Dynamic priority boosting for opportunistic mobile computing

S. Liang and S. Keshav
School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada, N2L 3G1

Abstract

Applications running on mobile terminals can benefit from exploiting transient connection opportunities: with an RTT of 2ms, even a 5s transient connection over a 50 Mbps 802.11a link represents 2500 RTTs and a chance to transfer up to 312.5 Mbytes. However, because of user mobility, such opportunistic time windows can be quite short. This paper investigates boosting CPU scheduling priorities of network-related processes on a mobile terminal to maximize benefit from communication opportunities. We explore the space of all possible boosting policies, propose two specific approaches, implement them in the Linux kernel, and study their performance. To our surprise, experimental results show that process re-prioritization improves application performance only slightly, and only under heavy load. With today's commodity hardware and commodity operating systems, both bulk transfer and interactive audio applications can be well-supported, under moderate loads, with no modifications.

1. Introduction

The purpose of most wireless links today is to merely replace a wire between a mobile terminal and a network access point. While useful, this does not fully exploit the potential of wireless technology. Many novel applications are enabled by an *opportunistic* style of communication, where mobiles opportunistically communicate with peers or a central server as they move past a wireless hotspot or a wireless hotspot moves past them. For instance, a user's email on a mobile could be handed off to a city bus as it moves past [1]. Similarly, a doctor could record patient observations into a mobile device and this information could be sent to a central server for voice recognition, indexing, and archival data storage when the doctor walks past a nurse's station. This style of opportunistic communication is the basis for a wide variety of interesting applications in areas ranging from healthcare, to environmental monitoring, to rural development.

Opportunistic communication imposes a number of stringent burdens on the network and operating system because the window of communication opportunity may be short--on the order of hundreds of milliseconds to a few minutes—and the wireless link is usually very noisy. Under these constraints, classical end-to-end communication using TCP/IP proves to be unusable [2]. In related work, we are examining architectural elements for improving the performance of opportunistic communication [4]. In this paper, we focus on a single sub-problem: that of maximizing the performance seen by communicating applications during opportunistic communication.

Specifically, we study the benefit gained by *boosting* the CPU scheduling priority of communicating processes during the time a mobile is present in a hotspot. Intuitively, this reprioritization of communicating processes over non-communicating processes would benefit opportunistic applications. This is similar to the approach taken in some of today's commercial operating systems, where processes running in windows with 'focus' are dynamically given higher priority [4]. In this paper, we study the benefits, in terms of performance improvement, of such an approach. In particular, we have implemented a simple priority boosting scheme in the Linux 2.6 kernel. We quantitatively study the effects of priority boosting for interactive audio and for bulk data transfer applications.

Our results indicate that, other than at extremely high loads, priority boosting does not significantly improve application performance, either for bulk data transfer or for interactive voice applications. This negative result shows that on current mobiles, most applications have entered the zone of abundance, where even simple scheduling disciplines suffice, because resource contention is rare.

2. Dynamic priority boosting

Dynamic priority boosting for opportunistic computing requires algorithms for four sub problems:

1. Determining that the mobile is in (or has left) a hotspot
2. Determining the set of communicating processes
3. Determining how to boost process priority when entering a hotspot
4. Determining how to reset process priority when leaving a hotspot

In this section we will outline our algorithms for each of these problems.

1. Determining that the mobile is in (or has left) a hotspot
This is a complex problem because the radio signal received by a mobile from a hotspot varies both spatially and temporally. Here, we will assume that an oracle reliably indicates to the priority boosting subsystem that a mobile has entered, or left a hotspot.

2. Determining the set of communicating processes
Every communicating process must use one or more of the following system calls: *read(), write(), select(), sendto,* and *recvfrom()*. When a mobile is not in a hotspot, the process executing one of these calls will eventually block on I/O, and be removed from the run queue. When a mobile enters a hotspot, communication resumes and the blocked process will be made schedulable again. Therefore, the set of communicating processes is simply the set of processes that are unblocked due to the transmission or receipt of a message when the mobile enters a hotspot. Our approach is to modify the kernel's implementation of the system calls noted above. We add a few lines of code to each implementation that boosts the priority of the process precisely at the point where it is unblocked and just before it is made schedulable. Note that we need to distinguish between unblocking from socket file descriptors (which boosts priority) and non-socket file descriptors (which does not). We do so simply by marking socket file descriptors returned by the *socket()* system call.

3. Determining how to boost process priority when in a hotspot
We classify priority boosting techniques into two classes: *static* and *dynamic*. With static boosting, the priority of a process is increased when the mobile is in a hotspot and the process is unblocked from a blocking communication system call. The priority is not modified over time, and process priority (potentially) changes only when the mobile leaves the hotspot. With dynamic boosting, the process priority changes over even if the mobile stays within a hotpot. There are two types of dynamic boosting: *priority incrementing,* and *priority decrementing.*

With priority decrementing, when a mobile enters a hotspot, an unblocked communicating process is initially given a priority boost (for instance, its Unix *nice* value is set to -20). Then, over time, as long as the mobile is in a hotspot, the priority monotonically decreases. The idea here is to give the maximum boosting to processes during the initial time that a mobile is in a hotspot. The longer the mobile stays in a hotspot, the less priority given to a communicating process. Over time, communicating processes are given no boost at all – they compete for the CPU just like any other process. This style of boosting is well suited to a mobile that is used both for opportunistic and traditional wireless communication. During opportunistic communication, communicating processes will get a priority boost, but when the mobile is a hot spot long enough, for example if the user is at home or in an office, there will effectively be no boosting.

In contrast, with priority incrementing, the process priority *increases* monotonically over time. The idea is that if a process was not able to finish its communicating task over some period of time, it should be given higher and higher priority, so that it can get its job done within the limited communication opportunity. This style of priority boosting is suited for mobile applications that critically require communication, so that the mobile dynamically allocates more and more resources to the communicating processes until its work is done. We envisage that future opportunistic communication applications may request one of three priority boosting styles (static, priority increasing, priority decreasing) using a system call.

With both dynamic priority policies, priority can be incremented or decremented in one of three ways:

- Every *N* events
- Every *T* time units

- Every *B* bytes of data transferred

Each of them has the obvious pros and cons. In particular, with the last scheme, when used with priority decrementing, over time, audio applications that periodically transfer small amounts of data will retain their priority far longer than bulk data transfer type applications, that transfer large amounts of data, and therefore will quickly lose their priority boost.

With all dynamic priority schemes, the maximum priority is clamped to a maximum value that is low enough that user-level processes do not interfere with essential system operations – in our work, this value is -20.

4. Determining how to reset process priority when leaving a hotspot
Once a mobile leaves a hotspot, the boosted priorities can either be reset or retained. The motivation for resetting is that the mobile is no longer in a hotspot, so a communicating process ought to give up its priority. On the other hand, the case for retaining priority is that communication processes are, in any case, going to be quickly blocked on I/O when the mobile is disconnected. So, their increased priority does no harm to other processes. When the mobile re-enters a hotspot, the remembered priority allows previous history to seamlessly influence future scheduling decisions.

These options for priority boosting are summarized in the table below:

What to do in a hotspot		*What to do on leaving a hotspot*
Statically boost priority		Reset priorities
Dynamically increase priorities, to a maximum	Per event	
	Per unit time	
	Per unit of data transferred	
Dynamically decrease priorities, starting at some initial value, to a minimum	Per event	Retain priorities
	Per unit time	
	Per unit of data transferred	

Note that just with linear priority boosting, at least fourteen separate policies are possible. These can be parametrized by (initial boost value, final boost value, slope_type (time, events, or data transferred), slope_val, reset_or_retain). If boosting is non-linear, or with a weighted combination of (time, events, data transferred), the space of possibilities further expands. To fix ideas, the figure below shows how process priorities would change over time using one of two schemes: dynamic decrease, per unit time, with priority reset; dynamic increase, per event, with priority retained.

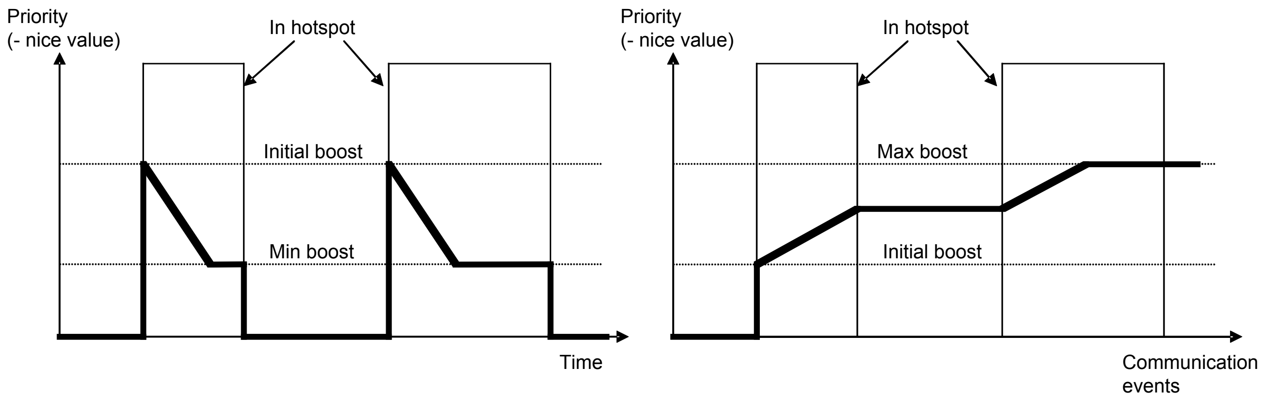

Figure 1: Two of the fourteen possible linear priority boosting policies

3. Experimental setup

We evaluated the effect of priority boosting on a server running Linux 2.6.1 that has been modified to emulate a PDA as follows:

1. We changed the clock speed and the network adaptor settings in order to mimic a mobile PDA: the CPU was clocked down from 2.5 GHz to 325 MHz and the network card was reset from 100 Mbps to 10 Mbps. This models modern PDAs that run at roughly 400 MHz and have a 11Mbps 802.11b link.
2. To model entry and exit from a hotspot, we introduced a new system call that is made by an oracle telling the system that it has entered or left the hotspot. The system call handler
    a. marks the NIC up or down by calling the netif_stop_queue and netif_start_queue procedures in the Intel Ethernet Pro 100 device driver
    b. implements the appropriate priority boosting policy
3. The oracle calls the hotspot entry and exit system calls using a simple mobility model. We model an exponentially distributed time in an hotspot, followed by an exponentially distributed time out of an hotspot. The mean of the exponential distribution is 5s.

We study the effect of priority boosting on the performance of two canonical applications. The first is a secure file transfer application (SCP) that is representative of a bulk data transfer application that also has a significant CPU load, due to the encryption of all data bytes (something that will be necessary in any real-world deployment of mobile computing). The second is an application that mimics a generic audio application transferring encrypted audio at a nominal rate of 32kbps. This application reads 80 bytes of data from the sound card every 20 ms, encrypts the data, then sends it on a UDP socket.

Besides the reference applications, we introduce competition for the mobile device's CPU with a synthetic workload process that we can control to take a specified fraction of the CPU. To model the decrease in wireless link capacity in fringe areas, we also implement a similar synthetic workload process that we can control to take up a specified fraction of the network bandwidth (thus effectively reducing the available bandwidth to the communicating process below even the 10Mbps delivered by the NIC).

For the bulk transfer application, the figure of merit is the mean and standard deviation of the time taken to transfer a file with and without priority boosting, when competing with one or more of our synthetic workload processes. For the audio application, the figure of merit is the delay jitter in the application, specifically the standard deviation in the interarrival time of outgoing packets, as measured by the kernel in the *sendto* system call. Ideally, with every outgoing packet being sent every 20ms, the standard deviation will be zero.

4. Results

We implemented and evaluated the two specific priority boosting policies shown in figure 1. The minimum priority level was a Unix nice level of -1, and the maximum priority level was a Unix nice level of -20 (in Unix lower nice levels get higher priority). For priority incrementing (INCR), we used the event based approach, with the priority incremented by 1 every event. For priority decrementing (DECR), we used the data transferred approach, with the priority decremented by 1 every 60Kbytes. Each experiment was repeated 10 times and our results show both the mean metric and the standard deviation.

The table below shows the mean and standard deviation of the time taken to transfer a 41 MByte file using SCP with 0, 1, 10, 20, and 30 competing synthetic CPU workload processes with the CPU speed set to 325 MHz and the NIC speed set to 10 Mbps.

File transfer time mean and standard deviations (in seconds) with increasing competition for CPU					
# synthetic cpu load processes	0	1	10	20	30
Original	80.8/0.63	81/0.82	81/0.82	82.4/2.55	82.7/2.54
INCR	80/0	80.1/0.32	80.1/0.32	80.1/0.32	80.2/0.42
DECR	80.4/0.52	80.4/0.52	80.8/0.79	81.2/0.92	81.7/2.26
T tests: P values of comparing INCR and DECR to original policy					
INCR	0.0031	0.0072	0.0093	<0.0001	<0.0001
DECR	0.1387	0.0652	0.5843	0.1880	0.3650

It is clear from the top half of the table that both priority boosting approaches improve performance only marginally. To confirm this hypothesis statistically, these three groups of file transfer times are compared using the standard T tests for the hypothesis that all three policies are the same, where the variance for calculating the P values is estimated by the Satterthwaite method [5]. The results show that the INCR policy is statistically different than the original policy, while the INCR policy is not. However, the absolute value of the difference is very small and negligible in practice.

We next studied the impact of network load on file transfer times, by running synthetic network workloads that took more and more of the network capacity (note that the competing network traffic was *also* priority boosted).

File transfer time mean and standard deviations (in seconds) with increasing network load						
# synthetic cpu load processes	0			1		
Cross traffic load (% of NIC capacity)	25%	50%	85%	25%	50%	85%
Original	87.5/1.18	99.2/1.48	108.3/0.82	87.4/0.70	98.4/0.84	108/0.47
INCR	85.7/1.25	96.5/0.85	107/1.15	85.3/1.06	96.9/0.57	107/0.47
DECR	87/0.67	98.3/0.48	107.8/0.79	86.8/0.79	98.1/0.74	107.4/0.52

We see that all three policies, though they differ dramatically in scope, deliver essentially the same performance. DECR is marginally better, but the improvement is slight. We conclude that boosting affords no advantage over commodity operating systems even with a heavy cross traffic network load.

Finally, we study and audio-like application that simulates a 32 kbps audio stream. The application reads 80 bytes from a sound card every 20 ms and uses UDP to send it to the destination. We are interested in the delay jitter experienced by the application, as measured in the *sendto()* system call. Our metric for delay jitter is the standard deviation of the absolute jitter (i.e. absolute deviation from 20ms interarrival spacing). The table below shows the effect of priority boosting on this application.

Standard deviation (microseconds) for an audio applications						
# synthetic cpu load processes	0			1		
Cross traffic load (% of NIC capacity)	0%	50%	85%	0%	50%	85%
Original	278	268	1066	847	672	575
INCR	896	1034	1150	1132	1147	661
DECR	433	594	1090	118	1174	549

As before, priority boosting does not seem to improve the performance of the application. In fact, it slightly degrades performance (a delay jitter increase of about 300-800 microseconds on a mean interarrival time of 20,000 microseconds, or about 1.5-4%) – this is because the

5. Conclusions and Future Work

Our results show that, surprisingly, priority boosting is marginally useful for bulk transfer applications when the competing CPU load is high (i.e. with more than 30 competing processes) and marginally worse for audio applications.

In general, resource availability can lie in one of three zones. In the zone of *scarcity* all resource scheduling algorithms perform poorly – there simply isn't enough resource to go around. In the zone of *abundance* all scheduling algorithms perform well, because there is little or no contention for resources. Intelligent scheduling algorithms, and techniques such as priority boosting, are effective only in the intermediate zone. Our results show that from the perspective of both audio and bulk transfer applications, today's PDAs and networks are sufficiently powerful that no particular care needs to be taken in scheduling access to CPU and the network. All of our priority boosting schemes perform as well as the unmodified Linux 2.6 scheduler. This is a good negative result in that unmodified PDAs of the near future should be able to support both voice calls and bulk transfer applications over a WiFi network.

Our contributions are the following:

- A precise statement of the priority boosting problem and an enumeration of all linear boosting policies
- An implementation of two priority boosting schemes in the Linux 2.6 OS, running on a server mimicking a PDA
- A strong and surprising negative result showing that, at current PDA speeds, standard Linux scheduling suffices at least for voice and bulk transfer applications

6. Acknowledgment

We would like to gratefully acknowledge our debt to Tim Brecht: our work merely expands his seminal idea of priority boosting for opportunistic communication.

7. References

1. Daknet, *http://www.daknet.org*, 2003.
2. K. Fall, "Delay Tolerant Networks for Challenged Internets," *Proc. ACM SIGCOMM 2003*, August 2003.
3. The Mindstream Project, http*://www.cs.uwaterloo.ca/~keshav/mindstream.html*, 2004
4. Inside Windows NT, 2nd edition, *Microsoft Press*, 2000.
5. John Neter, Michael Kutner, Christopher Nachtsheim, et al., "Applied Linear Statistical Models," 4th ed. pp.971-973.