ZipTx: Harnessing Partial Packets in 802.11 Networks

Kate Ching-Ju Lin
MIT/NTU

Nate Kushman
MIT

Dina Katabi
MIT

ABSTRACT

Current wireless protocols retransmit packets that fail the checksum test, even when most of the bits are correctly received. Prior work has recognized this inefficiency; however, the proposed solutions (PPR, HARQ, SOFT, etc.) require changes to the hardware and physical layer, and hence are not usable in today's WLANs and mesh networks. Further, they are tested using fixed modulation and coding schemes, whereas production 802.11 networks adapt their modulation and codes to maximize their ability to correct erroneous bits.

This paper makes two key contributions: 1) it introduces ZipTx, a software-only solution that harvests gains from using correct bits in corrupted packets with existing hardware, and 2) it characterizes the gains of partially correct packets for the entire range of operation of 802.11 networks, and in the presence of adaptive modulation and error correcting codes. We implement ZipTx as a driver extension and evaluate our implementation in both outdoor and indoor environments, showing that ZipTx significantly improves throughput.

Categories and Subject Descriptors C.2.2 [**Computer Systems Organization**]: Computer-Communications Networks

General Terms Algorithms, Design, Performance

1 Introduction

Current wireless networks operate in an all-or-nothing mode. When a received packet fails the checksum test, the entire packet is discarded and retransmitted, even if most of its bits are correct. Prior work has recognized this inefficiency and proposed a few mechanisms to exploit such partially correct packets [20, 18, 29]. The use of soft values from the physical layer is the principal technique underlying these solutions. Instead of presenting a simple 0-1 value for each bit, they require the hardware to expose a confidence value in each decoded bit. These confidence values can then be combined across multiple receptions to correct faulty bits [29], or used to direct the transmitter to only retransmit the bits that are likely to be wrong [18]. However, current WiFi hardware does not expose soft values to higher layers. Hence, any technique using soft values cannot be deployed on current production platforms, nor on production platforms available in the near future. In fact, all that we can get from the hardware are hard 0-1 values for each bit in a packet. Ideally, one would want a solution that is able to operate purely at the software layer to exploit partial packets in the absence of any information that selectively identifies correct bits.

Even if such a solution exists, would it be beneficial in actual 802.11 networks? The benefits of partial packets have been demonstrated using the GNU Radio platform, where packets have been sent at fixed modulation and code rate [18, 29]. However, production 802.11 networks, like most production networks, run autorate algorithms that adapt the coding and modulation to avoid scenarios with high packet error rates and approach the theoretical capacity of the medium [17]. It is unclear what further gains partial packets can milk in the presence of adaptive modulation and coding algorithms. Hence, there is a need to reexamine the benefits of partial packets in production 802.11 networks.

This paper presents ZipTx, a software-only solution that significantly increases wireless throughput and reduces dead spots by avoiding fate sharing of bits within a packet. ZipTx is implemented as a driver extension and hence is easy to deploy in existing WiFi networks. ZipTx turns off the checksum test and allows the hardware to pass up all packets including those that may contain bit errors. ZipTx uses error correcting codes to recover packets with low BER without knowing which bits in a packet are erroneous. However, given that error correcting codes have to add at least twice as much redundancy as the number of incorrect coding symbols, it is inefficient to use coding to recover packets with a high BER. ZipTx uses known pilot bits in each packet to identify high BER packets and recover them via retransmission instead of coding. This heterogeneous recovery strategy allows ZipTx to significantly increase throughput.

ZipTx's key feature is that it operates atop autorate, and hence presents the first work that reveals the subtle interaction between partially correct packets, and adaptive modulation and coding. We find that bit-rate adaptation fundamentally changes the gains from partial packets. Specifically, the achievable gains are vastly different in two different scenarios: typical indoor channels where SNRs are high, and nodes are stationary, and challenged environments, where SNRs are low, or nodes are mobile. In the typical stationary indoor WLAN setting, SNRs are usually high, and autorate is effective, i.e., it finds a bit-rate that exhibits a throughput close to the capacity of the channel. Here, partial packets are mostly beneficial when the channel capacity is between two bit-rate options, limiting the gains to the size of the gaps between the bit-rates available to the autorate algorithm. In contrast, in challenged outdoor environments, long multi-path delays can create high bit errors at all bit-rates [3], causing autorate to fail to find a bit-rate where most of the packets are correctly received. Harnessing partial packets in such scenarios can double or triple the throughput. Similar large gains occur in mobile scenarios, where autorate's long timescale adaptation is often unable to keep up with the fast changes in channel quality.

We have built a prototype of ZipTx as an extension to the MadWifi driver [1]. We evaluate our implementation in both indoor and outdoor scenarios. Our experiments reveal that ZipTx can effectively harness partial packets. Our key contributions are as follows:

- We present the first characterization of partial packet behavior

MobiCom'08, September 14–19, 2008, San Francisco, California, USA.

in operational 802.11 networks, which exhibit autorate, and use adaptive modulation and error correcting codes, to function across a wide range of channel conditions.
- We describe the design and implementation of ZipTx, a system that exploits partial packets to improve throughput across the entire range of SNRs, using current hardware and only software modifications.
- For typical 802.11 deployment scenarios in stationary high-SNR indoor networks, ZipTx can push the autorate algorithm to the next highest bit rate, typically producing a throughput gain of 10-20%.
- In outdoor environments which are challenged by low SNRs and long multi-path delays, ZipTx produces a median throughput gain of 2-3× and significantly alleviates dead spots.
- In mobile scenarios, where autorate may be too slow to track the optimal bit rate, ZipTx produces a median throughput gain of 2-3×.
- ZipTx is practical. It can operate a wireless link at the maximum rate of 54 Mb/s, while consuming only a few percent of the CPU.

2 Related Work

Recent papers have noted that wireless nodes retransmit a packet because of a few bit errors, ignoring that most of the bits have been correctly received. They address this inefficiency by either changing the physical layer or its interface to higher layers [18, 29, 19, 22, 8, 20]. Specifically, PPR [18] takes advantage of "soft information", a confidence value typically computed by the physical layer on its bit decoding decisions. It uses the soft information to find incorrect chunks in a packet and retransmit only those chunks. SOFT [29] also makes use of soft information, but instead exploits wireless spatial diversity to address the problem. Access points that hear the same transmission communicate over the wired Ethernet and combine their soft information to correct faulty bits in a corrupted packet. With Chase combining [8, 14], a receiver stores soft-information from corrupted packets and recovers from packet corruption by combining soft information across a packet and its retransmissions. Hybrid-ARQ (HARQ) with incremental redundancy is used for high speed down link packet access (HSDPA) in the 3GPP cellular networks [10]. Instead of retransmitting a corrupted packet, it transmits a new coded version of the packet.

ZipTx builds on this foundation but differs from it in two main ways. First, it is a pure software patch that works with existing hardware and hence delivers immediate benefits to the large population of 802.11 users. Second, while the above schemes are evaluated either via simulation [13, 14] or on USRPs using fixed modulations and coding [29, 18], ZipTx experiments with actual wireless cards and addresses the reality of an operational wireless channel with a variety of modulation schemes, underlying forward error correcting codes (FEC), and automatic bit-rate selection.

There is also a rich literature that uses coding to improve wireless throughput. Most wireless technologies use forward error correcting codes at the link layer as well as error detection codes like CRC [16, 9]. In [15], the authors increase the throughput of sensor networks by dividing a packet into multiple segments, each with its own CRC; instead of retransmitting an entire packet because of a few erroneous bits, the sender retransmits only the corrupted segments. The authors of [23] and [12] reduce wireless losses by combining multiple corrupted receptions of the same packet using majority voting or incremental CRC. Prior work also employs network coding to improve the throughput and reliability of mesh networks [7]. ZipTx usage of coding however differs from the above work since it employs an outer code that is implemented in the driver and operates on top of the 802.11 existing link-layer convolutional codes. Furthermore, in contrast to prior work which assumes fixed modulation and coding, ZipTx characterizes the gains of partially correct packets for the entire range of operation of 802.11 networks, and in the presence of adaptive modulation and error correcting codes.

Finally, our work is also related to past work on characterizing wireless errors. The communications field has spend significant effort characterizing errors on the wireless medium and their relation to SNR [28], whereas the networking community has looked at the characteristics of the link layer, considering only fully correct packets [11]. In contrast, this paper is the first to show experimental results characterizing errors in partially correct 802.11 packets, as perceived by the driver after the failure of the FEC in the PHY layer, and in the presence of rate adaptation.

3 Experimental Environment

We describe our experimental environment, which is used in later sections for both characterizing the 802.11 channel and evaluating ZipTx, a driver that exploits partial packets to improve throughput.

(a) Hardware All measurements are taken on 3.0 Ghz Intel Core2 Duo machines running Linux FC 6, and equipped with either a Netgear WAG311 card or DLink DWL-AG530 card, both of which are based on an Atheros chipset and work with the Madwifi driver. Some of our results are reported as a function of the RSSI, which on hardware using the Atheros chipset, is defined as the SNR in dB.

(b) Driver Configuration We run all experiments in the Madwifi monitor mode because this mode can be configured to deliver to the driver all packets received by the hardware including those that failed the CRC check. However, the interference mitigation option in the monitor mode is known to have a bug that reduces receiver sensitivity [2], and hence we turned this option off in our experiments.

(c) Indoor vs. Outdoor Scenarios Indoor scenarios include a set of 35 sender-receiver node pairs set up in various locations throughout our lab. Each run involved transmitting UDP packets between a node pair for one minute. These runs used 802.11a in order to avoid being affected by the existing 802.11g network in our lab. Outdoor scenarios include a set of 26 node pairs. For each pair, one node has an antenna attached to our building, placed outside the second story window. The second node is at ground level somewhere on the campus. These links were generally unable to receive packets at the higher rates of 802.11a or 802.11g, so all outdoor measurements were taken using 802.11b. Again, each run is one minute long. The use of multiple 802.11 modes allows us to illuminate crucial differences between them, as discussed in §4.

(d) Measurement Methodology For each sender-receiver pair, in the indoor and outdoor environments, we collect multiple traces. We want to use these traces to estimate the gains of partial packets when using the optimal bit rate for each channel. The problem, however, is that one cannot find the optimal bit-rate for a channel, unless one tries all bit-rates and compares their throughput. Thus, for each channel, the sender transmits 50 packets at each rate and repeats the cycle to collect a one-minute trace. The one-minute trace is then divided into a sequence of short traces each of them contains a full cycle across all rates. We then feed these short traces to an idealized autorate algorithm that identifies in each short trace the bit-rate that results in the maximum throughput. Operating over a sequence of short traces allows us to consider scenarios where the optimal bit rate changes over time for the same sender-receiver pair.

Finally, we note that we use offline processing of traces only to characterize the 802.11 channel. Our evaluation of ZipTx is done by

Term	Definition
Correct-byte throughput	Throughput measured over all correctly received bytes including those in erroneous packets
Correct-packet throughput	Throughput measured over fully-correct packets
Max-byte bit-rate	Bit-rate that maximizes the correct-byte throughput
Max-packet bit-rate	Bit-rate that maximizes the correct-packet throughput

Figure 1: Terms used in this paper

running the actual driver on the various channels in our indoor and outdoor environments.

4 Partial Packets in Production 802.11 Channels

802.11, like most modern wireless technologies (e.g., WiMax, cellular), is designed with multiple modulation and coding schemes to allow it to provide the highest possible throughput for a given underlying channel quality. An autorate algorithm typically runs in the driver and decides which of the available modulation and coding schemes to use at any given time. Such adaptation was not considered in prior works [18, 29, 15, 12]. In contrast, here we focus on the benefits of partial packets in 802.11 production networks, where autorate is a critical part of maintaining high throughput.

The goal of autorate is to select from a set of available bit-rates the bit-rate that maximizes the channel's throughput. It is important to note that the optimal bit rate may differ depending on whether the nodes can exploit correct bytes in partial packets. In particular, if the nodes cannot utilize partial packets, they will typically maximize their throughput by using a bit-rate at which the vast majority of the packets are received fully correct. In contrast, if the nodes can utilize correct bytes in partial packets, they may be able to increase their throughput by jumping to a higher bit-rate at which many packets contain errors. In this paper, we define the *correct-byte throughput* as the channel throughput measured over all correctly received bytes including those in erroneous packets, and the *correct-packet throughput* as the channel throughput measured over fully-correct packets. We also define the *max-byte bit-rate* as the bit-rate that maximizes the correct-byte throughput, and the *max-packet bit-rate* as the bit-rate that maximizes the correct-packet throughput.[1] Figure 1 lists these definitions.

Next, we investigate the gains from harnessing partial packets in both typical and challenged 802.11 channels.

4.1 Typical 802.11 Channels

Typical 802.11 production networks are indoor WLANs, where most nodes are stationary, and dense access point deployment ensures relatively high SNRs. Such SNRs allow 802.11 to operate in the *g* or *a* mode and pick from a large variety of available bit-rates, as shown in Figure 2.[2] This fine granularity of bit-rate selection enables the autorate algorithm to find a bit-rate that accurately matches the ideal rate sustainable by the channel.

[1] We focus on correct bytes rather than bits. Looking at correct bit throughput can be misleading because even a completely random packet will have 50% correct bits on average. We instead consider the correct bytes which more accurately reflect the information available from partial packets.

[2] The table ignores the 9Mb/s bit-rate specified by the standard because we find, as did past work [4], that it is never the optimal bit-rate choice and hence never chosen by the autorate algorithm.

Bit-Rate (Mb/s)	Max Throughput (Mb/s)	Relative Gap
6	5.7	n/a
12	10.9	2x
18	15.8	1.5x
24	20.0	1.25x
36	27.9	1.5x
48	34.3	1.25x
54	37.5	1.12x

Figure 2: 802.11a/g Autorate options: The table shows the available bit-rates as well as the resulting maximum throughput. The difference between the maximum throughput and the corresponding bit-rate comes from the practical overheads of packet headers and channel acquisition. It also shows the relative increase in bit-rate (and throughput) as autorate jumps from one bit rate to the next.

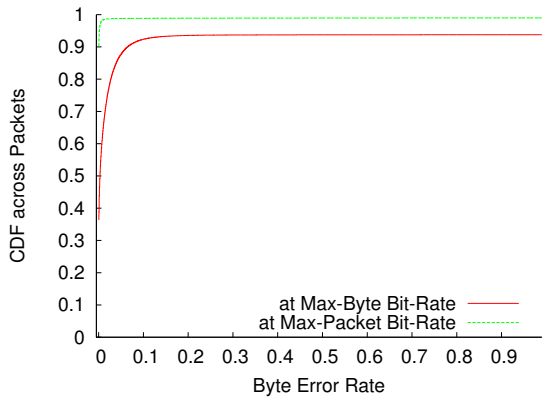


Figure 3: CDF of byte error rates in typical 802.11 channels: The figure plots the byte error rate for when the autorate algorithm maximizes the correct-packet throughput, as in today's networks, and when it maximizes the correct-byte throughput.

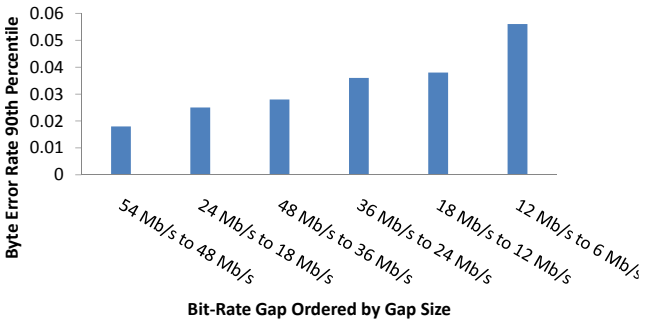


Figure 4: Byte error rates in partial packets increase with the size of the gap between the used bit-rate and the one below it.

(a) How often do we see partial packets in typical 802.11 channels, and how many errors do they have? The answer to this question depends, not only on the channel, but also on the bit-rate chosen by the autorate algorithm. However, the chosen bit-rate itself depends on whether the receiver can utilize partial packets. If it cannot it will operate at the max-packet bit-rate, whereas if can, it will instead operate at the max-byte bit-rate often choosing rates with a large fraction of partial packets.

Figure 3 plots the CDF of byte error rate taken over all packets, both for the case where the receiver cannot exploit partial packets and hence runs at the max-packet bit-rate, and when it can exploit partial packets and hence runs at the max-byte bit-rate. The figure shows that whether the receiver observes partial packets depends on whether it can harness them. Specifically, if the receiver cannot use partial packets, it will select a bit-rate, where the vast majority of the packets are received fully correct, and will hardly see any partial packets; whereas if it can utilize partial packets, it will push for a higher bit-rate, often ending up with a significant fraction of its packets being

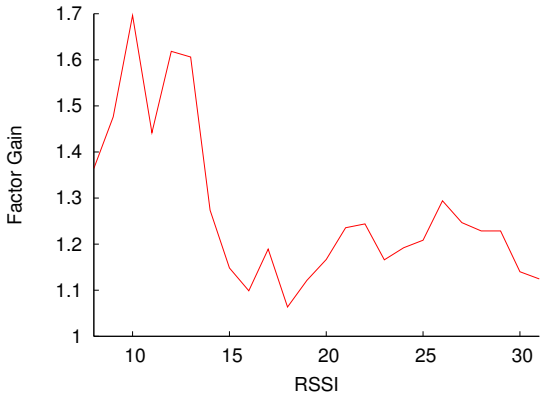


Figure 5: Ideal gains from partial packets in typical indoor channels: The figure plots the ratio of the maximum correct-byte throughput to that of the maximum correct-packet throughput as a function of the RSSI on the channel.

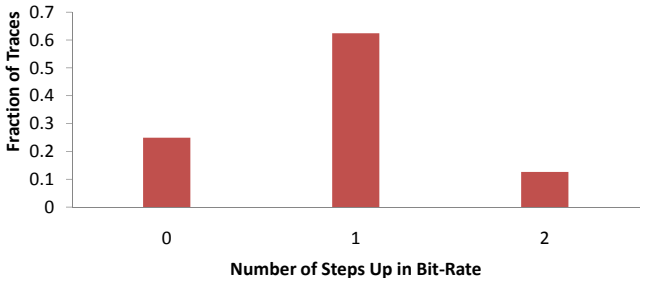


Figure 6: Difference between the max-byte bit-rate and the max-packet bit-rate in typical channels. The figure shows that in 70% of the traces the max-byte bit-rate is one step higher than the max-packet bit-rate, indicating that in most of the indoor scenarios the gains of partial packets arise from autorate jumping one bit-rate higher, and hence are limited by the gap between adjacent bit-rates.

partially correct. Specifically, in this scenario, the figure shows that 35% of the packets are fully correct, 55% are partially correct, and about 10% are erasures, i.e., not captured at all.

The figure also reveals that, in typical 802.11 channels, partial packets have low byte error rates, smaller than 5%. Hence, all one needs to recover these partial packets is to correct a few erroneous bytes. This low byte error rate can be attributed to the autorate algorithm. Specifically, if at the current rate, partial packets have too many erroneous bytes, this will reduce the correct-byte throughput at that rate, causing the autorate algorithm to move to a lower bit-rate with fewer byte errors. Furthermore, the byte error rate is typically lower if the difference between the current rate and the one below it is small. For example, the 54 Mb/s bit-rate experiences very low byte error rates because the relative drop in rate as we move down to 48 Mb/s is fairly small. Our empirical results confirm this argument. Figure 4 shows how the 90^{th} percentile error rate changes with the size of the gap between the current rate and the one below it. The figure clearly shows that as the gap increases the error rate in partial packets increases as well. Thus, in typical indoor channels, where autorate can choose from a fine-grained selection of bit-rates, partial packets have only a few percent byte errors.

(b) So, how much gain can we obtain from harnessing partial packets in typical 802.11 channels? We answer this question empirically using the traces from our indoor environments. For each trace we compute two values, the first is the correct-byte throughput assuming the autorate algorithm maximizes correct bytes, and the second is the correct-packet throughput assuming the autorate algorithm maximizes correct packets. We then compute the ratio of these two values, which provides an upper bound on the ideal gain one can hope to achieve from harnessing partial packets in typical indoor channels. Figure 5 plots this ideal gain as a function of the average RSSI in a trace. The figure shows that the gain from partial packets in typical indoor channels varies between 10% and 70%. While such a throughput improvement is significant, it is smaller than the gains reported by prior work [18, 29, 15, 12, 20], showing that the results from non-adaptive channels do not directly apply to channels with adaptive rate selection.

Bit-Rate (Mb/s)	Max Throughput (Mb/s)	Relative Gap
1	0.93	n/a
2	1.77	2x
5.5	4.06	2.7x
11	6.45	2x

Figure 7: 802.11b Autorate options: The table shows the available bit-rates as well as the resulting maximum throughput. It also shows that in 802.11b the relative increase in bit-rate (and throughput) as autorate jumps from one bit rate to the next is at least 2x, i.e., significantly higher than in the 802.11a/g modes.

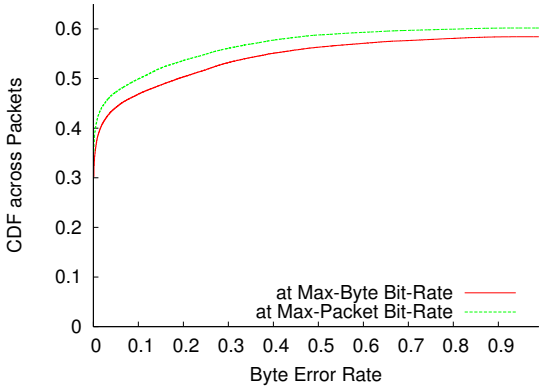


Figure 8: CDF of byte error rates in challenged outdoor channels: The figure plots the byte error rate for when the autorate algorithm maximizes the correct-packet throughput, as in today's networks, and when it maximizes the correct-byte throughput. In contrast to the indoor channels, both bit-rates have many partial packets. Note also that more than 40% of the packets are erasures.

Let us try to understand the origins of the gains in typical indoor channels and why they differ from those observed on channels without rate adaptation. Recall that we have shown that typical indoor 802.11 channels operating at the max-packet bit-rate, as they do today, hardly experience any partial packets. Thus, the only way to harness any gain from partial packet is to push to higher bit rates. Our measurements show that in most cases, the ability to harness partial packets allows the autorate algorithm to push the bit-rate one step higher. Specifically, Figure 6 plots the number of steps between the max-byte bit-rate and the max-packet bit-rate over all indoor traces (e.g., there is one step between adjacent bit rates such as 48 Mb/s and 54Mb/s). It shows that in 70% of the cases the max-byte bit-rate is just one step higher. Thus, the gains from partial packets in these networks are upper bounded by the size of the gap between adjacent bit rates. Hence, they can be significantly smaller than those in non-adaptive channels, where there is no such bound. Furthermore, since the gap between adjacent bit-rates is higher at lower bit-rates, i.e., at lower RSSIs, the gains are also higher in that range.

4.2 Challenged 802.11 Channels

While 802.11 is most commonly deployed as stationary indoor WLANs, the widespread availability of cheap hardware has motivated its deployment in a number of additional, more challenged environments, including outdoor mesh networks [25], rural wireless

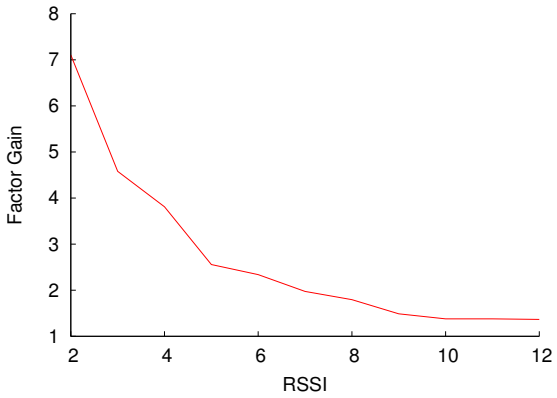


Figure 9: Ideal gains from partial packets in challenged outdoor channels: The figure plots the relative throughput gain from harnessing partial packets as a function of the RSSI. The gains are significantly larger than in typical indoor channels.

networks [27], vehicular networks [6], and mobile environments [5]. Bit-rate adaptation is less effective in such channels. Specifically, in the outdoor scenarios, the SNR is typically low, requiring the use of 802.11b, which offers the coarse selection of bit-rates shown in Figure 7. This reduces the ability of autorate to find a bit-rate that accurately matches the quality of the underlying channel and eliminate bit errors. Furthermore, long multi-path delays in outdoor channels can create high errors rates at all bit-rates [3], causing autorate to fail to find a bit-rate where most of the packets are correctly received. Similarly, autorate often struggles in mobile scenarios because it is unable to keep up with the fast changes in channel quality. In this section, we focus on stationary outdoor scenarios, while in §7.3, we show that mobile indoor scenarios exhibit similar behavior.

(a) How often do we see partial packets in challenged outdoor channels, and how many errors do they have? Similarly to the indoor case, we answer this question by plotting the byte error rate in transmitted packets, both for the case when autorate maximizes the correct-packet throughput and when it maximizes the correct-byte throughput. Figure 8 shows two interesting differences from the typical indoor scenario in Figure 3. First, the byte error rate is strikingly similar between the max-byte bit-rate and the max-packet bit-rate, and more importantly the max-packet bit-rate exhibits a large fraction of partial packets. Indeed more than 40% of all received packets are partial packets. This indicates that challenged channel can obtain large gains from harnessing partial packets because almost half of the packets they receive in today's networks are partial packets. Furthermore, since such channels can improve throughput without moving a step up in bit-rate, their gains are not as limited by the gap between adjacent channels.

Second, the figure shows that partial packets in challenged channels have much higher byte error rates than in typical indoor channels. Specifically, while in the indoor scenarios, almost all partial packets have less than a 5% byte error rate, only half of the partial packets in our outdoor channels have less than 5% of their bytes in errors, and many have error rates as high as 50-100%. In practice, there is an overhead associated with recovering erroneous bytes. For example, the receiver may have to inform the sender of the location of such errors [18]. This overhead increases with increased byte error rates. Hence, though challenged channels could offer large gains, collecting such gains comes at the price of higher practical overheads.

(b) So, how much gain can we obtain from harnessing partial packets in challenged outdoor channels? Figure 9 plots the ideal gain from partial packets as a function of RSSI. The ideal gain is computed as the ratio of the correct-byte throughput assuming the autorate algorithm maximizes correct bytes, to the correct-packet throughput assuming the autorate algorithm maximizes correct packets. The figure shows that the potential gain from partial packets in challenged channels could be as high as 7x. further, the gains are higher at lower RSSIs because partial packets account for a higher fraction of the received packets.

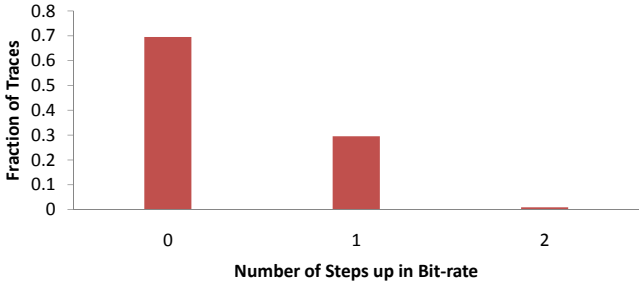


Figure 10: Difference between the max-byte bit-rate and the max-packet bit-rate in challenged outdoor channels. The figure shows that in about 70% of the traces the max-byte bit-rate is the same as the max-packet bit-rate, indicating that in most of the outdoor scenarios the gains of partial packets arise from keeping the same bit-rate that one would use to maximize the throughput of correct packets but fixing the partial packets on it.

Figure 10 gives a deeper insight about the origin of these gains. It shows that in 70% of the outdoor traces, the rate that maximizes the correct-byte throughput is the same as the rate that maximizes the correct-packet throughput. Hence, in most outdoor scenarios, collecting the gains of partial packets does not require autorate to jump to higher rates. The gains rather arise from harnessing the partial packets that naturally occur even on the rate that maximizes the throughput from fully correct packets.

5 Harnessing Partial Packets

Now that we have shown the potential gains from taking advantage of the correct bytes in partial packets, what software techniques can we use to harness these gains? As we have seen earlier, every channel has to deal with three types of packets: correct, erasure, and partially correct. Regardless of the protocol details, we believe that any technique to harness partial packets should adhere to the following guidelines on how to handle each type of packets:

- *Correct Packets:* Figures 3 and 8 show that even channels optimized for correct-byte throughput still see more than 30% fully correct packets. Thus, we would like a technique that does not introduce any extra overhead to correct packets.
- *Erasure Packets:* When a packet is lost completely and not captured by the hardware we cannot take advantage of any partial packet information. In this case, the optimal mechanism is to retransmit the packet. Any scheme that tries to do anything else will be wasting throughput.
- *Partially Correct Packets:* Since we are using off the shelf hardware, we cannot modify the physical layer in order to get additional information from it. Without information from the PHY about which bits are incorrect, there are really only two high-level approaches to harnessing partial packets: error detection typically done using CRCs, and error correction typically done using error correcting codes. Below we explain these two options in detail and compare their ability to harness partially correct packets.

5.0.1 Exploiting Partial Packets Using Per Block CRCs

The simplest technique for taking advantage of partial packets is to determine which parts of the packet are in error and retransmit only those parts instead of retransmitting the entire packet. One way to do this is for the sender to divide the packet into smaller blocks and compute a separate CRC for each of these blocks. Upon receiving a partial packet, the receiver recomputes the CRC for each block and any block that does not pass the CRC test must be retransmitted.

But what block size maximizes throughput? Choose too small a block size, and the overhead of sending the per block CRCs themselves negates the gain of partial packets. Choose too large a block size and most blocks will have incorrect bytes requiring retransmission of almost all blocks in a partially correct packet. We determine the optimal block size using empirical measurements. Figure 11 shows for various block sizes the difference in throughput between a per block CRC scheme and a scheme that uses only correct packets. The results are for a CRC size of 4 bytes, the same as the 802.11 packet-level CRC, and are computed ignoring the overhead of the receiver's communication of which blocks are in error. The results show that the optimal block size is 64, so we use this block size for our evaluation of the per block CRC scheme.

How effective is a CRC based scheme in harnessing the gains of partial packets? Clearly, any practical realization of such a scheme would incur overheads resulting from the need for receiver feedback and the associated communication delay. We consider two possible realizations. The first is an idealized per block CRC scheme that assumes infinite receiver feedback, i.e., it assumes the receiver will keep updating the sender about which blocks have errors until all blocks are correctly received, and that such feedback has no overhead and incurs no delay. The second is a more practical scheme that limits itself to two rounds of feedback. In this scheme the sender transmits all native data packets as normal. If the receiver fails to recover a packet, it uses a first round of communication to request the per-block CRCs, and a second round of communication to request the retransmission of only those blocks that fail the CRC check. If the packet is still undeliverable after the erroneous blocks have been retransmitted, it is considered as a loss.

Figure 12 plots the gain of harnessing partial packets with the above CRC-based schemes. We compute the gain as the ratio of the throughput delivered by the scheme to the throughput delivered using today's methods which drop all partial packets. The figure compares these gains to the ideal gains discussed in §4.1(b). The results are from the indoor traces and they are similar in nature to those from the outdoor traces. The figure shows that the two-round CRC scheme can garner about 50% of the ideal gains of partial packets, while the idealized CRC scheme captures slight more than 60% of these gains. The reason these schemes fail to collect all of the ideal gains is the natural trade-off between using a small block size but incurring a large overhead from sending the per-block CRCs, versus using a large block size but increasing fate-sharing among the bytes in partial packets.

5.0.2 Exploiting Partial Packets Using Coding

We have seen that simply detecting errors and retransmitting erroneous sections of the packet allow us to obtain 50% to 60% of the ideal gains available from partial packets. Can we do better? Without physical layer information, the only other way we can envision to take advantage of partial packets is through the use of error correcting codes. At the most basic level, error correction involves sending parity bits (i.e., redundancy) in addition to the original message, such that even in the face of some corruption in the transmission, the

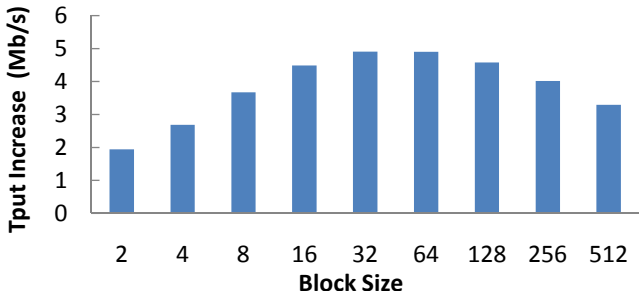


Figure 11: Throughput gain of the per block CRC scheme across block sizes: The figure plots for various block sizes the difference in throughput between a per block CRC scheme and a scheme that uses only correct packets, averaged over all indoor traces. A block size of 64 provides the best choice.

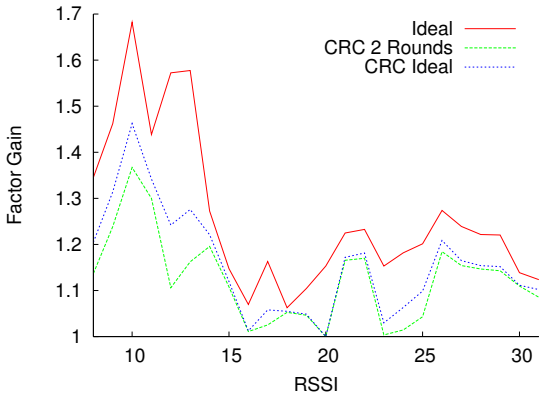


Figure 12: Effectiveness of the per block CRC schemes in harnessing partial packets: The figure shows that an idealized per block CRC scheme captures about 60% of the ideal gains available from partial packets, whereas a more realistic 2-round CRC-based scheme obtains 50% of these gains.

receiver is able to figure out the original message. Error correcting codes are a natural fit for scenarios where the receiver does not have access to physical layer information, because the receiver does not need to know which individual bytes are erroneous. As long as it has an estimate of the byte error rate, it can request sufficient redundancy to correct all the errors.

One may wonder, given that 802.11 already uses convolutional codes at the physical layer (PHY), whether there is any benefit from having an additional code at the higher layer. There are two problems with the current PHY-layer FEC. The first is that they are short convolution codes which cannot handle long error bursts. The second is that the added redundancy must be chosen before the packet is sent independently of how many errors occur in the packet's transmission. Error correcting codes at a higher layer can address both of these limitations. In fact, they work as outer codes on top of the inner convolution codes provided by the PHY. It is widely known [24] that the concatenation of outer and inner codes is useful for combating long error bursts. Further, working at a higher layer allows the transmitter to apportion the redundancy based on receiver feedback instead of making the decision before the packet is sent.

(a) Which Error Correcting Code Should We Use? Coding for error correcting is a vast field, with many trade-offs between the various coding options. We believe however that for our problem, we need a code that satisfies the following:

- *(a) Systematic Code:* Many codes turn the original data into some garbled set of bits, which is meaningless without decoding. Systematic codes, on the other hand, contain a copy of the original data

Symbol Size	Block Size	SER/BER	Worst SER / Average SER
4	15	1.11	35.1
6	63	1.21	7.4
8	255	1.25	2.5
11	2047	1.46	1.0
16	65535	1.67	1.0

Figure 13: Trade-off associated with the choice of a symbol size: Decreasing the symbol size decreases the average symbol error rate (SER) for a given bit error rate (BER), but increases the symbol error rate in the worst block in a packet.

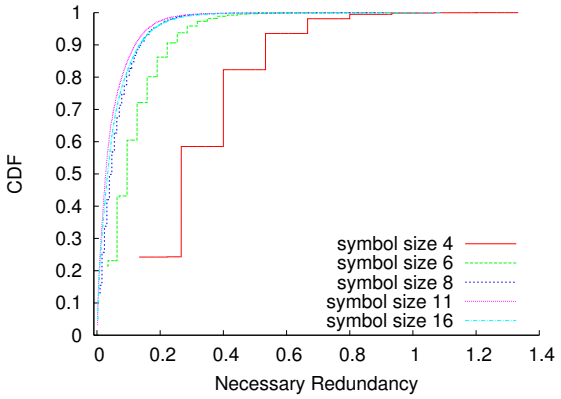


Figure 14: Percent of redundancy needed to correct a partial packet: This figure shows the CDF of the redundancy required to correct the worst block in a packet for different symbol sizes, i.e., the redundancy required to fully correct the packet. The figure shows that a symbol size of 11 bits minimizes the necessary redundancy, but also shows that any choice above 8 bits is effectively equally good.

as the first bits in the codeword. This allows us to send the native data with no parity information in the first transmission, and avoid any error correction overhead if no errors are introduced. Avoiding this overhead is important, because as we have seen in §4.1(a), a significant fraction of packets are received correctly.

- *(b) Incremental Redundancy:* Existing error correcting codes at the physical layer have course granularity, i.e., the auto-rate can choose from only a few bit-rates. This results in a loss of efficiency when the channel is between two rates. Additionally, these FEC codes choose the amount of redundancy before the packet is transmitted, hence all packets incur the same overhead regardless of how many errors they actually have. In contrast, we would like our code to support incremental redundancy, allowing matching of the redundancy to the channel quality at a fine granularity, and sending less redundancy for packets with fewer errors.
- *(c) Efficient Implementation:* Error correcting codes are usually implemented in hardware since the Galois fields typically used in their implementation are far more efficient in specialized hardware than in software implementations on top of general purpose computers. Thus it is important to choose a code that we can efficiently implement in software.

The preceding three principles lead us to choose the Reed-Solomon family of codes, as they are systematic codes that support incremental redundancy, and due to their popularity, have many high performance software implementations.

(b) Setting the parameters of Reed-Solomon: Reed Solomon codes are implemented by breaking the packet into symbols of s bits and then into chunks of c symbols. Each chunk is then coded into a block of size $b = c + r$ where r is the number of redundancy symbols in each block. Blocks are correctly decodable if less than $r/2$ symbols are in error.

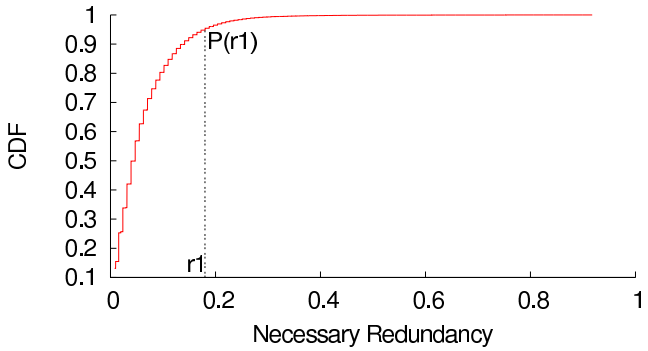
Hence Reed-Solomon codes have three parameters: the number of redundancy symbols, r, the symbol size, s, and the block size, b. How should we set these parameters? A block with e errors requires at least $2e$ redundancy symbols to be correctly decoded. However, when a packet does not pass the checksum test, the receiver only knows that at least one block has at least one error; it does not know how many errors there are, or in which blocks they lie. Thus, the transmitter cannot do better than sending the same amount of redundancy for each block. Since the packet can be delivered to higher layers only when all blocks are correct, one should pick the redundancy, r, to accommodate the observable error rate in the worst block in a packet.

How about the choice of symbol and block size? The choice of symbol size implicitly dictates a block size choice of 2^s [26]. Smaller symbols are desirable because a single incorrect bit in the symbol makes the whole symbol incorrect. However, smaller symbols lead to smaller blocks and hence many blocks per packet. Since the driver delivers only full packets to higher layer, it requires all blocks in the packet to be correct, an event whose probability decreases with the number of blocks in the packet. Hence there is a tension between large and small symbols. In Figure 13, we use measurements from the indoor channels to illustrates this tension for various symbol sizes. The figure shows that, as the symbol size increases the ratio of symbol errors to bit errors also increases. Hence, for the same underlying set of bit errors a higher fraction of symbols will be in error as the symbol size increases, arguing for choosing a smaller symbol size. On the other hand, increasing the symbol size increases the block size and and thus decreases the number of blocks per packet, reducing the symbol error rate in the worst block down closer to the average symbol error rate. Since we must pick the redundancy to accommodate the worst block, this argues for a larger block size and hence a larger symbol size. While Figure 13 illustrates the tension, the actual numbers show that the effect of the change in worst block to average block is much larger than the effect of the change in the ratio of symbol error rate to bit error rate, and hence larger symbols are likely to work better.

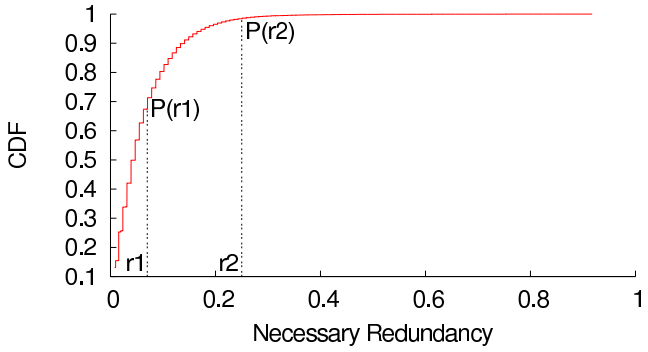
So, what is the best symbol size? The best symbol size is the one that requires the minimum amount of redundancy to recover the same fraction of packets and hence maximizes throughput. Figure 14 plots the CDF of the required redundancy to correct the packets in the indoor traces. The same analysis applies to the outdoor traces. Note that, as discussed earlier, the required redundancy is equal to twice the symbol error rate in the worst block. The figure shows that a choice of symbol size of 8 or higher is equally good with 11 being the optimal choice. Though 11 is slightly better, software implementations on modern day byte oriented computers are far simpler and more efficient when byte boundaries are conserved, leading us to choose a symbol size of 8.

(c) Coding in Rounds We said earlier that we need enough redundancy to correct the worst block in the packet. But how much is enough? The answer depends on how we send the redundancy. In its simplest form, we could send all the redundancy in one shot, i.e., if the transmitter does not receive an 802.11 synchronous ack, it sends all available redundancy for that packet. For such a one-round scheme, the only question is: how much total redundancy to send?

The optimal amount of redundancy to send should maximize the ratio of what gets delivered to higher layers to how much we transmit. We can compute this ratio for indoor scenarios from Figure 15(a) which uses the indoor traces to plot the total redundancy required to correctly deliver a packet to the higher layers, r as a CDF, $p(r)$. Again, a similar analysis can be done for the outdoor scenarios using the corresponding empirical distribution. The figure shows that if



(a) One-Round Scheme



(b) Two-Round Scheme

Figure 15: Sending redundancy in rounds: The top graph shows that if all redundancy is sent in one round, the transmitter incurs the same overhead for all partial packets. The bottom graph shows that if redundancy is sent in two rounds, the transmitter incurs only r_1 on low error packets and the savings allows it to send even more redundancy in the second round to correct more partial packets than if it used one round.

you transmit redundancy, r_1, in addition to the native packet then you can deliver $p(r_1)$ fraction of packets. Thus, we want to choose the r_1 that maximizes

$$\max_r \frac{p(r_1)}{1+r_1} \quad (1)$$

We solve this optimization numerically for the empirical CDF of the amount of redundancy required to recover a partial packet using a symbol size of 8 bits, i.e., the graph in Figure 15(a). Our solution shows that if one was to send all redundancy in one round, then the optimal amount of redundancy to send is 18% of the block size. This does not allow us to recover all partial packets. Further, it requires us to send much more redundancy than necessary for many packets. One can address these problems by increasing the number of rounds, although such a scenario will increase the delay.

In particular, let us compute the performance of a scheme that sends the redundancy over two rounds instead of one, as in Figure 15(b). In this scenario the sender will transmit a smaller amount of redundancy in the first round, r_1 trying to recover as many packets as possible with a low overhead. It then waits to hear from the receiver about which packets are still undecodable, and transmits additional redundancy, for only those undecodable packets such that the sum of the redundancy for those packets over round one and two becomes r_2. Similarly to above, we can find optimal values for r_1 and r_2 by maximizing the ratio of what is delivered to what is sent.[3] The solution to the optimization reveals that r_1 =7%, while r_2 =25%.

[3] A similar argument to the one-round case shows that the actual maximization in the two-round case is:

$$\max_{r_1,r_2} \frac{p(r_2)}{1+r_1+(r_2-r_1)(1-p(r_1))},$$

which we also solve numerically.

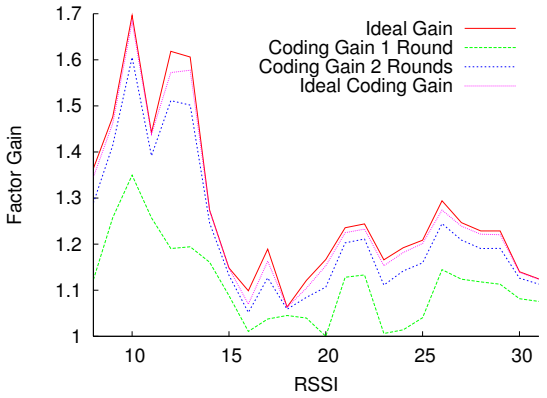


Figure 16: Efficacy of a coding scheme at harnessing partial packets: This shows that a coding-based scheme can harness most of the potential gain of partial packets. Further, it can do so with just two rounds of redundancy.

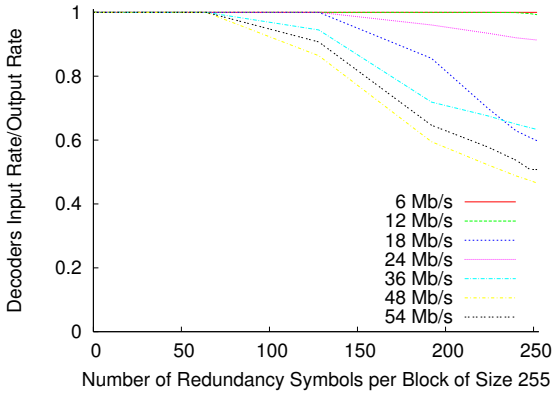


Figure 17: Software Decoding Performance: The figure shows that at all bit-rates, the decoder can keep up with the rate at which packets arrive as long as the redundancy is less than 25%. Since almost all partial packets on high-speed indoor channels have less than 5% byte error rate, this redundancy is sufficient for recovery.

This amount of redundancy allows us to recover 98.3% of all partial packets, which is higher than what we could have recovered with only one round. Clearly, one can continue increasing the number of rounds to increase the number of decodable partial packets, and hence the overall throughput. Such a large number of rounds significantly increases the delay and the overhead of receiver feedback, making it impractical. However, it serves as a useful benchmark to compares against the idealized CRC scheme in §5.0.1, and to determine how much of the performance gains from coding can be captured with only a small number of rounds. Such an infinite round scheme will send for each packet exactly the amount of needed redundancy, i.e., it follows the $p(r)$ curve.

How effective is a coding scheme at harnessing the throughput gains of partial packets? Figure 16 plots the throughput gains with one round, two rounds, and an idealized scheme of infinite rounds of redundancy, and compare these graphs to the ideal gains obtainable from partial packets. We can see from this that a coding based scheme can harness the vast majority of the partial packet gains. Additionally, much of this gain can be achieved with only two rounds without making impractical assumptions about infinite receiver feedback. Since this is better than what is achieved with a CRC-based approach, ZipTx's design employs a coding-based approach.

(d) Computational Efficiency We want to ensure that coding can be done at line speed for high bit-rate 802.11 channels. In Reed-

Solomon codes, most of the computational complexity comes at the decoder. Thus, we would like to check that our decoder can run at line speed, particularly for the case of typical indoor channels where the bit-rate can be as high as 54 Mb/s. To do so, we run the decoder on a 3Ghz Pentium machine and feed it with our indoor traces. We compare the output rate of the decoder to its input rate. If this ratio is one, then the decoder can always keep up with the line speed. If less than one, than the decoder starts lagging behind the speed at which packets are arriving, overflowing queues, and causing partial packets to be dropped. Figure 17 shows that all bit rates can be decoded at line speeds up until at least 25% redundancy. Such redundancy is more than sufficient for the high bit-rate indoor channels, where almost all partial packets have less than 5% byte error rates.

6 ZipTx

ZipTx is a modification to the Madwifi driver [1] that allows it to harness the gains of partial packets. It runs the Madwifi hardware abstraction layer (HAL) in monitor mode, which directs it to pass all packets to the driver regardless of whether they pass the checksum test. Thus, ZipTx requires no modifications to the kernel or applications, nor does it require changes to the HAL or the hardware.

ZipTx recovers erroneous packets using the two-round coding scheme in §5.0.2. At a high level, the transmitter first sends the original data packet, called the *native packet* without any parity bits. If the packet is received with errors, the receiver buffers it in the hope that the errors can be corrected later using parity bits. Every few packets, the receiver asynchronously responds back indicating for each packet whether it was received correctly, received with some errors, or not received at all. If a given packet was not received at all, then ZipTx retransmits the packet. If the packet was received correctly, the transmitter marks the packet as completed and moves on to the next packet. If the packet was received with errors, however, the transmitter sends a *coded packet* consisting of parity bits to allow the receiver to recover the original packet. Upon reception of the coded packet, the receiver attempts to decode using the combination of the original erroneous packet and the parity bits. Along with the next asynchronous response, it communicates back indicating to the transmitter whether the decoding was successful. This triggers one additional round of transmitting additional parity bits and the receiver trying to decode using all parity bits. If the second round also fails, the transmitter transmits the original packet again, repeating the entire process for a configured number of attempts.

6.1 Making Error Correction Work in Practice

The design of ZipTx addresses the following three practical issues.

(a) Packets with many errors As discussed in §4.2 many packets sent on the outdoor channel have a relatively high percentage of errors. Because it is so expensive to correct the errors in these packets, we are better off treating these packets as if they were never received at all, and just retransmitting them. Unfortunately however, the receiver does not know the BER of a transmitted packet until after it is decoded.

ZipTx solves this problem through the use of pilot bits. In particular, the transmitter inserts a known pilot bit into the transmitted packet once every 15 bytes, resulting in 100 pilot bits in a 1500 byte packet. The known pilot bits allow the receiver to fairly accurately estimate the overall BER of the packet with less than 1% overhead. The receiver then uses this estimate of BER and treats as erasures packets with a BER above a configurable threshold.

(b) How much coded data per packet? In §5.0.2, we showed how to pick the optimal amount of parity to send in each coded packet. This selection, while useful for bounding the gains of partial packets, is hard to apply in practice because it assumes that the driver knows the error distribution for the hardware and the environment in which it is operating, and that this distribution is static. In practice, we need a more dynamic decision process based on observed channel conditions. Doing so however requires fast feedback of an accurate metric of channel quality. A naive solution might use the RSSI metric typically available from 802.11 hardware for each packet. RSSI values are well known to be noisy, and unless averaged over long intervals, produce a poor estimate of channel quality. So, instead, ZipTx directly feeds back the metric that most matters: an estimate of the BER in the native packet. It does this by feeding back to the transmitter the BER estimate from the pilot bits which are discussed above.

(c) Countering coded packet overhead 802.11 is a CSMA protocol, and so every packet transmission requires the transmitter to spend time contending for the channel, an overhead that can account for 40% or more of channel time at high bit-rates, even with no other transmitters. Hence, sending parity bits in separate packets can create an excessive amount of overhead that negates any opportunity for throughput gain.

To handle this, ZipTx takes advantage of the fact that typical Internet packets are at most 1500 bytes while the maximum 802.11 frame size is 4095. Specifically, it opportunistically piggybacks the coded packets on top of native packets. If no native packet is sent for a configurable period of time however, the coded packet is sent as its own packet without piggybacking. In these cases the application has no new data to send, and hence throughput is not an issue.

6.2 Sending Receiver Feedback

A ZipTx receiver needs to inform the sender whether a packet was decodable, and whether it was even received at all. However, acquiring the medium to send separate packets is expensive. To counter the overhead of sending asynchronous acknowledgments, ZipTx uses two techniques. When packets are received with errors ZipTx sends asynchronous acknowledgments in batches, and when packets are received correctly ZipTx avoids sending acknowledgments altogether. Specifically, during periods with no packet errors, ZipTx can rely on the synchronous acks to convey data reception, and hence abstain from sending asynchronous acks. When errors or erasures occur, ZipTx waits until either it receives 8 more packets, or a timeout is reached.

The ack contains information about multiple recent erasures or partial packets. Erasures are detected using packet sequence numbers. For each packet, two bits encode what the receiver wants the transmitter to send – native, first coded packet, second coded packet, or no packet at all. Note that unlike standard 802.11, the receiver is actually in control of retries. Thus, if the receiver still cannot decode a particular native packet after receiving the two associated coded packets, the receiver moves back to re-requesting the native packet, effectively starting over again. Given that we picked the code redundancy to ensure that the vast majority of partial packets are decodable, such retrials should be infrequent.

6.3 Flow Control

In order to ensure that queue sizes and packet latencies do not grow unbounded, ZipTx implements a TCP-like flow control algorithm.

When the transmitter and receiver associate, they agree on the maximum number of outstanding native packets. Since both the transmitter and the receiver must maintain state for each outstanding native packet, this bounds the size of these buffers. Once the transmitter reaches this bound, it stops accepting new native packets from the kernel until it receives an acknowledgment that allows it to free a slot in its buffer.

One important issue is to ensure that deadlock cannot occur. Say that the channel is down for a relatively long interval, during which both packets and acks are erased. During this time, the sender will continue to send native packets until it reaches the maximum number of outstanding native packets, at which time it stops transmitting until it receives an ack from the receiver about the status of these packets. However, given that all packets were erased, and acks are triggered by packet arrivals, the receiver will also be waiting for the sender, creating a deadlock.

These deadlocks have to be broken by the transmitter because the receiver cannot distinguish this scenario from that when the transmitter has no packets to send. Thus, after waiting for a configurable timeout, the transmitter knows that either all of its outstanding packets were erased, or the resulting acks were erased. It assumes the former, and begins to retransmit all outstanding packets. It continues to cycle through the outstanding packets until it receives an acknowledgment or one of the packets in the buffer times out leaving a slot for another packet. If a much larger timeout is reached without receiving any acks then the transmitter will clear all of its state and attempt to re-associate with the receiver.

6.4 Auto-Rate

A key component of 802.11 is its use of multiple bit-rates allowing an auto-rate algorithm to choose an optimal modulation and FEC scheme for the given channel. ZipTx's use of error correction codes gives it a larger margin of error when choosing the bit-rate. However, obtaining optimal performance still requires choosing the optimal bit-rate because as the channel quality degrades, a given bit-rate eventually produces lower correct byte throughput than the bit-rate below it.

ZipTx bases its autorate algorithm on sample-rate, the default autorate algorithm in the Madwifi driver [4]. Sample-rate periodically samples the throughput of various bit-rates by sending every 20th packet at a randomly chosen bit-rate. It calculates the expected throughput based on the number of retries required in order to get the packet through, and moves to using the rate with the highest throughput.

ZipTx can use all of the mechanisms of Sample-rate, except for its computation of the expected throughput. This is because ZipTx's primary mechanism for reliability is not retransmission, but error correction codes. To calculate the expected throughput at a given rate, ZipTx calculates the number of bytes delivered to higher layers divided by the amount of channel time used to send those bytes. This channel time includes the time to send the native packet and each of its coded packets. Further, ZipTx uses retransmission as a secondary reliability mechanism, so the time must also include all failed attempts to retransmit the packet. Lastly, since the autorate algorithm runs at the sender, the receiver needs to include in its acks information about which packets were delivered to higher layers.

7 ZipTx Evaluation

Our evaluation compares the following two drivers:

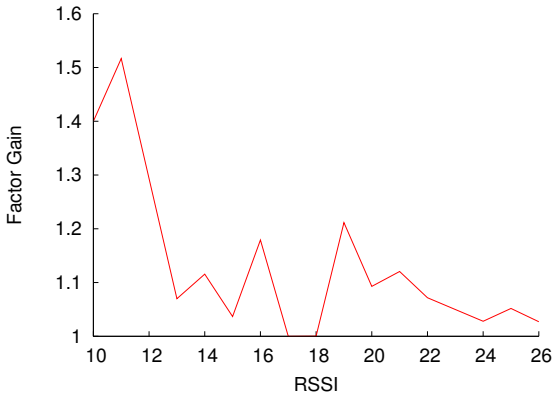


Figure 18: ZipTx's throughput gain in typical indoor channels: The figure plots the ratio of the throughput of ZipTx to that of the unmodified driver. It shows that, typically, ZipTx boosts the throughput by 10-20%, with some cases having gains as high as 50%.

- *Unmodified Driver:* This refers to the Madwifi v0.9.3 [1] driver. On Linux, Madwifi is the current *de facto* driver for Atheros chipsets and is a natural baseline.
- *ZipTx:* This is our implementation of ZipTx as an extension of Madwifi v0.9.3. Our implementation works in conjunction with auto-rate algorithms, carrier-sense, CTS-to-self protection, etc.

We experiment with ZipTx in the environments described in §3. For each link we run ZipTx three times and the unmodified driver three times. All measurements are calculated by the drivers themselves as we have instrumented both the ZipTx driver and the unmodified driver such that every second they output the average throughput and RSSI values. Note that on hardware using the Atheros chipset, the RSSI is defined as the SNR in dB. Additionally, all results are taken with a continuous stream of UDP traffic generate by Click [21].

7.1 Typical Indoor Channels

Indoor 802.11 networks typically offer a relatively high SNR as nodes are densely deployed in order to ensure access in all locations. In such networks, auto-rate algorithms can choose from a fine selection of bit-rates and work quite well. As a result, ZipTx provides practical throughput gains that are dependent on how close the max-packet bit-rate is to the ideal rate sustainable by the channel.

We evaluate ZipTx on such channels using the indoor testbed described in §3, which contains indoor links whose optimal bit-rate varies between 6 Mb/s and 54 Mb/s. For each link, we make a one-minute transfer first with ZipTx then the unmodified driver, and repeat the experiment three times. We compute ZipTx's fractional throughput gain as the ratio of ZipTx's throughput to that of the unmodified driver. Figure 18 plots the average gain as a function of the link's RSSI. We see that ZipTx's throughput gain follows a similar pattern to the ideal throughput gain in indoor channels, shown in Figure 5, though is a bit lower due to practical overheads. In particular, the throughput gain is typically around 10-20%, with some cases being as high as 50%.

7.2 Challenged Outdoor Channels

Outdoor 802.11 networks often suffer from long multi-path delay and have much lower SNR. Such networks use 802.11b links since they typically cannot sustain the higher rate modulation schemes utilized by 802.11g and 802.11a. As we discussed in §4.2, in such scenario,

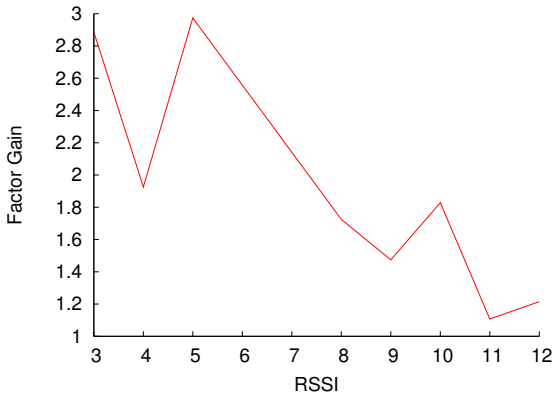


Figure 19: ZipTx's throughput gain in challenged outdoor channels: The figure plots the ratio of the throughput of ZipTx to that of the unmodified driver. ZipTx provides improvements ranging from 10% for the better quality links, to up to $3x$ for the lower quality links.

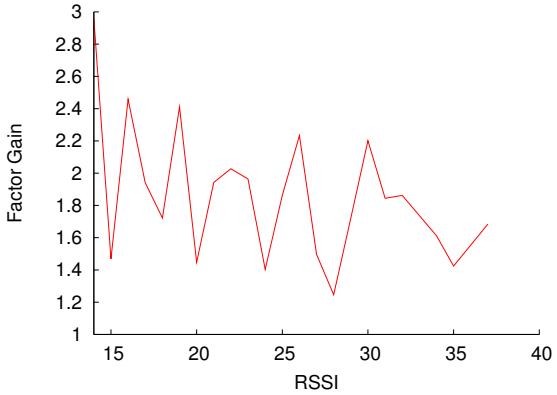


Figure 20: ZipTx's throughput gain in mobile channels: As a result of autorate's difficulty in picking a good rate, ZipTx improves the throughput by up to 3x, when moving at a walking pace.

autorate algorithms often struggle because of the coarse granularity of the available bit-rates and their high BER.

We evaluate ZipTx on the outdoor channel using the testbed described in §3 which contains links with channel qualities from just below those in the indoor range, down to the lowest SNR for which we were able to maintain association with the node acting as AP. From Figure 19 we see that, as expected from the analysis in §4.2, the throughput gain ZipTx provides is much larger in the outdoor scenario. It provides a 70% throughput improvement over the unmodified driver on average, while at the low end of the RSSI range it provides more than a 3x gain. For example, in these low RSSI scenarios, ZipTx provides a 1 Mb/s of throughput for a link that can only carry a couple hundred of Kb/s with the unmodified driver.

7.3 Mobile Channels

The main draw of wireless networks is that they make it easy for the user to move while still maintaining network connectivity. Thus in most common 802.11 usage scenarios, the user is at least somewhat mobile. Most of this movement is at relatively low speeds however, with the user walking from one location to another. These mobile channels run at high SNR, similar to our indoor channel. However, the autorate algorithm can be challenged by the movement, making them potentially function more like the low SNR outdoor channel.

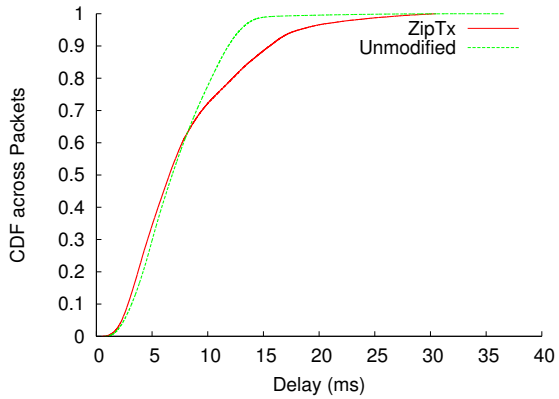


Figure 21: ZipTx's impact on delay: The figure shows the CDF of the packet delay across all indoor locations for both ZipTx and the unmodified driver. The figure shows that 70% of the packets experience more or less the same delay, 20% of the packets experience a delay increase of about 5 ms, and no packet sees an increase in delay that exceeds 11 ms.

To evaluate the throughput gain provided by ZipTx in this scenario, we set up a transmitter node in a conference room, and placed the receiver node on a cart. We then walked from the conference room down the hall to another conference room and back, repeating this loop for 1 minute. For each driver we ran three of these tests. During each test we output the throughput and RSSI once every second.

Figure 20 shows the results of this experiment. We see that, similar to the outdoor experiments, ZipTx provides throughput gains of more than 70% on average, and about 3x for some RSSIs.

This gain results from the auto-rate algorithm working poorly as a result of the movement. The SampleRate algorithm only occasionally updates its bit-rate selection since it must have time to gather sufficient samples from each of the different bit-rates. When the node is moving, however, the optimal bit-rate can change relatively quickly. Our modified version of Sample-Rate has the same problem, but ZipTx's use of partial packets provides it a much larger margin error in its bit-rate selection. This is because when standard 802.11 chooses too high a bit-rate, most of the packets are only partially correct, resulting in lost packets and creating a catastrophic drop in throughput. Since ZipTx is able to take advantage of the these partially correct packets, its throughput drops off much more gracefully, allowing it to provide reasonable throughput even when it chooses too high a bit-rate.

7.4 Delay

Since ZipTx's packet recovery employs two rounds of feedback and is implemented in software, it is important to check that it does not introduce unacceptable delay.

To measure the per packet delay, we sync up the CPU clock of our transmitter and receiver nodes using an ntp server on our local network. We use Ping tests to confirm that the clocks are sync-ed to within a 1 ms tolerance. For each packet the transmitter inserts into the body of the packet the time at which the packet is received from the kernel. When the packet is delivered to the kernel on the receiver side, it computes the end-to-end delay. We measure the delay for both ZipTx and the unmodified driver by running them back-to-back on the 35 indoor links in our testbed.

Figure 21 plots the per packet delay distributions for both ZipTx and the unmodified driver. The figure shows that ZipTx's delay stays within acceptable range. Specifically, 70% of the packets experience about the same delay as in the unmodified driver. As for cases that

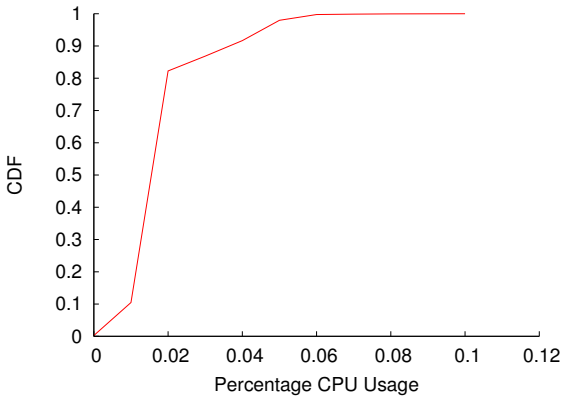


Figure 22: ZipTx's CPU usage: The figure plots the CDF of CPU usage across all indoor locations, showing that it stays low.

experience higher delays, the absolute increase is usually less than 5 ms, and in all cases stays bounded by 11 ms.

It should be noted that ZipTx has internal mechanisms to limit the delay increase. In particular, (1) it is limited to two rounds of feedback, (2) it sets small sizes on delay critical buffers, and (3) it times out packets that take too long to deliver.

7.5 CPU Usage

We showed in §4 that ZipTx's software decoder was able to decode at line rate even for high bit-rates. A natural question however is whether or not this decoding will load up the machine so much that it cannot be used for other activities during network transmissions.

We answer this question by measuring the CPU usage while we are running the indoor experiments from §7.1. To ensure that we account for all CPU usage caused by ZipTx we use *vmstat* to measure the average total CPU usage once every second. Our experiments run on 3.0 Ghz Intel Core2 Duo machines running Linux FC 6. The results are shown in Figure 22.

From this figure, we can see that the CPU usage never gets above 10% and is typically only 1 or 2%. The reason for this is that the decoding algorithm will only load the CPU when it has a packet with many errors to decode. As we saw from Figure 3, even at the max-byte bit-rate, more than 30% of the packets received are still fully correct, and a much higher fraction have only a few errors and are relatively fast to decode. Thus, high error rate packets are interspersed with low error packets and correct packets, ensuring that the average CPU load is relatively low.

8 Conclusion

This paper provides two major contributions. It characterizes the gains of partial packets across the entire range of operation of 802.11 networks, examining typical indoor WLAN scenarios, as well as challenged outdoor and mobile scenarios. Additionally, it presents the design and implementation of a purely software solution for leveraging the opportunity provided by partial packets. The ZipTx driver runs on off-the-shelf 802.11 hardware and provides average gains of 10-20% on typical indoor high SNR links and 2-3x on mobile and challenged outdoor links.

The main conclusion of our paper is that the gains of partial packets are highly dependent on the availability of rate adaptation and its effectiveness. A solution that harnesses partial packets is of a limited impact in scenarios where autorate is highly effective. However, when autorate is non-existent or ineffective because of mobility or other reasons, a solution that harnesses partial packets can compensate for this inefficiency and deliver large throughput gains.

Acknowledgments: We thank Hariharan Rahul and Shyamnath Gollakota for much help with the editing of this paper.

References

[1] Madwifi. `http://madwifi.org`.

[2] Madwifi bug report. `http://madwifi.org/ticket/705`.

[3] D. Aguayo, J. Bicket, S. Biswas, G. Judd, and R. Morris. Link-level measurements from an 802.11b mesh network. In *ACM SIGCOMM*, 2004.

[4] J. Bicket. *Bit-rate selection in wireless networks*. PhD thesis, Massachusetts Institute of Technology, 2005.

[5] V. Brik, A. Mishra, and S. Banerjee. Eliminating Handoff Latencies in 802.11 WLANs using Multiple Radios. In *Usenix IMC*, 2005.

[6] V. Bychkovsky, B. Hull, A. K. Miu, H. Balakrishnan, and S. Madden. A Measurement Study of Vehicular Internet Access Using In Situ Wi-Fi Networks. In *12th ACM MOBICOM*, 2006.

[7] S. Chachulski and S. Katti. Trading structure for randomness in wireless opportunistic routing. In *ACM SIGCOMM*, Kyoto, Japan, August 2007.

[8] D. Chase, I. CNR, and M. Needham. Code Combining–A Maximum-Likelihood Decoding Approach for Combining an Arbitrary Number of Noisy Packets. *Communications, IEEE Transactions on*, 33(5):385–393, 1985.

[9] G. Clark and J. Cain. *Error-Correction Coding for Digital Communications*. Perseus Publishing, 1981.

[10] R. Comroe and D. Costello Jr. ARQ Schemes for Data Transmission in Mobile Radio Systems. *Selected Areas in Communications, IEEE Journal on*, 1984.

[11] D. S. J. D. Couto, D. Aguayo, J. Bicket, and R. Morris. A high-throughput path metric for multi-hop wireless routing. In *Proceedings of the 9th annual international conference on Mobile computing and networking*, pages 134–146, 2003.

[12] H. Duboi-Ferriere, D. Estrin, and M. Vetterli. Packet combining in sensor networks. In *ACM Sensys*, 2005.

[13] F. Frederiksen and T. Kolding. Performance and modeling of WCDMA/HSDPA transmission/H-ARQ schemes. *Vehicular Technology Conference, 2002. Proceedings. VTC 2002-Fall. 2002 IEEE 56th*, 1, 2002.

[14] P. Frenger, S. Parkvall, and E. Dahlman. Performance comparison of harq with chase combining and incrementalredundancy for hsdpa. In *Vehicular Technology Conference, 2001. VTC 2001 Fall. IEEE VTS 54th*, volume 3, 2001.

[15] R. K. Ganti, P. Jayachandran, H. Luo, and T. F. Abdelzaher. Datalink streaming in wireless sensor networks. In *ACM Sensys*, 2006.

[16] M. S. Gast. *802.11 Wireless Networks: The Definitive Guide*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[17] P. Gupta and P. Kumar. The capacity of wireless networks. *Information Theory, IEEE Transactions on*, 46(2):388–404, 2000.

[18] K. Jamieson and H. Balakrishnan. PPR: Partial Packet Recovery for Wireless Networks. In *ACM SIGCOMM*, Kyoto, Japan, August 2007.

[19] S. Kallel and D. Haccoun. Generalized type II hybrid ARQ scheme using punctured convolutionalcoding. *Communications, IEEE Transactions on*, 38(11):1938–1946.

[20] S. Katti and D. Katabi. MIXIT: The Network Meets the Wireless Channel. In *ACM HotNets*. Massachusetts Institute of Technology Computer Science and Artificial Intelligence Laboratory, 2007.

[21] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[22] S. Lin and P. Yu. A Hybrid ARQ Scheme with Parity Retransmission for Error Control of Satellite Channels. *Communications, IEEE Transactions on*, 30(7), 1982.

[23] A. K. Miu, H. Balakrishnan, and C. E. Koksal. Improving Loss Resilience with Multi-Radio Diversity in Wireless Networks. In *11th ACM MOBICOM Conference*, 2005.

[24] E. C. Posner, L. L. Rauch, and B. D. Madsen. Vovager mission telecommunication firsts. `http://ieeexplore.ieee.org/iel1/35/2092/00057695.pdf`.

[25] Roofnet. http://www.pdos.csail.mit.edu/roofnet.

[26] L. Shu and C. Daniel. *Error Control Coding*. Prentice-Hall, 1983.

[27] S. Surana, R. Patra, S. Nedevschi, L. Subramanian, Y. Ben-David, and E. Brewer. Beyond pilots: Keeping rural wireless networks alive. In *ACM NSDI*, 2008.

[28] D. Tse and P. Vishwanath. *Fundamentals of Wireless Communications*. Cambridge University Press, 2005.

[29] G. Woo, P. Kheradpour, D. Shen, and D. Katabi. Beyond the bits: cooperative packet recovery using physical layer information. pages 147–158. ACM Press New York, NY, USA, 2007.