Revisiting Partial Packet Recovery in 802.11 Wireless LANs

Jin Xie, Wei Hu, Zhenghao Zhang
Computer Science Department
Florida State University
{xie,hu,zzhang}@cs.fsu.edu

ABSTRACT

In wireless LANs, partial packets are often received which usually contain only a few errors. According to the current 802.11 standard, such packets have to be retransmitted. Much effort has been invested recently in repairing such packets without retransmitting the entire packet, e.g., by using error correction (EC) code or retransmitting only the corrupted blocks. In this paper, we revisit this problem, and propose Unite, a framework for more efficient partial packet recovery. Unite is motivated by two key observations. First, the two repair methods, i.e., the EC-based and block-based, are not mutually exclusive and can be combined to achieve higher performance. Second, the recently introduced error estimators can be utilized to determine the optimal repair method depending on the condition of each partial packet. The combined method should achieve better performance than each individual method; in addition, it should remain low in complexity because the EC-based and block-based method are both simple in nature. Unite uses AMPS, a recently proposed estimator, for error estimation. We implement Unite on the Madwifi open source driver, and our experiments show that Unite outperforms other recovery schemes.

Categories and Subject Descriptors

C.2.2 [**Computer Systems Organization**]: Computer-Communications Networks

General Terms

Algorithms, Design, Performance

Keywords

Partial packet recovery, 802.11, Device driver

1. INTRODUCTION

Wireless Local Area Networks (LAN) are widely deployed for convenient access to the Internet. It is well known that wireless transmissions may result in partial packets, i.e., packets that have errors but still contain much correct information. Many recent works [3, 14, 5, 11, 4, 19] attempt to exploit partial packets for higher link efficiency. An appealing approach for 802.11 LANs, given the abundance of the already deployed hardware, is to extend the software [3, 19] or firmware [14] to handle partial packets. Existing solutions fall into two categories: those based on error correction code (EC-based) and those based on block retransmission (block-based). In the EC-based approach, the sender divides the packet into blocks and encodes each block into *codewords* according to an error correction code, and transmits parity bytes for each block to correct the corrupted data bytes. In the block-based approach, the sender divides packets into blocks and retransmits only the corrupted blocks, where a block is found to be corrupted if it fails the checksum test.

In this paper, we also focus on software extensions on top of commodity hardware for the exploitation of partial packets. We revisit this problem, because of two key observations. First, the EC-based and the block-based approaches are not mutually exclusive; rather, they should complement each other in many ways. A combined approach should achieve better performance than each of the individual approaches, and should not incur high complexity because the individual approaches are both simple in nature. Second, very recently, error estimators such as EEC [12] and AMPS [20] have been proposed that can estimate the number of errors in a received packet. The estimator provides key information which was not available to the earlier partial packet recovery schemes, and can be used to improve the performance by making intelligent decisions on the selection of the repair methods and repair parameters. For example, typically, a large percentage of partial packets have few corrupted blocks where each block has very few errors. In the block-based approach, the sender can locate the corrupted blocks and retransmit such blocks, in which often as high as 95% of the retransmitted bytes are unnecessary. An EC-based approach, with no knowledge of the error locations, will have to transmit parity bytes for all codewords and decode all codewords, while the majority of the transmission and decoding are unnecessary. Combining them, also assisted by the error estimator, leads to optimized repair method for such packets. To be more specific, if the estimated number of errors is small, the sender can locate the corrupted blocks by the checksum test and send parity bytes only for such blocks to correct errors in them. This saves both the amount of the transmitted data and the decoding time.

More fundamentally, the block-based approach may only be able to reap part of the potential gain [3] because it is still based on *repetition coding* which is inferior to more advanced coding schemes [1]. For example, the retransmitted blocks are still subject to the same channel condition and may be corrupted. The EC-based approach has the potential of achieving higher gain [3]; however, we find through our experiments that the computational resource may become the bottleneck, especially at high data rates because the decoding of an error correction code is computational intensive. Therefore, we propose Unite, a framework for optimized partial packet recovery considering the CPU load. Unite supports three repair methods. Among them, *Targeted Error Correction* (TEC) is described earlier, i.e., sending parity bytes only for the targeted erroneous blocks when the number of errors is low. If the number of errors is higher, Unite may use *Holistic Error Correction* (HEC) which first spreads errors evenly in the packet through *interleaving* then sends just enough parity bytes to correct errors in each codeword. As the decoding of error correction code may exceed the computational capacity of the host CPU, Unite may also resort to *Block Retransmission* (block-retran), which is basically the standard block-based approach.

Unite works under a CPU time constraint for decoding to avoid consuming excessive CPU resources. The key problem in the implementation is to determine the best repair method for the partial packets to minimize the number of transmitted bytes under this CPU constraint. The choice depends on many issues, such as the number of errors in the packet, the distribution of errors in the packet, the receiver's CPU load, etc. In addition, some of the information is only an estimate and cannot be known exactly, such as the number of errors. To this end, we design a practical algorithm that makes such decisions based on the feedback from the receiver.

We implement Unite based on the Madwifi [9] open source driver. We adopt AMPS [20] as the error estimator, because AMPS has a good accuracy and a small overhead of 8 bytes per packet. We test Unite with extensive experiments and compare it with other drivers, including the original Madwifi driver, a two-round retransmission driver, a pure block-based driver, and a pure EC-based driver enhanced with AMPS. We found that for the majority of the times, Unite achieves higher performance than all other drivers. In addition, Unite consumes CPU resources under the specified constraint.

The rest of the paper is organized as follows. Section 2 discusses related works. Section 3 describes the design of Unite. Section 4 briefly describes AMPS. Section 5 describes the link protocol we implement to evaluate Unite. Section 6 describes the experiments. Section 7 discusses possible extensions and future works. Section 8 concludes the paper.

2. RELATED WORKS

Partial packet recovery has drawn much attention in recent years. Unite is built on the foundation of the earlier works, with key distinctions discussed in this section.

ZipTx [3] proposes an EC-based recovery scheme on the Madwifi driver. The key differences between Unite and ZipTx include the following. First, Unite supports more repair options, which helps achieving higher performance. Second, Unite considers the computational constraint and ZipTx does not. This is motivated by our experiments which show that software decoding can be computational intensive in a number of cases. More importantly, we note that the link speed in the physical layer is fast increasing, e.g., the speed in 802.11n is as high as 300 Mbps, while the network may host devices with less computational power, e.g., tablets and smart phones. The experiments in [3] were based on a maximum speed of 54 Mbps on a 3 GHz dual core CPU, which may underestimate the CPU load at future higher data rates and slower CPUs. Unite, by taking the computational resources into account, may be applicable to more scenarios. Third, Unite employs AMPS for error estimation which provides valuable information that can be used to select the optimal repair method and parameters, while the advanced error estimators were not available to ZipTx. Fourth, Unite uses interleaving which randomly distributes errors across the packet such that each codeword receives similar number of errors, while ZipTx does not process packets to cope with bursty errors.

Maranello [14] adopts the block-based approach by modifying the firmware in the wireless card driver. A negative-ACK (NACK) is sent in the place of an ACK when the packet is corrupted. The NACK contains the checksums of the blocks in the packet, with which the sender can immediately determine which block to retransmit. Maranello uses only the block-based approach while Unite uses both the EC-based and block-based approach. Unite is a software implementation while Maranello needs to modify the wireless card's firmware. Noting that not all wireless card manufactures expose the firmware design, software-based solution like Unite can be applied to more platforms and bring immediate benefits.

Many other partial packet recovery schemes have also been proposed. The block-based approach has been implemented in [19]. PPR [4], SOFT [5], and MIXIT [11] rely on physical layer hint for recovering the corrupted sections of the packet. The limiting factor is that not all physical layers report such hints to the upper layers. In [7, 8, 6], opportunistic overhearing at third-party nodes is exploited for packet recovery. In this work, we focus on the recovery between two nodes, and leave the extension to multiple nodes to the future work.

In [12], an error estimation coding (EEC) scheme was proposed, and was also incorporated into the Madwifi driver to assist rate selection. Basically, EEC reports the error ratio of a data rate, and the autorate algorithm selects one rate that optimizes the throughput. The modified driver in [12] is very different from Unite, because it does not repair partial packets and relies on the upper layers to carry out the error correction. On the other hand, Unite hides the partial packet recovery from the upper layers. As not all upper layers handle partially correct data, Unite can work more seamlessly with the upper layers.

3. UNITE DESIGN

We discuss the design of Unite in this section. We begin with a high-level overview of the packet transmission procedure.

3.1 Packet Transmission Procedure

A packet transmission procedure may consist of a maximum of three rounds:

1. In the first round, the sender sends only the data bytes. If the received packet passes the packet level checksum test, it is delivered to the upper layer immediately and the transmission of this packet finishes. If the packet is *erased* in this round, i.e., no information is received or the header is corrupted, the packet transmission fails. Otherwise, it is a partial packet and the receiver runs the error estimator to find the number of errors in the packet, and also calculates the checksums of the *checksum blocks* which will be defined shortly in Section 3.2. When gets the opportunity to send, the receiver sends a feedback to the sender containing the estimated number of errors and the checksums.

2. In the second round, the sender selects a repair method and sends the repair data, either the parity bytes or certain checksum blocks. The receiver attempts to repair the packet based on the repair data. If the repair is successful, the packet is delivered to the upper layer and the transmission of this packet finishes. Otherwise, if the repair method is block-retran, the packet transmission fails. If the repair method is TEC or

HEC, the receiver sends another feedback and asks for a third round of transmission.

3. In the third round, the sender retransmits the corrupted checksum blocks according to block-retran. After receiving the repair data, the receiver attempts to repair the packet. If the repair is successful, the packet is delivered to the upper layer and the transmission of this packet finishes. Otherwise, the packet transmission fails.

A packet may attempt the above procedure three times, after which it is dropped.

3.2 Code Block, Interleaving, and Checksum Block

To support multiple repair methods, Unite organizes the data in multiple manners, namely the *code block* and the *checksum block*, which also involves *interleaving*. These concepts are explained in the following:

- Code Block: To support HEC, Unite divides a packet into code blocks. Data bytes in the same code block can be encoded into one codeword according to the Reed-Solomon (RS) code, where a codeword is basically the original data bytes followed by parity bytes. Code blocks should be of the same size, 150 in our current implementation, except that the last block may be less depending on the size of the packet. If encoding is needed, all code blocks are encoded with the same RS code. The RS code is used because of its strong error correction capability [2], as well as the availability of software implementations [13].

- Interleaving: In the first round, the fresh data undergoes an *interleaving* procedure, in which each byte is relocated to a random location based on a random permutation. At the receiver side, before attempting to run the estimator, the bytes undergo *deinterleaving*, i.e., the reverse of the random permutation, and are mapped to their original locations. Interleaving is a common technique in wireless communications to cope with bursty errors. As the errors in a wireless frame tend to be clustered in a few locations, without interleaving, it could happen that there are many errors in one received code block while very few in others. The interleaving procedure spreads the errors evenly across all codewords and significantly reduces the probability of such events.

- Checksum Block: To support block-retran and TEC, Unite divides a data packet into checksum blocks. In our current implementation, each checksum block is 64 bytes; the last block may be less depending on the size of the packet. Note that unlike the HEC which prefers the errors to be spread out evenly in the packet, TEC and block-retran prefer the errors to be clustered in as few blocks as possible, such that less parity bytes or less blocks need to be retransmitted. Therefore, the checksum blocks are defined based on the packet *after* the interleaving step.

For example, Figure 1 illustrates the definitions of the blocks, as well as the effectiveness of interleaving in spreading errors, for a packet with 8 bytes indexed from 0 to 7.

3.3 CPU Decoding Time Constraint

One of the key features of Unite is that it considers the constraint of CPU load. As a simple, qualitative test, we use the RS code implementation at [13] and measure the decoding time of RS codes

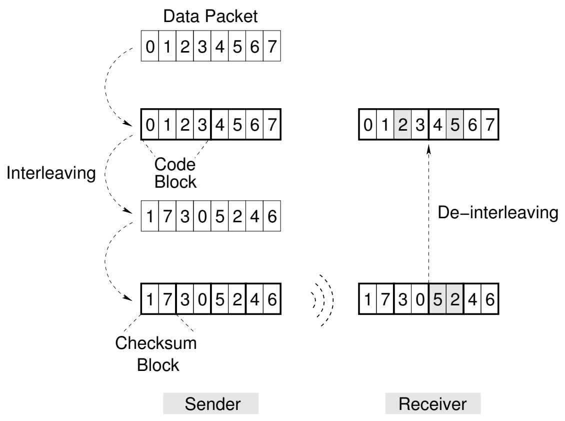


Figure 1: A simple illustration of the code block, interleaving, and checksum block. The shaded boxes represent bytes that have been corrupted.

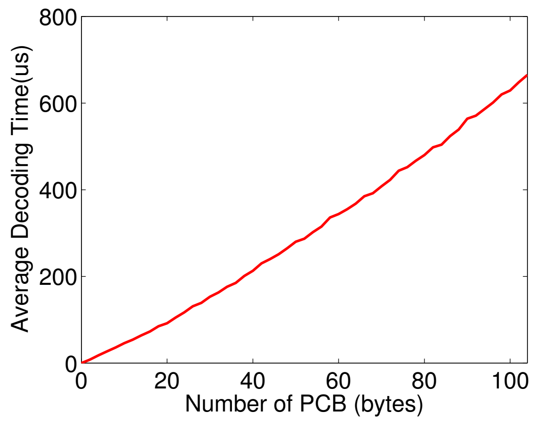


Figure 2: The average decoding time of RS code.

with code block size 255 and various number of parity checking bytes (PCB) on the Toshiba Satellite U405D-S2910 laptop with Dual Core 2.2GHz CPU running the Linux real time kernel, when there is no other application running on the machine. The average decoding time is shown in Figure 2, where it is clear that software decoding can be time consuming. The exact decoding time varies depending on the individual machine configurations; we provide a quantitative measure of CPU resource usage in Section 6.

3.4 Notations

We use β to denote upper limit of CPU time used in decoding. We use Y to denote the number of error bytes in a packet, and use Z to denote the maximum number of error bytes in a code block among all code blocks. Y and Z are random numbers, and their estimations are denoted as $\hat{y}$ and $\hat{z}$, respectively. The main notations are listed in Table 1.

3.5 Three Repair Methods

We have mentioned that Unite supports three repair methods. After giving the definitions of code block and checksum block, we now discuss these methods in more details.

3.5.1 *Block-Retran*

β	threshold of the CPU decoding time
Y	number of error bytes in a packet
$\hat{y}$	AMPS's estimate of Y
Z	maximum number of error bytes in a code block
$\hat{z}$	AMPS's estimate of Z

Table 1: List of Notations

The block-retran is the simplest among the three methods: the sender simply retransmits the corrupted checksum blocks. For the sender to locate the corrupted blocks, the receiver computes the checksums of the checksum blocks in the received packet, and sends them to the sender in the feedback. The sender computes the checksums of the same blocks based on the correct packet in its buffer, and locates a corrupted block if the two checksums do not match.

In our current implementation, the checksum we use is $x^{16} + x^{15} + x^2 + 1$ with the implementation available in the Linux kernel. We do not use the more popular and stronger 32-bit CRCs, because the 16-bit CRC has a reasonable error detection capability while requiring much less overhead. Note that a false negative in the block checksum test is not disastrous: even if an erroneous block passes the block checksum test, the reassembled packet still cannot pass the packet level checksum test and a retransmission can be attempted to recover the packet. Given that the false negative ratio of the 16-bit CRC is reasonably low, e.g., no more than 10^{-5} based on our tests, the amount of data saved by using the 16-bit CRC instead of the 32-bit CRC is far more than the amount of retransmitted data due to a false negative of the block checksum test.

3.5.2 HEC

With the HEC method, the receiver obtains $\hat{z}$ by AMPS, the estimate of the maximum number of errors in the code blocks, and sends it in a feedback. Based on $\hat{z}$, the sender sends parity bytes for the code blocks. The number of errors among the code blocks can be different; however, after interleaving, the variance should be minimized. Therefore, for simplicity, the same number of parity bytes are sent for all code blocks. After receiving the parity bytes, the receiver runs the decoding algorithm for each code block and reassembles the packet.

To be more specific, after getting $\hat{z}$, a code block is encoded into a codeword according to the $(255, 255 - 2\hat{z})$ RS code and the $2\hat{z}$ parity bytes are transmitted as the repair data. With $2\hat{z}$ parity bytes, the RS code guarantees that any no more than $\hat{z}$ errors in the received data bytes and parity bytes can be corrected. Choosing code adaptively based on the number of errors minimizes the time for decoding, because the decoding time monotonically increases with the number of parity bytes. It is likely that the codeword can accommodate more data bytes than the size of the code block; in this case the codeword is *shortened*, i.e., when encoding, dummy 0s are added to the beginning to the data bytes which need not be transmitted.

AMPS first obtains $\hat{y}$, then obtains $\hat{z}$ with a table look up. In our current implementation, a 1500-byte packet is qualified for HEC only if $\hat{y} \leq 47$. This is because with more errors, the variance of AMPS's output is larger, which leads to more underestimations, which in turn leads to decoding failure, wasted CPU time and additional rounds of transmissions. Setting such a threshold can significantly reduce the underestimation. In addition, for a 1500-byte packet to be repaired with HEC, we further process the output of AMPS according to Table 2 where we inflate $\hat{y}$ in some cases, as

$\hat{y} \leq 13$: $\hat{y} \leftarrow 13$	$\hat{y} \in [14, 18]$: $\hat{y} \leftarrow 18$
$\hat{y} \in [19, 23]$: $\hat{y} \leftarrow 23$	$\hat{y} \in [24, 29]$: $\hat{y} \leftarrow 29$
$\hat{y} \in [30, 35]$: $\hat{y} \leftarrow 35$	$\hat{y} \in [36, 47]$: $\hat{y}$ unchanged

Table 2: Processing $\hat{y}$

this does not increase much transmitted data but helps reducing underestimation.

3.5.3 TEC

If TEC is used, the sender locates the corrupted checksum blocks by comparing the checksums, and transmits parity bytes *only* for such blocks. The receiver receives the parity bytes and runs the decoding algorithm to repair the corrupted blocks. The TEC method is targeted for packets with very few errors clustered in few blocks. In our current implementation, a 1500-byte packet is qualified for TEC if $\hat{y} \leq 3$. TEC is introduced because as shown in [3, 14], a large percentage of partial packets contain very few error bytes. For such packets, both HEC and block-retran are not efficient. For example, if there is only one error byte, HEC still has to run the decoding algorithm for all codewords while only one code block is corrupted; block-retran has to retransmit one entire checksum block in which only one byte is necessary. However, TEC needs only transmit few parity bytes to correct one error and run decoding for only one codeword.

The simplest TEC would be to encode each corrupted checksum block into one codeword. However, as the checksum block is small and the number of errors is likely small, multiple blocks can be grouped together and encoded into one codeword. In our current implementation, up to three corrupted checksum blocks may be grouped together into one codeword.

The number of parity bytes to be sent for each codeword depends on the estimated number of errors, and is a tradeoff between reducing decoding time and reducing the failure probability. The challenge is that the estimated number of errors may be different from the actual number of errors, even with the best estimator. In our current implementation, if $\hat{y} \leq 1$, each codeword has 10 parity bytes; else each codeword has 20 parity bytes. The codeword is shortened when necessary.

3.6 Choice of Repair Methods

With the CPU time constraint, it may not be possible to decode all partial packets, and it would be naturally more advantageous to use TEC or HEC for packets requiring less decoding time, and use block-retran for packets requiring more decoding time. As the packets may have arbitrary numbers of errors and corrupted blocks, an algorithm is needed to find the best repair method for each partial packet to optimize the system performance. The two resources to be managed are the data transmission time and the CPU time. The algorithm must take advantage of the fact that the packet reception and the decoding run in parallel at the wireless card and the host CPU, respectively. The challenge is that the number of errors in a partial packet is only an estimate and may not be the actual number of errors.

3.6.1 The Algorithm

In our current implementation, the algorithm is run at the sender every time a feedback is received and determines the repair methods for the packets in this feedback. It takes an input W as the current *decoding time budget*, which is the total CPU time allowed for decoding these packets. For a packet, it takes the following parameters as input:

- g: the decoding time if repaired by TEC, if the packet qualifies for TEC.

- q: the number of required parity bytes of TEC, if the packet qualifies for TEC.
- d: the decoding time if repaired by HEC, if the packet qualifies for HEC.
- r: the number of required parity bytes of HEC, if the packet qualifies for HEC.
- b: the total size of the corrupted checksum blocks.

We suppose all inputs are given for now and will discuss how to obtain them in Section 3.6.2.

It is clear that the decisions are easy for some packets. First, if a packet is not qualified for HEC, it can only be repaired with block-retran. Second, if a packet is qualified for HEC but not for TEC, while its repair data transmission time with HEC is greater than that with block-retran, it obviously should be repaired with block-retran as block-retran requires no decoding time. Therefore, the algorithm first filters such packets out as block-retran packets. Suppose there are a total of n remaining packets where we denote a packet as P_i for $1 \leq i \leq n$.

We note that although there are three repair methods, for any packet, there are actually only two options, which simplifies the scheduling algorithm design. Note that a packet is either qualified for TEC or not:

- If qualified, it should not be repaired with HEC because HEC will require more data transmission time and more decoding time, hence the packet is repaired either with TEC or block-retran.
- If not qualified, it cannot be repaired with TEC and clearly can only be repaired with HEC or block-retran.

As a result, the algorithm should make *binary* decisions for each packet between using error correction or block retransmission. We therefore introduce a binary variable x_i for packet P_i where $x_i = 0$ means it should use block-retran and 1 otherwise.

For packet P_i, we define a *value* and a *weight* denoted as v_i and w_i, respectively. If P_i is qualified for TEC, $v_i = b_i - q_i$ and $w_i = g_i$; otherwise, $v_i = b_i - r_i$ and $w_i = d_i$. Note that the value is basically the extra number of bytes in the repair data if block-retran is used instead of TEC or HEC. The total number of bytes, given $\{x_i\}_i$, is

$$\sum_{i=1}^{n} x_i(b_i - v_i) + \sum_{i=1}^{n}(1 - x_i)b_i = \sum_{i=1}^{n} b_i - \sum_{i=1}^{n} x_i v_i.$$

As we want to minimize the total number of bytes under the CPU time constraint, the problem can be formalized as

$$\max \sum_{i=1}^{n} x_i v_i$$

under the constraint that

$$\sum_{i=1}^{n} x_i w_i \leq W.$$

This is exactly the Knapsack problem which is NP-hard. We therefore employ a greedy algorithm: in every iteration, we select the packet that has the largest ratio of value over weight, and mark it as using TEC or HEC, until the decoding time exceeds W, or until all packets have been marked.

3.6.2 *Obtaining the Input Parameters*

The decoding time budget W is determined by the CPU time constraint β. To get W, at the receiver, we maintain γ which is the average time that the receiver encounters a partial packet. If the receiver receives a partial packet and the last partial packet is received t seconds ago, γ is updated as

$$\gamma \leftarrow (1 - a)\gamma + at$$

where $a = 0.01$ in our current implementation. Note that γ is basically an estimate of the maximum CPU time the receiver can spend to cope with a partial packet by decoding. As the decoding should not take more than β fraction of the CPU time, we let

$$W \leftarrow \beta N \gamma$$

where N is the total number of partial packets in this feedback. W is sent by the receiver to the sender in the feedback.

Finally, note that for a packet, q, r and b can be found based on the size of the repair data. The decoding time g and d depend on the number of parity bytes as well as the speed of the receiver. The number of parity bytes is known. When installing the driver, the receiver can measure the decoding time for all possible number of parity bytes. This table can be sent to the sender during the association phase, such that given the number of parity bytes, the sender can find the decoding time with a table look up. The measurement should be carried out when the receiver is not running other applications, in which case the average decoding time is a good upper bound of the CPU time used in decoding.

4. THE ERROR ESTIMATOR

The error estimator is crucial to Unite. To date, there are two error estimators proposed, AMPS [20] and EEC [12]. We use AMPS, because it has a very low overhead of 8 bytes per 1500-byte packet, and is reasonably accurate based on our experiments. We also made comparisons between AMPS and EEC, which show that AMPS outperforms EEC for estimating the number of errors in a packet at a lower overhead [20].

For completeness, we give a brief description of AMPS; the details can be found in [20]. AMPS is based on the idea of *Amplified Sampling*. Basically, the sender finds the parity bit of multiple randomly selected bytes, and uses it as a *sample*. Multiple samples are computed, which are sent along with the data packet, guarded by CRC such that they are error-free if received. The received data may contain errors. The receiver recomputes the samples based on the received data and finds X, the number of mismatches between the locally computed samples and the transmitted samples. AMPS estimates Y based on the observed value of X using the maximum a posteriori (MAP) estimation. That is, it selects $\hat{y}$ that maximizes $P(Y = \hat{y}|X = x)$ among all possible values Y for a given observation $X = x$. Given $\hat{y}$, AMPS also computes an estimate of Z. The estimate is selected such that $P(Z \leq \hat{z}|Y = \hat{y})$ is greater than a threshold, set to be 0.95 in our current implementation. In total, AMPS uses 64 samples packed into 8 bytes. All the major steps are precomputed and stored in tables such that with the value of X, both $\hat{y}$ and $\hat{z}$ can be obtained with a constant time table lookup. The size of the table has been optimized to meet the requirements of Linux kernel code.

5. THE LINK LAYER PROTOCOL

We adopt a link layer protocol similar to that in [3] to evaluate Unite, described in the following.

5.1 Aggregated Feedback

The receiver sends feedback about packet reception status to the sender. To reduce the overhead of sending individual feedbacks, the receiver may aggregate multiple feedbacks into one feedback frame. In our current implementation, by default, the receiver sends a feedback in three cases, whichever occurs earlier: (1) when received 8 partial or erasure packets, (2) 10 ms after sending the last feedback while received a partial packet or an erasure packet in this interval or (3) 20 ms after the previous feedback is sent while received no response for the feedback. These thresholds can be configured. To increase the probability of correctly delivering the feedback frame, it is always sent at two rates lower than the current data rate; if no such rate exists, the lowest data rate.

5.2 Sender Queue and Receiver Queue

The sender considers a packet delivered when it receives an 802.11 ACK from the hardware or a software ACK from the receiver in the feedback frame. The sender maintains a queue of packets that have been sent but have not been ACKed. Whenever the queue is not full, the sender may keep sending packets. If the queue is full, the sender retransmits packets in the queue if no feedback is received after a configurable timeout, 20 ms in our current implementation.

At the receiver, packets that are received correctly are delivered to the upper layer without delay. The received partial packets are stored in a queue. The receiver may have several partial packets in the queue because the sender may send several packets before receiving a feedback.

5.3 Sender Frame Format

The frame sent by the sender is referred to as the *sender frame*, the format of which shown in Figure 3(a). After the MAC header are the sender header, two CRCs, the information of the repair data, AMPS samples for the full data packet, the repair data, and the full data packet. If the size of the repair data field is too large, the frame may not have the AMPS and the full data packet field.

The details of the fields are explained in the following:

- In the sender header, the type field contains information such as the type of repair data in this frame and whether the frame contains a full data packet. The feedback sequence number field indicates for which feedback this frame is generated. The sequence number field is the sequence number of the full data packet in this frame, if any. The repair information number field indicates the number of partial packets to be repaired by this frame. The retransmission counter field indicates the number of times the full data packet, if any, has been retransmitted.

- In the repair information field, the packet sequence number is the sequence number of the packet that needs the repair data. It also contains the information of the repair method of this packet, such as the decoder to be employed for the error correction code, or the bitmap of the corrupted checksum blocks.

- CRC1 is the checksum of the MAC header and the sender header. CRC2 is the checksum of the repair information and AMPS samples. A frame failed either of the CRCs will be considered an erasure.

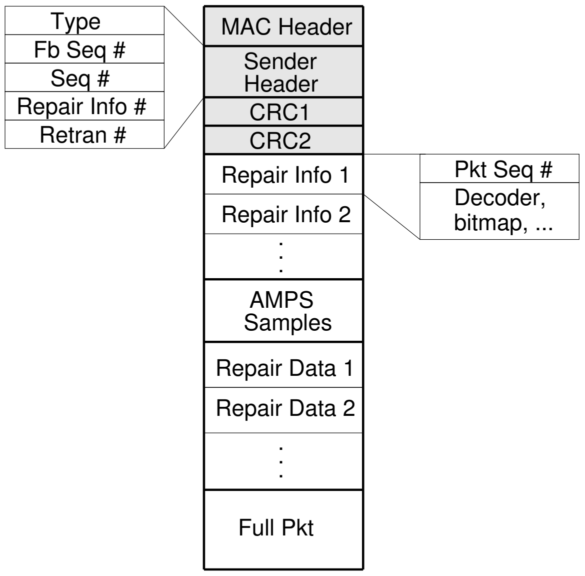
5.4 Feedback Frame Format

The feedback frame format is shown in Figure 3(b). After the MAC header are the feedback header field, the information of the repair method and parameter, and a list of checksums for each partial packet.

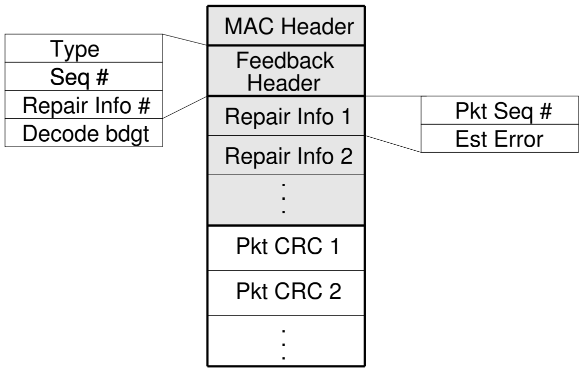
The details of the fields are explained in the following:

- In the feedback header, first is the type field indicating that the frame is a feedback frame, followed by the feedback sequence number, then followed by the number of packets whose information is carried in this frame, then followed by the decoding time budget for the packets in this frame.

- In the repair information field, the packet sequence number is the sequence number of the packet that needs repairing. It also contains information such as the estimated number of byte errors.



(a)



(b)

Figure 3: Frame format. The shaded fields are mandatory, others are optional. (a) Sender frame. (b) Feedback frame.

6. EXPERIMENTS

We implement Unite on the open source Madwifi driver [9] with the RS code implementation at [13]. The wireless card we use is the Cisco Aironet 802.11a/b/g wireless cardbus adapter[10]. We set the Madwifi driver in the *monitor* mode to allow the raw data frames to be delivered. The machines we use are two Toshiba Satellite

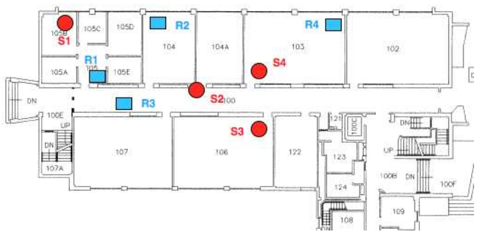


Figure 4: The sender and the receiver locations in four experiments. Circle: sender. Square: receiver.

U405D-S2910 laptop computers with 2.2GHz CPU with one core enabled. We classify packets as *erasures*, *partial packets*, and *correct packets*, the percentages of which are referred to as the *erasure ratio*, *partial ratio*, and *correct ratio*, respectively.

6.1 Comparing with Other Drivers

We first compare Unite with other drivers.

6.1.1 Compared Drivers

The compared drivers include:

- ORIG: The original Madwifi driver, with link retransmission enabled.
- BLCK: The original Madwifi driver enhanced with block-retran.
- EOLY: The original Madwifi driver enhanced with HEC and AMPS under the CPU time constraint.
- 2RND: The original Madwifi driver enhanced with a two-round, fixed packet repair schedule according to [3], with no CPU time constraint.

Among the compared schemes:

- BLCK is our implementation of the block-based driver.
- EOLY is an HEC-only driver complying with the CPU time constraint. It represents our approximation of the ZipTx driver [3] enhanced with AMPS under the CPU time constraint. It adopts two repair methods, either HEC or complete packet retransmission. A greedy scheduling algorithm similar to the scheduling algorithm used by Unite is used to select the repair method under the CPU time constraint: the packet with the largest ratio of packet size over the decoding time is selected for HEC in each iteration, until no packets are left or until the CPU time constraint is reached.
- 2RND implements the two-round fixed transmission schedule suggested in Section 5.2 of [3]: for a partial packet, in the first round, the sender transmits parity bytes 7% of the code block size; if the first round fails, in the second round, the sender transmits parity bytes 25% of the code block size. If both rounds of repair fail, this packet is retransmitted and error correction may be attempted again. Pilot bits are embedded in the data with which the Bit Error Ratio (BER) is estimated and packets with high BER (more than 0.3 in our current implementation) are not repaired with error correction. The 2RND driver is optimal based on the trace analysis in [3] when no information about the error number is available. It is, however, *not* the ZipTx driver which dynamically chooses the number of parity bytes [3].

Unless otherwise specified, the retransmission at the link layer is disabled because they are not as efficient as other repair methods. This may lead to an additional boost of performance except to ORIG because of the absence of additional exponential backoff upon a loss event; however, our main focus is the gain of Unite over other drivers that also disable link layer retransmission.

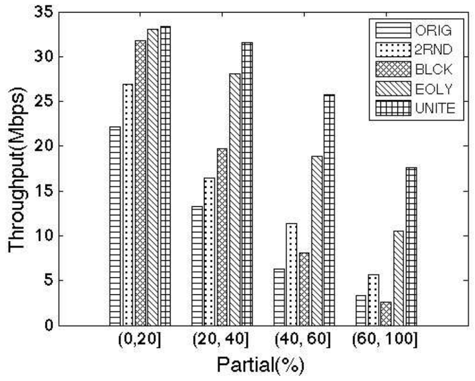


Figure 5: The measured link throughput of various drivers when the erasure ratio is no more than 0.2.

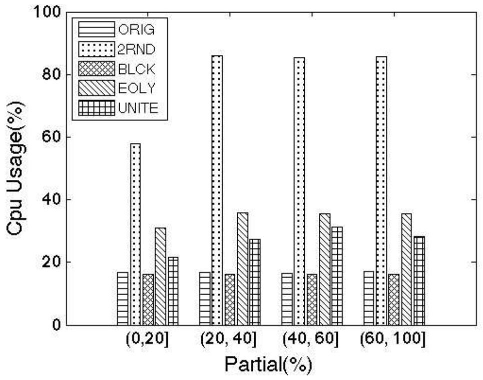


Figure 6: CPU load when the erasure ratio is no more than 0.2.

6.1.2 Experiment Setup and Methodology

We use one laptop acting as the sender and one laptop acting as the receiver where the receiver runs the real time kernel. We randomly choose 60 sender and receiver locations; for each location, each driver runs for 45 seconds at 54 Mbps. The sender and receiver locations in the some of the experiments, for example, are shown in Figure 4. The results of other data rates are not shown because they tend to have fewer partial packets than 54 Mbps, and basically duplicate the results of low partial ratio at 54 Mbps. The sender generates 3000 packets per second, where each packet is 1500 bytes. The generated load is slightly higher than the maximum data rate that can be supported by the link, after considering the overhead such as SIFS, DIFS, etc., to saturate the link. In this set of experiments, β is set to be 0.2. ORIG sometimes collapses under high partial ratios; we collect data for ORIG when its throughput is not zero.

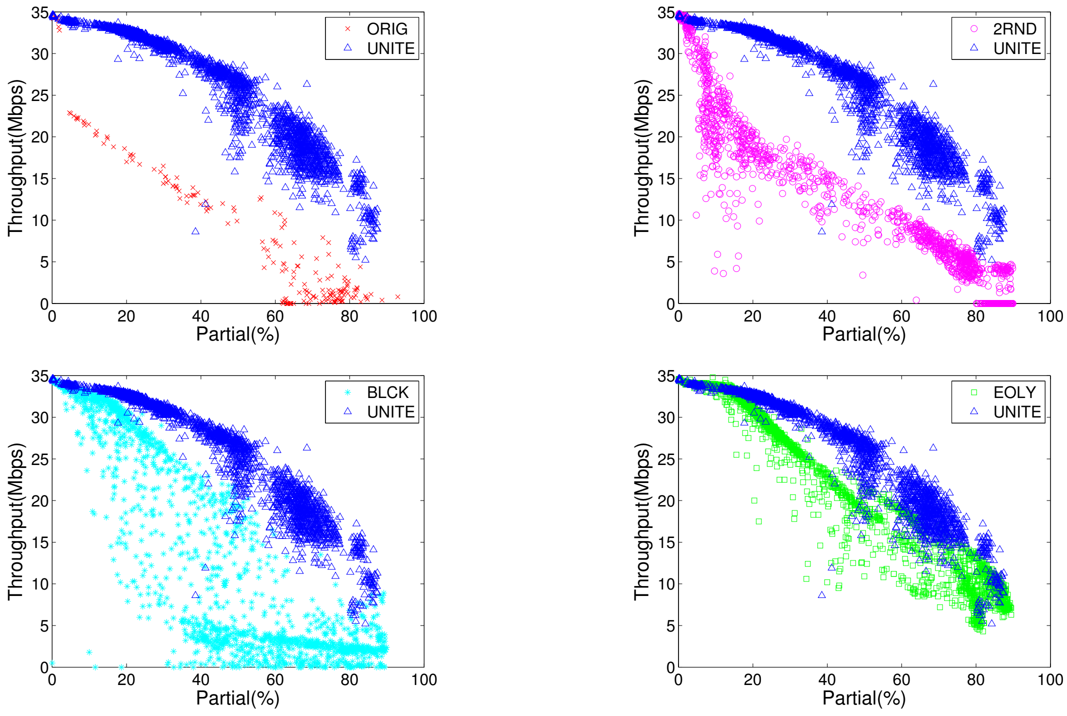


Figure 7: Throughput comparison of Unite with other drivers when the erasure ratio is no more than 0.2.

Ideally, the comparison between the drivers should be carried out under the same the channel condition. This is a challenge in practice because the wireless channel constantly fluctuates even when the sender and receiver are stationary, and it is impossible to replay the channel condition experienced by one driver to another driver. Therefore, we can only compare the drivers when they are under similar channel conditions. To obtain more samples for each channel condition, we divide an experiment trace into one-second segments and classify each segment according to the channel condition in this second. As the wireless channel may be fluctuating, the experiment in one sender and receiver location contributes to measurements under multiple channel conditions. Another practical challenge is the metric of the channel condition. It should be related to the physical layer characteristics; however, the wireless card we use reports only RSSI which is important but is known to be insufficient as the metric of the channel condition for data delivery with OFDM [16]. We note that the channel conditions most relevant to the performance of the repair schemes are the erasure ratio and partial ratio, which can be measured by the driver itself. The erasure ratio and the partial ratio are therefore used as the metric for channel conditions.

We classify channels into two bins, one with erasure ratio no more than 0.2 and the other with erasure ratio more than 0.2. We show more detailed analysis of the first bin because the second bin represents highly corrupted channels for the rate and is unlikely to be used by the rate selection algorithm. The measurement in the second bin is shown in Section 6.2.

6.1.3 *Throughput Comparison*

Figure 5 shows the average throughput for various partial ratios when the erasure ratio is no more than 0.2. We can see that Unite has higher throughput than other drivers in all partial ratios, and the gain is higher when the partial ratio is higher. This confirms that Unite achieves higher performance than other recovery schemes by better exploiting partial packets.

6.1.4 *CPU Load Comparison*

We also measure the CPU load of the receiver for different drivers during the experiment with *vmstat*. Similar to [3], we use the total CPU load as an estimate of the CPU load caused by the drivers, as no other application is running during the experiment and the operating system typically does not consume significant CPU time. When the receiver receives the first packet, a signal is sent to the program that collects the CPU load to synchronize the driver and the CPU load measurement.

Figure 6 shows the average CPU load for various partial ratios when the erasure ratio is no more than 0.2. As expected, ORIG and BLCK have the smallest CPU load, both around 17%, because they do not require any software decoding. Unite's average CPU load is below 31.4%. EOLY's average CPU load is below 35.8%. This suggests that Unite complies with the CPU time constraint which is set to be 0.2 in this set of experiments, noting that the constraint is only for the decoding time while the driver also needs CPU for other tasks which will consume similar CPU resources as ORIG. 2RND's CPU usage, however, is much higher than the rest of the drivers, which confirms that the software decoding may consume high CPU resources in practical settings.

6.1.5 *Fine-grained Throughput Comparison*

Figure 7 shows more detailed information about the throughput of the drivers when the erasure ratio is no more than 0.2, where we show the throughput of each one-second segment with regarding to the partial ratio. For better visibility, each plot shows the performance of Unite with one other compared driver. It can be seen that the performance of ORIG decreases almost linearly with the partial ratio, which is because to ORIG, a partial packet is a lost

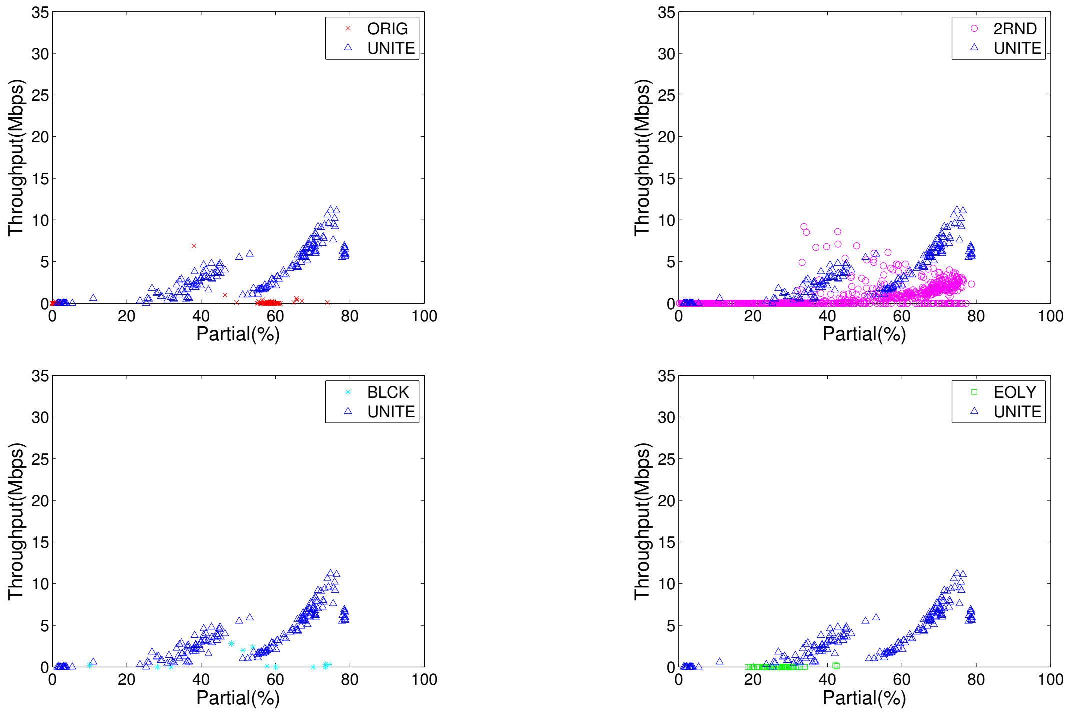


Figure 8: Throughput comparison of Unite with other drivers when the erasure ratio is more than 0.2.

packet and must be retransmitted. 2RND achieves much better performance than ORIG when the partial ratio is small, e.g., around 5%, because most partial packets have very few errors in this range which can be efficiently repaired by 2RND. However, as the partial ratio increases, the performance of 2RND drops sharply, which is because its fixed repair schedule cannot cope with the highly dynamic nature of the error distribution in the partial packets. BLCK performs reasonably well when the partial ratio is low; however, as the partial ratio increases, it drops very sharply. Note that the probability that the repair data is corrupted increases with the partial ratio. EOLY is better than these drivers but is still outperformed by Unite. Note that Unite adopts more repair methods than EOLY and TEC is much more efficient than HEC such that Unite can repair more partial packets with error correction.

6.2 Measurements when Erasure Ratio is More Than 0.2

Our experiments also include cases when the erasure ratio is more than 0.2, which are shown in Figure 8. As mentioned earlier, this represents channels that are highly corrupted. We collected a reasonable amount of data for Unite but not for some of the other drivers. In general, Unite still outperforms other drivers. It can be observed that in some cases, the throughput increases with the partial ratio, which is because an increase of the partial ratio likely means a decrease of the erasure ratio. This is not observed in Figure 7 because the erasure ratio there is bounded from above and an increase of the partial ratio likely means a decrease of the correct ratio.

6.3 Repair Method Failure Ratio of Unite

A repair attempt may fail due to a number of reasons. For block-retran, it is mainly due to the corruption of repair data. For HEC and TEC, it may be due to the underestimation of the number of errors such that the sender sends an insufficient number of parity bytes, or due to the corruption of repair data beyond the correction capability of the transmitted parity bytes. Figure 9 shows the fraction of partial packets repaired by HEC, TEC and block-retran that fail in the first repair attempt collected in the same set of data as in Section 6.1. We can see that the failure ratios of HEC and TEC are usually low, e.g., no more than 5%, except when the partial ratio is above 60%, in which case they are still no more than 10%. This implies that AMPS gives good estimations. Block-retran has the highest failure ratio, which is because it has no protection for the repair data while the methods based on error correction are more resilient to the errors in the repair data.

To verify the error estimation accuracy, we collected data for the packets repaired with HEC. We denote the actual maximum number of errors among the code blocks as z. An *underestimation* is defined as the event when $\hat{z} < z$, and the *overestimate number* is defined as $\hat{z} - z$ when $\hat{z} \geq z$. We find that the underestimation percentage is 2.52%. Figure 10 shows the PDF of the overestimate number, where we can see that the overestimate number is also very small in most cases with a median of 3.

6.4 Unite under Different CPU Time Constraint

Unite's performance is determined by how much recourse it is allowed to consume. We conduct another set of experiments when the CPU time threshold β is set to be 0.1, 0.2, 0.3, 0.4, and unlimited at same set of locations as the experiments in Section 6.1 at 54 Mbps. When the load is unlimited, no scheduling algorithm is run and all packets qualified for TEC are repaired with TEC; other packets, if qualified for HEC, are repaired with HEC.

Figure 11 and Figure 12 show the throughput and measured CPU load for different β, respectively. We can see that when the partial

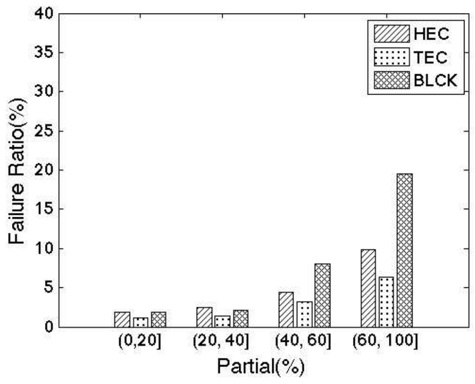


Figure 9: The percentage of HEC, TEC and block-retran that failed in the first attempt.

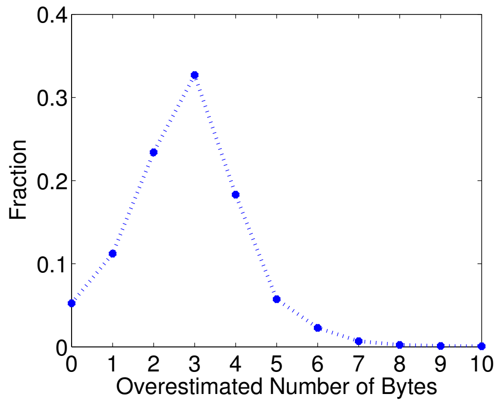


Figure 10: PDF of overestimated number of errors.

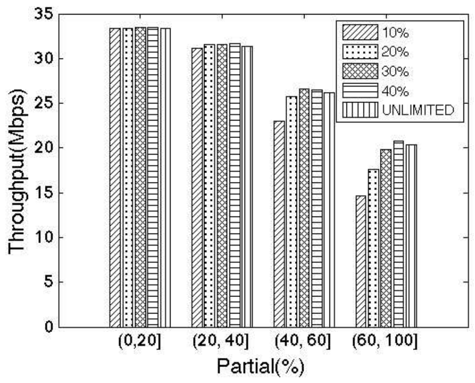


Figure 11: Throughput when varying the CPU time constraint for Unite.

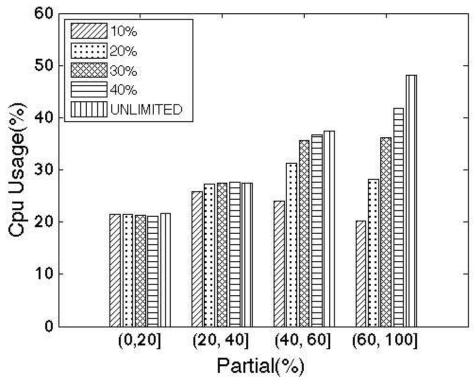


Figure 12: CPU load measurement when varying the CPU time constraint for Unite.

ratio is no more than 40%, the throughputs are similar which is because the demand for CPU time is still below the lowest threshold: as can be seen in Figure 12, the CPU loads for different β are similar. When the partial ratio is higher, larger β leads to larger throughput and also more CPU load. This confirms that Unite is capable of capitalizing the allowed CPU resource for higher link throughput. The measured CPU load is never higher than 50% even for the unlimited, because the heavily corrupted packets are not qualified for HEC or TEC and are repaired with block-retran, such that the demand of the decoding time is not excessively high.

The key feature of Unite is that it supports multiple repair methods and makes dynamic selections of the repair methods. Figure 13 shows the number of packets repaired by different methods per second for various partial ratios and various β. We can see that when partial ratio is no more than 40%, the majority of packets are repaired by TEC, and the selections under different β are similar. This is again because the demand of the decoding time is low such that the CPU time is not yet a constraint. For a partial ratio higher than 40%, with a larger β, more partial packets are repaired and more packets are repaired with HEC which is more time consuming. It can also been seen that as channel gets worse, more packets are repaired with block-retran.

6.5 Unite under Different Feedback Frequency

We conduct another set of experiments to find the impact of the

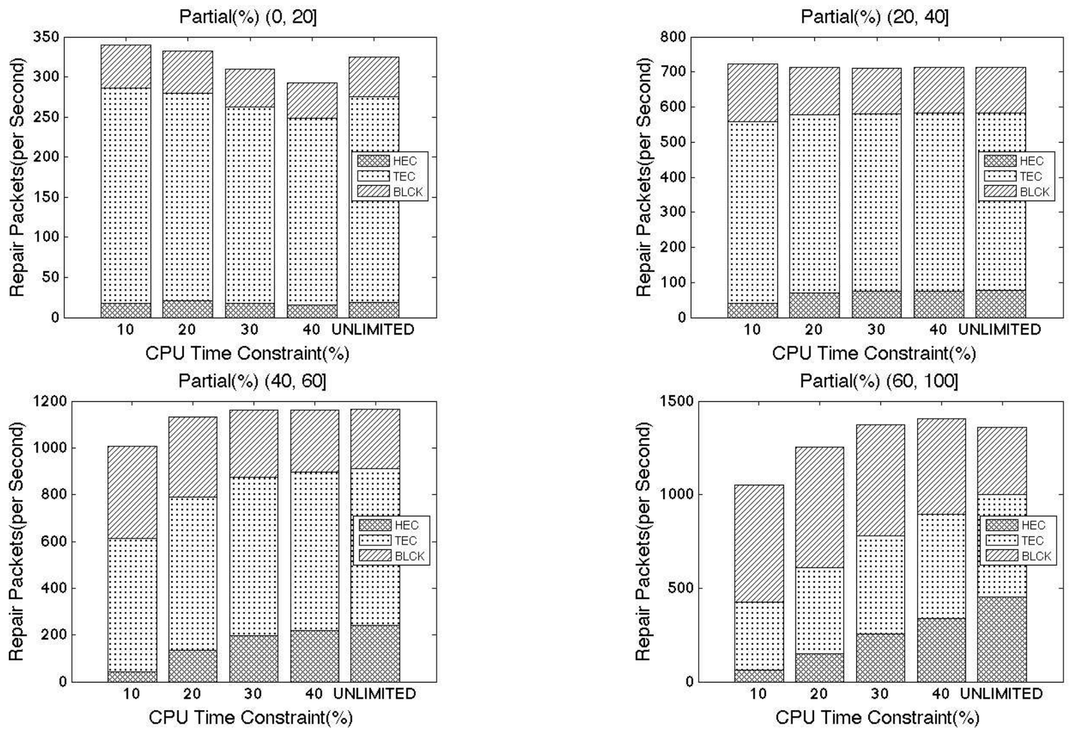


Figure 13: The choice of repair method of Unite under different partial ratio and CPU time constraint.

feedback frequency. We define the *delivery latency* as the time since a packet is realized by the receiver to the time the packet is correctly delivered. Intuitively, a higher feedback frequency should reduce the delivery latency; on the other hand, it leads to more feedbacks thus more overhead and a lower throughput. The most effective way of controlling the feedback frequency is to set the *packet count threshold*, which is the number of encountered partial and erasure packets to trigger a feedback. Figure 14 shows the throughput and the average delivery latency in one-second segments as a function of the partial ratio, when packet count threshold is 4, 8, and 16 in the same set of locations as the experiments in Section 6.1. Figure 15 shows the corresponding CDFs. We can see that varying the packet count threshold does have impact on the throughput; however, somewhat surprisingly, the throughputs are similar when the threshold is 4 and 8, while the throughput is actually smaller when the threshold is 16 when the partial ratio is higher than 40%. This suggests that under high partial ratios, more frequent feedback is actually preferred. It can also been seen that setting the threshold to 16 also leads to the highest delivery latency. It is worth noting that under all threshold values, the latency is less than 10 ms when the partial ratio is no more than 60%. For example, when the partial ratio is around 20%, the delivery latency is around 7ms.

7. DISCUSSIONS AND FUTURE WORK

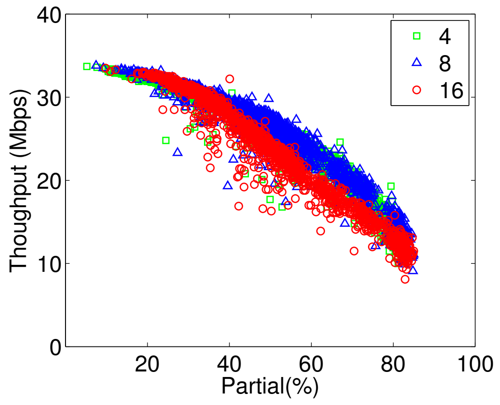
In this section, we discuss possible extensions and future works.

7.1 Extensions to Firmware/Hardware Implementation

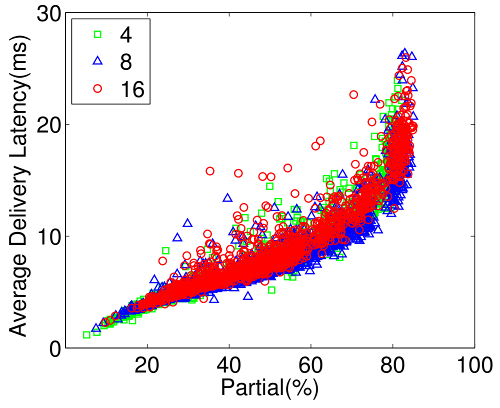
Unite is currently implemented in software as a driver. The advantage of the software implementation, as mentioned earlier, is that it allows more ubiquitous deployment than firmware implementations. It also possible to extend the idea of combining EC-based with block-based approach of Unite to firmware implementations. Basically, when receiving a partial packet, the checksum calculation and AMPS estimation can be carried out by the microprocessor on the wireless card. A negative-ACK can be sent immediately with the estimated number of errors and the checksums. The sender, when receives this NACK, chooses a repair method. The receiver, when receives the repair packet, recovers the original packet. The implementation will result in similar throughput performance as Unite; however, it has less jitter in packet latency because every partial packet is repaired immediately,

One issue, however, is that the decoding of error correction code requires a computational power that most likely exceeds the capacity of the microprocessor on the wireless card; therefore, this approach cannot be implemented in the existing wireless cards. It is still possible, however, if the decoding can be carried out by hardware. The wireless card can incorporate some additional hardware decoders, each capable of decoding up to certain number of errors per codeword. The complexity of the decoding circuit increases with the number of errors; fortunately, most partial packets contain very few errors, such that there is no need to allocate decoders for a large number errors. In case a partial packet contains too many errors, it can be repaired with block-retran. To further reduce hardware cost, it is also not necessary to allocate a decoder for each possible number of errors. Several decoders with suitable spacings can be allocated, and a partial packet can be sent to the smallest decoder capable of correcting all errors, at a cost of slightly increased number of transmitted parity bytes. For example, we may employ 3 decoders capable of decoding 3, 6, and 10 errors per codeword, respectively. The scheduling algorithm may have to be modified considering the decoding speed of the hardware decoders.

7.2 Unite and Rate Selection

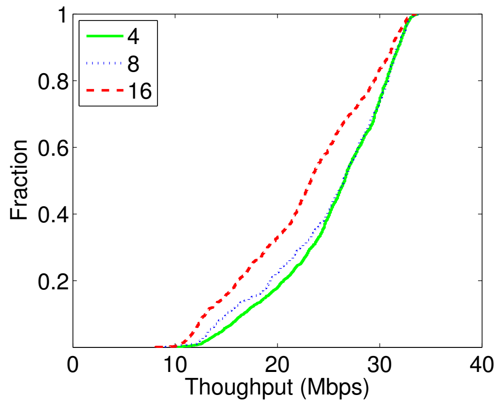


(a)

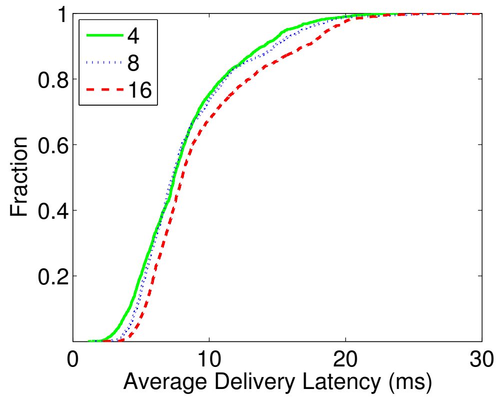


(b)

Figure 14: Varying feedback frequency. (a) Throughput. (b). Average delivery latency.



(a)



(b)

Figure 15: Varying feedback frequency. (a) CDF of throughput. (b). CDF of average delivery latency.

Unite provides novel packet repair methods which has implications on rate selection. The rate selection algorithm determines the best data rate under a channel condition. With Unite, the rate selection algorithm may be able to select a higher rate than with other drivers because Unite can function at such rate while others cannot.

Rate selection is an important topic of its own, and many algorithms have been proposed [18, 17, 16]. Our focus in this paper is the repair method for partial packets at any given data rate. The major challenge for rate selection is that it is difficult to estimate the packet delivery time of a rate when operating at another rate, because many issues influence the actual delivery time, including the number of errors, the location of the errors, and even the scheduling algorithm. One promising direction is to look into the physical layer information. Recent works show that the autorate algorithms can benefit significantly in terms of converging speed by using the channel state information reported by the 802.11n cards [16]. We will investigate how to incorporate the physical layer information into Unite for rate selection in our future work.

8. CONCLUSION

In this paper, we propose Unite, a framework for efficient partial packet recovery in 802.11 LANs. Unite combines the EC-based and block-based repair method, and uses an error estimator to assist intelligent selection among the methods. Our main contributions include the following. First, we propose to combine the EC-based and block-based repair methods and in particular, a novel repair method, namely TEC. Second, we design an algorithm to make selections among the repair methods, based on the error estimate and also considering the host's CPU load. We implement Unite on the Madwifi and our experiments show that Unite outperforms other recovery schemes.

9. REFERENCES

[1] D. Tse and P. Viswanath, "Fundamentals of Wireless Communication," *Cambridge University Press*, May 2005.

[2] S. B. Wicker, *Error Control Coding for Digital Communication and Storage*, Prentice-Hall, NJ, 1995.

[3] K. Lin, N. Kushman, and D. Katabi, "ZipTx: Harnessing partial packets in 802.11 networks," in *Proc. Mobicom, 2008*.

[4] K. Jamieson and H. Balakrishnan, "PPR: Partial packet recovery for wireless networks," in *Proc. Sigcomm, 2007*.

[5] G. Woo, P. Kheradpour, D. Shen and D. Katabi, "Beyond the bits: cooperative packet recovery using PHY information," in *Proc. Mobicom, 2007*.

[6] H. Duboi-Ferriere, D. Estrin, and M. Vetterli. "Packet combining in sensor networks," in *Proc. Sensys, 2005*.

[7] A. K. Miu, H. Balakrishnan, and C. E. Koksal. "Improving loss resilience with multi-radio diversity in wireless networks," in *Proc. Mobicom, 2005*.

[8] M.-H. Lu, P. Steenkiste, and T. Chen. "Design, implementation and evaluation of an efficient opportunistic retransmission protocol," in *Proc. Mobicom, 2009*.

[9] http://madwifi-project.org/

[10] Cisco Aironet 802.11a/b/g wireless cardbus adapter, http://www.cisco.com/.

[11] S. Katti, D. Katabi, H. Balakrishnan and M. Medard, “Symbol-level network coding for wireless mesh networks,” in *Proc. Sigcomm, 2008*.

[12] B. Chen, Z. Zhou, Y. Zhao, and H Yu, “Efficient error estimating coding: feasibility and applications,” in *Proc. Sigcomm, 2010*.

[13] http://www.ka9q.net/code/fec/

[14] B. Han, A, Schulman, F. Gringoli, N. Spring, B. Bhattacharjee, L. Nava, L. Ji, S. Lee, and R. Miller “Maranello: Practical partial packet recovery for 802.11,” in *Proc. NSDI, 2010*.

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

[16] D. Halperin, W. Hu, A. Sheth, and D. Wetherall, “Predictable 802.11 packet delivery from wireless channel measurements,” in *Proc. Sigcomm, 2010*.

[17] G. Judd, X. Wang, and P. Steenkiste, “Efficient channel-aware rate adaptation in dynamic environments,” in *Proc. MobiSys*, 2008.

[18] S. H. Wong, H. Yang, S. Lu, and V. Bharghavan, “Robust rate adaptation for 802.11 wireless networks,” in *Proc. Mobicom*, 2006.

[19] A. Padmanabha Iyer, G. Deshpande, E. Rozner, A. Bhartia, and L. Qiu, “Fast resilient jumbo frames in wireless LANs,” in *IEEE IWQoS, 2009*, Charleston, SC, June 2009.

[20] Z. Zhang, W. Hu, and J.Xie, “Employing coded relay in multihop wireless networks,” *arXiv:1012.4136v1*, December 2010.