidence intervals.

2.5.1.2 Synthetic Microbenchmarks

To better understand the behavior of Intentional Networking, we created synthetic network traces that emulate the two network scenarios shown in Table 2.3. These synthetic traces are intended to help us understand our system's behavior in controlled scenarios rather than precisely emulate actual network behavior. The first scenario replicates the network conditions that would be seen by a user with a high-bandwidth 3G network card sitting at a crowded WiFi hotspot. The 3G network offers higher bandwidth than the crowded AP, but it also inflicts significantly higher latency on network packets. Thus, each network is superior for different types of traffic. Empirically, we observed several instances of such scenarios in the network traces we collected, as described in the next section.

The second scenario emulates a vehicular setting in which a low-bandwidth, high-latency cellular network is continuously available. Opportunistic WiFi connections that offer better bandwidth and latency are intermittently available. We use empirical distributions from the Cabernet project [30] to model the availability of WiFi access points. The

distribution of access point encounters has a median of 4 seconds, a mean of 10 seconds, a 99th percentile of 250 seconds, and standard deviation of 0.4 seconds. The distribution of time between APs has a mean of 32 seconds and a median of 126 seconds. Our traces show several instances in which WiFi dominates 3G. However, the traces indicate that this is a simplified view: 3G may also dominate WiFi in many instances; one may offer better uplink bandwidth and worse downlink bandwidth; etc.

2.5.1.3 Trace-driven evaluation

While the microbenchmarks above help us understand the behavior of our system, we were curious to see how well it would perform in actual vehicular networking conditions. To generate repeatable experiments, we used a two-part process in which we first drove a vehicle with WiFi and Sprint 3G network interfaces. We continuously measured the downlink and uplink bandwidth and latency available through each network interface through active probing to a server at the University of Michigan. We also noted when each type of network was unavailable. The WiFi trace includes only those public APs to which we could associate and establish connections. We collected the traces in Ann Arbor, MI and Ypsilanti, MI at different times of the day. Trace 1 offers better 3G performance overall but encounters fewer public APs. Its median 3G bandwidth is 382 Kbps downlink and 57 Kbps uplink, with maximum bandwidth of 1.3 Mbps downlink and 78 Kbps uplink. Trace 2 has more WiFi access but poorer 3G performance. Its median 3G bandwidth is 368 Kbps downlink and 40Kbps uplink, with maximum bandwidth of 1.2 Mbps downlink and 74 Kbps uplink. Trace 1 has WiFi coverage only 7% of the time, with a median session length of 11 seconds; the longest session was 72 seconds. Trace 2 has WiFi coverage 27% of the time, with a median session length of 7 seconds; the longest session was 131 seconds. In both traces, there are periods where each type of network dominates the other, and where each type of network has better bandwidth but worse latency than the other. Thus, the network conditions are much more variable than in either of our microbenchmarks.

In the second step, we used the traces to drive the emulator in our testbed. Our traces lasted 138 and 36 minutes, respectively. Because our experiments run for different dura-

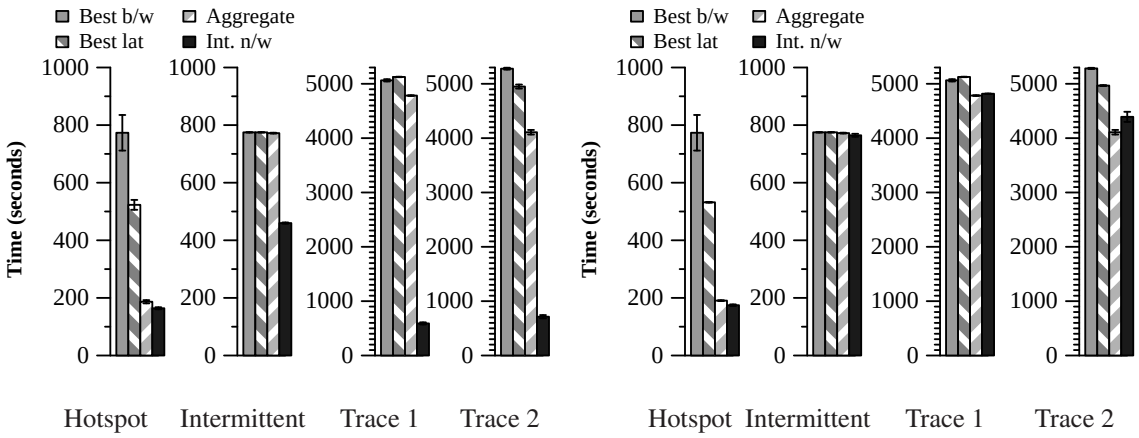
tions, we use the first portion of each trace for shorter experiments and loop the trace for longer ones.

We chose to use traces rather than measure application performance directly from the vehicle platform to provide repeatable conditions for different network management scenarios. Changing traffic conditions and external load on networks make it very difficult to achieve identical connectivity, even over multiple traversals of the same route. This variability would likely preclude meaningful comparisons across different trials.

2.5.1.4 Comparison strategies

For each application, we compare Intentional Networking with three strategies. The first two strategies use only a single network at a time but migrate connections to always use the best network according to a specific criteria. The first of these strategies always uses the network with the lowest round-trip time, while the second uses the network with the best bandwidth. We idealize a zero-cost migration by emulating a single virtual network connection that always has the bandwidth and latency of the best current network according to the selection criteria. For example, to create a virtual "best-latency" trace with a single network, we determine whether 3G or WiFi offered the lowest latency for the first second of the original trace, then use the recorded characteristics of that network for the first second of our new trace. We repeat the process for each second. Thus, these strategies show the maximum benefit that could be achieved by a migration strategy if an oracle chooses the best current connection and there is no migration cost.

We also compare Intentional Networking with an idealized version of an aggregation protocol, such as MultiNet or FatVAP, that multiplexes traffic over all available networks. We idealize aggregation by emulating a single virtual network connection that has bandwidth equal to the sum of the bandwidths of all networks and latency equal to the minimum of the latencies of all networks. This virtualized network is ideal in the sense that it offers better connectivity than any protocol that aggregates the individual networks could actually achieve. It therefore offers an upper bound on application-oblivious aggregation performance for each scenario.



(a) Interactive latency

(b) Background transfer time

Figure 2.1: BlueFS results

2.5.2 Results

2.5.2.1 BlueFS

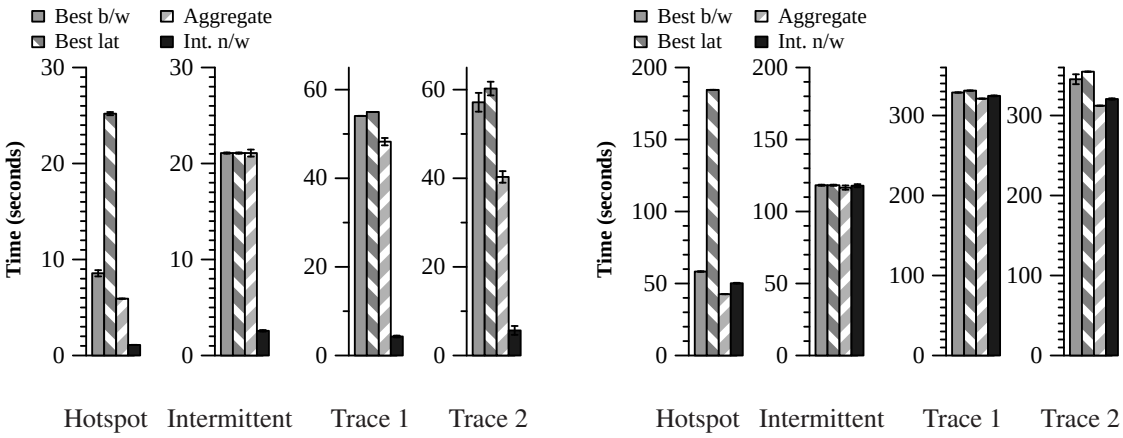
To evaluate BlueFS, we run a software development workload that rebuilds the lighttpd (version 1.4.25) Web server source tree. Such "Andrew-style" benchmarks have long been used to test file system performance [39]. Our particular benchmark deletes all object files from the build directory and then runs `configure` and `make` to build lighttpd. The benchmark begins with a cold client file cache, so all files are fetched from the server. We report the total time taken to execute the benchmark (i.e., the interactive performance), as well as the total time to finish propagating updates to the server in the background.

Figure 2.1a shows the interactive latency for BlueFS (the time to complete the software development benchmark). For the hotspot scenario, the best bandwidth strategy always uses the 3G network. The best latency strategy is an improvement because the workload is dominated by small fetches of 4 KB blocks. The idealized aggregation strategy works very well in this scenario because it is given maximum benefit from the diverse latency and bandwidth of each network. Yet, Intentional Networking still realizes a 14% speedup compared to aggregation by prioritizing foreground over background traffic. Intentional Networking improves interactive latency by 3x compared to the best latency strategy and by 4x compared to the best bandwidth strategy.

To verify that Intentional Networking does not unduly penalize background traffic, we also measured the total time to finish sending all background updates to the server, as shown in Figure 2.1b. Interestingly, Intentional Networking transfers all data 9% faster than the aggregation strategy in the hotspot scenario. At first, this seems anomalous because our idealized aggregation strategy should make maximum possible use of the networks. However, because the benchmark includes computation that depends on foreground transfers, compute episodes start earlier using Intentional Networking. This means that background data is generated sooner in the benchmark. Thus, Intentional Networking is able to use the uplink bandwidth earlier in periods where the aggregation strategy has no data to send. Where data dependencies exist, Intentional Networking can use the network more efficiently than even an idealized aggregation strategy that is unaware of application intent.

In the intermittent scenario, WiFi dominates 3G when it is available. Thus, the best bandwidth and best latency strategies both choose WiFi when available. The aggregation strategy derives a small additional benefit from also using 3G during these periods. Intentional Networking, however, reduces interactive latency by 40%. The benefit compared to aggregation is larger in this scenario because aggregation derives less benefit from its idealized use of two networks to offset Intentional Networking's benefit from understanding application intent. Intentional Networking's total transfer time for all data is 1% better than the other strategies.

The performance of Intentional Networking for latency-sensitive data is even better for the measured vehicular scenarios. Across the two traces, Intentional Networking improves interactive response time by 5-8x compared to aggregation, while increasing total background transfer time by only 1–7%. Compared to the best-bandwidth and best-latency strategies, Intentional Networking improves interactive latency by 7–8x and background transfer time by 5–17%. Despite the increased variability of network quality, Intentional Networking identifies and uses the best network for each type of traffic and thereby maximize benefit to the user.



(a) Average interactive delay

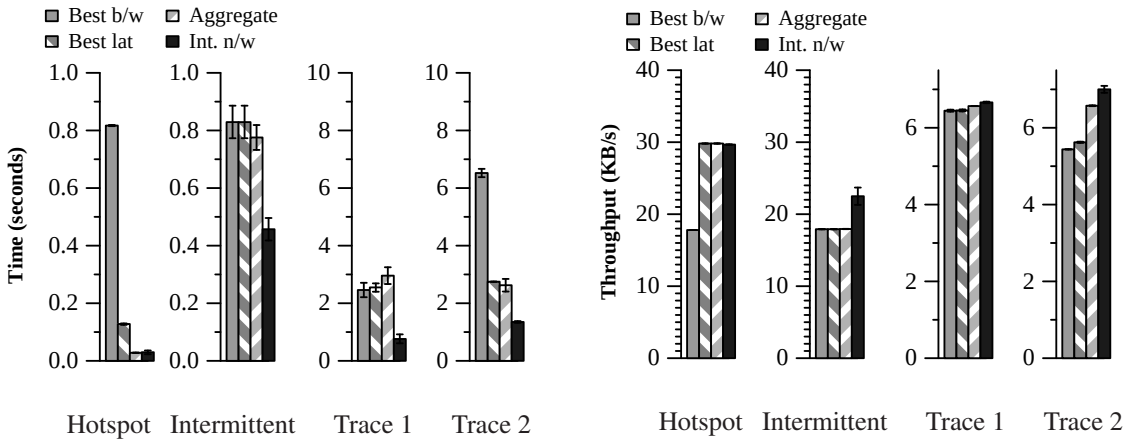
(b) Background transfer time

Figure 2.2: Thunderbird results

2.5.2.2 Thunderbird

In our Thunderbird benchmark, a user reads e-mail after a period of disconnection. The benchmark first fetches the e-mail headers of 100 messages, then downloads in the background the e-mail messages (with attachments), which range in size from 50 B to 256 KB. While the caching proxy is downloading these messages, the user selects 5 messages to read immediately based on the headers. We report the average interactive delay to fetch the on-demand e-mails, as well as the time to fetch all e-mails in the background.

Results for the Thunderbird e-mail benchmark are shown in Figures 2.2a and 2.2b. In contrast to the previous benchmark, the migration strategy that maximizes bandwidth is superior to the one that minimizes latency because transfer times are dominated by several large e-mails. Intentional Networking improves interactive latency compared to aggregation by 5x in the hotspot scenario and by 8x in the intermittent scenario. By reordering messages based on application semantics, Intentional Networking is able to deliver superior response time. Total background transfer time is 18% longer in the hotspot scenario, but 1% longer in the intermittent scenario. Results compared to the migration strategies are even better, with Intentional Networking fetching the on-demand e-mails 8–23x faster, while also improving total background transfers by up to 3x.



(a) Urgent update latency

(b) Background throughput

Figure 2.3: Vehicular Sensing results

For the two vehicular measurements, Intentional Networking improves interactive latency by 7–13x compared to the other strategies. The time to transfer all e-mails is within 1–3% of the idealized aggregation strategy and superior to both migration strategies.

2.5.2.3 Vehicular sensing

In our vehicular sensing benchmark, the vehicle uploads raw data to a server when network bandwidth is available, as described in Section 2.4.3. Our benchmark lasts for fifteen minutes. During that time, we have three episodes of urgent data transmissions. Since urgent messages are very often closely correlated in time, we send five messages in a period of seven seconds during each episode. We report the average response time for urgent events and the effective throughput of bulk sensor data, calculated over the entire 15-minute run time of the benchmark.

Figures 2.3a and 2.3b show results for the vehicular sensing application. In the hotspot scenario, the aggregate bandwidth is sufficient to prevent background data from interfering with urgent messages. Thus, both Intentional Networking and the aggregation strategy perform very well. The approximately 30 ms average latency for urgent updates is equivalent within experimental error for the two strategies. The aggregation strategy achieves the

maximum background data rate of 29 KB/s (a 4 KB summary and 25 KB of raw data per second), and Intentional Networking comes within 0.5% of this rate. Intentional Networking sends foreground data over 4x faster than the best-latency migration strategy.

In the intermittent scenario, Intentional Networking sends urgent events 41% faster than the aggregation strategy and also achieves 25% greater bulk data throughput. The throughput improvement comes from the use of thunks, which allow the Intentional Networking version to avoid polling and better schedule background transmissions.

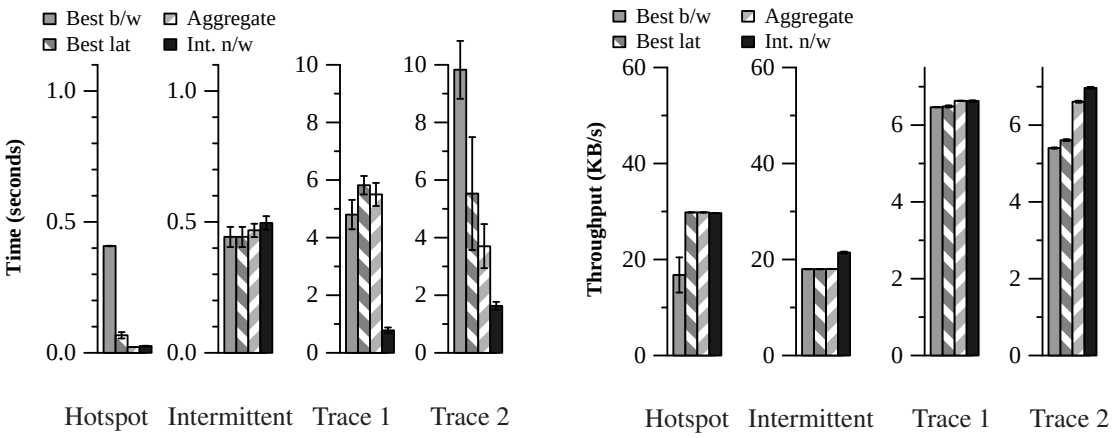
For the two vehicular traces, Intentional Networking improves urgent event response time by 2–5x compared to the other strategies. At the same time, Intentional Networking improves bulk sensor data throughput by 1–6% compared to the idealized aggregation strategy and by up to 29% compared to the idealized migration strategies.

2.5.2.4 Concurrent applications

Finally, we examined the effect of running multiple Intentional Networking applications concurrently by splitting the vehicular sensing application into two separate processes. The first process sends only the urgent messages; the second process sends only the raw sensor data. Figures 2.4a and 2.4b show results with two processes, including the two-process version of the application for each of the idealized strategies. The behavior of Intentional Networking with two processes is very similar to that with one process, showing that the cost of using shared memory to coordinate across multiple processes is not significant. The application-oblivious strategies see some benefits from multiple processes in the microbenchmark scenarios because the urgent updates and sensor data transmissions are now concurrent, yet Intentional Networking performance remains comparable to or better than the ideal strategies in all scenarios.

2.6 Summary

Mobile nodes face a changing array of diverse networking options, each of which may harbor different strengths and weaknesses. As a result, it is rarely the case that any one



(a) Urgent update latency

(b) Background throughput

Figure 2.4: Multi-app Vehicular Sensing results

networking option is the best choice for all traffic generated by all applications. By using the available options judiciously, an application may see significant improvements in service. Unfortunately, simply exposing the lower-level details of available networks, leaving everything to the application, is unlikely to gain much traction.

Intentional Networking addresses this impasse. It provides a simple, declarative interface for application to express the *intent* behind each network message. The system matches presented network traffic to the best available interface. If no available network is suitable, the traffic is deferred until such a network becomes available. Deferring some types of traffic but not others leads to reordering. Intentional Networking provides mechanisms to express mutual exclusion and ordering constraints over their traffic to match application constraints. Our results using vehicular wireless measurements show that these strategies improve interactive response time from 48% to 13x, while degrading throughput by no more than 7%.

CHAPTER 3

Spending budgeted resources to reduce user-visible delay

With Intentional Networking, we provided applications with a simple abstraction that allowed them to selectively introduce multi-networking while assuring correct operation, all while imposing only a small development burden. Applications can separate interactive traffic from background traffic, improving the user experience without degrading background throughput. Our initial work with Intentional Networking focused primarily on designing the right application interface and improving applications' interactive performance. Following this, we next began to think about other concerns that a multi-network abstraction would need to handle, energy consumption being at the forefront. We also continued exploring the common behaviors of real mobile applications and their implications for Intentional Networking.

Many popular mobile applications are largely user-centric, meaning that most of their network traffic occurs because the user made some request—opening a web page, for example. Since long delays on mobile networks can easily dominate the user experience, we soon realized that a common solution to this problem—*application-level prefetching*—is one of the most common forms of background traffic. By predicting future user requests and speculatively retrieving and caching the data, applications remove the network delay from the critical path of servicing the request.

In many ways, mobile computing is an ideal domain for prefetching. Applications that run on smart phones and tablets frequently fetch data from the cloud. Yet, mobile de-

vices must often rely on wireless networks, and such networks can exhibit low bandwidth, high latency, and intermittent connectivity. By prefetching and caching data, a mobile application can avoid on-demand use of poor and unreliable wireless networks, leading to a substantial improvement in interactive response time. Thus, the potential rewards of prefetching are high.

Unfortunately, the potential costs of mobile prefetching are also high. Since wireless network bandwidth is limited, care must be taken that prefetching does not interfere with interactive traffic and degrade application response time. Further, prefetching data from a remote server can consume battery energy and cellular data allotments, so profligate prefetching may result in a mobile device's battery expiring too soon or in overage charges on a cellular plan.

Unfortunately, balancing costs and benefits is complex. The prefetching system must consider at least three different concerns (performance, energy usage, and wireless data consumption), no two of which are measured by a common metric. It must understand how the costs of fetching data from a remote server vary as wireless network conditions change, and it must take into account network-specific oddities such as 3G tail time [83]. It must account for differences in hit rates caused by variation in behavior across users, applications, and even among different classes of data within the same application. Finally, it must ensure that prefetching does not substantially degrade foreground network activity.

Given this host of complexities and the limited resources of many mobile application developers, it is understandable that no current mobile application addresses all (or even most) of the above concerns when deciding when to prefetch data. Instead, mobile applications employ simple, yet often suboptimal, heuristics such as prefetch nothing, prefetch everything, or prefetch data subject to an arbitrary constraint (such as only prefetching data items with size less than 32 KB). Our evaluation shows that use of such heuristics represents a substantial missed opportunity.

Instead, we propose that the mobile computer system provide explicit prefetching support to all applications. Our prototype system, which we call IMP (Informed Mobile Prefetching), is structured as a library to which any mobile application may link. The

application interface is unobtrusive; applications must specify the items that could potentially benefit from prefetching, and they must inform IMP when those items are consumed or are no longer needed. The IMP library manages the complex mobile prefetching process, guiding applications to prefetch the right amount of data at the right time in a manner that improves performance while still meeting wireless data and battery lifetime goals. IMP both helps overzealous applications rein in prefetching to meet user budgets and allows tentative applications to make better use of available resources to improve the user's experience.

A key observation behind the design of IMP is that many of the prefetching complexities detailed above have been addressed individually, albeit not always within the domain of mobile computing. We view the major contributions of IMP as unifying these disparate ideas, applying them specifically to the domain of mobile prefetching, and providing prefetching services through a simple, easy-to-use interface that is well-suited for rapid mobile application development.

The design of IMP is inspired by Transparent Informed Prefetching (TIP), which employs a cost-benefit analysis to determine when to fetch data from an array of disks into a file cache [78]. Like TIP, IMP uses a shadowing strategy to predict hit rates for different prefetch depths. IMP uses a substantially different algorithm than TIP's to address unique features of mobile prefetching. IMP considers not just performance but also energy and wireless data usage. IMP also considers how aggregation of multiple prefetches can reduce energy usage. Finally, whereas cache space is the bottleneck resource for disk prefetching, IMP considers wireless network usage and its associated resource costs to be the bottleneck in mobile prefetching.

IMP leverages lower network layers to prioritize foreground network traffic over prefetching activity and to discover, characterize, and effectively match traffic to multiple available wireless networks. While IMP could be built on any layer that provides such services, it currently uses Intentional Networking (as described in Chapter 2) for these purposes.

Finally, IMP proposes a unified approach for dealing with *budgeted resources*. A budgeted resource has a fixed limit on the amount of the resource that can be consumed within

a specified time period. One example is cellular data allotments: many cellular plans allow usage of up to a specified amount of data for a fixed charge. Another example is battery energy: for predictable behavior, it is reasonable to expect the battery of a smart phone to last for an entire day so that it can be recharged at night. While the negative consequences of exceeding a budget have been widely observed, the negative consequences of under-utilizing a budget have received little attention. Yet, we observe that unused cellular data and leftover battery energy represent a substantial wasted opportunity when the mobile system has degraded user experience in order to conserve those resources. In response to this observation, IMP takes a "Price is Right" approach to budgeted resources—it tries to come as close as possible to the budget as possible without exceeding it. IMP continually measures how well it is doing in meeting its budgets and uses a control loop to adjust the cost of battery energy or cellular data usage relative to performance accordingly. This is a variation of Odyssey's goal-directed adaptation [33], which IMP modifies to manage multiple resources and applies to the domain of mobile prefetching.

We have implemented IMP as a Java library and modified two applications, the K9 e-mail client and the OpenIntents news reader, to use IMP. Our experimental results show that IMP is able to meet all specified energy and cellular data usage goals. Additionally, in most cases, whenever simple heuristic-based prefetching strategies also meet the goals, IMP outperforms those strategies on interactive fetch time, often by a factor of two or more.

3.1 Design considerations

In this section, we examine the tradeoffs involved in mobile prefetching, and discuss why the prefetching decision is more complex in the mobile domain than in many other areas.

3.1.1 Why is mobile prefetching so complex?

Most current prefetching systems have the sole goal of maximizing performance, subject to constraints such as cache space and network bandwidth. Prefetching is beneficial when a data item that was prefetched speculatively is later requested by a higher layer of the system. The latency of servicing the request is reduced by the time that would have been required to fetch the data. However, prefetching consumes resources. For instance, because prefetch requests are interwoven with demand fetches from the higher layer, a prefetch request might degrade the performance of those foreground requests. This interference mitigates the performance benefit of a successful prefetch. Further, when prefetched data is never consumed by the application, the interference with foreground activity degrades overall performance. Thus, prefetching systems employ heuristics to try to maximize performance by only prefetching data when the benefits of doing so exceed the costs.

Similar concerns arise in the domain of mobile prefetching when data is fetched from remote servers. The performance benefit of prefetching can be substantial, especially when wireless networks offer poor or intermittent connectivity. However, because wireless bandwidth is limited, the potential for prefetch requests to degrade other traffic is magnified.

In mobile computing, the prefetch decision must consider resources other than performance. Battery lifetime is often a critical concern. While the performance impact of prefetching can be mitigated by scheduling prefetches for intervals with little or no demand requests for data, the energy cost of prefetching data is difficult to mask. Yet, prefetching can still have substantial energy benefits. For instance, consider that the energy required to fetch data over a WiFi network is often less than that required to fetch data over a cellular network and that the energy required to fetch data over networks varies in proportion to the quality of the wireless connection. Thus, by prefetching data during periods of WiFi or good cellular connectivity, the mobile computer uses less energy than if it were to service demand fetches for the same data during periods of poorer connectivity. Further, prefetching allows the mobile computer to batch multiple requests. This saves energy by amortizing transition costs across multiple requests—for instance, the energy costs of 3G tail time can be ameliorated by sending multiple requests back-to-back.

Since it is increasingly common to cap cellular data consumption, prefetching systems must also consider cellular data usage when deciding whether to prefetch. Like energy usage, prefetching can impact cellular data usage either positively or negatively. Prefetching data over a cellular network will increase data usage if the prefetched item is never requested. On the other hand, the mobile computer can prefetch data over a WiFi network and thereby reduce data usage by avoiding a subsequent demand fetch over a cellular network. One could imagine setting a monthly budget for all prefetching and switching prefetching off if the budget is exceeded. However, it is unclear how to set such a budget accurately because non-prefetch data consumption can vary widely from month to month and from user to user.

In summary, mobile prefetching is complex because one must consider at least three metrics (performance, energy usage, and cellular data usage), whereas current prefetching algorithms often consider just performance. Further, it is difficult to compare the three metrics considered in mobile prefetching because no two are expressed in a single currency. For instance, if a potential prefetch would improve performance and save energy, but would also require additional cellular data transmission, should the mobile computer perform that prefetch? The next section describes our solution to this dilemma.

3.1.2 Balancing multiple concerns

Battery energy and cellular data allotments are examples of a resource class that we term *budgeted resources*. For such resources, there exists a fixed amount of the resource that must last for a period of time. For instance, a cellular data plan may provide 4 GB of data for a month. The consequence of exceeding the budget is severe (e.g., additional cost or degradation of quality of service). On the other hand, any resources not consumed during the budgeted period are wasted (unused data allotments do not roll over to the next month).

Battery energy is also best thought of as a budgeted resource. In this case, the budget is the amount of energy in the battery, and the budgeted period is the time until the battery is recharged. Current energy management strategies correctly worry about the negative

consequences of exceeding the budget (running out of battery energy). However, they do not focus nearly enough on the negative consequences of undershooting the budget. Since mobile computers increasingly degrade the user experience to preserve battery energy, any time the battery is recharged with a substantial amount of energy remaining represents a substantial wasted opportunity—the user experience was degraded for no purpose.

The inherent challenge is that the importance of a budgeted resource may change dramatically over time. Near the end of the month when there is substantial data left in a cellular plan, or near the end of the day, when the user has a mostly-full battery and plans to recharge soon, conserving a budgeted resource such as a data allotment or energy is relatively unimportant. Instead, performance should be maximized. On the other hand, when the battery is low or most of a data allotment is consumed, then the relative importance of the budgeted resource is very high.

Strategies that assign a fixed conversion rate for budgeted resources (e.g., saving a Joule of battery energy is worth a 10% degradation in application performance) are doomed to be incorrect as the relative importance of the budgeted resource changes. A fixed conversion rate will be too high when the budgeted resource is unimportant and too low when the resource is precious.

Instead, we argue that the management of the budgeted resource should be adaptive. IMP uses a control loop to adjust the conversion rate used to equate budgeted resources with performance. When the system is projected to exceed its budget based on measurements of current supply and demand, the conversion rate is increased to make the budgeted resource more precious. This causes future decisions (prefetch decisions in the case of IMP) to place more weight on conserving the budgeted resource. Thus, demand is reduced to match supply. On the other hand, when the system is projected to use significantly less than the budgeted amount, the conversion rate is decreased and the system becomes more aggressive about using the resource to improve performance (or reduce the use of other budgeted resources).

An additional benefit of the control approach is that the user experience becomes more predictable. For instance, users can come to expect that their phone battery will just last

all day in most circumstances, rather than having to constantly monitor the battery level and adjust their own behavior. Further, the budget itself is a simple knob through which the user can adjust system behavior. For instance, if one knows that one will recharge in an hour, one can just change the budget to reflect that decision rather than adjust numerous settings like screen brightness, processor speed, etc.

3.2 Design

We next discuss the design of Informed Mobile Prefetching. We first briefly describe how IMP uses Intentional Networking to simplify the use of multiple networks. We then describe how IMP decides whether to prefetch data in response to application hints.

3.2.1 Background: Intentional Networking

As described in Chapter 2, Intentional Networking is an abstraction for multiplexing traffic over multiple wireless networks. IMP uses Intentional Networking to simplify its use of multiple networks and to differentiate prefetch traffic from other application network activity. By designating prefetches as background requests, IMP causes Intentional Networking to prioritize other data over prefetches, ensuring that queuing delays due to excessive in-flight background data do not adversely affect foreground requests. IMP also gathers passive and active measurements of network quality from the Intentional Networking layer.

3.2.2 Prefetch decision algorithm

IMP decides when and how much data to prefetch using a cost/benefit analysis inspired by Transparent Informed Prefetching (TIP) [78]. The TIP algorithm regulates prefetching of file system data from an array of disk drives in a server environment. The core idea behind TIP is that the application should disclose hints that describe opportunities to prefetch data—these hints are predictions of future accesses. TIP dynamically decides when to

prefetch data corresponding to the disclosed hints based on a cost/benefit analysis that estimates how prefetching will impact application performance. One of the nice features of the TIP design is that applications do not have to know any details about the environment in which they are operating (e.g., resource constraints), nor do applications have to specifically estimate how effective prefetch hints will be at predicting future accesses. These details are managed by the prefetching system.

Since the use case that TIP targets (prefetching file system data from disk arrays) is considerably different than prefetching data to mobile computers, IMP retains the structure of the TIP cost/benefit model but changes most of the details. For instance, TIP considers cache buffers to be the bottleneck resource, and it attempts to optimize only performance. On the other hand, in mobile prefetching, the wireless network is the bottleneck resource and one must consider both battery energy and data usage in addition to performance.

We next describe how IMP estimates the potential cost and benefit of prefetching a data item for each of its three metrics: performance, energy use, and data consumption. IMP separately considers the impact of prefetching over each currently available network (e.g., it calculates the impact for cellular and WiFi when both are available).

3.2.2.1 Performance

The performance benefit of prefetching comes from decreasing the time to service a subsequent request for the prefetched data. The precise amount of benefit depends on the size of the data item and the network conditions that will exist at the time the data is requested. If the item is not prefetched, a demand fetch of the data will take time T_{fetch}, where

$$T_{fetch} = \frac{S}{BW_{future}} + L_{future}$$

as given by the size of the data item (S) and the future network bandwidth and latency at the time the demand fetch occurs (BW_{future} and L_{future}, respectively).

Although it may sometimes be feasible to predict future network conditions [68], for many applications it is quite difficult to predict when the user will request data. For this rea-

son, IMP uses the average network conditions observed in the past for the mobile computer as a reasonable approximation of future conditions.

For each type of network, IMP tracks the average availability, latency, and bandwidth of the network. To calculate averages, it obtains active network measurements and supplements those observations with passive measurements of network behavior that occur when data is prefetched or fetched on demand by applications. IMP uses network availability to merge the costs of performing the demand fetch over different network types. It assumes that if the lowest cost network is available, that network will be used. If not, the next lowest cost network will be used if available, etc. In practice, since IMP currently considers just cellular and WiFi transmission and since WiFi almost always dominates 3G when all costs (performance, energy, and data) are considered, this works out to:

$$T_{fetch-WiFi} \times Availability_{WiFi} + T_{fetch-cellular} \times (1 - Availability_{WiFi})$$

or the cost of fetching data over the two network types weighted by the observed average availability of WiFi.

IMP allows applications to specify the size of each data item when a prefetch is requested. For the applications we have modified to use IMP, it has been trivial for the application to retrieve the size of each data item from a remote server at the same instance that it learns of that item's existence. For example, when an email client requests a listing of all new messages, it can ask for the size of each email to be returned in the same request. If an application declines to supply the size of an item to be prefetched, IMP instead uses the average size of items it has fetched for that application in the past.

The calculation as described so far assumes that prefetch hints are always accurate; i.e., that the prefetched data will be requested at some time in the future. However, many prefetched items are never consumed by the application, so IMP must also consider the estimated *accuracy* of a prefetch hint in its calculations. This is calculated on a per-application basis as the number of prefetch hints for which the prefetched data was later consumed divided by the total number of prefetch hints issued. IMP increments the count of total hints when a new hint is issued, and it increments the count of consumed hints when an application first requests prefetched data. It uses shadow caching to determine the number

of hints for which it did not prefetch data but for which the application later requested the data. Specifically, IMP remembers all prefetch hints issued, regardless of whether or not it has yet decided to prefetch the hinted data. Applications inform IMP when a hinted request that has not yet been prefetched must be fetched on demand—this information is necessary since prefetching is no longer useful. IMP increments the count of consumed hints when this happens.

IMP allows applications to optionally differentiate among prefetch hints by associating each hint with a *class*. The choice of how to apply classes is left up to the application; in general, applications will benefit most if their prefetch classes separate prefetches by some application-specific or user-specific quality that makes some prefetches more likely to be requested than others. For instance, an email application might use one class for messages in a priority inbox, and another class for the remaining messages. A news reader application might associate a different class with each news feed. IMP maintains a separate estimate of accuracy for each class and uses the appropriate class estimator when new prefetch hints are issued. In this way, the abstraction we provide simplifies the creation of application-specific prefetch strategies; the application need only divide its hints into classes, and IMP takes care of discovering which classes have the most valuable hints.

In summary, IMP calculates the performance benefit of prefetching as the product of the average time to fetch the data on demand and the application-specific or class-specific accuracy estimate. The accuracy estimates allow IMP to help the application focus on prefetching the items that are most likely to be used. Note that since IMP relies on Intentional Networking to ensure that prefetch traffic does not interfere much with foreground application activity, it need not assign a performance cost to each prefetch.

3.2.2.2 Energy usage

IMP calculates the effect of prefetching on battery energy by comparing the energy that would be used to prefetch the data immediately ($E_{prefetch}$) with the expected energy cost of later fetching the item on demand (E_{fetch}). The previous section describes how IMP calculates T_{fetch}, the expected time to perform a future demand fetch of an item, as well as how it

calculates the expected size of the data item (if this is not specified by the application). IMP calculates $T_{prefetch}$, the time to perform a prefetch, the same way it calculates the time for a demand fetch, except that it uses the current estimates of bandwidth and latency for the network over which prefetching is being considered (recall that IMP considers prefetching over WiFi and cellular networks independently). IMP then uses models developed by PowerTutor [105] to calculate the energy impact of both an immediate prefetch and a possible future demand fetch. These power models are specific to the mobile device and cellular carrier. They are generated with a suite of automated tests that isolate and exercise different components of the device in turn. Once a model is derived, it can be re-used for many different purposes (as we are reusing models developed by others in our work).

For prefetching on WiFi, the power model provides a power coefficient $P_{WiFi-xmit}$ that encapsulates the average power usage of the WiFi interface when actively sending and receiving; thus, $E_{prefetch} = P_{WiFi-xmit} \times T_{prefetch}$. When prefetching on a 3G cellular network, IMP also includes the transition costs that may be incurred due to network activation. For instance, tail energy [83] has been shown to be a significant phenomenon in 3G cellular networks. 3G networks use inactivity timers to avoid the delays that can result from frequent channel release and reacquisition. This results in the network interface spending substantial time in a high-power state after the last transmission completes.

If a network transmission is predicted to cause a promotion to a high-power state, IMP includes the transition costs to and from the high-power state (which includes the tail energy cost for 3G networks) in the energy cost of the prefetch. However, if the radio is predicted to already be in a high-power state, only the additional time spent in the high-power state is included. Like PowerTutor, IMP monitors the number of bytes queued on the cellular interface to infer when power state transitions occur. When calculating the energy cost of an immediate prefetch, IMP queries the state transition model to learn whether or not the cellular radio is in a high power state. When calculating the energy cost of a future demand fetch, it uses the average distribution of power states observed for that interface to determine the likelihood of being in each power state. Thus, the energy cost of a 3G

prefetch is $(P_{3G-xmit} \times T_{prefetch}) + E_{tail}$, where

$$E_{tail} = \begin{cases} P_{tail} \times T_{tail} & \text{if transition occurs and tail begins} \\ P_{tail} \times T_{inactivity} & \text{if tail time is extended} \end{cases}$$

For simplicity, we here omit details such as the difference between DCH and FACH power states, though we include them in our calculations.

Finally, the net energy cost (or benefit) of a prefetch is $E_{prefetch} - (E_{fetch} \times Accuracy)$, since the energy cost of the future demand fetch is only paid if the user actually requests the item. Note that a prefetch can have an expected energy benefit if current network connectivity is strong (so less energy than average is required to fetch data) or a lower-power network is currently available, and the estimated accuracy of a prefetch is high. Batching (described in Section 3.2.2.5) can also lead to energy benefits for prefetching by amortizing transition costs across multiple requests.

3.2.2.3 Data consumption

IMP also considers the effect of prefetching on cellular data consumption. The estimated future data cost of a demand fetch is calculated as $D_{fetch} = S \times (1 - Availability_{WiFi})$. The benefit of prefetching over WiFi is D_{fetch}, since the cellular data cost on WiFi is zero. The cost of prefetching over a cellular network is simply S. As in the case of energy cost, when referring to the data cost of prefetching below, we denote it as $D_{prefetch} - Accuracy \times D_{fetch}$, where $D_{prefetch}$ is zero or S depending on which network is used.

3.2.2.4 Putting it all together

As explained in the last three sections, IMP separately calculates the cost or benefit of prefetching over each type of available network in terms of performance, energy, and data usage. Unfortunately, these three metrics are expressed in different units (seconds, Joules, and bytes), making it difficult to determine what to do when the metrics disagree about whether or not prefetching would be beneficial.

A common solution to this dilemma is to employ a conversion rate or utility function

to equate metrics expressed in different units (e.g., one could make 10 Joules of energy expenditure equal to 1 second of interactive delay). However, as argued in Section 3.1.2, the importance of each metric can change substantially over time, making static conversion rates arbitrary and dooming them to be incorrect at least some of the time.

Instead, IMP dynamically adjusts conversion rates using a feedback loop that takes as input how well the system is meeting its resource budgets. Specifically, IMP maintains two values, c_{energy} and c_{data}, that represent the current conversion rate between each respective budgeted resource and performance. For instance, if c_{energy} is 0.2, then 5 Joules is considered equal to 1 second of interactive delay.

IMP uses Odyssey's goal-directed adaptation to adjust these conversion rates. Since we did not modify this algorithm and it is described elsewhere in detail [31], we provide a simple overview of its design here. Note that Odyssey applied goal-directed adaptation only to energy; one of the contributions of this work is the observation that this strategy can also be used to regulate cellular data usage. Additionally, Odyssey used application-specific fidelity adjustments to regulate the amount of resource spending over its budget period, whereas we regulate spending by adjusting the amount of prefetching.

Once a second, IMP measures the remaining supply of a resource. Goal-directed adaptation uses an exponential smoothing algorithm to calculate the rate at which the resource is being consumed from recent observations of supply. It multiplies the estimated rate of consumption by the time remaining in the budgeted period to estimate the remaining demand for the resource.

IMP relies on having an estimate of the future time at which each budgeted resource will be replenished. Many cellular data plans have predictable periods coinciding with a monthly billing cycle, making it simple to estimate the budget period. Additionally, recent work [85] has identified personal patterns in different users' battery recharge behaviors, which suggests it may be possible to predict the budget period for the available battery energy. The accuracy of such predictions is important, however; a too-long or too-short prediction of the budget period will increase the likelihood that IMP underspends or overspends the budget, respectively.

Next, the goal-directed adaptation algorithm subtracts from the latest supply measurement 5% of the current remaining supply plus 1% of the original supply. This provides some insurance against future changes in the resource budgets, which can occur if the user downloads a new application that consumes a lot of data, or if they arrive home later than usual to recharge their phone. Finally, it computes a ratio that expresses how much resource consumption is currently over or under budget:

$$c_{adjustment} = \frac{estimated\_demand}{reduced\_supply}$$

IMP then updates the current conversion rate with this ratio: $c_{new} = c_{old} \times c_{adjustment}$. Thus, if energy consumption is over budget, a Joule becomes more precious compared to other resources. This feedback process leads IMP to bias further against prefetch decisions that cost energy, which will eventually bring the energy consumption within budget.

In summary, IMP calculates the net benefit of prefetching an item over a particular type of network as benefit minus cost, or:

$$T_{fetch} \times Accuracy - (c_{energy} \times (E_{prefetch} - Accuracy \times E_{fetch}) + c_{data} \times (D_{prefetch} - Accuracy \times D_{fetch}))$$

If this value is positive for a single network type, IMP prefetches the item over that network. If the value is positive for multiple network types, IMP tells the Intentional Networking layer that the data item may be prefetched over all such networks. Intentional Networking stripes the item over all such networks if the item is large enough that striping makes sense.

3.2.2.5 Batching

Because some costs (most notably 3G tail time) can be amortized across multiple prefetches, prefetching may be beneficial if several items are fetched in a batch, even though prefetching each item individually is estimated to have a negative impact. IMP checks for possible batching benefits by considering the impact of fetching one item, two items, etc. up to the number of currently outstanding application prefetch requests. If it predicts a positive impact for any batch size, it initiates prefetching.

3.2.2.6 Network disconnection

If a network becomes unavailable while a prefetch is being performed using that network, IMP recalculates the potential impact of prefetching over the remaining networks as above. If a potential benefit is found, it resumes the prefetch over the selected remaining networks. If not, the prefetch is delayed and resumed when an appropriate network is found. For simplicity, IMP currently relies on application-level support for resuming prefetch requests without retransmitting the data already received, though it could potentially provide this feature transparently with additional buffering.

3.2.2.7 Discussion

We have designed IMP to manage resources that have a fixed budget to be spent over a predictable time period. However, there are also cellular data plans where the subscriber pays for the amount of data actually consumed, rather than paying for a monthly budget. Though IMP does not handle such cases in its current form, we could imagine extending it to do so. Rather than targeting a single budget over a single time period, we could treat the pay-as-you-go data plan as a series of small budgets, in the amount of the data purchase increments. Once purchased, a unit of data can be treated in the same way as the budgets IMP currently considers, since there are no refunds for unused data and partially spent units are generally rounded up to the next full unit for billing. However, we would also need to consider the monetary cost of each data increment and also incorporate some notion of how much the user is willing to spend, which would require direct input from the user.

The concerns in this scenario differ from what we consider in IMP's current long-term budget approach, because of the added goal of saving the user money, the smaller units of data for which money is spent, and the fact that less money spent is always favorable. However, this scenario can still benefit from IMP's fixed-budget adaptation strategy for each allotted unit of data. In addition, the same strategy can be applied for each extra gigabyte that the user effectively purchases if they exceed their data cap on a pay-per-month plan.

Application → IMP		IMP → Application	
`prefetch(Fetcher)`	→ `Future`	`Fetcher.fetch(Labels)`	→ `Result`
`Future.get()`	→ `Result`		
`Future.cancel()`			
`Fetcher.setSize(int)`			
`Fetcher.setPrefetchClass(int)`			

This figure shows the IMP API used for hinting, executing, consuming, and canceling prefetches. Applications create a `Fetcher` that implements the `fetch` callback to execute a (pre)fetch. The `fetch` method takes as an argument the set of Intentional Networking *labels* to be used for the invoked fetch. The `prefetch` function takes a `Fetcher` and returns a `Future`, a handle that the application uses to retrieve the result of a fetch. The type of `Result` that a fetch returns is specified by the application. The application may optionally call `setSize` and `setPrefetchClass` to pass additional information to IMP.

Table 3.1: Informed Mobile Prefetching API

Caching systems typically have a limited amount of storage available for cached data, along with an eviction policy. We observe that battery energy and cellular data are usually far scarcer than storage on today's mobile devices, so we chose to focus on improving the use of those resources. We could also apply a TIP-like strategy to include the value of cache space in our cost-benefit analysis.

Finally, we note that some users have unlimited data plans, and some users are frequently able to charge their phones throughout the day so that they are rarely in danger of running out of battery. In these situations, IMP will prefetch more aggressively, since it is given larger resource budgets. In the case of unlimited data, the throttling that some carriers perform after the user has crossed a certain data limit can be viewed as a budget in itself, since the reduced bandwidth is undesirable.

3.3 Implementation

This section describes the IMP's Application Programming Interface and the applications we have modified to use IMP.

3.3.1 API

IMP is implemented as a Java library targeted at the Android platform. Its prefetching API is inspired by Java's ExecutorService and Future [36], which simplify programming tasks with arbitrary asynchronous computation. The interface, which is shown in Table 3.1, is designed so that IMP need not replicate the application-specific details of (pre)fetching data.

3.3.1.1 Prefetch hinting and execution

Applications use the `prefetch` method to provide a prefetch hint to IMP. As described in Section 3.2.2, IMP uses a cost/benefit analysis to decide when and if to act on the hint. The application supplies a `Fetcher` object that both identifies the data item to be fetched and implements the application-specific `fetch` method that retrieves it from the server. This method is used for both prefetches and on-demand fetches. In the former case, IMP initiates the prefetch by calling `fetch`. The application can optionally supply the size of the item to be fetched, as well as a *prefetch class*. Applications can use classes to indicate prefetch priority. IMP will prefetch items in classes with higher observed accuracy in preference to items in other classes. Within a class, IMP prefetches objects in the order that prefetch hints are supplied.

3.3.1.2 Prefetch consumption

Like the `submit` method in Java's ExecutorService, IMP's `prefetch` method returns a Future—this is a handle that allows the application to later retrieve the result of the fetch operation by calling its `get` method. If the prefetch has started but not yet completed, the `get` method blocks until the prefetch completes or until an optional timeout expires. If the prefetch has not yet started, calling `get` triggers an immediate demand fetch of the item (which is effected through the `fetch` callback).

3.3.1.3 Prefetch cancellation

If the application no longer needs a item for which it has issued a prefetch hint, it cancels the prefetch by either calling the `cancel` method on the Future object or by simply removing the last reference to the Future object, in which case the prefetch is implicitly canceled when the Future object is garbage-collected.

3.3.2 Applications

We have modified the K9 [46] open-source email client and the OpenIntents [75] news reader to use IMP.

3.3.2.1 K9 email

K9 [46] is is an open-source email application for Android, originally based on the stock Android email client. By default, K9 prefetches messages that are less than 32 KB (the threshold can be adjusted from the application). We added IMP support by using an IMAP proxy to intercept traffic between K9 and our IMAP server.

On application start, the proxy downloads the user's email headers (including the size of each message), decides which emails to prefetch, and issues prefetch hints for those messages (including their attachments), in order from newest to oldest.

3.3.2.2 News reader

We also added prefetching to OpenIntents, an open-source Android Atom/RSS feed reader. OpenIntents stores the content available in a feed but does not prefetch linked articles. Since the application uses HTTP for its network communication, we modified Apache HttpComponents [11], a Java HTTP library included in the Android SDK, to add support for article prefetching.

Frequently, Atom and RSS feeds only contain a summary and link to a given article, rather than the full contents of the article. Our modified news reader therefore grabs the

full-article links from each feed and decides which articles to prefetch. The application-specific fetcher specified via the `Fetcher` object issues an HTTP request for the full content of the article plus any embedded images, and it then stores the fetched data persistently for later retrieval.

Our news reader associates a prefetch class with each feed and uses the `setPrefetchClass` method to specify the class for each hint. To evaluate the impact of using classes, we also created a version of the news reader that does not specify prefetch classes.

3.3.3 Discussion

IMP needs some information from the application in order to make good prefetching decisions, but we must avoid making the developer's task onerous, or else applications will not use our system. Thus, we have only added features when our applications required them (*e.g.* prefetch classes). We could imagine applications that would desire additional features, such as the ability to retrieve a partial prefetch result instead of waiting for all data to arrive. Our API design would make such an addition straightforward.

In general, applications must be modified to work with IMP. We can support unmodified email clients via an email proxy, as features of the IMAP protocol help us discover which items the user might request and when items are requested on-demand or deleted. For other applications such as OpenIntents, it is difficult to gather this information without modifying the application.

3.4 Evaluation

Our evaluation compares IMP with the prefetch strategies most commonly employed by current mobile applications. We compare results across three metrics: application performance, energy usage, and data consumption. In addition, we examine how well IMP meets budgets for energy and data usage. Finally, we quantify the benefit of using prefetch classes.

3.4.1 Experimental Setup

3.4.1.1 Testbed

We run K9 and OpenIntents on the AT&T version of the Nexus One phone, running Android version 2.3.4. We modified Android system software to allow the simultaneous use of WiFi and cellular networks and added Intentional Networking support. To generate repeatable experiments, the phone connects to an isolated WiFi access point and a private Cisco MicroCell that is connected to AT&T's network. Since the MicroCell acts as a miniature cellular tower, our evaluation captures the effects of the cellular wireless medium on network bandwidth and latency. We emulate network conditions by passing all traffic through a computer that inserts delays with the netem [56] network emulator and throttles throughput using the Linux Advanced Routing and Traffic Control tools [55]. We run servers for each application on a Dell Optiplex GX270 desktop with a 2.8 GHz Pentium 4 processor and 1GB DRAM.

We measure energy usage by applying the Nexus One power model used in the implementation of PowerTutor [105]. We measure cellular data usage by reading the number of bytes sent and received through the Linux sysfs interface.

When we compare with strategies that do not use IMP, we allow those strategies to use Intentional Networking to isolate prefetching traffic from other foreground traffic and to take advantage of both cellular and WiFi networks. Thus, the results reported in this work show only the *additional* benefit that IMP provides on top of the benefit already provided by Intentional Networking. All reported results are the mean of five trials; graph error bars are 95% confidence intervals. Where there is a resource budget set for IMP, the budget is indicated by a horizontal line above the bar.

To quantify the benefit of different prefetching strategies, we report the average fetch time over all emails or news articles in a run. We report the average rather than the median because the median does not distinguish between two prefetch strategies that produce more than a 50% cache hit rate, and it is the long fetch delays that dominate the user's experience when not enough data is prefetched. Though the individual fetch times can vary

considerably due to several factors—data size, network bandwidth at the time of the fetch—lower average response time generally indicates more cache hits and thus more successful prefetching.

3.4.1.2 Trace-driven evaluation

To evaluate how our system performs in realistic mobile networking conditions, we use the vehicular network trace previously gathered in Ypsilanti, MI, as described in Section 2.5.1.3. Trace-driven evaluation provides experimental repeatability and allows for meaningful comparison between different prefetch strategies.

We gathered this trace by driving a vehicle equipped with WiFi and Sprint 3G network interfaces and continuously measuring the downlink and uplink bandwidth and latency via active probing to a server at the University of Michigan. The median 3G bandwidth is 368 Kb/s downlink and 40 Kb/s uplink, with maximum bandwidth of 1.2 Mb/s downlink and 74 Kb/s uplink. WiFi is available 27% of the time, with a median session length of 7 seconds; the longest WiFi session length is 131 seconds.

We collected a second trace by walking in downtown Ann Arbor and across the University of Michigan campus, carrying a Nexus One phone and measuring the available open and campus WiFi and the AT&T cellular network. The median 3G bandwidth is 695 Kb/s downlink and 216 Kb/s uplink, with maximum bandwidth of 1.3 Mb/s downlink and 358 Kb/s uplink. WiFi is available 18% of the time, with a median session length of 56 seconds; the longest WiFi session length is 99 seconds.

When running benchmarks, we replay the traces on the emulation computer, which throttles bandwidth and delays packets for each network according to the conditions observed. When no WiFi or cellular coverage is observed in a trace, the throttling computer causes the connection to drop—the Android OS typically discovers the network disconnection after several seconds. Since the collected traces are longer than our experiments, we use only the first portion of each trace.

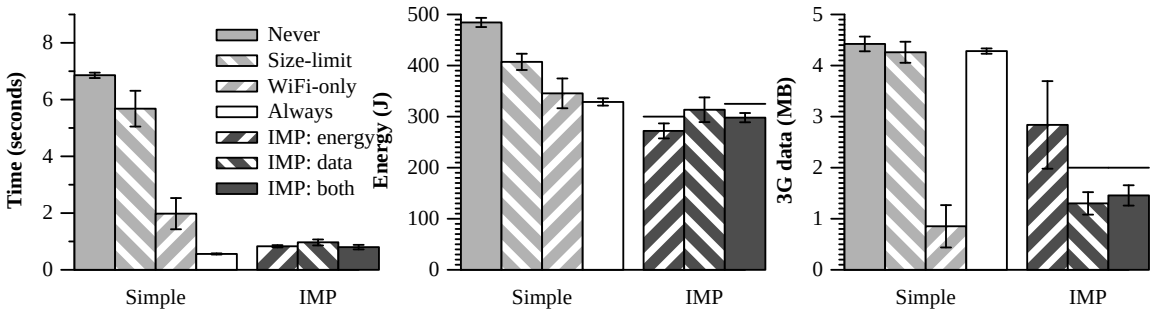
3.4.1.3 Comparison strategies

For each application, we compare IMP's performance to four prefetch strategies commonly used by mobile applications today. Unlike IMP, none of these strategies explicitly considers the cost and benefit of prefetching. The strategies are: never prefetch anything, prefetch items with size less than an application-specific threshold, prefetch over WiFi when it is available but never over the cellular network, and always prefetch everything.

3.4.1.4 IMP scenarios

IMP attempts to maximize performance subject to one or more resource constraints. For each application, we consider three scenarios: one where energy usage is constrained, one where cellular data usage is constrained, and one where both resources are constrained. Although we do not show the results, we did verify that when neither budgeted resource is constrained, IMP (correctly) emulates the always-prefetch strategy because that strategy maximizes performance.

To run a large number of experiments, we limit execution time to 20 minutes. The goals for energy and data usage are scaled proportionately. Note that even though longer experiments would likely show a wider variety of network conditions and usage behaviors, shorter experiments like these are much more challenging for a feedback strategy like the one employed by IMP because there is little time to react to under-usage or over-usage of a budgeted resource. For instance, if IMP is over-using the cellular data budget after half the budgeted time, it has 15 days to react during a month-long experiment, but only 10 minutes to bring the usage back under budget in these experiments. Similarly, IMP would have more time to react to unexpected changes in a user's mobility patterns over the course of a long experiment—for example, taking a different route to the office and encountering far fewer open WiFi APs. Thus, we expect that if IMP can meet budgets in these shorter experiments, it should do quite well in longer ones.



(a) Average fetch time (b) Energy usage (c) Data usage

In each plot, the left set of bars shows results for simple heuristic-based prefetching strategies; the right set of bars shows results for IMP when energy, cellular data, or both are constrained. Where there is a budget set for IMP, it appears as a solid line above the bar. Each bar is the mean of 5 trials. The error bars are 95% confidence intervals.

Figure 3.1: Email application, driving trace

3.4.2 Results

3.4.2.1 Email

We constructed an email workload from day-long traces of email activity within our department collected during prior projects. We use a trace of an email user fetching messages from a server [9] to derive the timing of email fetches and the interleaving user think time. The size of each email is randomly sampled from a distribution of email sizes in another trace [93].

At the start of the benchmark, the proxy fetches the list of 35 emails from the server along with their sizes. It issues prefetch hints for all emails. The reported accuracy of Gmail's Priority Inbox is 80% [6]. Since this seems a reasonable mechanism for deciding which email should be hinted, we assume that 28 of the emails are read during the experiment while the other 7 are deleted before they are read. This models a scenario in which a user flips through the inbox and decides, based on the subject line and message preview text, whether or not to read the email. Thus, the accuracy of prefetch hints is 80%. Of course, IMP is not told which hinted emails will be read and which will be discarded.

K9's default settings specify that email messages less than 32 KB will be fetched automatically upon inbox update. We use that value for the size-limit strategy.

Figure 3.1a shows the average email fetch time for each strategy. As expected, the always-prefetch strategy produces the lowest fetch time, since every fetch is a cache hit. However, as shown in Figures 3.1b and 3.1c, this strategy uses more energy and data than any IMP strategy.

Interestingly, the never-prefetch strategy uses more energy than any other strategy. This is due to a combination of the tail energy incurred with each demand fetch and poorer 3G bandwidth in later portions of the trace causing the 3G interface to remain in the high-power state longer.

When resources are scarce, IMP is able to meet resource budgets that are set lower than the amounts used by most simple strategies. In the constrained-energy scenario, we set the energy budget to use no more than 300 Joules over 20 minutes (this is indicated by the solid line over the `IMP: energy` bar in Figure 3.1b). Despite this goal being lower than the energy usage of any simple strategy, IMP comes in under budget in every trial. Further, IMP provides average fetch time within 300 ms of the (performance-optimal) always-prefetch strategy. Compared to the other strategies, IMP improves average fetch time by 2–8x and reduces energy usage by 21–43%. The fact that IMP produces a lower average fetch time than the common WiFi-only prefetching strategy lends credence to our assertion that prefetching on 3G is often a wise way to spend resources when the budget allows for it, since doing so provides a clear benefit to the user.

It is perhaps surprising that IMP uses less energy than the WiFi-only strategy in this scenario, especially since WiFi transmission is usually more energy-efficient than cellular transmission. Our analysis shows that because the WiFi-only prefetching strategy does not prefetch over the cellular network, it cannot prefetch some items before the user requests them. This leads to demand fetches later in the experiment that must be serviced over cellular because WiFi is unavailable at the time. Further, the cellular network quality at the time is poor, causing even more energy usage than normal. IMP avoids these demand fetches through more aggressive prefetching. Additionally, although IMP sends more data

over cellular than the WiFi-only strategy, it reduces energy usage due to 3G tail time by batching multiple requests together. In contrast, the demand fetches cannot be batched due to user think time.

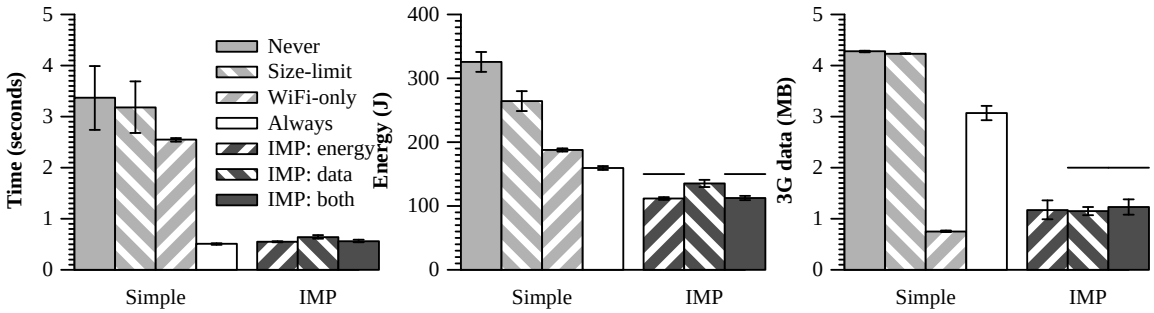
In the constrained-data scenario, we set the budget to use no more than 2 MB of data. Figure 3.1c shows the 3G data usage for each prefetching strategy. IMP meets this goal in all trials, whereas WiFi-only is the only simple strategy to use as little data. However, the WiFi-only strategy more than doubles average fetch time compared to IMP. Meanwhile, IMP is still able to provide average fetch time within 410 ms of that of the always-prefetch strategy and 2–7x less than that of the never-prefetch and size-limit strategies.

To constrain both resources, we set an energy budget of 325 Joules and a data budget of 2 MB. The results are similar to the single-budget scenarios. IMP provides average fetch time within 240 ms of the always-prefetch strategy and 2–8x lower than the other strategies, while reducing energy usage by 9–38% compared to the simple strategies. Compared to the always-prefetch, never-prefetch, and size-limit strategies, IMP reduces cellular data consumption by 3x.

IMP sometimes undershoots its budgets because it is hedging against future spikes in demand. In this experiment, however, undershoot occurs simply because IMP runs out of items to prefetch.

Figures 3.2a, 3.2b, and 3.2c show the results for the email application on the walking trace. In the walking scenario, the best strategy is to always prefetch, because the best WiFi is available at the beginning of the trace, and the little available WiFi that comes later is poor. In all scenarios, IMP emulates always-prefetch closely, achieving average fetch time within 40–150 ms of that of the always-prefetch strategy and 4–6x lower than the other strategies.

For the energy-constrained experiments using the walking trace, we set an energy budget of 150 Joules. IMP meets the goal in all trials, reducing energy usage by 30–65% compared to the simple strategies. For the data-constrained experiments, we set a 3G data budget of 2 MB. IMP meets this goal as well in all trials, reducing 3G data usage by 2–4x compared to the never-prefetch, size-limit, and always-prefetch strategies. Only the WiFi-



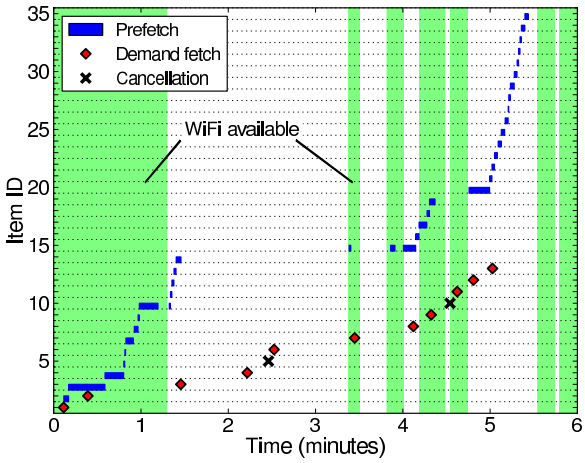
(a) Average fetch time (b) Energy usage (c) Data usage

In each plot, the left set of bars shows results for simple heuristic-based prefetching strategies; the right set of bars shows results for IMP when energy, cellular data, or both are constrained. Where there is a budget set for IMP, it appears as a solid line above the bar. Each bar is the mean of 5 trials. The error bars are 95% confidence intervals.

Figure 3.2: Email application, walking trace

only strategy meets the data goal, but it increases average fetch time by over 4x compared to IMP.

To provide some insight into the decisions that IMP makes over the course of an experiment, we show in Figure 3.3 the activity of the email application in one run of the driving trace, with IMP running with the energy budget only. IMP starts prefetching on WiFi, until the WiFi fades shortly after the 1-minute mark. At this point, the 3G interface is already active due to an Intentional Networking control message, so IMP starts prefetching on 3G. It completes a few prefetches before deciding that the next prefetch is too large to send on 3G, so it subsides. A few minutes later, it begins encountering some spotty WiFi, which it uses to attempt prefetching the next message. Soon after it starts making forward progress on WiFi, the WiFi network fails again. IMP then decides that it has enough budget to send a large batch of prefetches over 3G (taking the tail time savings into account), so it finishes prefetching the rest of the messages.



Email prefetch and demand fetch activity for one run of IMP, energy-constrained, on the driving trace. The horizontal bars show issued prefetches. The diamonds show demand fetches. The shaded regions show periods when WiFi is available. Each canceled prefetch hint is marked with an X. Since prefetching finishes early in the experiment, only the first six minutes are shown.

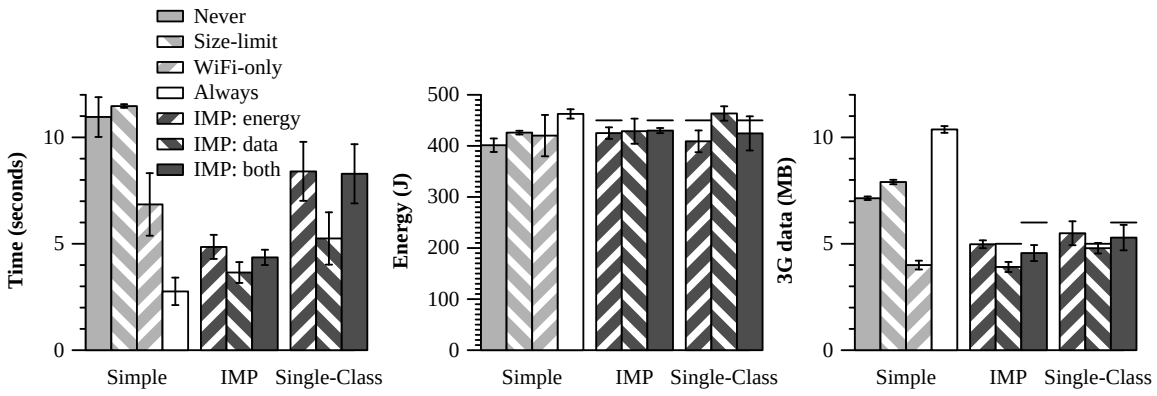
Figure 3.3: Email + IMP prefetch activity

3.4.2.2 News reader

We constructed a news reader workload based on one of the authors' Google Reader subscriptions. Google Reader provides statistics for a user's feeds for the past 30 days, including information such as the number of articles published and the number of articles read, along with the calculated read percentage. We choose five feeds with articles widely varying in size (primarily due to attached images). We used the relative feed volume over the past 30 days to choose a subset of each feed's articles for the benchmark, starting with the most recent articles. There are 25 articles across the five feeds.

Our simulated user selects a feed and begins iterating through the articles in that feed, opening and reading some of them and marking others as read without actually opening them. The fraction of articles that the user actually reads is determined by the read rate for that particular feed. For the chosen feeds, the read rate ranges from 25% to 100%, and the total read rate across all feeds is 64%.

Lacking a trace for newsreader user behavior, we chose to model a user's think time between article fetches as a random variable. Recent work in modeling web user dwell



(a) Average fetch time (b) Energy usage (c) Data usage

In each plot, the left set of bars shows results for simple heuristic-based prefetching strategies. The middle set of bars shows results for IMP when energy, cellular data, or both are constrained. The right set of bars shows results for IMP with the same constraints when IMP does not separate prefetch hints by class. Where there is a data budget set for IMP, it appears as a solid line above the bar. Each bar is the mean of 5 trials. The error bars are 95% confidence intervals.

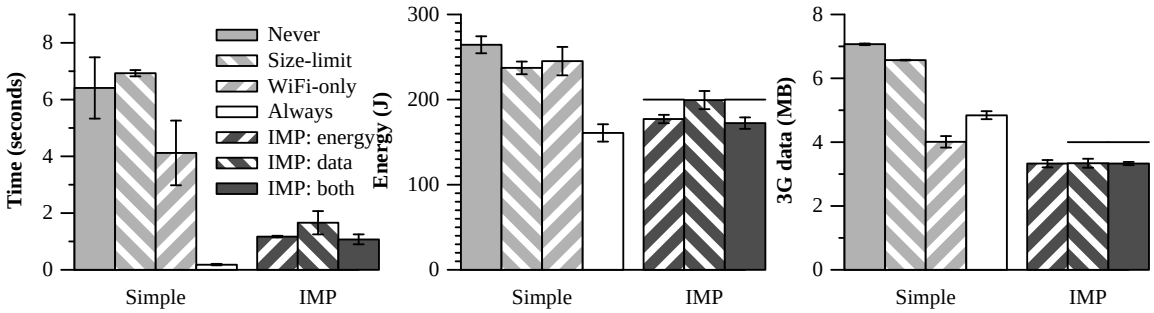
Figure 3.4: News reader application, driving trace

times observes that the majority of page visits last one minute or less [57]. Thus, after the user reads an article, the benchmark pauses for a random think time uniformly distributed between 30 and 60 seconds, then moves on to the next article in the feed, continuing until the feed is exhausted. The user does not pause after choosing to skip an article. As with the email application, the benchmark lasts 20 minutes.

Since the news reader does not prefetch article contents by default, we set the value used in the threshold strategy to 128 KB, which is just above the median article size.

Our news reader associates a different class with each feed. The results of this evaluation are shown by the middle set of bars in Figures 3.4a, 3.4b, and 3.4c. In the next section, we evaluate the benefit of prefetch classes by using only a single class. For ease of comparison, these results are shown by the rightmost set of bars in each figure.

As with the email benchmark, the always-prefetch strategy produces the best average fetch time, though the greater inaccuracy of the prefetch hints in this experiment reduces the benefit of prefetching compared to the email experiment.



(a) Average fetch time (b) Energy usage (c) Data usage

In each plot, the left set of bars shows results for simple heuristic-based prefetching strategies. The right set of bars shows results for IMP when energy, cellular data, or both are constrained. Where there is a budget set for IMP, it appears as a solid line above the bar. Each bar is the mean of 5 trials. The error bars are 95% confidence intervals.

Figure 3.5: News reader application, walking trace

In contrast to the email benchmark, aggressive prefetching has an adverse affect on energy usage. This is primarily due to the lower prefetch hint accuracy and the large size of some articles. Nevertheless, when given an energy budget of 450 Joules, IMP performs enough prefetching to reduce average fetch time by 29–58% compared to the never-prefetch, size-limit, and WiFi-only strategies, while also meeting the energy goal. As we discuss in the next section, IMP benefits from separating prefetch hints into classes and first prefetching articles most likely to be read. While the always-prefetch strategy has better performance than IMP, it does not meet the energy goal.

When we constrain data usage to 5 MB, IMP has more freedom in scheduling prefetches than in the energy-constrained scenario and reduces average fetch time by 47–68% compared to the never-prefetch, WiFi-only, and size-limit strategies. IMP meets the cellular data goal in all runs, reducing 3G data usage by 45–62%.

When constraining both resources, we kept the 450 Joule energy budget and set a 6 MB data budget. IMP's behavior is similar to its behavior in the energy-constrained scenario: it meets each goal in all runs and reduces average fetch time by 36–62% compared to the never-prefetch, size-limit, and WiFi-only strategies.

IMP uses less than the given cellular data budget, for the same reasons as above: IMP finishes prefetching all the hinted articles, after which no more articles are sent on the cellular network.

Figures 3.5a, 3.5b, and 3.5c show the results for the newsreader application on the walking trace. As in the case of the email application, the best strategy for this trace is to always prefetch. In this application, IMP comes closer to the average fetch time of always-prefetch than any of the other simple strategies, achieving average fetch time 2–6x lower than those strategies.

For the energy-constrained experiments using the walking trace, we set an energy budget of 200 Joules. IMP meets the goal in all trials, reducing energy usage by 25–35% compared to the never-prefetch, size-limit, and WiFi-only strategies.

IMP uses 7–10% more energy than the always-prefetch strategy on average. The reason for this is that IMP decides to pause prefetching when it encounters a particularly large prefetch early in the benchmark. At this time in the experiment, IMP has sent a few prefetches on 3G to amortize the tail time, but it decides the potential savings of the tail time is not worth the energy that will be spent to fetch the large article, especially since the article is less likely to be read than the articles that IMP has already prefetched. This results in two demand fetches on 3G later in the benchmark, since IMP does not resume prefetching until it has ceased spending energy for a while to catch up to its target rate. Thus, IMP incurs additional tail time compared to the always-prefetch strategy.

For the data-constrained experiments, we set a 3G data budget of 4 MB. IMP meets this goal as well in all trials, reducing 3G data usage by 17–53% compared to the simple strategies. Interestingly, IMP reduces 3G data usage even compared to the WiFi-only strategy. The reason for this is that IMP uses prefetch classes, whereas WiFi-only does not, which causes IMP to prioritize articles most likely to be read. Since WiFi is limited in this trace, the WiFi-only strategy fails to prefetch some particularly large articles that later must be demand-fetched over 3G.

3.4.2.3 Benefit of prefetch classes

To evaluate the benefit of prefetch classes, we executed the same benchmark, using the driving trace, without having the news reader use classes to differentiate prefetch hints of articles in different feeds.

The benefit of prefetch classes is greatest in the scenarios in which energy is constrained; in these two scenarios, the multi-class news reader reduces average fetch time by 42–47% compared to the single-class news reader scenarios. Though the energy constraint causes both versions of the news reader to prefetch cautiously, the multi-class news reader benefits greatly by first prefetching the articles with the highest likelihood of being read.

When the application uses prefetch classes, IMP meets all goals in every trial. However, without prefetch classes, IMP meets only 87% of the resource goals, failing to meet the data goal in 1 of 5 trials of the data-constrained scenario and the energy goal in 1 of 5 trials of the dual-constraint scenario. In the two trials in which IMP missed the goal, it overshot by less than 5.3% due to a late demand fetch in the benchmark. While no adaptive system will ever be perfect, we expect that a longer experiment with larger goals (typical of mobile phone usage today) would give IMP substantially more room for error and probably eliminate these overshoots.

Nevertheless, prefetch classes provide an added margin of safety by allowing IMP to make more informed decisions. IMP is able to target its prefetch effort to articles that are more likely to be read. Further, IMP's per-item predictions are more accurate since they are based on per-feed statistics rather than overall hit rates.

3.4.2.4 Discussion

From the above results, we conclude that IMP does an excellent job of meeting budgets for constrained resources. IMP meets all budgets in every trial for both the email and news reader applications (as long as the latter uses prefetch classes to differentiate feeds).

In the newsreader experiment on the walking trace, when energy is the only constraint,

the always-prefetch strategy is best for both fetch time and energy usage, as discussed above. However, in all other cases, when one or more budgetary constraints are specified, IMP outperforms all heuristic-based prefetching strategies we examined that also meet the budgetary constraints. Often, the performance improvement is a factor of two or more.

The benefit of prefetch classes is apparent not only in improving the percentage of budgets met for the news reader from 87% to 100%, but also in substantially improved response time. There is clearly a tradeoff between complexity and accuracy in the application interface. IMP can make better decisions if the application provides more information, but each additional specification places more burden on the application developer. In the case of prefetch classes, our results indicate that the small effort of differentiating hints by class (in the case of the news reader) can substantially improve results. This indicates that other, more contextual methods for differentiating prefetch accuracy could be beneficial.

3.5 Summary

Prefetching is an invaluable tool for improving the performance of mobile applications. However, prefetching can be costly in terms of the energy and cellular data consumed, and these costs are exacerbated by the possibility of prefetching data that the user never requests. As a result, many mobile applications employ simple but suboptimal heuristics as a means of grasping some benefit while avoiding undue complexity.

We propose that mobile computing systems provide explicit prefetching support to applications. IMP shifts the burden of complex prefetching decisions from the application to the system. It assumes responsibility for estimating the cost and benefit of prefetching, tracking the importance of energy and cellular data over time, and scheduling prefetches to improve interactive performance while meeting specified resource usage goals.

Our evaluation shows that IMP is able to meet specified energy and cellular data usage goals. Additionally, we show that in most cases, whenever simple heuristic-based prefetching strategies also meet the goals, IMP outperforms those strategies on interactive fetch time, often by a factor of two or more.

CHAPTER 4

Incorporating predictor uncertainty in decision-making

One major takeaway from our work with Informed Mobile Prefetching is the notion that conserving energy or cellular data is not always the most appropriate goal. Often, these resources are in surplus, sufficiently so that it becomes advantageous to spend them in pursuit of reducing user-visible delays. With IMP, we sought a principled approach to decide when resources are sufficiently in surplus and spend them accordingly, using prefetching as the mechanism for improving performance; we now turn to another application of this principle.

Current mobile applications greatly overestimate their ability to predict the future. As circumstances change, mobile systems and applications adapt their behavior to take best advantage of their environments. These adaptive decisions are made based on *predictions* of the future—network performance and availability, expected computational loads, the presence and capabilities of support services in the infrastructure, etc.

Unfortunately, these predictions are rarely certain, yet they typically are used as if they were perfectly precise. Overconfidence in prediction leads to incorrect adaptations and lost opportunities, with consequences visible to the end user in terms of performance, power, and network costs. The problem is that mobile applications typically modularize their decision processes. First, an application calculating estimated values for such quantities as bandwidth, and application compute needs. Even though the calculation of an estimated value takes into account underlying distributions and measurement uncertainty, the act of

collapsing the information into a single scalar value means that the inherent notion of error in the underlying prediction is no longer captured. Second, applications use the estimated values to choose the single option that maximizes the difference between estimated benefit and estimated cost. As a result, even applications that select the optimal strategy based on the predicted values are bound to make wrong decisions at least some of the time—when reality does not match the predictions.

We argue that the system and applications should *explicitly* consider the uncertainty of their predictions when making decisions based on them, and they should use new information to re-evaluate those decisions when necessary. In particular, considering prediction uncertainty throughout the decision process lets applications properly consider the benefit of *redundant strategies*. For example, a network selection application may decide to send the same data redundantly on multiple networks when it is quite unsure which network will provide the best response time. By using the first to complete and terminating the second, the application gets the best performance possible by masking the effects of the predictor uncertainty.

However, the advantages gained by employing redundancy come at a price. Sending data on two networks rather than one spends additional energy and potentially also spends cellular data, which for many users is limited to a fixed budget per month. From the perspective that resource conservation is of utmost importance, redundancy may appear to be obvious folly, since it always spends more resources than choosing the correct strategy from the start. In the face of uncertainty, however, the correct strategy cannot always be known. Faced with uncertainty, redundancy is a powerful mechanism that can spend resources to purchase an improvement in performance and a reduction in variability. In this way, redundancy can be seen as an insurance policy against inaccurate predictions.

Of course, the use of redundancy as a hedge against uncertainty is not a new idea. It is used in several other domains, including cloud servers [27] and route selection in wireless networks [7]. But such techniques have not been applied throughout the systems and application stack in a principled way, and we argue that mobile applications are missing substantial opportunities by not doing so.

To explore these ideas, we have implemented a framework that applications can use to capture the uncertainty in their predictions and incorporate that uncertainty in its decision-making. Our framework implements three different methods for making decisions while taking predictor uncertainty into account, each with different tradeoffs. Each explicitly weighs the importance of performance and resource conservation, employing redundancy only when spending resources to purchase better performance is likely to be cost-effective.

We have modified two applications to use our framework: network selection and speech recognition. Our experimental results show that there is significant benefit to be gained from redundancy when resources are sufficient to justify the tradeoff. Compared to the both the simple strategies that these applications typically employ, as well as adaptive strategies that consider performance and cost without considering predictor error, our framework improves application performance by up to a factor of two.

Thus, the major contributions are:

- Demonstrating that propagating uncertainty throughout the decision process improves mobile applications.
- Demonstrating that redundant strategies are better than non-redundant ones in many realistic scenarios.
- Providing a library-based abstraction that greatly simplifies the use of predictor error and the consideration of redundant strategies.
- Measuring the advantages and disadvantages of three methods for quantifying predictor uncertainty.

4.1 Example

We illustrate the ideas in this chapter with a (very simple) motivating example. Consider two servers that can execute an offloaded computation, with the following known distributions on response time: server A takes 10 seconds half the time and 20 seconds otherwise, and server B takes either 12 or 22 seconds with 50-50 probability. A system that only considers non-redundant solutions would calculate the expected response time

for each server (15 and 17 seconds, respectively) and execute the computation on server A. But, since the response times are independent, a system that considers redundant strategies would calculate the expected fastest response time from either server over the joint distributions as 13 seconds (the fastest response with equal probability will be 10, 10, 12, or 20 seconds).

Thus, if response time is the only consideration, redundant execution offers an expected benefit of 2 seconds over the best non-redundant solution. However, redundancy uses additional resources. A principled approach would balance the expected 2 second benefit against the 2x server resource usage and choose redundancy only if the value of improved latency exceeds the added resource cost.

In addition to improving expected response time, redundancy can help mask outlier behavior. Consider a distribution in which each server takes 10 seconds 99.9% of the time and 100 seconds 0.1% of the time. Redundant execution reduces the chances of the user experiencing the outlier behavior from 0.1% to 0.0001%. Cloud systems currently apply this well-known principle to reduce tail latency [27].

Finally, it is important to reevaluate decisions based on new information. While it is often unreasonable to modify applications to provide explicit notifications about progress, the absence of a response can often provide valuable insight.

For instance, in the previous example, consider a system that made the decision to execute on one server (because the resource cost of redundant execution is high and outlier behavior is unlikely). Assume that the computation has run on that server for 11 seconds without a response. Now, the conditional probability distribution reveals that the expected completion time is an additional 89 seconds (the outlier behavior is essentially certain in this simplified example). A new evaluation at this point reveals that starting a computation on a second server is extremely likely to substantially reduce response time. Thus, the negative information embodied by a lack of response changes the decision about whether or not to employ redundancy. While timeouts or other failover methods can provide an ad-hoc solution, consideration of the conditional distributions provides a more general and precise method of handling unexpected delay.

We advocate for the *principled* use of redundancy in mobile applications. Too often, mobile systems eschew redundant strategies altogether by collapsing predicted distributions into expected values (as in the first example), or they do not consider negative information (as in the final example). They therefore miss many opportunities to improve the user experience.

4.2 Uncertainty and redundancy

Uncertainty and redundant strategies go hand-in-hand. The following conditions make redundancy attractive:

- There must be multiple strategies available to accomplish some task. The strategies must not interfere with each other, or such interference must be minimal.
- Future conditions on which strategy selection depends (e.g., resource supply and demand) must be uncertain.
- The benefits of better performance must outweigh the costs of greater resource expenditure.

These conditions are present in the cloud computing domains; for instance, Google systems mask variable component response time with hedged requests that initiate a redundant request to a second server if the first server does not respond quickly [27]. Redundancy is also used extensively in wireless networks to compensate for unreliable links between nodes [7] where transmissions can be scheduled to minimize interference.

These conditions are also present in mobile application design. Specifically, non-interfering strategies exist in network selection (data may be sent over cellular and WiFi networks simultaneously) and in cyber foraging (computation may be concurrently executed on both a mobile computer and a remote server). Compared to controlled data center environments, mobile environments are even more variable and future conditions are even more difficult to predict. While resource usage such as energy consumption is a very

important concern in mobile computing, performance is also a paramount concern (otherwise, users would not buy the latest and greatest smartphone with the fastest and most power-hungry processor).

The paradox is that, in hindsight, redundancy is always wrong! Sending on an extra network always spends more energy than if the application had just chosen the best network to begin with. With perfect predictors, therefore, redundant strategies are never attractive. However, when predictions may be wrong, redundancy offers a performance benefit by giving applications an opportunity to hedge their bets. As the expected error increases, the performance benefit of redundancy improves.

4.3 Design

We define a *predictor* to be a software component that estimates the future value of some quantity. For example, commonly-used predictors in mobile computing estimate resource supply, such as network bandwidth and latency; failure probability, such as the chance of moving out of range of a hotspot; and application or user behavior, such as the length of a speech utterance.

Applications use the output of predictors to select from among different *strategies*, which we define to be different means of accomplishing the same task. Strategies may be non-redundant, such as sending data over a cellular or WiFi network, or redundant, such as sending the same data over both types of network.

Adding redundancy through the simultaneous execution of multiple strategies tends to improve performance since the application can use the result of the fastest strategy to complete. However, executing multiple strategies increases resource utilization since each new strategy consumes network bandwidth, battery energy, etc. A redundant strategy should therefore be used when its expected benefits in terms of improved performance exceed its expected costs in terms of additional resource usage.

Meatballs currently considers response time, energy consumption, and cellular data usage in deciding which strategy to employ. In order to meaningfully compare quantities

expressed in different units, it assumes that the current relative importance of the metrics is specified through importance factors given by the user. For example, an importance factor of 0.0001 expresses the following tradeoff: "I am willing to spend 10 J or 10 KB to avoid 1 second of waiting." In this work, we focus on the effects of these factors rather than how to set them; such factors could be set by users through system settings, or by the system using goal-directed adaptation as in Odyssey [72] and IMP (Chapter 3 of this dissertation).

The *weighted cost* of a strategy is calculated by multiplying the raw values for time, energy, and data usage with their respective importance factors and summing the weighted values together. The best strategy is the one that has the lowest weighted cost.

Note that when uncertainty is low and measurements are probably accurate, a non-redundant strategy should be preferred. This is because a decision that prefers one non-redundant strategy over the others is likely to be correct. On the other hand, when prediction uncertainty is high, the value of redundancy is greater because the likelihood of choosing the wrong strategy increases. Hedging bets by choosing multiple strategies has a greater likelihood of significantly improving application performance.

There are many possible methods for incorporating uncertainty in decision-making, each of which has strengths and weaknesses. Since we were not sure which method would work best in mobile computing, we implemented three such methods in Meatballs. The first two, which are described in Sections 4.3.1 and 4.3.2, directly measure prediction error by comparing the measured value of a quantity with the previously-predicted value. The sequence of error measurements collected over time is used to predict the uncertainty of future measurements. In the third method, described in Section 4.3.3, Meatballs measures decision error by comparing the strategy selected by the application with the strategy it should have taken.

So far, we have described only how Meatballs makes an initial decision to employ a redundant or non-redundant strategy. Such decisions should be reevaluated when new information arrives in the form of updated predictions or even in the form of negative information such as the lack of response from a remote server. Section 4.3.4 describes how Meatballs performs reevaluation.

4.3.1 Brute force

Our first method quantifies predictor error by measuring it over time and maintaining an empirical distribution. When a new measurement is made, Meatballs calculates the relative error of a prediction as the measured value divided by the previously predicted value. A relative error greater than one indicates that the last prediction underestimated the true value, whereas an error sample less than one indicates that it overestimated the true value. The set of relative errors observed over time forms the empirical distribution of predictor error.

In order to incorporate the error distributions into a redundancy decision, Meatballs calculates the expected completion time and resource costs for each non-redundant strategy as a weighted sum over the joint error distributions of the predictors. For each redundant strategy, the completion time at each point in the joint distribution is the minimum of the completion time of any component strategy and the resource costs are the sum of the resource costs of all component strategies. These values are likewise calculated as weighted sum over the joint error distributions. Meatballs combines the individual values to calculate a weighted cost for each strategy, as described in the previous section.

A number of possible methods exist for storing distributions; e.g., storing all samples forever, binning samples in histograms, random sampling, etc. We note that predictor error is constantly changing; e.g., a quantity such as network bandwidth that was unpredictable seconds ago may have since settled. Thus, Meatballs ages samples using an exponential weighted moving average so that the most recent error samples are given greater weight. Aging also limits the number of samples stored; older samples are removed after they drop beyond a threshold where their impact on the calculation is negligible.

The expected advantage of the brute force method is simplicity and high accuracy; the expected disadvantage is computational cost. For instance, with a four-year-old Nexus One phone, decisions for the applications in Section 4.4.3 can take hundreds of milliseconds. Meatballs mitigates this cost by moving brute force calculations off the critical path. For instance, a network selection application can make an initial non-redundant decision about which network to use, then asynchronously run the slower redundancy decision to decide

whether to use an additional network. Additionally, the predictor error distributions change slowly, so error calculations are cached and re-used for several decisions.

4.3.2 Probabilistic error bounds

Our second method measures and ages samples of predictor error in the same way as the brute force method, but it uses probabilistic bounds rather than calculations over an empirical distribution to make decisions. Compared to the brute force method, use of probabilistic error bounds is less accurate but has substantially lower computational costs.

Previous work in network queuing uses Chebyshev bounds to give the worst-case error in packet delay prediction [38]. We initially tried this approach but found the resulting bounds too loose to be useful. Instead, Meatballs uses a Student's-t distribution over the observed relative error samples to construct a statistical *prediction interval* ($\alpha = 0.05$) on the mean error value. Whereas a *confidence interval* represents the likelihood that the true value of a sampled population lies within a computed range, a prediction interval represents the likelihood that the *next* sample of the quantity falls in that range. Since we are attempting to make decisions based on observations of quantities such as network bandwidth and latency, whose true values are neither known nor fixed, the prediction interval fits our purpose more closely.

Given the calculated error bounds, we must decide how to use them to make a decision. First, we can use the bounds on all predictors to calculate the resulting bounds on time, energy, and cellular data; for example, the minimum network transfer time occurs at the upper bound of bandwidth and the lower bound of latency. Then, if no two non-redundant strategies have overlapping time bounds, this means that, with the chosen confidence, redundancy does not offer benefit. In this case, we simply choose the best non-redundant strategy.

If any non-redundant strategies' time bounds do overlap, we then evaluate redundancy as follows. First, we identify the best non-redundant strategy as above. Next, we calculate the maximum possible benefit from redundancy and the minimum additional cost, and we

choose redundancy if the benefit exceeds the cost. Since a redundant strategy's completion time is the time of its fastest non-redundant strategy, redundancy provides the most benefit when the supposedly best non-redundant strategy turns out to be worse than expected and the supposedly worse non-redundant strategy turns out to be better than expected. Therefore, the maximum benefit from redundancy is the minimum time of the redundant strategy subtracted from the maximum time of the best non-redundant strategy. Similarly, the additional cost of redundancy is calculated using the bounds of the additional strategies used, not including the best non-redundant strategy.

Because the error bounds method captures the distribution of error with less precision than the empirical distribution of the brute force method, and because the computed bounds are fairly loose, the error bounds method tends to overestimate the benefit of redundancy. The choice to use the upper bound on redundancy benefit and the lower bound on redundancy cost also biases this method towards redundancy. These effects are seen in our evaluation and discussed in Section 4.5.

4.3.3 Bayesian estimation

Our last method quantifies uncertainty by calculating the *posterior distribution* of the actual predicted quantities with the predictor values as *evidence*. This process is known as Bayesian inference or estimation. It centers around the Bayesian view of statistics and probability, which provides a way to formulate sound beliefs about an environment given some prior beliefs and new evidence. Prior work applied this technique to computation offload but only considered the bandwidth of one network and did not consider redundancy [103].

In our case, the prior beliefs are our historical measurements of predicted quantities such as bandwidth and latency, and the new evidence is the most recent decision made by Meatballs; e.g., "based on the predicted bandwidth and latency, WiFi is better than cellular." From Bayes' Theorem, the *posterior* distribution of bandwidth and latency is proportional to the product of the *prior* bandwidth and latency distributions and the *likelihood* of the

evidence (the predictors' decision) given the prior observations:

$$\text{posterior} = \frac{\text{likelihood} \times \text{prior}}{p(\text{evidence})}$$

where $p(\text{evidence})$ is a normalizing factor that ensures that the posterior distribution is a PDF (that it sums to 1).

It may not be immediately clear how the Bayesian formulation incorporates the *error* of the predictors, as was explicitly done in the prior two methods. To clarify this, we draw an analogy to a textbook example of Bayesian reasoning: a simple problem of weather forecasting, answering the question, "will it rain tomorrow?" In this problem, the prior is the historical frequency of rain, and the evidence is a forecast (a prediction of "rain" or "no rain"). Since weather forecasting is imperfect, a forecast will sometimes predict rain when none occurs, and sometimes it will predict clear skies when rain occurs. This inaccuracy is captured in the likelihood measure, which answers the questions: "when it *does* rain, how often does the forecast agree?" and "when it *doesn't* rain, how often does the forecast agree?"

The likelihood measure answers similar questions for Meatballs; e.g., "when bandwidths and latencies have been such that WiFi was better, how often has WiFi been predicted to be better?" Thus, the Bayesian formulation captures the accuracy of the predictors, but in a more abstract sense than the previous methods: by measuring how the inaccuracy affects the correctness of the resulting decisions.

In order to compute posterior distributions, Meatballs keeps empirical records of the information specified in Bayes' Theorem: the prior distributions, the likelihood of the predictor decisions given the priors, and the normalizing factor $p(\text{evidence})$, which in our case is the strategy frequency regardless of the priors. These values are binned and kept as histograms to reduce storage costs.

Each time we obtain a new measurement, we check the predictors' decision (e.g., which network is best) *just before* the measurement was taken. We update the $p(\text{evidence})$ histogram with the indicated strategy, and we update one histogram in the likelihood array, using the new measurements to choose which histogram to update. Note that this is in

keeping with the Bayesian formulation of the problem discussed above; we are tracking how often the predictors have agreed with the next observed measurements, for various values of those measurements.

Given the posterior distributions of predicted quantities, Meatballs calculates the expected benefit and cost of redundancy in the same way as in the brute force method. That is, it iterates over the joint prior distributions and uses the information that it has tracked to compute the posterior joint distribution. It then uses the joint posterior distribution to compute the weighted sums as before, and it uses the same method to decide whether the benefit of redundancy is worth the cost.

As in the brute force method, Meatballs ages samples so that older observations have decreasing impact. In the Bayesian method, it accomplish this by aging the bins in each prior distribution's histogram; each bin in the histogram is now a weight rather than a count. When a new sample is added to a bin, the weight for all other bins in that distribution decreases.

It may at first appear that the Bayesian method is equivalent in performance to the brute force method, since it still produces a joint distribution over all predictors. In practice, however, the likelihood distribution is sparsely populated; when Meatballs calculates the weighted sum, it avoids including the portions of the joint prior distribution not included in the likelihood distribution, because it knows they are zero. Thus, whereas the the brute force method has complexity $O(n^m)$ with m predictors having n samples each, the Bayesian method has complexity $O(m \times n)$, or the total number of samples across all predictors.

4.3.4 Reevaluation from new information

The arrival of new information can change the decision about which strategy to employ. New information may arise from an explicit event such as association with a new access point. Alternatively, it may arise from the lack of an expected event, such as not receiving a response from a remote server within some time.

Meatballs reevaluates a decision by repeating the prior calculations using conditional probability distributions for the predictors, based on the new evidence. Regardless of the evaluation method chosen, Meatballs restricts consideration to only the portion of the distributions that fit within observed conditions. For instance, given that a remote operation has already taken at least `x` seconds, Meatballs, using the brute force or error bounds method, considers only the portion of the joint error distributions that agree with a duration of at least `x` (e.g., the region of bandwidths and latencies that would cause the operation to take at least `x` seconds). Similarly, if Meatballs is using the Bayesian approach, it only considers the regions of the prior distribution that agree, which produces a conditional posterior distribution.

The notion of what information is sufficient to trigger reevaluation is application-specific, so Meatballs allows an application to trigger a reevaluation whenever it observes a sufficient change in the environment. In the case of an explicit event, deciding to trigger reevaluation is straight-forward. However, it can be challenging to determine when the lack of information is sufficient to justify a reevaluation, so Meatballs provides additional support for this case.

On application request, Meatballs will calculate the "tipping point" at which the conditional error distributions calculations will cause the decision to change. It does a binary search over the range of a given predictor, noting the strategy chosen at each point. The search terminates when the tipping point is found within a specified granularity. Since the strategy selection is cleanly separable from the details of how a method is implemented, this binary search works for all three methods without modifications.

Note that the binary search strategy assumes that only one predicted value changes based on new information. More complex applications for which this assumption is not valid may simply schedule periodic reevaluations instead.

Application → Meatballs	Meatballs → Application

`make_strategy(time_fn, energy_fn, data_fn, s_arg)` → `strategy`
`make_redundant_strategy(strategy[])` → `strategy`
 	
`create_predictor()` → `predictor`
`get_predictor_value(predictor)` → `double`
`add_measurement(predictor, measurement, new_prediction)`
`set_predictor_bound(predictor, bound, {UPPER,LOWER})`
`clear_predictor_bounds(predictor)`
 	
`register_strategies(strategy[], enum method)` → `evaluator`
`choose_strategy(evaluator, c_arg)` → `strategy`
	`time_fn(s_arg, c_arg)` → `double`
	`energy_fn(s_arg, c_arg)` → `double`
	`data_fn(s_arg, c_arg)` → `double`

This table shows the API for incorporating uncertainty into application decisions. Applications call `make_strategy` to specify a strategy in terms of time, energy, and cellular data calculation functions. `make_redundant_strategy` creates a redundant strategy composed from multiple non-redundant strategies. The application informs Meatballs about a predicted quantity via `create_predictor`, and it passes its measurements and predictions to Meatballs via `add_measurement`. To make a decision, the application first calls `register_strategies`, passing its list of strategies and chosen evaluation method and receiving an evaluator. It then calls `choose_strategy`, and Meatballs calls into the application's time, energy, and data functions to perform error-adjusted calculations. In those functions, the application calls `get_predictor_value` to retrieve error-adjusted values according to the selected evaluation method. Once finished, Meatballs returns the best strategy.

Table 4.1: Meatballs API

4.4 Implementation

4.4.1 Library

We built Meatballs, a library that makes it easy for mobile applications to consider predictor error in their decisions. Meatballs encapsulates common code needed to execute the three methods described in the prior section. Table 4.1 lists the main functions of Meatballs' API. An application first specifies the non-redundant strategies from which it will decide. Each specification includes strategy-specific functions that calculate the time, battery energy, and data usage of executing the strategy given predicted quantities. Next,

the application creates redundant strategies; each such strategy is composed of two or more non-redundant strategies that will be executed concurrently. Meatballs will calculate the time and resource usage of the redundant strategies using one of the three methods in the previous section.

An application next specifies the predictors, each of which is responsible for tracking a single quantity such as network bandwidth or latency. It may specify any combination of custom predictors or the generic predictors that we describe in the next section. Meatballs tracks the error in the predictions. Thus, each time an application makes a new measurement, it passes the measurement, the previous prediction for the quantity, and the new prediction for the quantity to Meatballs. In turn, when Meatballs runs its evaluation method, it uses its error-adjusted estimates when invoking the application-supplied strategy functions, which those functions obtain via the predictor interface.

The predictor interface also allows applications to bound the distribution of possible values based on new information. For instance, if a cyber foraging application has not received a response in time `x`, it sets a lower bound of `x` on the response time. Meatballs will then use only the portion of the distribution with values greater than `x` to estimate response time.

Finally, an application passes the list of strategies to Meatballs and receives back an *evaluator*. An evaluator is simply a handle by which the application can ask Meatballs to choose the best strategy. The evaluation method is specified when the evaluator is created, and the evaluator provides a uniform interface for choosing a strategy and scheduling deferred reevaluation. The evaluator also takes responsibility for caching previously computed decisions and invalidating the cached decisions when new measurements arrive.

4.4.2 Predictors

We provide several generic and customizable predictors for applications to use. We briefly describe here the specific prediction algorithms that we drew from the literature.

Network bandwidth and latency. The network predictor implements a flop-flip filter [49] to smooth estimates of network quality. Separate latency and bandwidth estimates are calculated via linear interpolation over varying transmission sizes. The predictor provides separate estimates for each available network (typically WiFi and cellular).

Network dwell time. This predictor estimates the time the mobile computer will remain in range of the current WiFi access point. It fits the dwell time to a Weibull distribution via maximum likelihood estimation using the methods reported by Lee et al. [54]. The effect of this is that it has low confidence in the reliability of a newly-associated WiFi network, but the confidence increases over the time of association. It provides per-AP estimates given sufficient samples for the current AP; else, it provides a generic prediction.

Application compute time. This predictor estimates the time to complete a application-specific computation using an approach similar to that of Narayanan et al. [65]. It posits a linear relationship with computation parameters and performs a regression to generate the best fit. This is application-specific, so the application must specify the value of the regression parameters for each observation.

Application energy usage. This predictor uses the PowerTutor models [105] to estimate energy usage as a function of communication and computation. Similar to the previous predictor, the amount of computation and communication is a linear fit over application-specified parameters.

4.4.3 Applications

We have modified two applications to use Meatballs: network selection and speech recognition.

4.4.3.1 Network selection

Network selection allows mobile data to be sent over either cellular or WiFi networks, as conditions warrant. Prior work often focuses on throughput and either sends data over a single network at a time or stripes data by sending different data over different networks.

Our work focus on user-facing activities, so latency, not throughput, is the relevant metric. In such scenarios, it may make sense to redundantly transmit the same data over multiple networks and use the first instance to arrive at the remote endpoint.

We chose to modify Intentional Networking (see Chapter 2), though several systems provide similar multi-network functionality. Our modified version estimates network bandwidth, latency, dwell time, and energy usage for each available network using the predictors described above. For each transmission, it calls into Meatballs for a decision as to whether the transmission should be sent using WiFi, cellular, or both networks redundantly.

The decision of when to reevaluate is application-specific. Intentional Networking reevaluates its choice of strategy whenever the mobile device associates with a new network, or a network it is currently using fails. In addition, the strategy choice is reevaluated when a response is not received after a delay of two times the expected completion time. This avoids the wasted time and resources of spurious retransmissions when networks are stable, but still allows for quick recovery from real failures. The reevaluation is repeated every 200 ms until the decision changes or the transfer completes. The effect of reevaluation is a graceful response to failure in which a redundant transfer is started on a second network as Intentional Networking loses confidence in its initially chosen network.

4.4.3.2 Speech recognition

Speech recognition is a classic application of cyber foraging [13], the offloading of computation from a mobile client to a remote server. We modified the PocketSphinx [23] library to perform recognition on the mobile device, on a remote server, or concurrently at both locations.

This application uses the above predictors for network latency, bandwidth, and dwell time, as well as the predictors for application compute time and energy usage. The regression parameter for compute time and energy usage is the length of the spoken utterance. However, since recognition starts immediately and proceeds in parallel with the speaking of the utterance, the precise length of the utterance to be spoken is not known when the initial decision about where to execute recognition is made.

Scenario	WiFi Coverage	WiFi session length		3G Bandwidth (Kbps)			
				Downlink		Uplink	
		Median	Max	Median	Max	Median	Max
Walking	69%	41 sec	5 min	137	737	48	454
Driving	27%	7 sec	2 min	368	1200	40	74

Table 4.2: Details of the network traces used for evaluation

Therefore, this application also uses an application-specific predictor to estimate the length of the utterance using the utterance lengths that have been observed in the past. The output of this predictor is used to compute recognition time (measured from the point when the user finishes speaking to the time the recognized text is displayed) and energy (measured over the entire recognition).

In our experience, the upload bandwidth of both 3G and LTE networks is too low for remote execution—it is faster and more energy-efficient to run recognition locally. Thus, remote recognition takes place only when WiFi is available.

The speech application reevaluates its decision in three cases. First, a decision to execute locally is reevaluated if the mobile device associates with a new WiFi network (since remote recognition may now make sense). Note that if the local recognition is likely to complete soon, starting a remote recognition may not make sense even if network quality is good. Second, the utterance may be significantly longer than predicted. The application uses Meatballs' binary search feature to precalculate the tipping point for utterance size and changes its recognition strategy if the tipping point is reached (e.g., by starting a redundant remote execution). Finally, the application also uses Meatballs' binary search feature to calculate a tipping point in case remote recognition takes longer than expected. This is the point at which a redundant local recognition is started.

4.5 Evaluation

Our evaluation answers the following questions:

- When do the benefits of redundancy outweigh the costs?
- How useful is considering prediction uncertainty in identifying opportunities to employ redundancy?
- How do the three evaluation methods (brute force, error bounds, and Bayesian) compare?

4.5.1 Experimental Setup

4.5.2 Testbed

We run all applications on the AT&T version of the Nexus One, running Android 2.3.4. We modified Android to allow the simultaneous use of WiFi and cellular networks. To ensure repeatable experiments, the phone connects to an isolated WiFi access point and a private Cisco MicroCell that is connected to AT&T's network. Since the MicroCell acts as a miniature cellular tower, our evaluation captures the effects of the cellular wireless medium on network bandwidth and latency. We emulate network conditions by passing all traffic through a computer that inserts delays with the netem [56] network emulator and throttles throughput using the Linux Advanced Routing and Traffic Control tools [55]. We run servers for each application on a Dell Optiplex GX270 desktop with a 2.8 GHz Pentium 4 processor and 1GB DRAM.

We measure energy usage by applying the Nexus One power model used in the implementation of PowerTutor [105]. We measure cellular data usage by reading the number of bytes sent and received through the Linux sysfs interface.

Usage	Expected battery life	Battery life reduction		
		Low-cost (100 J)	Mid-cost (10 J)	High-cost (1 J)
Idle	125 hours	37.5 min	3.75 min	22.5 sec
Average	20 hours	6 min	36 sec	3.6 sec
Heavy	2 hours	36 sec	3.6 sec	0.36 sec

This figure shows the approximate battery life impact of spending various amounts of energy to save 1 second of delay in different evaluation scenarios.

Table 4.3: Performance-energy tradeoff in various scenarios

4.5.3 Scenarios

Performance, energy, and cellular data usage have varying importance to different users. We capture this through the importance factors described in Section 4.3. Our experimental results are thus given as a weighted cost that is derived from using these factors to equate the diverse metrics of time, energy consumption, and data usage. One can contextualize these factors in order to give them more concrete meaning; e.g., "I am currently willing to spend X Joules to save Y seconds of waiting."

We evaluate our results in four different scenarios, each of which reflects a different potential tradeoff between performance (interactive delay) and resource consumption (energy and data usage). The first is the `no-cost` scenario, in which performance is the only consideration (energy and data usage are not considered). Table 4.3 shows the time-energy tradeoff in the remaining three scenarios: `low-cost`, `mid-cost`, and `high-cost`. In the `low-cost` scenario, a user is willing to spend 100 Joules to save 1 second of waiting. With that amount of energy, the user could watch 36 seconds of streaming video, or the user could leave the phone idle for an additional 37 minutes. Based on the average behavior (including both idle and usage periods) given by a recent study [74], this would equate to 6 minutes of extended battery lifetime. In the `mid-cost` scenario, 1 second of waiting equates to 3.6 seconds of video watching, 3.75 minutes of idle time, or 36 seconds of av-

erage usage. In the `high-cost` scenario, 1 second of interactive delay purchases only 0.36 seconds of video playing or 22.5 seconds of battery lifetime on a completely idle phone. This already seems like a very poor tradeoff, so we did not investigate higher cost scenarios.

Similarly, in the `low-cost` scenario, 1 second of waiting time equates to saving 100 KB of data usage. In the `mid-cost` and `high-cost` scenarios, 1 second of waiting equates to 10 KB and 1 KB of data, respectively.

4.5.4 Trace-driven evaluation

We use trace-driven emulation to provide experimental repeatability and allow for meaningful comparison between strategies. We gathered these traces by continuously measuring the bandwidth and latency via active probing to a server at the University of Michigan. We use two traces that illustrate different mobility scenarios: a walking trace gathered on the University of Michigan's north campus, and a vehicular networking trace collected in Ypsilanti, MI (one of the vehicular traces described in Chapter 2). The vehicular trace illustrates a highly-mobile scenario with challenging network dynamics. The walking-trace has longer associations with WiFi access points and better overall connectivity. Table 4.2 details the network characteristics for the two traces.

When running benchmarks, we replay the traces on the emulation computer, which throttles bandwidth and delays packets for each network according to the conditions observed. When no WiFi or cellular coverage is observed in a trace, the throttling computer causes the connection to drop. The Android OS typically discovers the network disconnection after several seconds. Since the collected traces are longer than our experiments, we use only the first portion of each trace.

4.5.5 Results

4.5.5.1 Network selection

To evaluate the effectiveness of redundancy in network selection for small, interactive transfers, we constructed an experiment in which the application executes a series of request-response exchanges, using Intentional Networking.

These requests and responses are of random sizes, normally distributed about 1 KB and 4 KB, respectively. The application pauses between requests for a think time uniformly distributed between 10 and 30 seconds.

We compare the results using Meatballs to two simple network selection strategies: only use cellular, or use WiFi when available and fall back to cellular when it is not. WiFi-preferred is effectively the strategy currently employed by both Android and iOS.

For comparison, we also built an adaptive strategy that uses the same cost weights and formulas as Meatballs to calculate the time, energy, and data cost in order to select either cellular or WiFi (when available). This is a standard adaptive strategy that treats predictions as oracles and does not incorporate uncertainty. It also does not reevaluate its decisions based on new information.

Each experiment runs for 20 minutes. We report the total weighted cost over this 20-minute interval. All results are the mean of 5 runs; error bars are 95% confidence intervals. Performance is the time the user spends waiting for a response to each request. We first discuss the results for the brute force method, and later compare the three methods.

Figure 4.1 shows weighted cost results for the walking trace, normalized to the weighted cost of the cellular-only strategy. Redundancy provides the most benefit in the `no-cost` scenario, when performance is of utmost importance. Compared to a cellular-only strategy, the user spends 24% less time waiting when redundancy is employed. The reduction in waiting time due to redundancy is greater than a factor of two compared to both the WiFi-preferred and the adaptive strategies.

It may seem surprising that the slower cellular-only strategy provides better performance than the WiFi-preferred strategy that mobile devices commonly employ. The im-

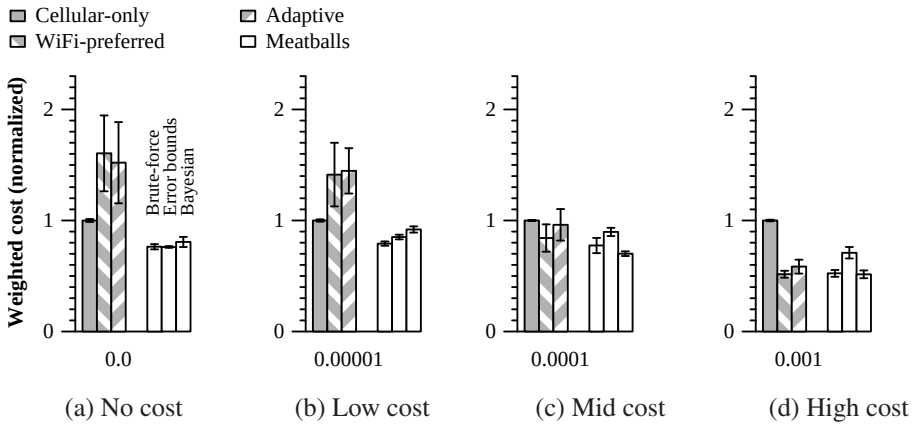


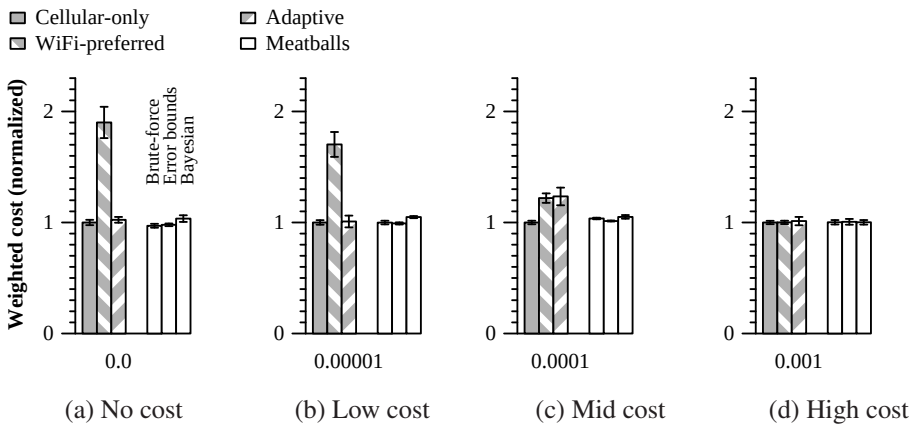
Figure 4.1: Network selection, walking trace

pact of WiFi failure detection delay accounts for this gap; even though it often provides better performance, WiFi is less reliable than cellular when the user is moving. A few long delays due to failure detection and failover when the mobile device moves out of range of an access point dwarf the performance difference between cellular and WiFi.

The `low-cost` scenario shows similar results to the `no-cost` scenario. However, since energy and cellular data now have non-zero weight, the WiFi-preferred and adaptive non-redundant strategies show improved results. Meatballs provides an improvement of 21% over cellular-only and 44% over WiFi-preferred and adaptive non-redundant.

The utility of redundancy drops as resource conservation becomes more important. In the `mid-cost` scenario, the strategies that prefer to send less cellular data have improved considerably. Meatballs' redundant strategies are equivalent within experimental error to the best non-redundant strategy (WiFi-preferred). However, for the brute force and Bayesian methods, both the average and the variance are still lower than WiFi-preferred, due to redundancy reducing the effect of the outliers caused by WiFi failover delay. This contributes to a less variable user experience.

In the `high-cost` scenario, the brute force and Bayesian strategies are equivalent within experimental error to the best non-redundant strategy (WiFi-preferred). Thus, Meatballs achieves benefit from redundancy when it is available and correctly chooses to avoid redundancy when it is not helpful.



(a) No cost (b) Low cost (c) Mid cost (d) High cost

Figure 4.2: Network selection, driving trace

Note that since the cellular-only strategy never uses WiFi, a user could potentially achieve slightly better energy usage by disabling WiFi on their phone. However, the energy savings from doing so are minimal (1% reduction in weighted cost in the `low-cost` scenario to 4% in the `high-cost` scenario). Any potential gains are therefore dwarfed by the poor performance of the cellular-only strategy. Further, the user would lose the opportunity to employ WiFi when it is best (as it is in the `high-cost` scenario).

Figure 4.2 shows the results for the driving trace. Unlike the walking trace, opportunistic WiFi provides little benefit, because most sessions are short and unreliable. Thus, cellular only is best in all scenarios. Redundant transmission simply offers little benefit because the disparity between WiFi and cellular quality is too great. Meatballs correctly selects the non-redundant cellular strategy in almost all cases. This scenario demonstrates an important property: Meatballs eschews redundancy when it has high confidence in the underlying predictions, so it avoids doing harm when redundancy has little benefit.

The unshaded bars in Figures 4.1 and 4.2 compare the relative weighted cost realized by the three redundancy evaluation methods. The results are mixed when comparing our three methods. The Bayesian method is often best in higher cost scenarios and the brute-force method is often best in lower cost scenarios. The brute force method has higher computational complexity, which may result in higher weighted cost. The Bayesian method incorrectly sends some transfers non-redundantly in the lower cost scenarios because it is less accurate than the brute-force method.

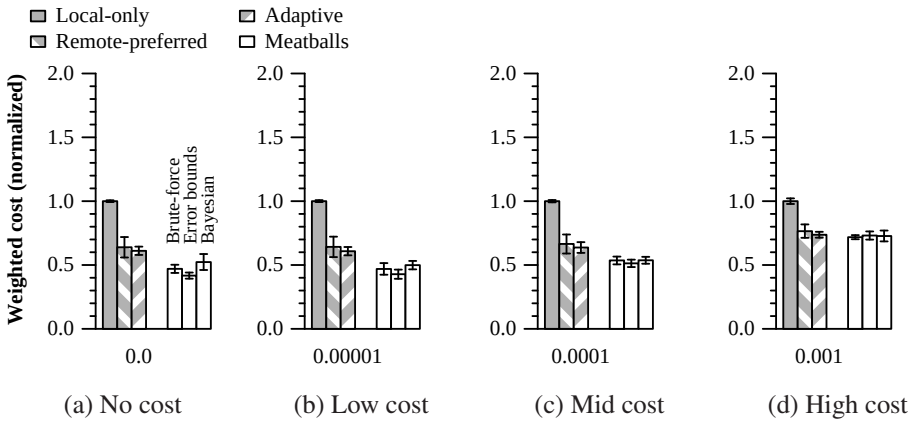


Figure 4.3: Speech recognition, walking trace

The error bounds method performs the poorest in general. It spends resources more readily, resulting in higher weighted cost in the scenarios where resource conservation is more important. Since the bounds are loose, the method is prone to having an inflated view of predictor uncertainty and thus naturally tends more towards redundancy.

4.5.5.2 Speech recognition

To evaluate the effectiveness of redundancy in a speech recognition application, we selected 20 utterances ranging in length from 1 to 8 seconds. We simulate a user speaking at the original sampling rate of 20 KB/sec, looping over the 20 utterances at 30-second intervals until 20 minutes have elapsed. The performance metric is the recognition delay, or the interval between the time when the user finishes speaking and the time with the transcribed text is available. We report the total weighted cost, which reflects only recognition delay and energy usage in these experiments because cellular usage is negligible.

The two non-redundant strategies are to always execute locally or to prefer remote recognition if WiFi is available (since remote recognition using cellular transmission is always inferior to local recognition). We also compare against an adaptive strategy that considers time and cost but not predictor error. The redundant strategy executes both locally and remotely.

Figure 4.3 shows the results for speech recognition on the walking trace. Note that the approximate break-even point for redundancy is even higher in this application—that is, redundancy still shows significant benefit in the `mid-cost` scenario. Speech recognition generally takes longer than a single small request/response exchange, so there is more delay to be saved for a given cost.

In the `no-cost` scenario, redundant strategies are generally best, with performance improvement of 23–35% over the remote-preferred and adaptive non-redundant strategies, and improvement of more than a factor of two over the local-only strategy. The exception is the Bayesian method, which shows high variance and thus is equivalent within error to the remote-preferred and adaptive non-redundant strategies.

In the `high-cost` scenario, the remote-preferred strategy dominates due to its reduced energy usage, and executing a local redundant recognition is not worth the cost. Meatballs correctly selects the non-redundant strategy in this scenario.

Note that, as was the case in network selection, no single strategy is always best, and Meatballs reduces weighted cost by up to a factor of two compared to the simple strategies in scenarios where they are not best. Also, even though local-only is never the best strategy in any scenario, using local execution in addition to remote execution does provide benefit in reducing the uncertainty of remote execution, as it mitigates the impact of failover delays.

We also ran this application with the driving trace. We do not show the results because they mirror the previous application. The poor network quality causes local recognition to always dominate remote recognition. Meatballs achieves equivalent performance by correctly selecting the local recognition in almost all cases.

Finally, we ran the speech application in a scenario designed to show the impact of server load. In this experiment, a high-quality WiFi connection is always available, but we add a background load of concurrent clients requesting speech recognition. Concurrent recognitions are performed by separate event-based server processes that compete for the CPU resource on the server. We vary the number of concurrent requests according to a Poisson distribution, with the average load increasing from 6 to 10 concurrent clients during the course of a 10 minute experiment. Since the network is stable, this experiment

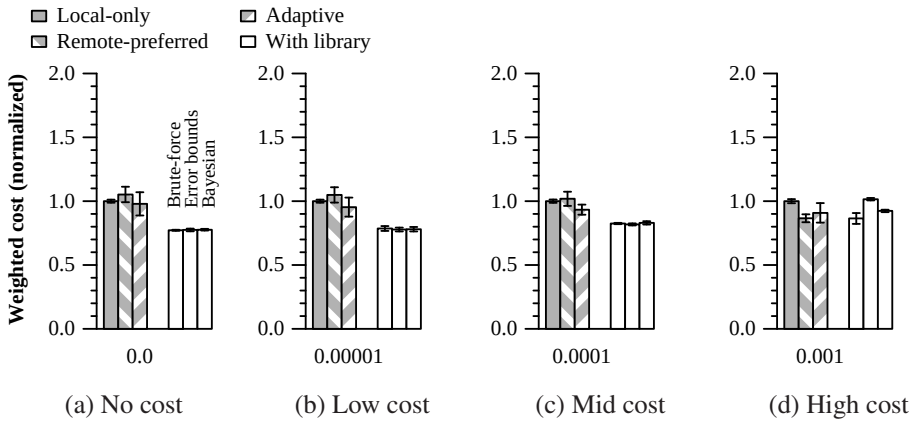


Figure 4.4: Speech recognition, server load trace

shows how Meatballs reacts to changes in recognition time due to server load and varying utterance lengths.

As Figure 4.4 shows, Meatballs reduces recognition delay in the `no-cost` scenario by 23% compared to the local-only and remote-only strategies. The adaptive non-redundant strategy also performs poorly in this scenario because it frequently makes incorrect decision due to the difficulty in predicting future server load. Meatballs achieves improved performance by hedging its bets with redundancy. Meatballs achieves similar benefit in the `low-cost` and `mid-cost` scenarios by choosing local recognition in periods where the server load is likely to be high, remote recognition in period where the server load is likely to be low, and hedging its bets when there is uncertainty about which strategy is best. In the `high-cost` scenario, remote recognition is almost always superior, though the server load causes high variance. However, Meatballs sometimes hedges its bets via redundancy. The brute-force method does this in only one of five runs, due to a single high load measurement; it is therefore equivalent within error to the remote-only method. The error bounds method is equivalent within error to the local-only method due to energy usage, and the Bayesian method outperforms the local-only strategy but under-performs the remote-only strategy (saving time but using more energy).

For this application, the error bounds method performs the best on the network trace. The brute force and Bayesian methods are generally equivalent within experimental error and best on the server load trace. As with the previous application, the Bayesian method

is prone to underestimating uncertainty and choosing redundancy less readily. The brute force method again has high computation complexity.

4.5.5.3 Discussion

Overall, we observe significant benefit from redundancy: user wait time is reduced by up to a factor of two. Further, our simple back-of-the-envelope estimations suggest that sufficient resources are available quite often for the average user.

When redundancy offers no benefit, Meatballs usually approximates the best non-redundant strategy. The one exception is the `high-cost` scenario for speech recognition with CPU load: in this case, Meatballs overestimates the inherent variability and sometimes chooses redundancy inappropriately.

Finally, we observe that the brute force method appears to best capture and act upon the uncertainty in the applications' predictions, whereas the error bounds and Bayesian methods tend to overestimate and underestimate predictor error, respectively. On the other hand, the brute force method has the highest computational cost, which penalizes it in high-cost scenarios. Now that we understand these factors, we plan to add a feature in which Meatballs chooses the best method based on observed application characteristics.

4.6 Summary

Mobile applications operate in environments fraught with uncertainty, yet they frequently depend on predictions made about this environment, such as network bandwidth and latency. Because applications commonly treat these predictions as oracles, they are bound to make wrong decisions some of the time, which leads to user-visible delays that could have been avoided.

We propose that applications should consider uncertainty as a fundamental aspect of their decision-making. Rather than simply returning an expected value, predictions should include some measure of uncertainty. Applications should take this uncertainty into ac-

count when choosing a strategy to complete the task at hand. When applications include predictor uncertainty in their decisions, it often becomes prudent to employ redundant strategies, spending a modest amount of energy and cellular data resources to purchase a reduction in delay and variability, both of which improve the user experience.

We have implemented Meatballs, a library that provides three different methods for quantifying predictor uncertainty and helps applications make decisions with this uncertainty in mind. We have modified network selection and speech recognition applications to use this library. Our experiments show that there is significant benefit to be gained by employing redundancy when sufficient resources are available. In the face of uncertainty, Meatballs reduces user-visible delays by up to a factor of two.

CHAPTER 5

Related work

This chapter reviews prior systems related to multinetworking, mobile prefetching, and consideration of predictor uncertainty. We summarize the approaches in these prior works and explain where we have borrowed, been inspired, and differed from these.

5.1 Intentional Networking

There is a large body of work that seeks to route network traffic over multiple interfaces. Prior work largely falls into one of two categories: *application-oblivious*, in which the network over which data is sent is chosen based on system-wide goals such as maximizing throughput and without consideration of application intent, and *application-alone*, in which each application must manage the details of selecting among multiple networks on its own and the system's role is only to expose the details of possible options to the application.

Virtual WiFi [20] is one application-alone solution. It virtualizes a device's wireless interface, fooling applications into believing the device is connected simultaneously to different APs on different channels. This is a step in the right direction, because devices can now exploit all available connectivity in their vicinity. Unfortunately, Virtual WiFi places the burden of access point selection entirely on the application. In contrast, Intentional Networking presents applications and users with a single unchanging network interface that accepts declarative intent.

Application-oblivious systems are more numerous. FatVAP [47] presents an infrastructure similar to that of Virtual WiFi, but operates only within a single layer of an overlay network, and is concerned only with maximizing throughput, without concern for other application-level preferences. Other systems attack the bandwidth aggregation problem by designing new multi-path transport protocols to replace TCP, such as R-MTP [60], pTCP [40], mTCP [106], and SCTP [97]. SCTP also supports multi-streaming of independent byte streams; in contrast, Intentional Networking allows applications to specify ordering and atomicity constraints over data sent to a destination computer. Multi-path transport has also been built into the kernel just above the transport layer [88]. Chebrolu et al. [21] use a modified network layer at the mobile host and at a remote proxy to hide the use of multiple networks, and the resulting reordering of packets, from the transport and application layers. Though all of the above application-oblivious systems are simple for applications to use, they only focus on throughput maximization and cannot take into account other application-specific or request-specific goals such as minimizing latency.

In contrast to application-oblivious and application-alone strategies, Intentional Networking splits the burden of network selection among applications and the system. Applications disclose qualitative hints about their intentions in using the network, and the system reasons about how traffic labeled with those hints should be mapped to specific networks based on their current characteristics.

Rather than target throughput maximization, Wiffler [15] opportunistically routes data over WiFi to minimize cellular usage. Others [21, 81, 104] have argued that throughput maximization is not the only goal of interest to mobile applications and users, and that the ability to specify network usage policies on a per-application basis would be useful. We differ from these prior works in two ways. First, we argue that the application, not the user, should set policies. Application network usage patterns may change quickly, and the proper choice of policy changes likewise; it would place too great a burden on the user to understand their applications' behavior and constantly update the policies. Second, we propose, implement, and evaluate a specific mechanism for applications to set fine-grained policies by describing the intent of each network message.

The push toward ubiquitous computing makes automatic service discovery in new environments more important than ever [90]. Existing work, however, has focused more on enabling application-level services [25, 35, 95] than on choosing and managing a diverse set of network connections from an application's point of view.

Several systems seek to allow clients of one wireless service provider to access foreign wireless hotspots when roaming [18, 29, 61, 89] or between public and private networks [62]. Our work is complementary, since users must find and associate to an access point before negotiating such roaming agreements. This service discovery is similarly critical for grassroots wireless collective initiatives [17, 73, 92].

Contact Networking [19] hides the differences between local and remote communication from users. All communication appears to be local—like a direct Bluetooth connection between two devices—even if infrastructure such as the Internet is actually involved. Like us, the authors recognize that mobile devices typically have several heterogeneous wireless radios at their disposal. Contact Networking is also conscious of the properties of different link layers. Their primary focus, however, is on neighbor discovery, name resolution, and (ultimately) the preservation of application-level sessions in the face of user mobility. Our work does find common ground with the idea that all network connectivity options are not equivalent and the operating system should dynamically assign data flows to the most appropriate link.

Zhao et al. [107] attack problems similar to those addressed by Contact Networking. Their work lies firmly within the framework of Mobile IP [80] as well. The user's Home Agent is required to arbitrate the routing of various data flows. Further, applications must explicitly bind a data flow to a specific interface through their `SO_BINDTODEVICE` socket option. We propose a decentralized solution and envision the operating system automatically assigning flows to the optimal interface, aided at most by simple hints from applications.

Much recent work has argued that the multiple networking options available to mobile devices are a blessing, not a curse. Johansson et al. [45], among others, show how Bluetooth radios are often preferable to IEEE 802.11 for short-range, low-power communication. Bahl et al. [12] illustrate scenarios where multiple radios can help devices save energy,

enhance their data communication capacity, make wireless AP handoff more seamless, and better tolerate wireless link problems. Draves et al. [28] show how overall throughput can be increased for multi-radio nodes in mesh networks by dynamically choosing the "best" outbound link when forwarding a given packet. Stemm and Katz [96] recognize the hierarchical nature of overlapping wireless networks. Much like cache hierarchies in computer architecture, multiple wireless networks commonly cover one spot, with the utility (e.g., bandwidth) of a network usually inversely proportional to its coverage radius.

Labels are partially inspired by the use of hints to guide power management decisions in STPM [8]. Both projects share the goal of having applications disclose a minimal amount of information to guide resource management decisions. Yet, the domains to which these hints are applied are very different. STPM sets wireless network power management modes, while Intentional Networking changes the scheduling and routing of network messages.

5.2 Informed Mobile Prefetching

One of the primary contributions of IMP is the unification of prior solutions to a seemingly disparate collection of problems to meet the unique challenges of mobile prefetching. In this section, we describe prior work in prefetching and in managing limited resources in mobile computing, and we discuss how IMP unifies and builds upon these ideas.

5.2.1 Prefetching

Prefetching is a long-studied technique used in a wide variety of computing domains. Processors predict which instructions or data will be needed and populate cache lines in advance [94]. File systems and databases attempt to infer application access patterns and fetch data items before they are needed [51, 77]. Distributed file systems fetch files from servers in advance of their use, both to improve performance and to maintain availability in the face of unreliable network connections [50].

The design of IMP is inspired by Transparent Informed Prefetching (TIP), which uses cost/benefit analysis to manage allocation of disk cache buffers between competing consumers: prefetched data and the LRU cache [78]. Such work recognizes that prefetching must be done carefully, lest it harm more than it helps. We observe the same high-risk/high-reward nature of prefetching in mobile computing and construct a cost/benefit analysis based on user-perceived performance improvement and the cost of prefetching in terms of battery energy and cellular data.

In the domain of mobile computing, prefetching has long been recommended as a primary technique for improving application performance [102]. Prior theoretical work in prefetch algorithm modeling and analysis observes, as we do, the tension between improving data access performance and spending limited bandwidth and energy resources to achieve that end. Persone et al. develop a prefetch cost model by which to numerically evaluate prefetch strategies based on mobility patterns [26]. However, their work considers different costs of prefetching in isolation from each other and from the benefit of prefetching. In contrast, IMP explicitly considers how different costs, such as energy usage and 3G data consumption, may be meaningfully combined and weighed against the benefits that prefetching can provide.

Lee et al. seek to improve the efficacy of prefetching and counteract its inherent uncertainty by improving the accuracy of predictions, using location-specific or context-specific information [52]. Web prefetching has long used spatial locality to predict what data users will request next [76]. Such efforts are beneficial and complementary to IMP. IMP allows the application to decide which data should be prefetched and instead addresses the decision of when to prefetch given limited resources and changing network conditions.

As we have demonstrated, having more accurate prefetch hints from the application allows IMP to deliver better results. Thus, IMP might benefit from prediction of network availability and quality, as done systems such as BreadCrumbs [68]. By knowing with greater accuracy what future bandwidth and WiFi coverage to expect, IMP can make better-informed decisions about whether prefetching now or waiting until later would be more cost-effective.

5.2.2 Limited resources in mobile networking

Several recent projects have proposed techniques for dealing with the different strengths and weaknesses of WiFi and cellular networks. Wiffler [15] explored the use of intermittently-available WiFi to reduce 3G data usage and ease pressure on cellular networks, subject to application-specific delay tolerance. Bartendr [91] observed that the energy cost of sending data on a cellular network increases significantly as signal strength drops and that energy savings can be realized by predicting periods of good signal strength and, when possible, delaying transmissions to target those periods.

Additionally, much recent work targets the poor interaction between naive applications' cellular data usage patterns and the energy cost of 3G tail time. Balasubramanian et al. measured the significant amount of energy wasted in 3G tail time and developed TailEnder to amortize tail energy cost with batching and prefetching [16]. The Tail Optimization Protocol [83] predicts long idle periods and direct the cellular radio to release radio resources and enter the low-power idle mode without waiting for the tail time. TailTheft [58] "steals" parts of the tail time for small transmissions without extending the tail time.

We share these systems' goal of reducing energy and cellular data costs, and indeed, many of the techniques they describe are applicable to IMP as well. However, we also observe that resource conservation, while a worthy goal, is not always the most important goal for a mobile user, and that often, energy and cellular data can and should be spent more freely in order to improve the user's experience. We also observe that prefetching is not always beneficial, and that aggressively prefetching data that will not be used is unnecessarily wasteful. Hence, IMP explicitly considers the changing relative importance of these goals to tune its decisions over time, and it also considers the observed accuracy of prefetch hints to determine the value of prefetching.

IMP uses Intentional Networking, as described in Chapter 2, to simplify the use of multiple wireless networks and to prevent prefetch traffic from penalizing the performance of interactive traffic. IMP also benefits from the power modeling work of the PowerTutor project [105], which enabled automatic derivation of power models through a series of tests that exercise different components of a mobile device in turn and isolate the power

consumption of each. Finally, IMP adopts goal-directed adaptation techniques developed in Odyssey for energy savings [33], which we additionally apply to the cellular data resource. IMP applies these ideas to the new domain of mobile prefetching.

5.3 Coping with predictor uncertainty

We apply our techniques to two domains: code offload and network selection.

5.3.1 Code offload

Many prior systems offload computation by migrating software components to remote servers. MAUI [24] partitions applications by running methods either locally or remotely. It monitors method runtime, energy, and network conditions and uses a global optimization to choose the partitioning that minimizes energy usage while keeping added latency under 5%. It starts a new computation only if it detects a remote failure via timeout or dropped connection.

CloneCloud [22] partitions applications by running threads either locally or remotely. It profiles applications offline to generate optimal partitions for various network, CPU, and energy conditions. It measures these conditions when an application starts and chooses a partition via table lookup.

Spectra [32] also runs software components either locally or remotely. It measures supply and demand of CPU, network, energy, and storage at runtime. It calculates an expected value for each and uses those values to decide whether local or remote execution is best.

Chroma [14] builds on Spectra and so uses the same techniques. It selects from a wider set of tactics (strategies for partitioning and adapting the fidelity of applications). Additionally, Chroma provides a mechanism for executing redundant computation on multiple servers; however, it does not provide any policy for determining when redundancy should be employed. It is precisely such a policy that is the focus of our work. Because Chroma

does not implement such a policy, it is similar to Spectra in that its decisions consider only the expected values of resource supply and demand.

Like Chroma, Slingshot [98] also runs computation on multiple servers, but it always instantiates redundant computations when multiple servers are available. Thus, it does not use estimates and, like Chroma, it provides no policy for deciding whether or not to execute redundantly.

Odessa [84] partitions stages in pipelined processing by executing each stage locally or remotely. Additionally, it varies data parallelism within a stage by adding or removing threads; all threads of a stage run on a single computer. It makes such decision by considering average processing time, CPU speeds, and network and bandwidth estimates.

In summary, none of these prior solutions, nor any other offloading system of which we are aware provide policies for considering when to execute redundantly. This is because all partition computation by running components either on the mobile device or on one or more servers. The two systems that execute redundantly on more than one server (Chroma and Slingshot) provide a mechanism but no policy for deciding whether or not to execute redundantly. Thus, our work would improve all of these prior solutions by providing a policy that identifies when to employ redundancy.

5.3.2 Network selection

Current mobile devices (e.g., iOS and Android) generally use a trivial network selection policy: prefer WiFi over cellular when available, unless the user has disabled WiFi. This assumes WiFi will outperform cellular, which is not always the case, especially when the user is moving and with modern cellular technology such as LTE.

Several prior systems send data concurrently over multiple mobile networks. R-MTP [60], pTCP [40], and mTCP [106] attempt to aggregate throughput by striping data across multiple networks. Note that striping is *not* a redundant strategy; any given unit of data is only sent on a single network. Striping hinders low latency (the focus of our work) because the aggregate transmission experiences the latency of the slowest network. Careful

scheduling is required because a single network estimated poorly is enough to stall the entire pipe when ordered delivery is required; the designers of FatVAP eschew striping within flows for exactly this reason [47]. Before our modifications, Intentional Networking (see Chapter 2) identified latency-sensitive traffic, but it sent such traffic over only a single network. Compared to these prior systems, our work is the only one to explicitly consider redundantly sending the same piece of data over multiple networks to lower latency.

Network measurement is itself a well-studied area, encompassing a variety of passive [49, 44, 100, 53] and active [41, 87, 43] techniques. Most of these techniques produce spot measurements of available bandwidth and latency without providing a measure of the uncertainty in those measurements. Jain and Dovrolis [43] characterize available bandwidth with a variation range, but offer no policy for using the range to make decisions.

5.3.3 Other domains

Redundancy is used in cloud computing [10, 27] to reduce tail latency of compute jobs by sending them to multiple servers. Routing in ad-hoc sensor networks has long used multiple redundant paths to provide fault tolerance in the face of packet loss and node or path failure (e.g., [7]). These prior systems use domain-specific algorithms to determine when to employ redundancy. We focus on the more general domain of mobile applications, and therefore offer more generic algorithms for quantifying prediction error and deciding when to employ redundant strategies.

CHAPTER 6

Conclusion

In this final chapter, we summarize the contributions of the dissertation and reflect on its limitations and practical impact. We also describe promising areas of future work in balancing performance and resource tradeoffs in mobile computing.

6.1 Contributions

In this dissertation, we have argued that mobile systems can significantly improve applications' user-visible performance, simply by providing abstractions that enable principled use of multiple networks and careful spending of limited battery energy and cellular data resources. In support of this thesis, we have described three such abstractions that contribute to accomplishing this goal.

First, we described *Intentional Networking*, a system that provides applications with a simple abstraction for using multiple mobile networks effectively. Applications decorate their network traffic with simple, qualitative labels, and the system uses those labels to match traffic to networks and reorder traffic for improved interactive performance. Since sending traffic on multiple networks introduces reordering, Intentional Networking provides abstractions for ensuring atomicity and ordering, analogous to the locks and condition variables used in concurrent programming. Our experiments with three applications modified to use these abstractions show that Intentional Networking indeed helps appli-

cations achieve lower interactive response time with only modest background throughput overhead.

Next, identifying prefetching as a promising mechanism for spending resources to purchase improved performance, we described *Informed Mobile Prefetching*, a system that provides applications with a simple abstraction that helps them prefetch the right amount of data at the right time. IMP tracks the spending rate of battery energy and cellular data, aiming to neither overspend nor underspend these *budgeted* resources. Further, since prefetching can be wasteful if the user never requests the prefetched data, the system keeps track of the likelihood that the user will request a given data item and incorporates this knowledge when deciding the value of a prefetch. Experiments show that IMP is able to meet specified resource goals and that in most cases, when simple heuristic-based strategies also meet the goals, IMP outperforms them on interactive response time, often by a factor of two or more.

Finally, we argued that mobile applications should explicitly consider the *uncertainty* involved in using predictors such as network bandwidth and latency. Mobile applications commonly rely on such predictors to make decisions about e.g. which of two networks to use, and the user suffers delays when the decisions turn out to be wrong. Instead, applications could execute *redundant* operations in parallel and use whichever result finishes first, thereby eliminating the effects of the uncertainty and achieving interactive performance approximately equivalent to a perfect strategy with knowledge of the future. We implemented *Meatballs*, a library that enables applications to consider uncertainty when making such decisions. We implemented three techniques for quantifying the uncertainty of predictors and the resulting expected benefit of redundancy. Since redundancy has a cost in terms of battery energy and cellular data, our system considers these costs when deciding whether to employ redundancy, based on the relative importance of performance and resource conservation at a point in time. Experiments show that, when sufficient resources are available and uncertainty exists, considering predictor uncertainty improves interactive performance by up to a factor of two.

Providing mobile application developers with the right abstractions for balancing tradeoffs between user-visible performance and the careful spending of limited resources will

enable more developers to achieve better experiences for their users by making better use of the available networks. The applications of these principles articulated in this dissertation represent a substantial contribution towards achieving this goal.

6.2 Limitations

Though we have shown in our evaluation that the approaches in this dissertation have substantial benefit in realistic scenarios, there are also scenarios where the benefit of our strategies is limited. For example, if an application is primarily composed of one type of traffic (e.g., interactive), or if a user does not have significant background traffic, the utility of labeling traffic with Intentional Networking is limited. Similarly, if a high-quality network is always available, then the decision of which network to use or when to fetch data is less interesting, since the impact of the decision is minimal. Further, in such a scenario, there would not be much benefit to considering uncertainty in predictions, since little uncertainty exists. For this reason, it is especially important that our strategies are *adaptive*; our systems can detect scenarios when little potential benefit exists and ensure that they do no harm.

The broadening deployment of high-speed LTE networks raises the question of whether opportunistic WiFi usage will continue to provide benefit going forward. For instance, if LTE or some future technology becomes ubiquitous, and if its energy usage improves greatly or if battery technology turns a corner, it may be reasonable to expect the impact of the tradeoffs discussed in this dissertation to decline. However, past and present experience agree that, even as wide-area cellular connectivity improves, local-area wireless connectivity tends to be superior [48]. Further, we can see anecdotally that users' appetite for more content, higher fidelity, and more powerful devices has kept pace with the advance of mobile technology. If there is in fact an asymptote approaching where mobile network, processor, and battery technology is almost always sufficient to meet user demand, it appears to be far from the present.

All of our techniques require modifications to application source code. We argue that the benefits we see in our evaluation would not be possible without the application and user knowledge gleaned through our APIs. Nevertheless, in order to broaden the deployability of our techniques, it would be useful to reduce or eliminate the amount of modification required as far as is possible. We describe some potential approaches for this in Section 6.4.

6.3 Practical impact

Since the completion of the work described in this dissertation, several of the ideas have appeared in similar form in practical real-world contexts. Multipath TCP [34] has been integrated into recent Linux kernels [86] and has even seen adoption by Apple for communication related to Siri [1]. MPTCP, like several of the prior works discussed in Section 5.1, has no knowledge of application intent and thus cannot match the performance of Intentional Networking for interactive traffic. Nevertheless, it is encouraging that system support for multinetworking has been realized in a real-world context, and we look forward to seeing how these deployments evolve and gain wider adoption.

QUIC (Quick UDP Internet Connections) [3] is an experimental network protocol from Google which shares some of the goals of this dissertation. QUIC is being developed in the context of the Chromium open-source web browser project and Google's SPDY replacement HTTP protocol, with the goal of reducing the number of round-trips involved in web browsing, especially with regard to the overhead of TLS. Because it uses UDP internally, QUIC is able to avoid the "head of line blocking" which can occur in multinetwork TCP implementations, where a lost packet on one network stalls the connection on all networks. QUIC is also designed specifically with mobile clients in mind, providing support for fast session suspension and resumption as networks come and go. Further, though the API design for QUIC is in flux as of this writing, the intended API for QUIC streams (distinct sequences of data analogous to streams in SCTP or IROBs in Intentional Networking) includes application-specified priorities. Though it remains to be seen how well the eventual API will allow applications to express their intent without an onerous development burden,

it is again encouraging to see high-profile actors such as Google investing in technology that aims to realize ideas similar to those articulated in this dissertation.

With regard to prefetching and background transfer in general, Microsoft has released APIs specifically tailored to background transfers [5] and prefetching [4] with Windows and Windows Phone 8. Apple provides a background data fetch API for iOS developers [2]. Google provides a great deal of advice to developers seeking to use networks efficiently for background transfers, but to the best of our knowledge, Android does not provide support via a system API. In general, on the platforms that provide system support for prefetching, it is not clear how each system decides when to invoke application prefetching, or whether the system considers and balances the cost and benefit as IMP does. As with the other examples, we look forward to seeing how these services evolve and gain adoption. It would be especially interesting in the future to see a cross-platform survey of the various support APIs and their utility in terms of performance, resource budgeting, and user/developer overhead.

6.4 Future work

We envision two primary directions for future work: improving the ease-of-use of Intentional Networking and Informed Mobile Prefetching, and improving Meatballs' consideration of aging error observations.

6.4.1 Intentional Networking

Applications must currently be modified to use Intentional Networking. As discussed in Section 2.4, these modifications have not been onerous. Nevertheless, to broaden the applicability of Intentional Networking, our future plans include providing mechanisms to disclose hints on behalf of unmodified applications.

It may be possible to identify on-demand activity by intercepting user actions and correlating them with network usage. We may even be able to classify opportunistic behavior

by observing UI updates that do (or, importantly, do not) happen together with I/O activity. Alternatively, we are planning to combine stack introspection techniques from the security community [101] with causal analysis techniques recently used to create high-performance file systems that provide strong persistence guarantees [71]. This scheme tracks user and UI behavior through the operating system, identifying the set of inputs that can possibly have influenced a set of outputs. Of course, this set is possibly too large because it tracks any relationships that might have been causal. We can prune the set via offline analysis, either by observing many executions of similar code paths and eliminating candidate causal events that only happen some of the time [59] or by using taint checking to profile causality within a process [66].

Our current implementation also requires that both ends of a network connection be modified to use Intentional Networking. When one cannot modify the server, we believe the best solution is to run a proxy in the cloud that converts Intentional Networking traffic from the client to a single TCP connection to the server. The application client can thus use Intentional Networking to manage the wireless connection, which is where the majority of benefit from network diversity is likely to be found.

6.4.2 Informed Mobile Prefetching

Like Intentional Networking, Informed Mobile Prefetching requires application modifications to gain the knowledge it needs to make decisions. However, since a large portion of the content consumed on mobile devices comes from HTTP traffic, it may be possible to integrate IMP with the cross-application HTTP cache already present on a mobile device. This could potentially transfer the responsibility of code modification to another system component rather than the application developer, allowing applications to transparently gain the benefit of HTTP prefetching and caching without modification.

6.4.3 Coping with predictor uncertainty

During our evaluation of Meatballs, we observed multiple instances in which one strategy became the clear winner—for example, the current WiFi network degrades enough that there is no longer any benefit to using it for redundant transmission. In the extreme, this can result in the other strategy never being attempted, and thus its predictors never again measured. In this situation, even if conditions change such that WiFi becomes competitive again, because Meatballs has decided not to use WiFi, it will never measure WiFi and thus will never discover that WiFi has recovered.

In the case of Intentional Networking, we mitigate the impact of this problem by resetting the WiFi error distributions each time a new access point is observed. For speech recognition, there is no similar moment at which the error distribution for server load can be reset. In both cases, a more general, principled approach is desirable. We view this as an important piece of future work.

Intuitively, an old measurement should be treated as more uncertain than a recent measurement. This added uncertainty will eventually cause the decision to tilt back towards redundancy, resulting in the lapsed strategy to be used and measured again. The implementation of this age-based uncertainty is specific to the evaluation method being used, but like other details, it can be encapsulated inside Meatballs.

Besides age-based uncertainty, it may be worth considering incorporating periodic active measurements for predictors that show lapses in passive measurements. For example, realistic mobile phone usage involves long periods of idleness in between bursts of activity. Periodic active measurement is an additional way to spend resources to purchase improved performance—though in that case, what's actually being purchased is better information, which may in turn lead to better decisions and better performance. Meatballs could track user activity and calculate the probability that a strategy will be used in the near future, and use this to gauge the importance of obtaining up-to-date measurements.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Apple seems to also believe in Multipath TCP. http://perso.uclouvain.be/olivier.bonaventure/blog/html/2013/09/18/mptcp.html.

[2] Background execution and multitasking. https://developer.apple.com/library/ios/documentation/iphone/conceptual/iphoneosprogrammingguide/ManagingYourApplicationsFlow/ManagingYourApplicationsFlow.html#//apple_ref/doc/uid/TP40007072-CH4-SW20.

[3] Experimenting with QUIC. http://blog.chromium.org/2013/06/experimenting-with-quic.html.

[4] Windows dev center - ContentPrefetcher class. http://msdn.microsoft.com/en-us/library/windows/apps/windows.networking.backgroundtransfer.contentprefetcher.aspx.

[5] Windows dev center - transferring data in the background. http://msdn.microsoft.com/en-us/library/windows/apps/xaml/Hh452975(v=win.10).aspx.

[6] ABERDEEN, D., PACOVSKY, O., AND SLATER, A. The learning behind Gmail priority inbox. In *NIPS 2010 Workshop on Learning on Cores, Clusters and Clouds* (Mount Currie, British Columbia, Canada, December 2010).

[7] AKKAYA, K., AND YOUNIS, M. A survey on routing protocols for wireless sensor networks. *Ad Hoc Networks 3*, 3 (2005), 325–349.

[8] ANAND, M., NIGHTINGALE, E. B., AND FLINN, J. Self-tuning wireless network power management. In *Proceedings of the 9th Annual Conference on Mobile Computing and Networking* (San Diego, CA, September 2003), pp. 176–189.

[9] ANAND, M., NIGHTINGALE, E. B., AND FLINN, J. Ghosts in the machine: Interfaces for better power management. In *Proceedings of the 2nd International Conference on Mobile Systems, Applications and Services* (Boston, MA, June 2004), pp. 23–35.

[10] ANANTHANARAYANAN, G., GHODSI, A., SHENKER, S., AND STOICA, I. Effective straggler mitigation: Attack of the clones. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation* (Lombard, IL, USA, April 2013), pp. 185–198.

[11] APACHE HTTPCOMPONENTS. `http://hc.apache.org/`.

[12] BAHL, P., ADYA, A., PADHYE, J., AND WOLMAN, A. Reconsidering wireless systems with multiple radios. *Computer Communication Review 34*, 5 (2004), 39–46.

[13] BALAN, R., FLINN, J., SATYANARAYANAN, M., SINNAMOHIDEEN, S., AND YANG, H.-I. The case for cyber foraging. In *the 10th ACM SIGOPS European Workshop* (Saint-Emilion, France, September 2002).

[14] BALAN, R. K., GERGLE, D., SATYANARAYANAN, M., AND HERBSLEB, J. Simplifying cyber foraging for mobile devices. In *Proceedings of the 5th International Conference on Mobile Systems, Applications and Services* (San Juan, Puerto Rico, June 2007).

[15] BALASUBRAMANIAN, A., MAHAJAN, R., AND VENKATARAMANI, A. Augmenting mobile 3G using WiFi. In *Proceedings of the 8th International Conference on Mobile Systems, Applications and Services* (San Francisco, CA, June 2010), pp. 209–221.

[16] BALASUBRAMANIAN, N., BALASUBRAMANIAN, A., AND VENKATARAMANI, A. Energy consumption in mobile phones: a measurement study and implications for network applications. In *IMC '09: Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference* (New York, NY, USA, 2009), ACM, pp. 280–293.

[17] Bay area wireless users group. `http://www.bawug.org/`.

[18] BRUNATO, M., AND SEVERINA, D. WilmaGate: A new open access gateway for hotspot management. In *Proceedings of the 3rd ACM International Workshop on Wireless Mobile Applications and Services on WLAN Hotspots (WMASH)* (Köln, Germany, September 2005), pp. 56–64.

[19] CARTER, C., KRAVETS, R., AND TOURRILHES, J. Contact networking: a localized mobility system. In *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services* (San Francisco, CA, May 2003), pp. 145–158.

[20] CHANDRA, R., AND BAHL, P. MultiNet: Connecting to multiple IEEE 802.11 networks using a single wireless card. In *Proceedings of the 23rd Annual IEEE Iternational Conference on Computer Communications* (Hong Kong, March 2004), pp. 882–893.

[21] CHEBROLU, K., RAMAN, B., AND RAO, R. R. A network layer approach to enable TCP over multiple interfaces. *Wireless Networks 11*, 5 (September 2005), 637–650.

[22] CHUN, B.-G., IHM, S., MANIATIS, P., NAIK, M., AND PATTI, A. CloneCloud: Elastic execution between mobile device and cloud. In *Proceedings of the 6th ACM European Conference on Computer Systems* (Salzburg, Austria, April 2011).

[23] CMU SPHINX. `http://cmusphinx.sourceforge.net/`.

[24] CUERVO, E., BALASUBRAMANIAN, A., KI CHO, D., WOLMAN, A., SAROIU, S., CHANDRA, R., AND BAHL, P. MAUI: Making smartphones last longer with code offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications and Services* (San Francisco, CA, June 2010), pp. 49–62.

[25] CZERWINSKI, S., ZHAO, B., HODES, T., JOSEPH, A., AND KATZ, R. An architecture for a secure service discovery service. In *Proceedings of the 5th International Conference on Mobile Computing and Networking* (Seattle, WA, August 1999), pp. 24–35.

[26] DE NITTO PERSONÈ, V., GRASSI, V., AND MORLUPI, A. Modeling and evaluation of prefetching policies for context-aware information services. In *Proceedings of the 4th International Conference on Mobile Computing and Networking* (Dallas, TX, USA, October 1998), pp. 55–65.

[27] DEAN, J., AND BARROSO, L. A. The tail at scale. *Communications of the ACM 56*, 2 (February 2013), 74–80.

[28] DRAVES, R., PADHYE, J., AND ZILL, B. Routing in multi-radio, multi-hop wireless mesh networks. In *Proceedings of the 10th International Conference on Mobile Computing and Networking* (Philadelphia, PA, September 2004), pp. 114–128.

[29] EFSTATHIOU, E. C., AND POLYZOS, G. C. A peer-to-peer approach to wireless LAN roaming. In *Proceedings of the 1st ACM International Workshop on Wireless Mobile Applications and Services on WLAN Hotspots (WMASH)* (San Diego, CA, September 2003), pp. 10–18.

[30] ERIKSSON, J., BALAKRISHNAN, H., AND MADDEN, S. Cabernet: Vehicular content delivery using WiFi. In *Proceedings of the 14th International Conference on Mobile Computing and Networking* (September 2008), pp. 199–210.

[31] FLINN, J. *Extending Mobile Computer Battery Life through Energy-Aware Adaptation*. PhD thesis, Department of Computer Science, Carnegie Mellon University, December 2001.

[32] FLINN, J., PARK, S., AND SATYANARAYANAN, M. Balancing Performance, Energy, and Quality in Pervasive Computing. In *Proceedings of the 22nd International Conference on. Distributed Computing Systems* (Vienna, Austria, July 2002).

[33] FLINN, J., AND SATYANARAYANAN, M. Managing battery lifetime with energy-aware adaptation. *ACM Transactions on Computer Systems (TOCS) 22*, 2 (May 2004), 137–179.

[34] FORD, A., RAICIU, C., HANDLEY, M., AND BONAVENTURE, O. TCP Extensions for Multipath Operation with Multiple Addresses. RFC 6824, January 2013.

[35] FRIDAY, A., DAVIES, N., WALLBANK, N., CATTERALL, E., AND PINK, S. Supporting service discovery, querying and interaction in ubiquitous computing environments. *Wireless Networks 10*, 6 (November 2004), 631–641.

[36] FUTURE (JAVA 2 PLATFORM SE 5.0). `http://docs.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/Future.html`.

[37] GARLAN, D., SIEWIOREK, D. P., SMAILAGIC, A., AND STEENKISTE, P. Project aura: Toward distraction-free pervasive computing. *IEEE Pervasive Computing 1*, 2 (2002), 22–31.

[38] GARROPPO, R., GIORDANO, S., PAGANO, M., AND PROCISSI, G. On traffic prediction for resource allocation: A Chebyshev bound based allocation scheme. *Computer Communications 31*, 16 (Oct. 2008), 3741–3751.

[39] HOWARD, J. H., KAZAR, M. L., MENEES, S. G., NICHOLS, D. A., SATYANARAYANAN, M., SIDEBOTHAM, R. N., AND WEST, M. J. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems 6*, 1 (February 1988), 51–81.

[40] HSIEH, H. Y., AND SIVAKUMAR, R. A transport layer approach for achieving aggregate bandwidths on multi-homed mobile hosts. In *Proceedings of the 8th International Conference on Mobile Computing and Networking* (Atlanta, GA, September 2002), pp. 83–94.

[41] HU, N., AND STEENKISTE, P. Estimating available bandwidth using packet pair probing. Tech. Rep. CMU-CS-02-166, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, September 2002.

[42] IYER, S., AND DRUSCHEL, P. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous i/o. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles* (Banff, Canada, October 2001), pp. 117–130.

[43] JAIN, M., AND DOVROLIS, C. Ten fallacies and pitfalls on end-to-end available bandwidth estimation. In *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement* (Taormina, Sicily, Italy, October 2004), pp. 272–277.

[44] JAISWAL, S., IANNACCONE, G., DIOT, C., KUROSE, J., AND TOWSLEY, D. Inferring TCP connection characteristics through passive measurements. In *Proceedings of the 23rd Annual IEEE Iternational Conference on Computer Communications* (Hong Kong, China, March 2004), pp. 1582–1592.

[45] JOHANSSON, P., KAPOOR, R., KAZANTZIDIS, M., AND GERLA, M. Personal area networks: Bluetooth or IEEE 802.11? *International Journal of Wireless Information Networks 9*, 2 (April 2002), 89–103.

[46] K-9 MAIL. http://code.google.com/p/k9mail/.

[47] KANDULA, S., LIN, K. C.-J., BADIRKHANLI, T., AND KATABI, D. FatVAP: Aggregating AP backhaul capacity to maximize throughput. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation* (San Francisco, CA, April 2008), pp. 89–103.

[48] KATZ, R. H., AND BREWER, E. A. The case for wireless overlay networks. In *Proceedings of SPIE Multimedia and Networking Conference (MMNC '96)* (San Jose, CA, January 1996).

[49] KIM, M., AND NOBLE, B. D. Mobile network estimation. In *Proceedings of the 7th International Conference on Mobile Computing and Networking* (July 2001), pp. 298–309.

[50] KISTLER, J. J., AND SATYANARAYANAN, M. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems 10*, 1 (February 1992).

[51] KOTZ, D., AND ELLIS, C. S. Practical prefetching techniques for parallel file systems. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems* (Miami Beach, FL, USA, December 1991), pp. 182–189.

[52] LEE, D. L., XU, J., ZHENG, B., AND LEE, W.-C. Data management in location-dependent information services. *IEEE Pervasive Computing 1*, 3 (2002), 65–72.

[53] LEE, H., KIM, S., LEE, O., CHOI, S., AND LEE, S. Available bandwidth-based association in ieee 802.11 wireless LANs. In *MSWiM '08: Proceedings of the 11th International Symposium on Modeling, Analysis, and Simulation of Wireless and Mobile Systems* (Vancouver, British Columbia, Canada, October 2008), pp. 132–139.

[54] LEE, K., LEE, J., YI, Y., RHEE, I., AND CHONG, S. Mobile data offloading: How much can wifi deliver? In *The 6th International Conference on emerging Networking Experiments and Technologies (CoNEXT)* (Philadelphia, PA, USA, December 2010), pp. 536–550.

[55] LINUX ADVANCED ROUTING AND TRAFFIC CONTROL. *http://lartc.org/*.

[56] THE LINUX FOUNDATION. *netem*. http://www.linuxfoundation.org/collaborate/workgroups/networking/netem.

[57] LIU, C., WHITE, R. W., AND DUMAIS, S. Understanding web browsing behaviors through Weibull analysis of dwell time. In *Proceedings of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval* (Geneva, Switzerland, July 2010), pp. 379–386.

[58] LIU, H., ZHANG, Y., AND ZHOU, Y. TailTheft: Leveraging the wasted time for saving energy in cellular communications. In *Proceedings of the 6th ACM International Workshop on Mobility in the Evolving Internet Architecture* (Washington, D.C., USA, June 2011), pp. 31–36.

[59] LU, S., PARK, S., HU, C., MA, X., JIANG, W., LI, Z., POPA, R. A., AND ZHOU, Y. MUVI: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles* (Stevenson, WA, October 2007).

[60] MAGALHAES, L., AND KRAVETS, R. Transport level mechanisms for bandwidth aggregation on mobile hosts. *IEEE International Conference on Network Protocols* (2001).

[61] MATSUNAGA, Y., MERINO, A. S., SUZUKI, T., AND KATZ, R. Secure authentication system for public WLAN roaming. In *Proceedings of the 1st ACM International Workshop on Wireless Mobile Applications and Services on WLAN Hotspots (WMASH)* (San Diego, CA, 2003), pp. 113–121.

[62] MIU, A. K., AND BAHL, P. Dynamic host configuration for managing mobility between public and private networks. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS)* (San Francisco, CA, March 2001), pp. 147–158.

[63] MOZILLA THUNDERBIRD. *http://www.mozillamessaging.com/en-US/thunderbird/*.

[64] MUMMERT, L., EBLING, M., AND SATYANARAYANAN, M. Exploiting weak connectivity in mobile file access. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (Copper Mountain, CO, Dec. 1995).

[65] NARAYANAN, D., FLINN, J., AND SATYANARAYANAN, M. Using history to improve mobile application adaptation. In *Proceedings of the 3rd International Workshop on Mobile Computing Systems and Applications (HotMobile)* (Monterey, CA, August 2000), pp. 30–41.

[66] NEWSOME, J., AND SONG, D. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *In Proceedings of the 12th Network and Distributed Systems Security Symposium* (February 2005).

[67] NICHOLSON, A. J., CHAWATHE, Y., CHEN, M. Y., NOBLE, B. D., AND WETHERALL, D. Improved access point selection. In *Proceedings of the 4th International Conference on Mobile Systems, Applications and Services* (Uppsala, Sweden, 2006), pp. 233–245.

[68] NICHOLSON, A. J., AND NOBLE, B. D. BreadCrumbs: Forecasting mobile connectivity. In *Proceedings of the 14th International Conference on Mobile Computing and Networking* (San Francisco, CA, September 2008), pp. 46–57.

[69] NICHOLSON, A. J., WOLCHOK, S., AND NOBLE, B. D. Juggler: Virtual networks for fun and profit. *IEEE Transactions on Mobile Computing 9*, 1 (January 2010), 31–43.

[70] NIGHTINGALE, E. B., AND FLINN, J. Energy-efficiency and storage flexibility in the Blue File System. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (San Francisco, CA, December 2004), pp. 363–378.

[71] NIGHTINGALE, E. B., VEERARAGHAVAN, K., CHEN, P. M., AND FLINN, J. Rethink the sync. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, WA, October 2006), pp. 1–14.

[72] NOBLE, B. D., SATYANARAYANAN, M., NARAYANAN, D., TILTON, J. E., FLINN, J., AND WALKER, K. R. Agile application-aware adaptation for mobility. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles* (Saint-Malo, France, October 1997), pp. 276–287.

[73] NYCWireless. `http://nycwireless.net/`.

[74] OLIVER, E., AND KESHAV, S. An empirical approach to smartphone energy level prediction. In *13th International Conference on Ubiquitous Computing* (Beijing, China, September 2011), pp. 345–354.

[75] OPENINTENTS NEWS READER. `http://www.openintents.org/en/newsreader`.

[76] PADMANABHAN, V. N., AND MOGUL, J. C. Using predictive prefetching to improve world wide web latency. *Computer Communication Review 26*, 3 (1996), 22–36.

[77] PALMER, M., AND ZDONIK, S. B. Fido: A cache that learns to fetch. In *Proceedings of the 17th International Conference on Very Large Data Bases* (Barcelona, Spain, September 1991), pp. 255–264.

[78] PATTERSON, R. H., GIBSON, G. A., GINTING, E., STODOLSKY, D., AND ZELENKA, J. Informed prefetching and caching. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles* (Copper Mountain, CO, December 1995), pp. 79–95.

[79] PEEK, D., AND FLINN, J. EnsemBlue: Integrating distributed storage and consumer electronics. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, WA, November 2006), pp. 219–232.

[80] PERKINS, C. IP mobility support for IPv4. RFC 3344, August 2002.

[81] POPA, O. Multipath TCP on mobile phones. Master's thesis, Computer Laboratory, University of Cambridge, 2010.

[82] QIAN, F., WANG, Z., GAO, Y., HUANG, J., GERBER, A., MAO, Z. M., SEN, S., AND SPATSCHECK, O. Periodic transfers in mobile applications: Network-wide origin, impact, and optimization. In *Proceedings of the 21st International World Wide Web Conference (WWW 2012)* (Lyon, France, April 2012), pp. 51–60.

[83] QIAN, F., WANG, Z., GERBER, A., MAO, Z. M., SEN, S., AND SPATSCHECK, O. TOP: Tail optimization protocol for cellular radio resource allocation. In *Proceedings of the 18th IEEE International Conference on Network Protocols* (Kyoto, Japan, October 2010), pp. 285–294.

[84] RA, M.-R., SHETH, A., MUMMERT, L., PILLAI, P., WETHERALL, D., AND GOVIDAN, R. Odessa: Enabling interactive perception applications on mobile devices. In *Proceedings of the 9th International Conference on Mobile Systems, Applications and Services* (Washington, DC, June 2011).

[85] RAHMATI, A., QIAN, A., AND ZHONG, L. Understanding human-battery interaction on mobile phones. In *Proceedings of the 9th ACM International Conference on Human Computer Interaction with Mobile Devices and Services* (Singapore, September 2007), pp. 265–272.

[86] RAICIU, C., PAASCH, C., BARRÉ, S., FORD, A., HONDA, M., DUCHÈNE, F., BONAVENTURE, O., AND HANDLEY, M. How hard can it be? designing and implementing a deployable multipath TCP. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation* (San Jose, CA, April 2012), pp. 399–412.

[87] RIBEIRO, V., RIEDI, R., BARANIUK, R., NAVRATIL, J., AND COTTRELL, L. pathChirp: Efficient available bandwidth estimation for network paths. In *4th Passive and Active Measurement Workshop (PAM 2003)* (La Jolla, California, USA, April 2003).

[88] SAKAKIBARA, H., SAITO, M., AND TOKUDA, H. Design and implementation of a socket-level bandwidth aggregation mechanism for wireless networks. In *WICON '06: Proceedings of the 2nd Annual International Workshop on Wireless Internet* (Boston, MA, 2006).

[89] SALEM, N. B., HUBAUX, J.-P., AND JAKOBSSON, M. Reputation-Based Wi-Fi Deployment Protocols and Security Analysis. In *Proceedings of the 2nd ACM International Workshop on Wireless Mobile Applications and Services on WLAN Hotspots (WMASH)* (Philadelphia, PA, October 2004), pp. 29–40.

[90] SATYANARAYANAN, M. Pervasive Computing: Vision and Challenges. *IEEE Personal Communications 8*, 4 (August 2001), 10–17.

[91] SCHULMAN, A., NAVDA, V., RAMJEE, R., SPRING, N., DESHPANDE, P., GRUNEWALD, C., JAIN, K., AND PADMANABHAN, V. N. Bartendr: A practical approach to energy-aware cellular data scheduling. In *Proceedings of the 16th*

International Conference on Mobile Computing and Networking (Chicago, IL, USA, September 2010), pp. 85–96.

[92] SeattleWireless. `http://seattlewireless.net/`.

[93] SHAH, S., AND NOBLE, B. D. A study of e-mail patterns. *Software: Practice and Experience 37*, 14 (2007), 1515–1538.

[94] SMITH, A. J. Cache memories. *ACM Computing Surveys 14*, 3 (September 1982), 473–530.

[95] SNOEREN, A., BALAKRISHNAN, H., AND KAASHOEK, F. Reconsidering Internet mobility. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS)* (Schloss Elmau, Germany, May 2001), pp. 41–46.

[96] STEMM, M., AND KATZ, R. H. Vertical handoffs in wireless overlay networks. *Mobile Networks and Applications 3*, 4 (December 1998), 335–350.

[97] STEWART, R., XIE, Q., MORNEAULT, K., SHARP, C., SCHWARZBAUER, H., TAYLOR, T., RYTINA, I., KALLA, M., ZHANG, L., AND PAXZON, V. Stream control transmission protocol. Tech. rep., IETF, June 2000.

[98] SU, Y.-Y., AND FLINN, J. Slingshot: Deploying stateful services in wireless hotspots. In *Proceedings of the 3rd International Conference on Mobile Systems, Applications and Services* (Seattle, WA, June 2005), pp. 79–92.

[99] TERRY, D. B. Caching hints in distributed systems. *IEEE Transactions on Software Engineering 13*, 1 (January 1987), 48–54.

[100] VASUDEVAN, S., PAPAGIANNAKI, K., DIOT, C., KUROSE, J., AND TOWSLEY, D. Facilitating access point selection in IEEE 802.11 wireless networks. In *Proceedings of the 5th ACM SIGCOMM Conference on Internet Measurement* (Berkeley, CA, USA, October 2005), pp. 293–298.

[101] WALLACH, D. S., BALFANZ, D., DEAN, D., AND FELTEN, E. W. Extensible security architectures for Java. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles* (Saint-Malo, France, October 1997).

[102] WATSON, T. Application design for wireless computing. In *Proceedings of the First Workshop on Mobile Computing Systems and Applications (HotMobile)* (Santa Cruz, CA, USA, December 1994), pp. 91–94.

[103] WOLSKI, R., GURUN, S., KRINTZ, C., AND NURMI, D. Using bandwidth data to make computation offloading decisions. In *2008 IEEE International Symposium on Parallel and Distributed Processing* (April 2008), pp. 1–8.

[104] ZAHARIA, M., AND KESHAV, S. Fast and optimal scheduling over multiple network interfaces. Tech. Rep. CS-2007-36, University of Waterloo, 2007.

[105] ZHANG, L., TIWANA, B., QIAN, Z., WANG, Z., DICK, R. P., MAO, Z. M., AND YANG, L. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis* (Scottsdale, AZ, USA, October 2010), pp. 105–114.

[106] ZHANG, M., LAI, J., KRISHNAMURTHY, A., PETERSON, L., AND WANG, R. A transport layer approach for improving end-to-end performance and robustness using redundant paths. In *Proceedings of the USENIX Annual Technical Conference* (Boston, MA, 2004).

[107] ZHAO, X., CASTELLUCCIA, C., AND BAKER, M. Flexible network support for mobility. In *Proceedings of the 4th International Conference on Mobile Computing and Networking* (Dallas, TX, October 1998), pp. 145–156.