Maranello: Practical Partial Packet Recovery for 802.11

Bo Han*, Aaron Schulman*, Francesco Gringoli†, Neil Spring*, Bobby Bhattacharjee*
Lorenzo Nava†, Lusheng Ji‡, Seungjoon Lee‡, Robert Miller‡
* University of Maryland † University of Brescia ‡ AT&T Labs – Research

Abstract

Partial packet recovery protocols attempt to repair corrupted packets instead of retransmitting them in their entirety. Recent approaches have used physical layer confidence estimates or additional error detection codes embedded in each transmission to identify corrupt bits, or have applied forward error correction to repair without such explicit knowledge. In contrast to these approaches, our goal is a practical design that simultaneously: (a) requires no extra bits in correct packets, (b) reduces recovery latency, except in rare instances, (c) remains compatible with existing 802.11 devices by obeying timing and backoff standards, and (d) can be incrementally deployed on widely available access points and wireless cards.

In this paper, we design, implement, and evaluate Maranello, a novel partial packet recovery mechanism for 802.11. In Maranello, the receiver computes checksums over blocks in corrupt packets and bundles these checksums into a negative acknowledgment sent when the sender expects to receive an acknowledgment. The sender then retransmits only those blocks for which the checksum is incorrect, and repeats this partial retransmission until it receives an acknowledgment. Successful transmissions are not burdened by additional bits and the receiver needs not infer which bits were corrupted. We implemented Maranello using OpenFWWF (open source firmware for Broadcom wireless cards) and deployed it in a small testbed. We compare Maranello to alternative recovery protocols using a trace-driven simulation and to 802.11 using a live implementation under various channel conditions. To our knowledge, Maranello is the first partial packet recovery design to be implemented in commonly available firmware.

1 Introduction

Partial packet recovery approaches attempt to repair corrupt packets instead of retransmitting them. Packet recovery relies on the observation that packets with errors may have only a few, localized errors, or at least some salvageable, correct content. Various approaches have been proposed: some rely on physical layer information to identify likely corrupt symbols (related groups of bits) to be retransmitted [12], while others embed block checksums into oversized frames to allow the receiver to recognize partially correct transmissions [11]. Some avoid explicit knowledge and adaptively transmit forward error correction information that is likely to be sufficient to repair bit errors [14]. These approaches have found substantial potential in partial packet recovery, particularly when auto-rate selection mechanisms, which dynamically change the transmission rate to maximize throughput without too many errors, may choose too high a rate, thus creating errored packets to be recovered.

Motivated by the potential of these recent approaches, we set out to construct a partial packet recovery scheme using commonly available 802.11 hardware and evaluate it in live networks. The key challenge in working within 802.11's typical operation is timing, in particular, performing all acknowledgment-related computation within one short inter-frame space (SIFS) interval (10 μs for 802.11b/g or 16 μs for 802.11a). This requirement all but precludes bus transfers to the driver and complex processing on the network devices. To be deployable today, partial packet recovery must exploit features available to programmable *firmware*.

In this paper, we present *Maranello*, a block-based partial packet recovery approach implemented (primarily) in firmware for widely-available Broadcom cards. Maranello takes the following design decisions. We use *block-based recovery*, meaning that we identify incorrect blocks of consecutive bytes for retransmission, as opposed to aggregating by symbol or estimating bit error rate. We transmit independent *repair packets* that contain only the blocks being retransmitted, in contrast to other approaches that may bundle repair information with subsequent transmissions to save on medium acquisition time. Repair packets, by being shorter, are more likely to arrive successfully than full size retransmissions and take less time to transmit, improving performance over 802.11. Using immediate repair packets also limits the amount of buffering (of out of order, incomplete packets) required at the receiver side. We use the *Fletcher-32* checksum [5] to isolate errors to individual blocks; this checksum is sufficient to find all single bit errors, burst errors in a single 16-bit block, and two-bit errors separated by at most 16 bits [25]. Fletcher-32 is also efficient

enough to be computed block-by-block in software during frame reception. Finally, we exploit the deference stations give to acknowledgments of overheard packets: because stations sending acknowledgments have priority over the medium right after a transmission, there is time for a receiver to grab the medium and send prompt feedback about received blocks. Through these decisions, we construct a partial packet recovery scheme that (a) introduces no additional bits in the common case of successful transmissions, (b) decreases recovery time after failed transmissions, (c) is compatible with unmodified 802.11 devices, and (d) can be implemented on typical off-the-shelf hardware and deployed incrementally.

Our goal in constructing a practical partial packet recovery scheme was to permit evaluation both in simulation and on live networks. We apply two strategies. First, we construct a trace-driven simulation to evaluate the performance Maranello would have when run with various combinations of operating system, driver, and chipset, as well as the performance Maranello would have compared to idealized PPR [12] and ZipTx [14]. We study the retransmission behavior of 802.11 implementations so that we might simulate Maranello on each: performance improvement depends on how aggressively the existing firmware retransmits, in particular, whether it performs proper exponential backoff and how it reduces transmission rate. We survey retransmission rate fallback selection schemes and show that Maranello increases throughput regardless of retransmission rate fallback: if the rate chosen is too high, Maranello may increase the delivery probability with a short repair packet [8]; if too low, Maranello decreases the time to transmit relative to retransmission.

Our implementation permits us to evaluate Maranello in terms of delivered throughput and latency in realistic settings. We compare the link throughput of Maranello and that of the original 802.11 in three different environments: an industrial research lab, a home, and a campus office building. We show that Maranello can significantly improve the delivered link throughput. We also verify that, even in the presence of bit corruption, Maranello can maintain or reduce the link latency, in terms of the time to deliver an individual packet and receive an acknowledgment. We also deploy Maranello on programmable access points running OpenWRT to ensure scalability and compatibility by associating both Maranello-enabled and unmodified 802.11 devices. Surprisingly, we find that ACK frames can be modified to report the feedback information of received blocks, without causing errors on coexisting unmodified 802.11 devices.

In the following section, we present an overview of prior wireless error recovery mechanisms including partial packet recovery schemes and those that rely on wireless communication diversity. In Section 3, we present the high-level design of Maranello, show how wireless errors cluster enough to support block-based recovery, and justify the choice of Fletcher-32. In Section 4, we evaluate these design choices in simulation, showing the potential throughput gains by interpreting detailed packet traces. In Section 5, we implement Maranello using the OpenFWWF firmware and a slightly modified driver within the Linux kernel. Section 6 presents performance comparisons collected in our testbeds using this implementation. We offer a discussion in Section 7 and conclude in Section 8.

2 Related Work

In this section, we classify various wireless error recovery protocols. Table 1 summarizes wireless error recovery protocols. We categorize these protocols along two dimensions: the main repair techniques that they employ and the features they provide. The main repair techniques include:

Block checksum (Section 2.1) When transmissions fail, receivers can aid recovery by sending feedback about corrupted blocks based on the per-block checksums transmitted with data packets. Seda [6] and FRJ [11] are protocols in this category.

Forward error correction (Section 2.2) Protocols like ZipTx [14] avoid explicit knowledge about where the error bits are and adaptively transmit error correction bits that are likely to be sufficient to repair corrupted packets.

PHY layer hints (Section 2.3) The PHY layer of GNU Radio systems can provide the confidence of each symbol's correctness. PPR [12] and SOFT [27] benefit from this information to identify corrupt bits without extra error detection codes.

Wireless communication diversity (Section 2.4) Wireless packet losses are path and location dependent. A packet corrupted at its destination may be correctly received by other radios, due to the broadcast nature and diversity of wireless communication. Several protocols exploit this diversity to perform error recovery, such as MRD [20], SPaC [4], and PRO [16].

These error recovery protocols provide the following features:

No extra bits for correct packets Most of the protocols introduce no additional bits for successful transmissions, except Seda and FRJ, which transmit block/segment checksums with all packets, and ZipTx, which sends pilot bits in each transmission.

Technique	Protocol	No extra bits for correct packets	Maintain link latency	Compatible with 802.11	Incremental deployment	Partial Packet Recovery
Checksum	**Maranello**	✓	✓	✓	✓	✓
	Seda [6]			N/A	✓	✓
	FRJ [11]				✓	✓
FEC	ZipTx [14]				✓	✓
PHY layer hints	PPR [12]	✓	✓	N/A		✓
	SOFT [27]	✓	✓	N/A		
Diversity	MRD [20]	✓	✓		✓	
	SPaC [4]	✓	N/A	N/A	✓	
	PRO [16]	✓	✓	✓	✓	

Table 1: Desired behavior and functionality of wireless error recovery protocols

Reduce recovery latency Seda, FRJ, and ZipTx may increase the recovery latency by aggregating feedback for a group of corrupted packets. MRD and SOFT may also increase the recovery latency for the packets that cannot be repaired by frame combining.

Compatible with 802.11 Among the protocols designed for 802.11 wireless networks, MRD, FRJ, and ZipTx disable the retransmission protocol at the MAC layer and thus do not interoperate with native 802.11.

Incremental deployment Most of the protocols are implemented using commercial hardware, either 802.11 cards or MICA motes, and thus can be incrementally deployed on widely available wireless devices. In contrast, PPR and SOFT use physical layer information provided by GNU Radio systems.

Partial packet recovery Protocols like PRO and SOFT always retransmit the entire packet when the original cannot be recovered.

Table 1 shows that none of these protocols achieve all these features simultaneously.

2.1 Block Checksum

Acknowledgment frames can be extended to include feedback to help error recovery protocols. Seda [6] is a recovery mechanism designed for data streaming in wireless sensor networks. In Seda, a sender divides each packet into blocks and encodes each block with a one-byte sequence number and a (one-byte) CRC-8 for error detection. A receiver, after receiving several packets, will test the block-level CRC-8's for packets that fail the CRC-32 (if any) and request retransmission of those blocks.

FRJ [11] uses jumbo frames to increase wireless link capacity. Each jumbo frame comprises 30 segments and each segment has its own CRC checksum. The receivers can check these segment checksums to perform partial retransmissions when the segments are corrupted. FRJ uses both MAC-layer ACKs and its own ACKs. FRJ sends its own ACKs after 100 ms or 64 received frames.

Unlike Seda and FRJ, Maranello introduces no extra bits for correctly received frames and performs retransmission immediately after corrupted frames are detected.

2.2 Forward Error Correction

Forward error correction codes are beneficial to error recovery because they do not require explicit information about error locations. ZipTx [14] uses a two-round forward error correction mechanism to repair corrupted packets. In the first round, the transmitter sends a small number of Reed-Solomon bits for a corrupted packet, based on the feedback provided by the receiver. If the receiver still cannot recover the corrupted packets using these parity bits, the transmitter sends more parity bits in the second round. If both rounds fail, the receiver requests a retransmission of the whole packet. To reduce the number of feedback frames, ZipTx receivers accumulate feedback information to be transmitted after receiving eight packets or after a timeout.

Although ZipTx increases throughput, it may also increase recovery latency. This is because it disables MAC layer retransmission and generates its own ACKs for a group of packets in the driver. As a result, the delay for the recovered packets may be significantly higher than that of the retransmitted native 802.11 packets. Maranello repairs corrupted packets immediately after transmission fails and thus can reduce recovery latency.

2.3 PHY Layer Hints

Error recovery protocols can benefit from physical layer information beyond the best guess at the received symbol, although most commercial 802.11 cards do not expose such extra information. PPR [12] requests retransmissions of only those symbols that are likely corrupted. PPR also provides a compact encoding of the ranges of bits requested for retransmission and replicates the preamble to a “postamble” so that receivers may recover

correct bits at the end of packets that lack a good preamble. PPR was implemented and evaluated on an 802.15.4 (ZigBee) protocol stack.

Driven by per-bit confidence from the PHY Layer, SOFT [27] combines several received versions of a corrupted frame to produce a correct frame. To repair packets sent to an AP, several APs share bit confidence over a wired link. To repair packets sent to a client, the client combines per-bit confidence from a corrupted transmission and one or more retransmissions.

Due to performance limitations of software radio platforms, these protocols are evaluated only at low bit rates. In contrast, Maranello is implemented using readily available commercial 802.11 hardware, and thus it can be immediately realized at speed and deployed. We also show that Maranello provides increased performance even with the encodings used for high bit rates.

2.4 Wireless Communication Diversity

Correcting errors with wireless diversity complements Maranello's packet repair. Diversity approaches attempt to correct packets by observing different copies of the same packet, either as received at different stations or as received in (corrupt) retransmissions. When failure happens, MRD [20] combines many received versions of a given packet at different APs, which may have error bits at different locations, to recreate the original packet. If the original packet cannot be recovered through frame combining, a retransmission protocol, called Request For Acknowledgment (RFA), is proposed to retransmit the whole packet. SPaC [4] exploits the spatial diversity of multihop wireless sensor networks to combine several corrupted receptions of a packet at its destination. These corrupted receptions may be retransmitted by different neighboring nodes to repair the original transmission. PRO [16] is an opportunistic retransmission protocol for 802.11 wireless LANs that allows overhearing relay nodes to retransmit on behalf of the source node after they know that a transmission failed.

Other protocols can benefit from wireless communication diversity, but these are typically evaluated only by theoretical analysis or simulation study. For example, MRQ [24] keeps all the erroneous receptions of a given packet and recovers the original packet by combining these receptions. Like PRO, HARBINGER [28] improves the performance of Hybrid ARQ, by exploiting retransmitted packets from relays that overhear the communication. The approach of Choi et al. [3] uses the error correction bits transmitted in data packets to recover corrupted blocks. It retrieves uncorrected blocks from later retransmissions of the packets and combines them with previous blocks to recover the original packets.

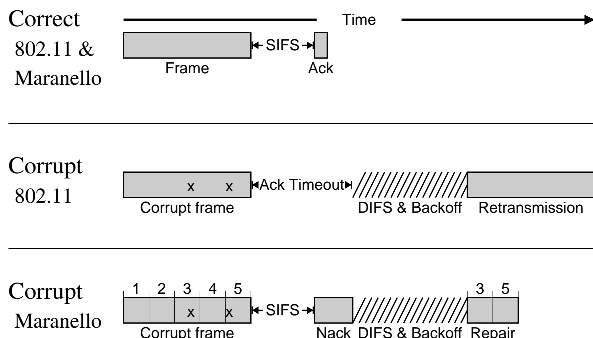


Figure 1: Maranello reacts to packet corruption by sending a NACK when the sender awaits an ACK. The time to repair should decrease relative to retransmission. (Diagram not to scale.)

3 Maranello Design

In this section, we present an overview of Maranello, describe how it achieves the key design goals of a practical partial packet recovery scheme, and justify the choices of block-based recovery and the Fletcher-32 checksum computation. We analyze this design in isolation in the following section (4) before presenting implementation details (Section 5) and evaluating the implementation on real hardware.

3.1 Overview

Figure 1 presents an overview of the Maranello protocol, compared to 802.11. When a Maranello-supporting device receives a frame with errors, it divides the frame into 64-byte blocks (the last block may be smaller) and computes a separate checksum for each block. Then it replies to the transmitter with a NACK that includes these checksums. It saves the corrupted original packet in a buffer, waiting for the sender to transmit correct blocks. This negative acknowledgment is sent when the transmitter expects to receive a positive acknowledgment. A Maranello-supporting transmitter will then match the receiver-supplied checksums to those of the original transmission and send a repair packet with only those blocks of the original transmission that were corrupted. Once the repair packet is received correctly, the receiver sends a normal 802.11 ACK.

Devices that do not support Maranello interoperate easily. Unmodified senders will treat the negative acknowledgment as garbage and retransmit as normal. Unmodified receivers will fail to transmit a Maranello NACK, and cause a Maranello sender to retransmit after timeout.

At very low transmission rate, the NACK for a large packet may be longer than other stations expect to defer to the acknowledgment (i.e., it may extend beyond the

Network Allocation Vector); if it does, we rely on carrier sense to inhibit collisions with the end of the NACK.

The cases when a Maranello-specific packet are lost are straightforward. If a NACK is lost, the transmitter will retransmit the packet as in 802.11. If this retransmission has errors, the receiver will send another NACK. If a repair packet is lost or received with errors, the receiver will transmit nothing. One could alter the protocol to send an abridged NACK to recover correct blocks from errored repair packets, but we expect minimal gain from the added complexity.

3.2 Design Goals

Maranello is a practical partial packet recovery design with four primary goals, described below.

Require no extra bits in correct packets Maranello embraces systems design principles of optimizing the common case, successful transmission, and doing no harm (not increase the size or delay of retransmissions). No additional error checking information, beyond the existing CRC-32, is added to normal packets.

Reduce recovery latency Maranello ensures that recovery latency is smaller than retransmission time by using the time reserved for positive acknowledgments to, in the event a positive acknowledgment is not warranted, send negative acknowledgments.

(In the unlikely event that the entire packet is corrupt, the longer NACK may require more time than an ACK and the retransmission of entire packet may not be avoided, leading to an overall increase in retransmission time.)

Compatibility with existing 802.11 802.11 is widely deployed, cheap, and useful. To extend it requires obedience to key inter-frame spacing and backoff requirements. The receiver must be able to construct and send a NACK before the transmitter decides to retransmit the entire packet, ideally immediately after the SIFS (short inter-frame space) interval when the transmitter expects an ACK. That is, the implementation must support extremely quick computation of block checksums in order to respond to the sender. At the same time, a Maranello sender cannot send repair packets any more quickly than 802.11 sends retransmissions: collisions are a potential cause of transmission error and must be addressed by proper exponential backoff. These two features are necessary for coexistence with 802.11 networks.

Incremental deployability on existing hardware Wireless networks are dynamic: Maranello should not require negotiation or, worse, ubiquitous deployment within a service area. By transmitting Maranello messages such that unmodified 802.11 devices are not confused, Maranello can coexist. In effect, the Maranello NACK is a negotiation; a Maranello station may infer that the receiver does not support Maranello if no NACKs are sent. (Reserved bits in the capability-information field of beacon and association-request frames are also available; it is possible to negotiate protocol features when necessary.) Further, by implementing Maranello in the firmware of existing wireless cards, this partial packet recovery protocol can be deployed today for users just by updating the firmware.

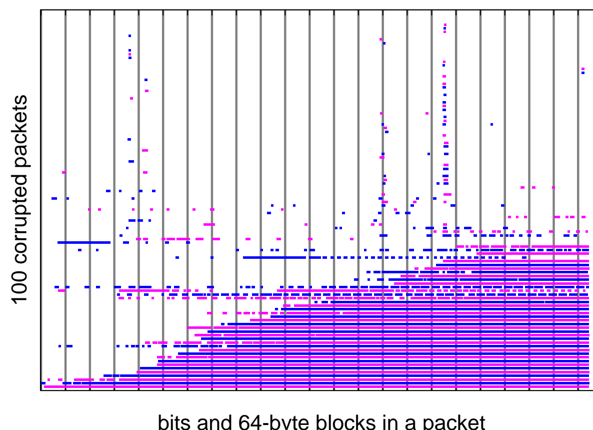


Figure 2: Shaded areas indicate bit errors. Within-packet (horizontal) correlations are likely due to interference or loss of clock synchronization; across-packet (vertical) correlations may be caused by subcarrier fading.

3.3 Block-Based Recovery

Broadly speaking, a partial packet recovery approach can use various means for receivers to solicit retransmission of parts of the packet and various means for transmitters to correct those errors. Maranello sends negative acknowledgments with checksums over blocks; transmitters determine which blocks must be retransmitted and send repair packets in place of retransmissions. (Alternate approaches may report abstract bit error estimates, request retransmission of individual symbols, or piggyback repair on subsequent transmissions, as described in Section 2.)

Block-based recovery, however, relies on a key assumption: that errors are clustered within a packet. In Figures 2 and 3, we present two views of error clustering. Figure 2 shows the positions of bit errors in 100 packets chosen at random from the errored packets in a larger trace of packets. For packets with few bit errors, those errors are constrained within 64-byte blocks. For packets with many bit errors, those errors are similarly often bound within consecutive 64-byte blocks.

Figure 3 plots 17,961 packets by the number of 64-byte blocks that would be needed to repair errors. The x-axis represents the fraction of corrupt packets: each packet occupies the same horizontal space along the axis,

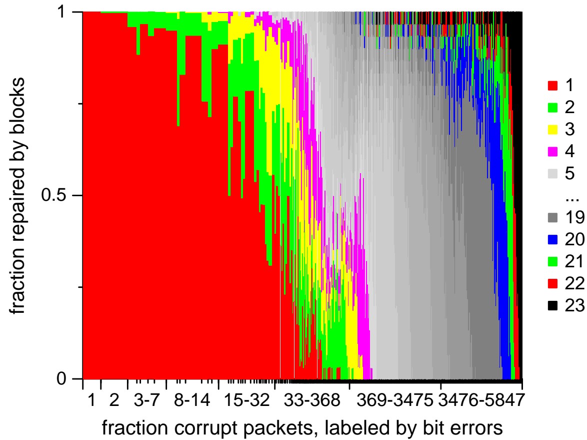


Figure 3: 64-bit blocks required to repair corrupt packets in a trace. Most packets having bit errors have few corrupt blocks; even those with many bit errors typically have a few correct blocks.

sorted in ascending order of the number of bit errors observed in that packet. A stacked bar graph extends above, showing the fraction of those packets required by different numbers of blocks. At the left side of the graph, the dominant color represents the single block's ability to repair all 1-bit errors (of course), 99.7% of two-bit errors, 96% of three-bit errors, etc. This is in contrast to a random bit-error model in which two bit errors in a 1500-byte packet would have only a 4% chance of corrupting only one 64-byte block. At the right end of the graph, relatively few packets require complete retransmission. (This graph may underestimate the number of irreparable transmissions; those that the hardware cannot receive at all would not appear.)

3.4 Fletcher-32

The block checksums a receiver puts into a NACK must be completely computed before the SIFS expires. One approach might be to reprogram the hardware-accelerated CRC-32 engine used by the device to compute whole-packet CRCs. Unfortunately, this engine does not appear to be programmable. Instead, we compute a different checksum, the Fletcher-32 [5] which is more efficiently computed on the wireless card's microprocessor. Historically, the IETF considered Fletcher checksums as an alternative for TCP checksums [30].

To verify the effectiveness of Fletcher-32 to detect bit errors, we perform the following trace-driven simulation. We take the 99,118 corrupted frames from a packet trace, and identify error bit positions in each frame. Then, we apply the error patterns to randomly generated packet contents to construct 9,911,800 errored packets. Finally, we apply CRC-32 and Fletcher-32 to detect corrupted blocks with 64-byte size. All the corrupted blocks can be detected by both CRC-32 and Fletcher-32.

Even with the efficient Fletcher-32 checksum, the microprocessor is still not powerful enough to compute each of the block checksums during the SIFS interval: A single checksum for a 64-byte block can take up to 4 μs. To solve this problem, we exploit an interesting feature of the chipset. The microprocessor, in fact, is idle during the reception of a frame! Instead of allowing it to sleep until the packet is completely received, we modify the firmware to copy partially received packets and begin computation of block checksums during reception of the next block. This approach leaves enough time at the end of a corrupted frame to compute the last checksum (if needed) and to build the NACK.

4 Simulation

Before we describe and evaluate the implementation, we evaluate the design of Maranello in simulation. Maranello's gains depend on the specified, but not always followed, 802.11 backoff and the unspecified retransmission rate fallback behavior implemented in 802.11 drivers and chipsets. We want to see if Maranello improves throughput for cards (we consider both the manufacturer's driver and chipset) that behave unlike Broadcom's, which we implemented Maranello on.

Each card implements a different suite of error control algorithms, including auto-rate selection, retransmission rate fallback, and backoff. 802.11's backoff behavior is defined in the specification, however our observations and those of Bianchi et al. [2] indicate that there are many different interpretations of 802.11 backoff. Although the 802.11 specification dictates backoff behavior, it leaves implementors to decide on auto-rate selection and retransmission rate fallback. 802.11 does not contain definitions for these algorithms because no algorithm will work in all wireless environments. For example an optimistic auto-rate selection may yield higher throughput on some links, but may also result in many errors on others. Our simulated results indicate Maranello can help increase the throughput from optimistic rate selection.

4.1 Maranello Increases Throughput for Popular 802.11 Cards

To characterize a variety of 802.11 backoff and retransmission rate fallback policies, we observed the retransmissions sent by three popular 802.11 cards. We ran the cards on Windows XP to observe the behavior of the most common driver. To analyze many instances of the card retransmitting its maximum number of retransmissions, we prevented the receiver from sending any acknowledgments. For each card, Figure 4 depicts the me-

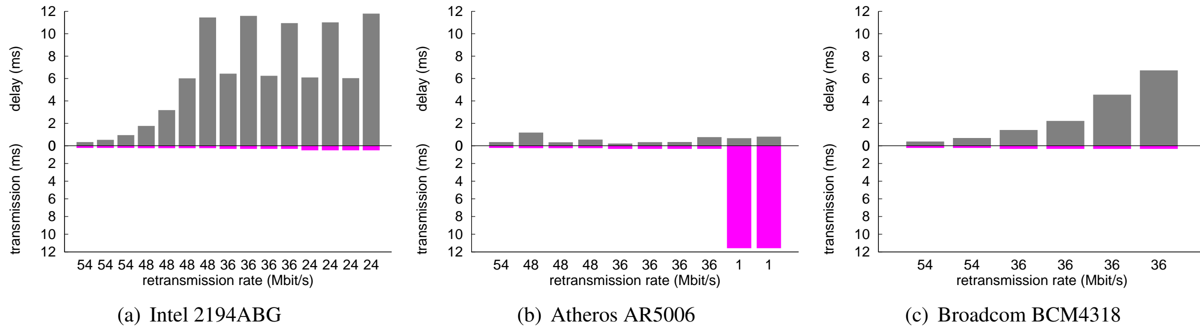


Figure 4: Popular 802.11 cards exhibit different exponential backoff behavior (top) and retransmission rate fallback (x labels show the rate, bottom bar shows transmission duration).

dian inter-retransmission delay and time to transmit for the observed retransmission rate fallback. For backoff, some cards appear to follow 802.11: Intel and Broadcom's median interval between retransmissions doubles for each retransmission. We did not observe Atheros doubling the backoff window after failed retransmissions.

Retransmission rate fallback also varies between cards. Each card appears to attempt a different number of retransmission rates (Intel 4, Atheros 4, Broadcom 2). Atheros does not experience much loss because the card will eventually attempt to retransmit a packet at the lowest possible rate defined in 802.11. Maranello helps Atheros because it will increase the probability that transmission is successful in the first few retransmissions, eliminating or at least reducing the size of retransmissions sent at the lowest bit rate. Intel retransmits at optimistic rates so it may need to retransmit more times than a card that quickly lowers the retransmission rate. For Intel, Maranello will help because it increases the probability of receiving a retransmission correctly, rewarding optimistic retransmission rate selection.

4.2 Trace-Driven Simulation

A trace-driven simulation of Maranello indicates that successfully retransmitting earlier increases throughput for several interpretations of 802.11. The simulator operates on a trace of packets with known payloads. Knowledge of the payload provides several desirable properties: (1) The simulator can determine the number of corrupted blocks in a packet. (2) The simulator can determine if the repair blocks fit inside a contiguous region of correct bits at the beginning of a (potentially corrupted) retransmission packet. (3) Resulting from (2) the simulator can subtract excess retransmissions seen after a successful repair. Table 2 shows the speedup obtained from simulating Maranello for the three popular cards. Intel appears to achieve significant gain because Maranello mitigates the errors caused by retransmitting at an optimistic rate, avoiding long, although standard, backoff times.

card	avg throughput	avg speedup	avg ERR	avg rate
Atheros	8.64	1.05	0.03	15.85
Broadcom	11.92	1.05	0.05	40.98
Intel	8.14	1.17	0.06	33.09

Table 2: In simulation Maranello increases throughput for the Intel chipset by correcting errors caused by optimistic behavior. ERR is the 64-byte block error rate.

4.3 Repair Size

Compared to other partial packet recovery protocols, Maranello does not need to send significantly larger repair packets. We simulated each of the repair protocols (Figure 5) with traces of data packets sent from a Broadcom card. To vary the bit error rate of these traces, we changed the distance between the sender and the receiver. The symbol size (1–216 bits) for symbol based repair (PPR) corresponds to the packet bit rate. We simulated an ideal version of ZipTx that assumes the indexes of corrupted bits are known, so it can pick the smallest number of redundancy bytes for the repair.

To repair corrupted bits, all of the repair protocols must send significantly more repair bits. For traces with a low BER, Maranello requires marginally more bits than the other repair protocols. ZipTx is able to retransmit so few bits because Reed Solomon works well when there are few bit errors. For corrupted packets with a high average BER, PPR's symbol-based repair needs to transmit the least number of bits to repair the packets. However, symbol based repair requires additional hardware to measure the confidence of symbols. Although Maranello needs more bits than symbol based repair, it requires fewer bits than ZipTx. If the packet contains errors clustered in one block, ZipTx's Reed Solomon will waste many repair bits for correct blocks because ZipTx chooses its coding rate based on the BER of the most corrupted block.

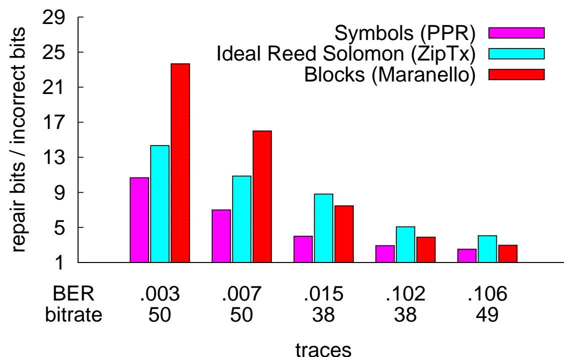


Figure 5: To repair corrupted packets with a high average BER, Maranello uses fewer repair bits than ZipTx. For low-average-BER corrupted packets, Maranello uses more repair bits than the other techniques. The bitrate shown is the average rate chosen by minstrel.

5 Implementation

We implement Maranello using OpenFWWF [7] open firmware and b43 Linux device driver [1] for Broadcom chipsets. In the following, we first discuss why several other potential platforms are not suitable for Maranello. We then present the implementation details of Maranello.

5.1 Why Other Platforms Are Unsuitable

To use the airtime reserved for ACK frames, receivers must construct and send NACK frames within SIFS, which is the defined inter-frame space between data packets and ACK frames [9]. Commercial 802.11 wireless NICs implement this time-critical operation in firmware or hardware.

5.1.1 Driver space of 802.11 wireless NICs

Recently, several wireless research platforms, such as FlexMAC [15] and SoftMAC [21], have been proposed to develop new MAC protocols. They are extensions of the MadWifi driver [18] for Atheros chipsets which runs in Linux kernel space. To determine how fast an implementation in driver space can send back NACK frames for corrupted frames, we perform the following experiment. When the test receiver gets a corrupted packet, it copies the first 100 bytes directly into a NACK frame, and sends it out immediately without performing backoff and using SIFS. From packets traced by a monitor node, we found that the minimum gap between the data packets and NACK frames is higher than 70 μs. This delay is mainly caused by bus transfer delay and interrupt latency and is consistent with the measurement results in Lu et al. [15]. This high latency makes the driver space unsuitable for the implementation of Maranello. Jitter due to DMA transfers makes timing too variable.

5.1.2 GNU Radio

GNU Radio platforms are slow and expensive. However, due to their flexibility, they have attracted increasing attention from the wireless research community and there are 802.11 implementations for them [22, 26]. In GNU Radio, the wireless signal is decoded at the host machine and the delay, depending on the length of the packets, is usually higher than 1000 μs [22]. The decoder could be put into the FPGA (Field-Programmable Gate Array) on the Universal Software Radio Peripheral (USRP), but the FPGA is much slower than the digital signal processor on the wireless NICs. Moreover, another challenge is to generate NACK frames for corrupted packets within SIFS, which is difficult to implement on these platforms.

5.2 Maranello Implementation

We first briefly introduce OpenFWWF and review the architecture of wireless device drivers in Linux kernel. Then we present the implementation of Maranello, focusing on NACK generation and repair packet construction, which are time-critical operations implemented in the firmware. Finally, we describe other operations implemented in the Linux driver.

5.2.1 Background

A microprocessor executes a typically proprietary microcode (firmware), written in assembly language, that handles various operations on wireless cards. OpenFWWF [7] attempts to replace the proprietary firmware with an open source firmware for Broadcom chipsets. It can support almost all the 802.11 primitives in the 2.4GHz frequency band. By changing the standard code path, it is possible to implement from scratch a completely different channel access mechanism, subject to a few basic hardware constraints, such as the PHY layer carrier sensing, the CCK and OFDM modulation schemes.

To better understand how the Maranello implementation works, we briefly review in the following the basic building blocks that equip the Broadcom chipset. The internal microprocessor drives the data exchange between different blocks using two main paths: transmit (TX) and receive (RX). The firmware is built as a main loop that reacts on external conditions such as a new frame's arrival from the air, a channel free indicator, and (programmable) timer expiration. The basic building blocks include:

TX and RX FIFO queues – The microprocessor pulls frames from the TX queue and moves them into the serializer when a transmission opportunity comes. On the opposite path, it moves a received frame from a buffer into the RX queue and raises an IRQ so that the host kernel can retrieve the frame.

Internal shared memory (SHM) – The microprocessor maintains several state variables which can be monitored or even changed by the host kernel.

Template RAM – The microprocessor can compose an arbitrary frame in this memory and transmit the resulting packet as if it came from the TX FIFO.

Internal registers and external conditions (EC) – The microprocessor sets these hardware registers in response to changes in the EC to program the radio interface and set up timers.

The current Linux kernel uses mac80211 [17] for device driver development. mac80211 is an abstraction layer that bridges between the kernel's networking stack and almost all the low-level wireless device drivers. For example, the rate control algorithms are usually implemented in mac80211 and shared by all the drivers. These drivers then act as stage-two bridge since all the 802.11 low level operations, such as retransmissions, acknowledgments, and virtual carrier sense, must be performed by either firmware or hardware, due to hard timing constraints that can not be met by a host-controlled approach.

5.2.2 NACK generation

As we mentioned in Section 3, to compose the NACK, the receiver computes block checksums for corrupted frames in the firmware. Due to hardware limitations, the Maranello block size should be a multiple of 32 bytes. We use 64 bytes as the block size. Longer blocks increase computation efficiency and shorten NACKs, while shorter blocks are parsimonious with repair bytes. In our experience, the 64-byte block represents a good compromise at typical rates, though we discuss possibilities for dynamic adjustment in Section 7.

For some transmission rates, a Maranello NACK uses more airtime than a MAC ACK, which may cause problems in the presence of hidden terminals. The size of an ACK frame is only 14 bytes. A Maranello NACK frame, based on 64-byte blocks, is at most 96 bytes longer than an ACK frame (4-byte checksum for each block, 24 blocks maximum). For a Maranello link, a hidden terminal of the receiver may hear from the transmitter the network allocation vector (NAV) and the earliest time it can start its own transmission is DIFS, 50 μs for 802.11b/g, after the end of NAV (suppose its backoff time is 0). There will be no collision when NACK's bit rate is higher than 12 Mbps. Otherwise, a transmission from the hidden terminal may collide with our NACK frames, which causes the retransmission of the whole packet. Preliminary experiments on a hidden terminal topology, indicate that even in this scenario, enabling Maranello can increase overall throughput.

5.2.3 Repair packet construction

Maranello transmitters must handle both ACK and NACK frames.

- Like 802.11, after a transmitter sends an original data packet or a recovery packet, it will set up an ACK timer.
- If the transmitter gets an ACK frame from the receiver, it will release the resource allocated for the original or recovery packets.
- If the transmitter gets a NACK frame from the receiver, it divides the original packets into blocks, computes the checksums for these blocks, and only retransmits the blocks whose checksums do not match those in the NACK. In practice, the block checksums are precomputed in the driver on the host processor.
- After the transmitter's ACK timer expires, and it does not receive a frame, but it previously attempted to repair the packet, it retransmits the repair packet. Otherwise it retransmits the whole packet

After a transmitter gets a NACK, it compares the received block checksums with the locally computed checksums and decides which block to retransmit inside a repair packet. We always retransmit the first block of a packet, which contains the important headers of various layers. For a repair packet, we reuse the 8-byte LLC header, only for data frames, by (1) changing the first byte to distinguish repair packets from other packets; (2) using the following 3 bytes as a bitmap of retransmitted blocks; and (3) appending an extra checksum (CRC-32 or Fletcher-32) in the last four bytes. The receiver uses this checksum, as an extra measure of safety, to verify that the recovered packet is correct.

Maranello uses the same 802.11 retry limit; each repair packet will increase the retry counter by one. Also before transmitting repair packets, it doubles the contention window.

5.2.4 Driver functionality

We implement non-time-critical operations in the driver, including the pre-computation of block checksums at the transmitter, and the reconstruction of frames at the receiver. We compute the block checksums for data packets in the driver, because the CPU on the host machine is much more powerful than the microprocessor of the wireless card. Checksums are sent to the firmware with each data packet. After the transmitter receives a NACK frame, its firmware can use these checksums directly, without recomputation. Checksums computed at the transmitter are used only to match those in the NACK frames and they are not transmitted. The receiver's driver combines a buffered corrupt packet with a correct recovery packet to reconstruct the original. Recovered packets

that cannot pass the extra Fletcher-32 checksum test are discarded.

6 Evaluation

In this section, we evaluate the throughput and latency performance of Maranello in implementation, isolate the factors that reduce recovery time, and run Maranello alongside unmodified 802.11 senders to ensure cooperative interaction.

We used 802.11b/g channels 1, 6, and 11 in environments with active APs and stations. This experimental approach has the advantage of injecting real-world interference and collisions as sources of packet error, but has the disadvantage of reducing the repeatability of experiments since contention varies. We enable auto rate feedback for all of the experiments and use Linux "minstrel" [19] as the rate control algorithm, which supports multiple rate retries and is the only rate control algorithm enabled in the Linux kernel 2.6.28 and above. (Our driver implementation is in 2.6.29-rc2.)

6.1 Maranello Increases Link Throughput

In the following, we show that Maranello can increase throughput for UDP traffic. We construct testbeds in three different environments: an industry research lab, a home, and a university building. We run Iperf [10] on randomly selected links from these testbeds to generate a CBR UDP stream to saturate the wireless channel. We focus on UDP to isolate link capacity from TCP dynamics.

We compare the throughput of Maranello and unmodified 802.11 in Figure 6. In these plots, the x-axis represents the throughput of 802.11 and the y-axis is the throughput of Maranello. Each point represents a pair of one-minute executions of Iperf, typically separated by less than 15 seconds. This separation is needed because we reload the firmware and driver, set up wireless interfaces, and initialize minstrel's bit rate table. Each point belongs to a group of ten points collected from randomly selected sender and receiver locations. In other words, we collected ten points, moved the receiver or sender station to another location, collected ten points again, and repeated. These figures include 370 (industry research lab), 390 (home), and 1000 (university building) points. The position of a point indicates the apparent throughput gain. For example, if a point is on the line marked "2X", the throughput gain is 2. We divide the points into 5 regions based on their throughput gain and show the percentage of points in each region in these figures. A point on a line is counted in the region above that line.

Figure 6 shows that Maranello can increase the throughput for UDP traffic; often by 30% or more. The university building environment shows higher throughput gain, because of increased contention and poorer channel conditions, than those observed in the other environments. There are more than 10 access points deployed for each of the 802.11b/g channels, 1, 6, and 11, and they are used by many people. For the other environments, each channel usually has fewer than four access points and relatively few users. To estimate the variability in the measurement of throughput over adjacent intervals, we also compare the throughput of 802.11 with itself. We pair the throughput of two consecutive runs with 802.11 into a point. Figure 6(d) shows the results for experiments done in our office building. The uncertainty in the throughput of adjacent measurements of unmodified 802.11 appears comparable to those of measurements between 802.11 and Maranello. Put simply, Maranello does not appear to increase the variability in throughput performance.

6.2 Maranello Reduces Recovery Latency

We define latency in this context to be the interval between when the firmware fetches the pending packet from the head of the TX FIFO to when an ACK is received. This includes the time spent inhibited by carrier sense, waiting for a transmission opportunity, and represents the time that the device is occupied with the transmission of an individual frame. We randomly select a link, then run Iperf for one minute for Maranello and 802.11 separately to get the per-packet latency. We use the firmware to record the measured time directly using the internal board clock: a 64-bit counter incremented every microsecond.

One might consider alternate definitions of latency. One might ignore contention and backoff time required by CSMA/CA; even though the card is occupied in the process of transmitting a packet, no signal is yet being transmitted. Such would be appropriate for measuring peak performance. Alternately, one might consider the time to successful delivery and ignore cases when the ACK is lost; the transmitter, of course, is still occupied.

We plot the CDF of latency for packets that need retransmissions in Figure 7. To make the comparison clear, we omit the latency for packets without retransmission, and we plot the latency of only one configuration of sender and receiver locations (other configurations are qualitatively similar but not composable). Maranello can deliver 90% of the packets that need retransmission within a latency of 4.16 ms. In contrast, 10% of 802.11 recovery latencies are above 17.1 ms. The small modes near 16 and 32 ms for 802.11 represent low-rate retransmissions: The minstrel default retransmission rate fallback attempts retransmissions at the original rate twice, followed by 1 Mbit/s up to four times if need.

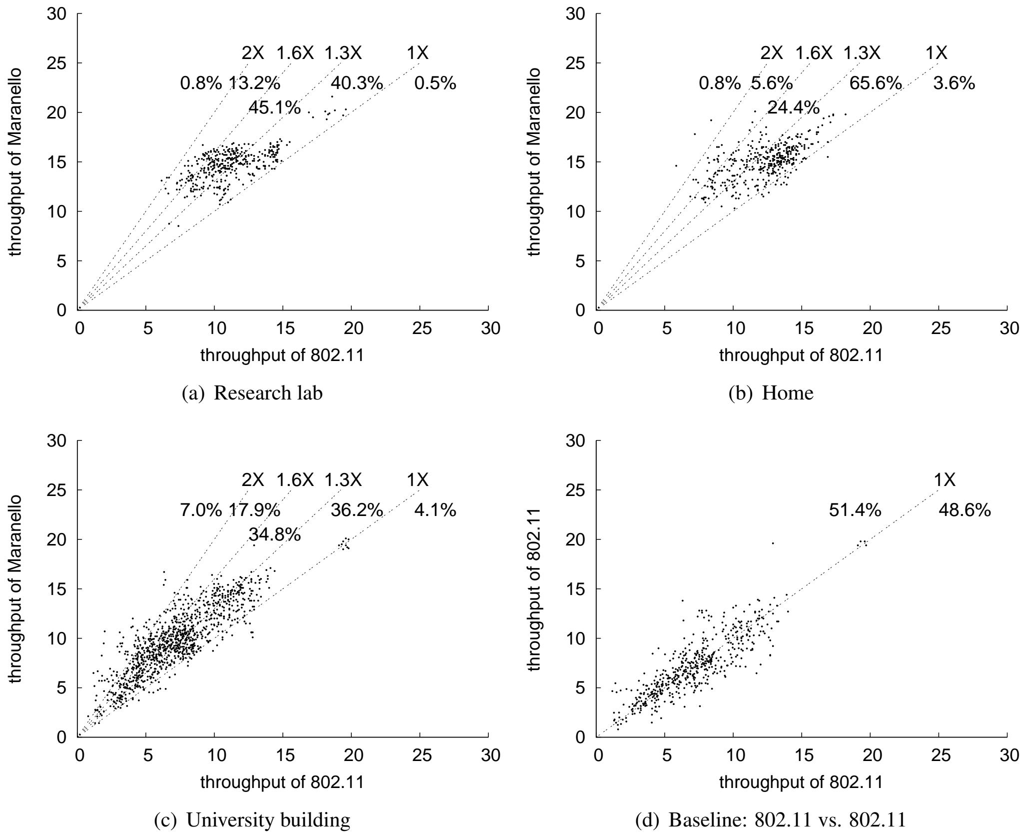


Figure 6: Maranello has a higher throughput than 802.11. Each figure compares 802.11 with Maranello in a different environment, or to show the uncertainty of the comparison, with 802.11 itself. Each point represents the performance of back-to-back one-minute UDP throughput measurements; ten points were collected for each configuration of sender and receiver stations.

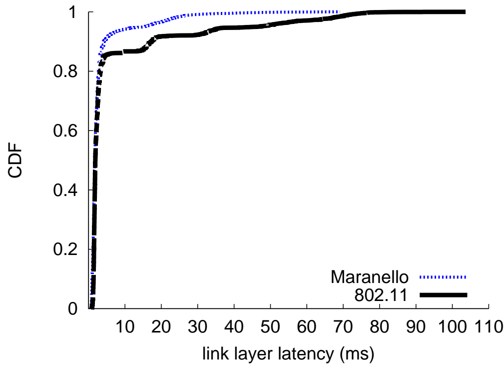
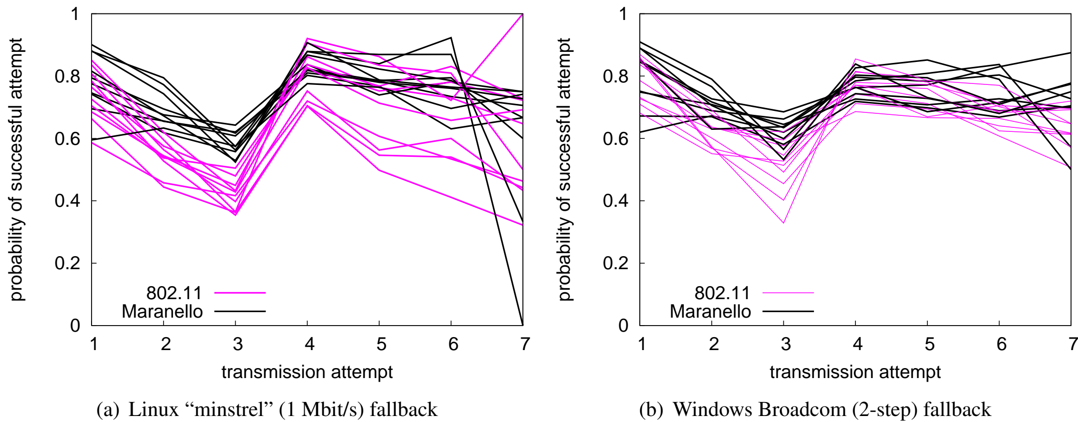


Figure 7: With block-based repair, Maranello recovers packets faster than 802.11's retransmissions.

6.3 The Sources of Throughput Gain and Latency Reduction

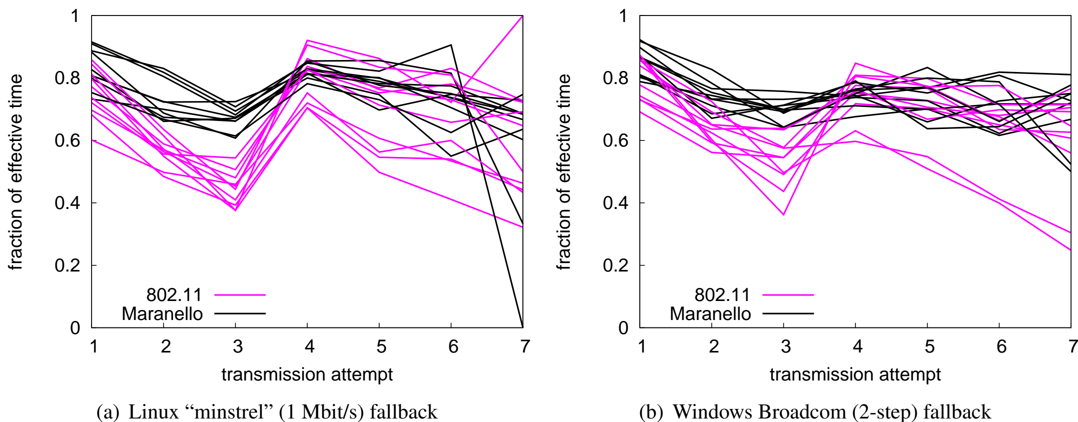
To break down the sources of performance improvement, we enhance the transmission status report for each packet with the following information: (1) whether a repair packet was used, (2) if used, at which attempt, and (3) the number of retransmitted blocks in the repair packet. The original report also includes (1) whether the packet is successfully delivered, (2) the number of attempts, (3) the bit rate used for the packet. With this information, we can calculate the delivery probability at each attempt, the transmission airtime and the number of transmitted bytes for each attempt. We run Iperf for one minute for 10 randomly selected links and plot in Figure 8 the probability of successful attempt for two retransmission rate fallback schemes: Linux "minstrel" fallback which always uses 1 Mbps as fallback rate, and 2-step fallback which drops the bit rate selected by minstrel for the initial transmissions by 2 steps (if possible) and uses it as fallback rate. The two-step fallback selection emulates the Broadcom driver for Windows XP (Section 4.1). In this figure, the x-axis is transmission attempt. The retry limit of Broadcom cards is 7, 1 initial transmission, and at most 6 retransmissions. The y-axis is the probability that an attempt can succeed.

Figure 8 shows that the probability of successful re-



(a) Linux “minstrel” (1 Mbit/s) fallback (b) Windows Broadcom (2-step) fallback

Figure 8: Maranello can successfully retransmit a packet earlier than 802.11. Each line represents a link measured either with 802.11 or Maranello; the probability that Maranello’s recovery packets are delivered is typically higher.



(a) Linux “minstrel” (1 Mbit/s) fallback (b) Windows Broadcom (2-step) fallback

Figure 9: Maranello can use airtime more effectively for packet transmissions. Each line represents a link measured either with 802.11 or Maranello; Maranello spends more time transmitting bits not yet correctly received.

transmission for Maranello is usually higher than that of 802.11. Because the retransmission rate fallback does not budge for the first two retransmissions, the probability of successful retransmission can be thought of as the conditional probability that, given a packet (or two) recently failed to be delivered at the chosen rate, this next transmission at the same rate will be delivered. Not surprisingly, for 802.11, this probability descends more steeply than for Maranello. Maranello, in contrast, can send shorter repair packets, which are less likely to be corrupted [8], even at the original bit rate.

The delivery probability increases at the fourth attempt because the firmware reduces the bit rate for the last four attempts. The successful attempt probabilities for the first three attempts are more important, because most packets can succeed at the first two retransmissions. The estimate of the delivery probability for the seventh attempt (after three previous attempts at 1 Mbit/s) is uncertain due to the dearth of data. For example, the 7th attempt that had 0.0 delivery probability of Maranello, only one packet was transmitted seven times. For the 7th attempt with 1.0 delivery probability of 802.11, there were 5 packets transmitted 7 times and all succeeded at this last attempt.

We also plot the fraction of effective time for each transmission attempt in Figure 9. Effective time represents the time spent transmitting correct blocks; Maranello can use airtime more effectively, because the correct bits in corrupted packets may be combined with recovery packets to reconstruct the original packets and the transmission time of these correct bits is effective.

6.4 Deployment on Access Points

To show that Maranello can increase overall network performance and does not interact poorly with unmodified 802.11 devices, we deploy Maranello on Linksys wireless routers running OpenWRT [23]. We associate two desktop stations, A and B, with the Maranello AP. We

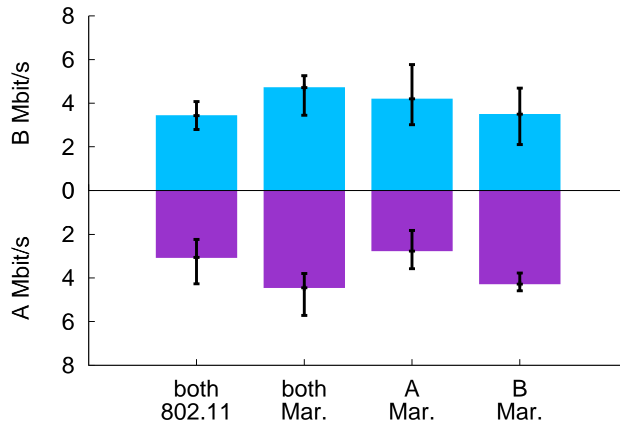


Figure 10: With two clients sending to an AP, on average, Maranello increases their individual and overall throughput. Error bars indicate min and max for five one minute runs.

run four types of experiments: A and B both running Maranello, both running 802.11, A running Maranello and B running 802.11, and vice versa. We connect a third station, C, to the AP using Ethernet, to act as an Iperf server. We do not run the Iperf server on the AP directly due to its limited CPU power. During a single one-minute experiment, A and B send UDP packets to C as fast as they can. Although experimenting with downlink traffic might be more typical of access point use, in that situation, that AP would be the only transmitter and would not show how Maranello transmitters interact with unmodified 802.11 transmitters.

Figure 10 plots the throughput of these two stations using a stacked bar graph. There are two key notes. First, running Maranello does not decrease the performance of the unmodified 802.11 station. That is, Maranello does not "cheat" the existing station of throughput. Second, when both stations run Maranello, the throughput is significantly increased for both stations. An interesting observation is that it appears not to help A or B to individually run Maranello when in contention. (The results in Section 6.1 imply that each station gains individually when running Maranello without a persistently competing station.) We plan to investigate this surprising result that Maranello is more social than selfish when competing with an unmodified station.

7 Discussion

In this section, we discuss how Maranello can be complementary with frame aggregation, which is used in 802.11n, and how the block size affects the performance of Maranello.

7.1 Frame Aggregation and Maranello are Complementary

To increase throughput, 802.11n reduces the 802.11 protocol overheads, such as interframe spacing, PHY layer headers and acknowledgment frames, by aggregating data packets into jumbo frames. Aggregated packets that are received incorrectly are indicated in a block acknowledgment which is sent back to the transmitter. The transmitter can then send a new chunk that contains only the corrupt packets. Even though only part of a packet may have errors in it, 802.11n frame aggregation must retransmit whole packets: correctly received bits are wasted.

Frame combining can improve throughput, but it also significantly increases latency, as senders must wait to aggregate enough frames to fill a jumbo frame. Block acknowledgments provide a complementary aggregation of feedback for 802.11n, where ACKs may be buffered together and sent as a group, similarly increasing per-packet latency. Maranello is complementary with these frame aggregation techniques because by repairing corrupted aggregated packets, Maranello can further increase link throughput.

7.2 Optimal Block Size

The Maranello block size is 64 bytes, primarily because it is the smallest multiple of 32 that can be supported by hardware (Section 5.2.2). A larger block size would increase computation efficiency somewhat and shorten NACKs, which may be useful at low bit rates. When the error rate is low, however, larger blocks may lead to repair packets with unnecessary extra bytes, wasting channel time.

We consider an interesting future direction of research to be dynamically adjusting the block size. The ideal block size may vary based on an estimate of wireless channel conditions and the bit rate chosen by the transmitter, which determines the bit rate of the acknowledgment and thus the transmission time of the NACK. When the NACK is transmitted at a low rate, it may be better for global throughput to keep NACK transmissions short than to be precise about the blocks in error. A similar tradeoff exists in the FEC systems between the coding rate of error correction bits and recovery efficiency. Another approach to determine the optimal block size that we intend to explore is to use theoretical models of wireless communication errors [13, 29].

8 Conclusion

In this paper, we design, implement, and evaluate Maranello, a practical partial packet recovery protocol for 802.11 wireless networks. Maranello has the following features simultaneously: (a) it introduces no extra

bits in correct transmissions, (b) it reduces recovery latency, except in rare cases, (c) it is compatible with the 802.11 protocol, and (d) it can be incrementally deployed on widely available 802.11 devices.

We implemented Maranello using OpenFWWF open source firmware. This implementation, and Maranello's compatibility with 802.11, allowed us to test in three different, live environments over heavily used 802.11b/g channels where contention and interference are realistic. We found significant throughput gains when running Maranello over 802.11 in consecutive intervals. We also installed Maranello on access points running OpenWRT to demonstrate that Maranello does not compete unfairly with unmodified 802.11 devices and that the processing requirements of Maranello do not preclude performance improvement. Moreover, we evaluate Maranello's performance compared to recently-proposed recovery protocols using a trace-driven simulation.

9 Acknowledgment

We thank the anonymous reviewers, Daniel Halperin, Aravind Srinivasan and Dave Levin, and our shepherd Srini Seshan, for insightful comments and discussion. Vinko Erceg answered our questions about Broadcom WiFi chipsets. This work was supported in part by NSF ITR Award CNS-0426683, NSF Award CNS-0626636 and NSF Award CNS-0643443. Part of this work was done when Bo Han was a summer intern at AT&T Labs – Research.

References

[1] b43 linux kernel driver for broadcom chipsets. http://linuxwireless.org/en/users/Drivers/b43/.

[2] G. Bianchi, *et al.* Experimental assessment of the backoff behavior of commercial IEEE 802.11b network cards. In *INFOCOM*, 2007.

[3] S. Choi, Y. Choi, and I. Lee. IEEE 802.11 MAC-level FEC scheme with retransmission combining. *IEEE Transactions on Wireless Communications*, 5(1):203–211, 2006.

[4] H. Dubois-Ferriere, D. Estrin, and M. Vetterli. Packet combining in sensor networks. In *SenSys*, 2005.

[5] J. G. Fletcher. An arithmetic checksum for serial transmissions. *IEEE Transactions on Communications*, 30(1):247–252, 1982.

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

[7] F. Gringoli and L. Nava. Open firmware for WiFi networks. http://www.ing.unibs.it/openfwwf/.

[8] B. Han and S. Lee. Efficient packet error rate estimation in wireless networks. In *TridentCom*, 2007.

[9] IEEE std 802.11, 2007.

[10] Iperf. http://sourceforge.net/projects/iperf/.

[11] A. P. Iyer, *et al.* Fast resilient jumbo frames in wireless LANs. In *IWQoS*, 2009.

[12] K. Jamieson and H. Balakrishnan. PPR: Partial packet recovery for wireless networks. In *SIGCOMM*, 2007.

[13] A. Kopke, A. Willig, and H. Karl. Chaotic maps as parsimonious bit error models of wireless channels. In *INFOCOM*, 2003.

[14] K. C.-J. Lin, N. Kushman, and D. Katabi. ZipTx: Harnessing partial packets in 802.11 networks. In *MOBICOM*, 2008.

[15] M.-H. Lu, P. Steenkiste, and T. Chen. Using commodity hardware platform to develop and evaluate CSMA protocols. In *WinTech*, 2008.

[16] M.-H. Lu, P. Steenkiste, and T. Chen. Design, implementation and evaluation of an efficient opportunistic retransmission protocol. In *MOBICOM*, 2009.

[17] Linux kernel mac80211 framework for wireless device drivers. http://linuxwireless.org/en/developers/Documentation/mac80211/.

[18] Madwifi linux kernel driver for WLAN devices with Atheros chipsets. http://madwifi-project.org/.

[19] Minstrel rate control algorithm. http://linuxwireless.org/en/developers/Documentation/mac80211/RateControl/minstrel/.

[20] A. Miu, H. Balakrishnan, and C. E. Koksal. Improving loss resilience with multi-radio diversity in wireless networks. In *MOBICOM*, 2005.

[21] M. Neufeld, *et al.* SoftMAC – flexible wireless research platform. In *HotNets-IV*, 2005.

[22] G. Nychis, *et al.* Enabling MAC protocol implementations on software-defined radios. In *NSDI*, 2009.

[23] OpenWrt linux distribution for embedded devices. http://openwrt.org/.

[24] P. S. Sindhu. Retransmission error control with memory. *IEEE Transactions on Communications*, 25(5):473–479, 1977.

[25] J. Stone, M. Greenwald, C. Partridge, and J. Hughes. Performance of checksums and CRC's over real data. *IEEE/ACM Transactions on Networking*, 6(5):529–543, 1998.

[26] K. Tan, *et al.* Sora: High performance software radio using general purpose multi-core processors. In *NSDI*, 2009.

[27] G. Woo, P. Kheradpour, D. Shen, and D. Katabi. Beyond the bits: Cooperative packet recovery using physical layer information. In *MOBICOM*, 2007.

[28] B. Zhao and M. C. Valenti. Practical relay networks: A generalization of Hybrid-ARQ. *JSAC*, 23(1):7–18, 2005.

[29] M. Zorzi, R. R. Rao, and L. B. Milstein. Error statistics in data transmission over fading channels. *IEEE Transactions on Communications*, 46(11):1468–1477, 1998.

[30] J. Zweig and C. Partridge. TCP alternate checksum options. IETF RFC-1145, 1990.