Efficient Software Partial Packet Recovery in 802.11 Wireless LANs

Jin Xie, Wei Hu, and Zhenghao Zhang, *Member, IEEE*

Abstract—In 802.11 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 study software partial packet recovery and propose a comprehensive scheme called UNITE with the following key features. First, UNITE supports both the EC-based and block-based methods, because they are suited for different scenarios and can complement each other to achieve higher performance. Second, UNITE employ AMPS, an error estimator designed by us, which is capable of estimating the error condition in each individual partial packet and providing key information for repairing the packet. Third, UNITE selects the repair method for each packet to optimize the link performance under configurable system resource constraint, such as CPU or power. We implement UNITE on the Madwifi open-source driver. Our experiments show that UNITE outperforms other recovery schemes while not over-consuming the system resources.

Index Terms—Partial packet recovery, 802.11, Device driver.

1 Introduction

802.11 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 [14], [8], [18], [12], [10], [9] 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 [14], [9] or firmware [8] 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; if the data is corrupted, the sender encodes each block into *codewords* according to an error correction code and transmits the parity bytes for each block to correct the errors. In the block-based approach, the sender divides packets into blocks and retransmits the corrupted blocks, where a block is found to be corrupted if it fails the checksum test.

J. Xie and W. Hu were with the Computer Science Department, Florida State University, Tallahassee, FL 32306 during this work. Z. Zhang is with the same department. Email: {xie,hu,zzhang}@cs.fsu.edu.

In this paper, we study software-only partial packet recovery and propose a comprehensive scheme called UNITE, motivated by the following observations. First, the EC-based and the block-based approaches are not mutually exclusive; rather, they 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, error estimators can estimate the number of errors in a received packet and provide key information for repairing the packet; such information was not available to the earlier partial packet recovery schemes because error estimators were only proposed recently. Third, the existing repair schemes may consume a large amount of system resource such as the CPU and power; a practical scheme, on the other hand, should avoid over-consuming the system resources.

The key features of UNITE include the following. First, UNITE supports three repair methods, which are best suited for packets with different numbers of errors. The methods include *Holistic-EC* (HEC) and block-retran, which are the standard error correction and block-based approaches, respectively, as well as a new method called *Target-EC* (TEC), which is specifically designed for packets with very few errors because such packets often exist. Second, UNITE employs AMPS, an error estimator proposed by us, and gets an estimate of the error number in each partial packet which guides the selections of the repair methods and repair parameters. Third, UNITE may operate under configurable CPU and power constraints, and employs an algorithm that optimizes link performance under such constraints. We implement UNITE based on the Madwifi [2] open-source driver. We test UNITE with extensive experiments and compare it with other drivers, including the original 802.11 driver, a two-round retransmission driver, a pure block-based driver, and a pure EC-based driver enhanced with AMPS. We find that in the majority of the cases, UNITE achieves higher performance than all other drivers. In addition, UNITE consumes the system resources under the specified constraint.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 describes the protocol according to which UNITE transmits and repairs packets. Section 4 describes AMPS. Section 5 describes the system resources usage models. Section 6 describes the algorithms for repair method selection under configurable resource constraints. Section 7 describes the experiments and evaluation. Section 8 discusses possible extensions. Section 9 concludes the paper.

2 Related Work

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 [14] 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 methods, which helps achieving higher performance. Second, UNITE is capable of limiting its CPU or power consumption under given constraints. This is motivated by our experiments which show that software decoding can consume a large amount of CPU as well as power. ZipTx, on the other hand, does not have this mechanism. In ZipTx, power consumption was not considered and it was argued that the CPU load will not be high; however, we note that the experiments in ZipTx were carried out in high end machines which may underestimate the CPU load in low end machines such as smartphones and tablets. 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.

The block-based approach has been implemented in the Madwifi dirver [9] and in firmware [8]. For example, in Maranello [8], 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. We note that UNITE combines both the EC-based and block-based approach and can achieve higher performance. We also note that UNITE is a software-based solution that can be applied to many platforms; firmware solution such as Maranello needs to modify the wireless card firmware which may become a limiting factor in deployment because not all wireless card manufactures expose the firmware design.

Many other partial packet recovery schemes have also been proposed. PPR [10], SOFT [18], and MIXIT [12] recover corrupted packet with the assistance of the physical layer, i.e., the physical layer is modified to report additional information about the received bits, such as the confidence levels of the bit values, to allow the upper layers to locate the corrupted sections of the packet or to better determine the bit values by a soft combining of multiple copies of the packet. While these approaches may efficiently recover corrupted packets, they rely on specialized hardware and thus are not applicable to the existing devices. In [17], [16], [6], opportunistic overhearing at third-party nodes is exploited for packet recovery. For example, in PRO [16], a relay node will retransmit a lost packet on behalf of the source node if the relay node has a better link to the destination node. In this paper, we focus on the recovery between two nodes, and leave the extension to multiple nodes to future work.

UNITE was first presented in [20]; in this paper, we have enhanced our earlier work with substantial new materials and new results, including detailed description of AMPS, the power consumption model, and many improvements in the implementation.

3 The UNITE Protocol

In this section, we discuss the protocol used by UNITE. We begin with a high-level overview of the packet transmission procedure.

3.1 Packet Transmission Procedure

In the initial transmission, the sender sends only the data bytes. Depending on the outcome:

- If the packet is received correctly, it is delivered to the upper layer immediately. The transmission of this packet finishes.
- If the packet is received partially, the receiver computes AMPS samples. It also divides the packet into blocks and calculates the checksums of these blocks. When the receiver gets the opportunity to send, it sends a feedback to the sender containing the sequence number, the AMPS samples and the block checksums for the partial packet. After the sender receives the feedback, it runs the error estimator to find the number of errors, as well as comparing its local block checksums with the received block checksums to find the corrupted blocks. The sender selects a repair method and sends the repair data, which is either the parity bytes or the corrupted blocks.
- If the packet is erased, detected by the receiver according to the absent sequence number, the sender is informed and may retransmit the packet up to three times until the packet is received correctly or becomes a partial packet.

When the receiver receives the repair data for a partial packet, it attempts to repair the packet:

- If the repair succeeds, the packet is delivered to the upper layer and the transmission of this packet finishes.
- If the repair fails:
  - If the packet is repaired with block-retran, it is repaired with block-retran again for a maximum of two times.
  - If the packet is repaired with HEC or TEC, it will be repaired with block-retran for a maximum of three times.

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:

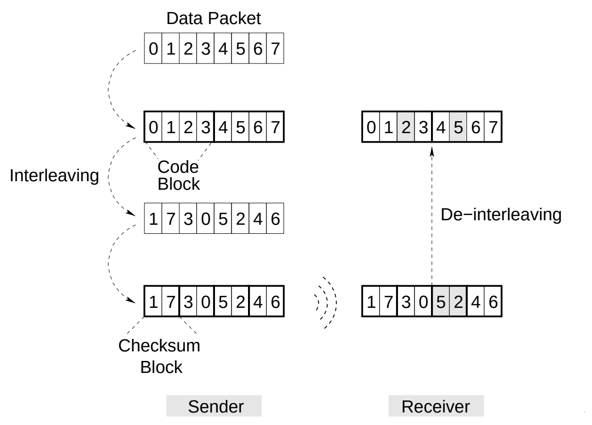


Fig. 1. A simple illustration of the code block, interleaving, and checksum block. The shaded boxes represent corrupted bytes.

- *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 the original data bytes followed by parity bytes. In our current implementation, the maximum length of a codeword is 255 bytes and the size of the code blocks is 150 bytes, 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 [15], as well as the availability of software implementations [11].
- *Interleaving*: When transmitting the original data packet, the packet undergoes an *interleaving* procedure, in which each byte is relocated to a random location based on a random permutation. At the receiver side, the packet undergoes an *deinterleaving* procedure, which is the reverse of the interleaving, such that the bytes are mapped to their original locations. Interleaving is a common technique in wireless communications to cope with bursty errors. As errors in a wireless frame tend to be clustered in a few locations [17], 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, UNITE also 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 the error correction code prefers the errors to be spread out evenly in the packet, while the block-retran prefers the errors to be clustered in as few blocks as possible, such that less blocks need to be retransmitted. Therefore, the checksum blocks are defined based on the packet *after* the interleaving step.

For example, Fig. 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 Three Repair Methods

UNITE supports three repair methods, namely block-retran, HEC, and TEC. In this section, we discuss the repair methods in details. We use Y and Z to denote the number of error bytes in a packet and the maximum number of error bytes in a code block among all code blocks, respectively. Y and Z are unknown to the receiver and their estimates provided by AMPS are denoted as $\hat{\mathcal{Y}}$ and $\hat{\mathcal{Z}}$, respectively.

3.3.1 Block-Retran

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 received partial 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. Block-retran is the fallback repair method in UNITE; all packets can be repaired with block-retran.

In our current implementation, the checksum is 16 bits according to the generator polynomial $x^{16} + x^{15} + x^2 + 1$ with the implementation available in the Linux kernel. We did not use the more popular 32-bit CRCs which have stronger error detection capabilities, due to the better tradeoff between detection probability and overhead of the 16-bit CRC.

3.3.2 HEC

With HEC, the sender sends parity bytes for the code blocks. The number of errors among the code blocks may be different; however, after interleaving, the variance should have been minimized. For simplicity, the same number of parity bytes are sent for all the code blocks. To ensure that all codewords can be decoded correctly, the number of parity bytes should be sufficient for the most corrupted block; therefore, the number of parity bytes is determined according to $\hat{\mathcal{Z}}$. After receiving the parity bytes, the receiver runs the decoding algorithm for each code block.

To be more specific, the sender encodes a code block into a codeword according to the (255, 255-2$\hat{\mathcal{Z}}$) RS code and transmits 2$\hat{\mathcal{Z}}$ parity bytes as the repair data. With 2$\hat{\mathcal{Z}}$ parity bytes, the RS code guarantees to correct any no more than $\hat{\mathcal{Z}}$ errors in the received data bytes and parity bytes. Choosing the code adaptively based on the number of errors minimizes the transmission time and the decoding time. If the codeword can accommodate more data bytes than the size of the code block, the codeword is *shortened*, i.e., when encoding, dummy 0s are added to the beginning of the data bytes while such 0s do not need to be transmitted.

In our current implementation, a 1500-byte packet is qualified for HEC if $\hat{\mathcal{Y}} < 100$. Only packets with less than 100 estimated error bytes qualify for HEC, because for packets with more errors, the variance of AMPS's output is larger which leads to more underestimations. Underestimation causes insufficient number of parity bytes to be sent, which results in decoding failure, wasted CPU time, wasted energy, and additional rounds of transmissions. Setting a threshold at 100 can significantly reduce the probability of underestimation. The threshold can be empirically chosen for packets of other sizes.

3.3.3 TEC

With TEC, the sender locates the corrupted checksum blocks by comparing the checksums, and transmits parity bytes to correct errors in these blocks. TEC is targeted for packets with very few errors clustered in few blocks for which neither HEC nor block-retran is efficient. For example, if there is only one error byte, HEC still has to run the decoding algorithm for all codewords and block-retran has to retransmit an entire checksum block; on the contrary, TEC only sends a few parity bytes and decodes one codeword.

In our current implementation, a 1500-byte packet is qualified for TEC if $\hat{\mathcal{Y}} < 15$ and the number of corrupted checksum blocks is no more than 3. As the size of the checksum block is small, all corrupted checksum blocks are grouped together and encoded into one codeword. If $5(t-1) \leq \hat{\mathcal{Y}} < 5t$ where $t \in \{1,2,3\}$, $10t$ parity bytes are transmitted. The number of transmitted parity bytes are more than the required number of parity bytes according to $\hat{\mathcal{Y}}$, because this reduces the repair failure probability without significantly increasing the overhead. The parameters can be empirically chosen for packets of other sizes.

3.4 The Link Layer Protocol

The details of the link layer protocol are described in the following.

3.4.1 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 which is the success repair status in a feedback frame. The sender maintains a queue of packets that have been sent but have not been delivered. If the queue is not full, the sender may keep sending new packets from the upper layer; otherwise, the sender stops sending any new packets. To cope with feedback loss and packet erasure, the sender retransmits packets in the queue if the queue is full for a certain amount of time while no feedback is received; the threshold is set to be 20 ms in our current implementation.

At the receiver, packets received correctly are delivered to the upper layer immediately. The partial packets and the sequence numbers of the detected erasure packets are stored in a queue. There may be several partial packets and erasure packets in the queue, because, for example, the sender may send several packets before receiving a feedback.

The length of the sender queue is the same as the length of the receiver queue, set to be 32 in our current implementation.

U	the number of data bytes in a packet
B	the number of code blocks in a packet
K	the number of selected bytes for a sample
S	the number of samples
X	the number of error samples
Y	the number of error bytes in the packet
Z	the maximum number of errors in a code block among all transmitted code blocks

TABLE 1
List of notations for AMPS

3.4.2 Feedback

The aggregated feedback mechanism in UNITE is very similar to that in ZipTx [14]. The receiver sends feedback about packet reception and repair 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, the receiver sends a feedback frame in three cases, whichever occurs first:

- When the receiver has received ϕ partial or erasure packets, where ϕ is set to be 8 in our current implementation.
- T_1 after receiving a partial or an erasure packet. This can reduce the packet delivery latency for strong signal channels that need a long time to accumulate enough number of partial packets or erasure packets. T_1 is set to be 10 ms in our current implementation.
- T_2 after the previous feedback is sent, while no corresponding repair data is received. This happens when the previous feedback is erased or corrupted in the transmission. Therefore, the receiver need to retransmit the feedback. In our current implementation, T_2 is set to be 20 ms.

In order to increase the probability of correctly delivering the feedback frame, it is always transmitted at two rates lower than the current data transmission rate; if no such rate exists, the lowest data rate.

4 AMPS – THE ERROR ESTIMATOR

We design AMPS for error estimation, which is based on the idea of *Amplified Sampling*. It is well-known that most partial packets have very few errors, which poses a challenge because an error estimator may not encounter an error byte. AMPS computes a *sample* with multiple bytes, which *amplifies* the raw byte error ratio into a much larger sample error ratio, such that it may better detect the error condition in the packet.

4.1 AMPS Design

In our current implementation, AMPS is invoked for each new data packet that has been received partially. We denote the size of the packet as U. The receiver randomly samples K data bytes, and computes their parity bit; each parity bit is a *sample*. The probability that the sample's parity is flipped is much larger than the probability that a byte is corrupted; for example, if the data byte error ratio is 0.01 and $K = 25$, the probability that the sample's parity is flipped is approximately $(1-0.99^{25})/2 = 0.11$. The receiver sends the AMPS samples in the feedback and the sender calculates the samples in exactly the same way based on the original packet. The sender's samples may be different from the receiver's samples, because the receiver's samples are calculated from the corrupted data. We call a mismatching sample an *error sample*. We denote the number of error samples as X, which is used as the input to AMPS. As mentioned earlier, AMPS estimates Y, the number of error bytes in the packet, as well as Z, the maximum number of errors in a code block among all transmitted code blocks. Table 4.1 lists the main notations related to AMPS.

4.1.1 Estimation of Y

AMPS outputs $\hat{\mathcal{Y}}$ as the estimate of Y, if $\hat{\mathcal{Y}}$ maximizes the conditional probability $P(X = x|Y = \hat{\mathcal{Y}})$ under the constraint that $\hat{\mathcal{Y}}$ is no more than R, where R denotes the maximum number of errors in a packet.

To see why AMPS adopts this policy, we note that in the ideal case, $\hat{\mathcal{Y}}$ should maximize $P(Y = \hat{\mathcal{Y}}|X = x)$, which can be easily found to be equivalent to maximizing $P(X = x|Y = \hat{\mathcal{Y}})P(Y = \hat{\mathcal{Y}})$. $P(X = x|Y = \hat{\mathcal{Y}})$ can be calculated; however, the practical challenge is that the prior distribution $P(Y = \hat{\mathcal{Y}})$ can be costly to obtain in practice because it is different in different channels. However, we find that the numbers of corrupted bytes and in the vast majority of partial packets are typically below a threshold which we denote as R; for example, we find that a reasonable R is 200 for packets of 1500 bytes. Therefore, in AMPS, we basically apply a very simple prior distribution which is uniform from 0 to R and is 0 for other values.

Calculating $P(X = x|Y = y)$: We note that the error bytes are randomly distributed in the packet after interleaving. We assume that the error bytes take random values. Therefore, the probability that a sample is an error sample when there are y error bytes, denoted as η_y, is

$$\eta_y = [1 - \binom{U-y}{K} / \binom{U}{K}]/2. \qquad (1)$$

For simplicity, we treat the samples as independent. As a result, the probability that there are x error samples follows the Binomial distribution:

$$P(X = x|Y = y) = \binom{S}{x} {\eta_y}^{x} (1-\eta_y)^{S-x}. \qquad (2)$$

Obtaining $\hat{\mathcal{Y}}$: According to our policy, $\hat{\mathcal{Y}} = \min\{\hat{y}, R\}$ where $\hat{y}$ is the value that maximizes Eq. 2. In the following, we focus on finding $\hat{y}$. When $x < \frac{S}{2}$, $\hat{y}$ should be

$$\hat{y} \approx (1 - \sqrt[K]{1 - \frac{2x}{S}})U. \qquad (3)$$

To see this, we note that Eq. 2 is the standard Binomial distribution and is maximized when $\eta_y = \frac{x}{S}$ for given x. Therefore, $\hat{y}$ satisfies

$$\begin{aligned} \frac{x}{S} &= [1 - \binom{U-\hat{y}}{K} / \binom{U}{K}]/2 \\ &\approx [1 - (1 - \frac{\hat{y}}{U})^K]/2 \end{aligned}$$

with which Eq. 3 can be derived. When $x \geq \frac{S}{2}$, Eq. 2 is still maximized when $\eta_y = \frac{x}{S}$; however, such values of η_y cannot be taken because $\eta_y < \frac{1}{2}$ according to Eq. 1. Therefore, when $x \geq \frac{S}{2}$, $\hat{y}$ approaches U and corresponds to a η_y as close to $\frac{1}{2}$ as possible; such values will not be used because of the cap at R.

4.1.2 Estimation of Z

Z is estimated based on the conditional probability $P(Z = z|Y = y)$. We calculate it iteratively on the number of code blocks. To be more specific, we use $P_i(Z = z|Y = y)$ to denote the probability that $Z = z$ when there are i code blocks. By definition, $P(Z = z|Y = y) = P_B(Z = z|Y = y)$. First, when there is only one code block, clearly,

$$P_1(Z = z|Y = y) = \begin{cases} 1 & z = y \\ 0 & \text{otherwise.} \end{cases}$$

For notational simplicity, we use $\delta_t^{y,i}$ to denote the probability that among the y error bytes, t bytes are in one particular code block when there are i code blocks. Because the error bytes are randomly distributed,

$$\delta_t^{y,i} = \binom{y}{t} (\frac{1}{i})^t (1 - \frac{1}{i})^{y-t}.$$

Given $P_i(Z = z|Y = y)$, $P_{i+1}(Z = z|Y = y)$ can be found by conditioning on the number of error bytes in a particular code block. That is, we single out one code block, called the *tagged code block*, and check the number of error bytes in this code block. The event that $Z = z$ occurs when the tagged code block has less than z errors while the maximum number of errors in a code block among the remaining code blocks is exactly z, or when the tagged code block has exactly z errors while the maximum number of errors in a code block among the remaining code blocks is no more than z. Therefore,

$$\begin{aligned} P_{i+1}(Z = z|Y = y) &= \sum_{t=0}^{z-1} \delta_t^{y,i} P_i(Z = z|Y = y-t) \\ &+ \delta_z^{y,i} \sum_{z'=0}^{z} P_i(Z = z'|Y = y-z). \end{aligned}$$

After getting $\hat{\mathcal{Y}}$, AMPS outputs $\hat{\mathcal{Z}}$ such that $P(Z \leq \hat{\mathcal{Z}}|Y = \hat{\mathcal{Y}})$ is greater than a threshold, set to be 0.95 in our implementation.

4.2 Choices of Parameters

The key parameters of AMPS are S and K. We discuss the choice of S and K for packets of size 1500 bytes as an example; the same method can be easily applied to other packet sizes.

The values of S and K are both related to the number of errors in the packet. S is the number of samples, and is the overhead of AMPS measured in bits. A system prefers overhead as small as possible; on the other hand, the number of samples must allow a reasonable granularity of estimate. Given the condition that $R = 200$, we choose $S = 64$, i.e., the overhead of AMPS is only 8 bytes.

K determines the amplifying factor. It is desired to choose K to minimize a certain cost function; however, we note that a meaningful cost function cannot be defined because it must involve the prior distribution of Y which varies depending on the particular channel condition and the wireless card. Therefore, we choose K such that the maximum number of estimated errors matches R. From earlier discussions, it can be found that the largest estimate corresponds to that when $x = \frac{S}{2} - 1$. According to Eq. 3, the maximum number of estimated error is:

$$(1 - \sqrt[K]{1 - \frac{1}{2S}})U,$$

and to match it with R, K should be determined by

$$K \approx \frac{\ln \frac{1}{2S}}{\ln\left(1 - \frac{R}{U}\right)}. \quad (4)$$

It can be found that K is 25 when $U = 1500$ and $R = 200$.

4.3 Table Lookup Implementation

We note that both $P(X = x|Y = y)$ and $P(Z = z|Y = y)$ can be precomputed. To reduce the computation complexity in real time, we precompute the tables for the estimation of Y and Z such that the estimates can be obtained by a simple table lookup in constant time. We note that the sizes of the tables are very small, because Y and Z are determined by lookups on X and Y, respectively, while X and Y both have no more than S+1 distinct values in our system. To handle more frame sizes, tables can be built for a set of representative frame sizes; for a received frame of any size, the table for the closest frame size can be used.

4.4 AMPS Performance

We discuss the performance of AMPS in this section.

4.4.1 *Comparing with EEC and the Pilot Method*

Error estimation can also be achieved by the simple pilot method, i.e., by inserting pilot bits and using the fractions of flipped pilot bits as the estimate of bit error ratio, as well as by EEC [4], which is another recently proposed error estimator. We compare the performance of AMPS with these methods and show the results in Fig. 2, where the performance is measured by the *estimation error*, i.e., the difference between the estimated and the actual number of errors. As AMPS and EEC/pilot estimate byte and bit errors, respectively, we consider 1500-byte and 1500-bit packets for AMPS and EEC/pilot, respectively. We use simulations to obtain the results for AMPS and EEC and derive the results for the pilot method mathematically. In the simulations, we inject random byte and bit errors into the packets where the errors range from 1 to 100 bytes/bits at a step of 1 byte/bit. For each error number, we run the simulation 10,000 times. For the pilot method, we note that if there are U data bits and S pilot bits in the packet, given there are Y flipped bits, the mean estimation error is

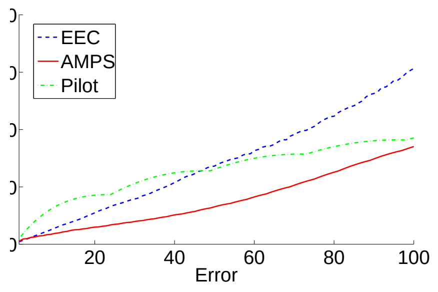


Fig. 2. Comparison between AMPS and EEC.

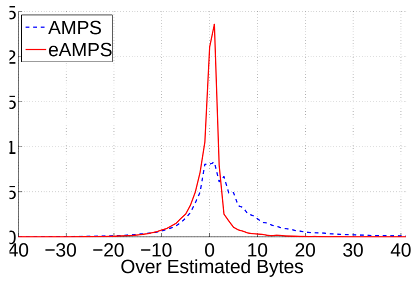
$$\sum_{e=0}^{\min(Y,S)} \frac{\binom{U}{Y-e}\binom{S}{e}}{\binom{U+S}{Y}} |round(\frac{(U+S)e}{S}) - Y|. \quad (5)$$

Based on Section 3.5 of [4], EEC needs 7 sample levels from level 3 to level 10; therefore, EEC has 28 bytes of samples. For the pilot method, we set the number of pilot bits to be 64 such that it has the same overhead as AMPS. Fig. 2 suggests that AMPS outperforms EEC because the estimation error of AMPS is only about half of EEC in all cases, while the overhead of AMPS is only 8 bytes compared to the 28 bytes of EEC. We can also see that performance of the pilot method is poor when the number of errors is small but improves when the number of errors increases; this is expected because the main problem of the pilot method is that it may fail to sample any error bits when the number of errors is small.

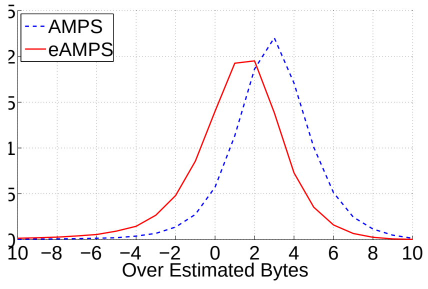
4.4.2 *AMPS in Real Channels*

We also run AMPS for 1,500-byte packets in real-world experiments and show the probability density function (PDF) of estimation errors in Fig. 3. We show two versions of AMPS, one denoted as AMPS and is the AMPS estimator described in this paper, the other denoted as eAMPS and is AMPS enhanced with more accurate prior distribution of Y and is expected to perform better. The two versions of AMPS differ only in the estimation of Y; after obtaining an estimate of Y, the same method is used for the estimation of Z. The details of the eAMPS are described in [19] and are not included in this paper due to the limit of space.

We can see that AMPS performs reasonably well; the average overestimation and underestimation of Y are about 5 bytes and 1.5 bytes, respectively, and the average overestimation and underestimation of Z are about 3.9 bytes and 0.07 bytes, respectively. We can also see that the performance of the AMPS is reasonably close to that of eAMPS. We prefer the AMPS over eAMPS because AMPS does not require accurate prior distribution of Y which may be costly to obtain.



(a)



(b)

Fig. 3. AMPS performance in real wireless channels. (a). PDF of AMPS's Estimate of Y. (b). PDF of AMPS's Estimate of Z.

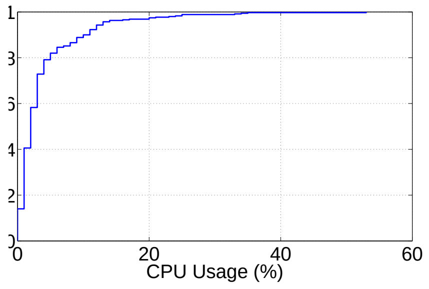


Fig. 4. The CPU usage estimation.

5 Resource Usage Models

We develop models to understand the resource usage caused by the repair methods, which enable the correct choice of repair methods under various constraints.

5.1 CPU Usage

The EC-based methods use significant CPU time in decoding the codewords; the block-retran hardly uses any significant CPU time due to its simplicity. Therefore, we use the total decoding time of the error correction code to estimate the CPU time usage in the partial packet recovery. To illustrate the accuracy of the estimation, we show the estimated CPU usage and the CPU usage measured with the Linux *vmstat* tool which reports the CPU usage percentage in a 350-second experiment. We divide the experiment trace into one-second segments, and show the cumulative distribution function (CDF) of the absolute estimation errors in Fig. 4. We can see that the simple estimation method is reasonably accurate; for example, the median of the estimation error is only 3%.

5.2 Power Usage

We measure the power consumption of the machine with an external power meter *Wattsup* [3]. Wattsup reports the wattage value every second and is accurate to 1.5% + 0.3 Watts of the load [3]. For example, the load for our machine without running any application and wireless driver is about 39 Watts; at such load, the meter's displayed value should be between 38.1 - 39.9 Watts. However, we find that without resetting the power meter, the displayed value of the power meter is stable for the same power load.

For simplicity, we adopt a linear model, i.e., we assume the power consumption can be represented as a linear combination of certain parameters. To learn the power consumption model, we obtained a total of 1200 one-second experimental results. We use half of the results to learn a *multiple linear regression model* [5], and test the learned model on the other half. We evaluate the following 3 models, which differ in the numbers and types of parameters:

- Model 1 which uses the CPU time consumption as the only parameter.
- Model 2 has 21 parameters:
  - *The number of received packets*, including all correct packets, partial packets and packets with corrupted MAC headers.
  - *The number of sent packets.*
  - *The number of packets repaired by block-retran.*
  - *The number of packets repaired by TEC*, in total 3 parameters, one for each possible number of parity bytes.
  - *The number of packets repaired by HEC*, in total 15 parameters, one for each possible number of parity bytes. The possible number of parity bytes is 15 because $\hat{\mathcal{Z}} \leq 15$ when $\hat{\mathcal{Y}} < 100$ according to AMPS.
- Model 3 has two parameters: the CPU time consumption and the total number of packets, including the transmitted packets and the received packets.

Fig. 5 shows the CDF of the estimation errors of the three models. We can see that Model 1 is the least accurate, while the other two models are almost identical. We note that Model 2 exhausts all possible parameters known to the partial packet recovery scheme and represents the upper limit of the estimation accuracy. Considering that Model 3 achieves a very similar performance as Model 2 while using much less parameters, we use Model 3 as our power consumption model.

6 Choice of Repair Methods

The EC-based repair methods usually need a less amount of repair data than the block-based repair methods, but may consume more system resources. On devices with weaker CPU or run on battery power, the system resource consumption must be considered and limited under certain constraints. In this section, we study the choice of the repair methods under given constraints. We begin by considering

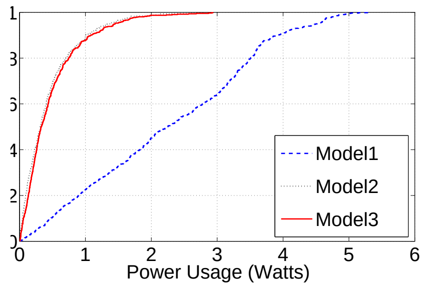


Fig. 5. The power consumption estimation.

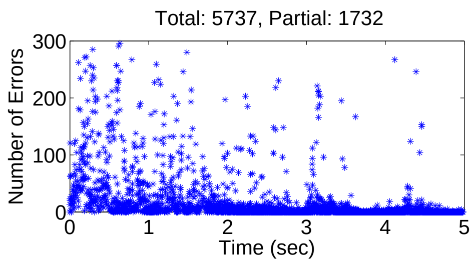


Fig. 6. The number of error bytes in one experiment.

only the CPU constraint, then argue that the power constraint is equivalent to the CPU constraint based on our model in practical systems.

We note that UNITE selects repair method for each individual packet, because the best repair method is mainly determined by the number of errors in the packet, while the number of errors may vary significantly even in consecutive packets. We collected data on 54 Mbps links for 1500-byte packets, and Fig. 6 shows the high variance in one typical experiment, in which a point (x, y) represents a packet received at time x with y errors. We note that it is affordable to select the best repair method for individual packet because the decision is made by a simple algorithm described in the following.

6.1 Scheduling with the CPU Constraint

For CPU usage, we select the repair method such that the CPU time consumed by UNITE does not exceed a threshold configured by the user, denoted by β, which is the fraction of CPU time used in partial packet recovery. With such constraint, it may not be possible to decode all partial packets. 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 checksum blocks, an algorithm is needed to find the best repair method for each partial packet.

In our current implementation, the algorithm runs at the sender every time a feedback frame is received and determines the repair methods for the packets in this feedback frame. 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.
- q: the number of required parity bytes of TEC.
- d: the decoding time if repaired by HEC.
- r: the number of required parity bytes of HEC.
- b: the total size of the corrupted checksum blocks.

The decoding time budget W is determined by the CPU 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 was received t seconds before, γ is updated as $\gamma \leftarrow (1-a)\gamma + at$ where $a = 0.01$ in our current implementation. Note that γ is an estimate of the maximum CPU time the receiver can spend on the decoding of a partial packet. As the decoding should not take more than β fraction of the CPU time, we let $W \leftarrow \beta M \gamma$ where M is the total number of partial packets in this feedback.

It is clear that the decisions are easy for some packets. First, if a packet is not qualified for TEC and HEC, it can only be repaired with block-retran. Second, if a packet requires more data transmission time with HEC or TEC than with block-retran, it 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 $1 \le i \le n$.

We note that although there are three repair methods, for any packet, there are actually only two options. Depending on whether a packet is qualified for TEC or not:

- If yes, 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, it cannot be repaired with TEC and clearly can only be repaired with HEC or block-retran.

As a result, the algorithm makes *binary* decisions for each packet between using block-retran or one of the error correction methods. 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 its *value* and *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 the extra number of bytes in the repair data if block-retran is used instead of TEC or HEC. The total number of transmitted 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 constraint, the problem can be formalized as $\max \sum_{i=1}^{n} x_i v_i$ under the constraint that $\sum_{i=1}^{n} x_i w_i \le W$. This is exactly the Knapsack problem which is NP-hard [7]. We therefore employ a greedy algorithm: in every iteration, we select the packet that has the smallest weight over value ratio, and mark it as using TEC or HEC, until the decoding time exceeds W or until all packets have been marked.

6.2 Equivalency of CPU and Power Constraints

Our power consumption model in Section 5.2 indicates that the power consumption can be predicted by the CPU usage and the number of packets. The partial packet recovery scheme should only control the power consumed by itself, but not the power consumed by the upper layers or the applications. While it is possible to control the additional CPU consumption as in Section 6.1, controlling the number of additional packets can be challenging, because it is related to the manner the upper layers and the applications generate packets. We note that the additional packets sent by the recovery scheme are the feedbacks which are usually a small number because UNITE aggregates multiple feedbacks in one frame in many cases. On the other hand, it can be argued that UNITE does not receive additional packets compared to when no recovery scheme is used, because the partial packets are produced by the wireless channel and UNITE will only reduce the number of packets transmitted to recover the partial packets. Therefore, for simplicity, a practical choice to control the power consumption is to only control the CPU usage; in this sense, the CPU and the power constraints are equivalent and we will only consider the CPU constraint in the rest of the paper.

7 Experiments and Evaluation

We modify the open-source Madwifi driver [2] to implement UNITE and other compared schemes. We run experiments on two Toshiba Satellite U405D-S2910 laptop computers with 2.2GHz CPU running Ubuntu 10.04 LTS with kernel version 2.6.32.18. The wireless card we use is the Cisco Aironet 802.11a/b/g wireless cardbus adapter [1] based on the Atheros 5212 chipset.

7.1 Comparing with Other Drivers

We first compare UNITE with other drivers. We set the CPU constraint β and the feedback aggregation threshold ϕ for UNITE to be 0.2, and 8, respectively; other drivers do not have any constraint on CPU usage.

7.1.1 Compared Drivers

The compared drivers include:

- ORIG: The original unmodified Madwifi driver, with hardware retransmission enabled.
- BLCK: The modified Madwifi driver that uses block-retran as the only repair method.
- EOLY: The modified Madwifi driver that uses HEC as the only repair method.
- 2RND: The modified Madwifi driver enhanced with a two-round, fixed packet repair scheme according to the descriptions in [14].

BLCK, EOLY and 2RND are implemented on the same code base as UNITE, and adopt a similar link layer protocol. In BLCK, the feedback only contains the block CRCs. In EOLY, the feedback only contains AMPS samples. 2RND implements the two-round fixed transmission scheme suggested in Section 5.2 of [14]: for a partial packet, in the first round, the sender transmits parity bytes 7% of the codeword size; if the first round fails, in the second round, the sender transmits the rest parity bytes which is 18% of the codeword size. The receiver can use the parity bytes sent in both rounds to decode the partial packet. 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; packets with high BER, more than 0.3 in our implementation, are not repaired with error correction. The 2RND driver is optimal for the trace collected in [14] when no information about the error number in the packet is available. It is not the ZipTx driver which dynamically chooses the number of parity bytes [14]. We do not have a replica of the ZipTx driver because it is unclear to us how to determine the number of parity bytes based on the BER estimation of the current packet and the measured BERs of earlier packets described in Section 6.1(b) of [14].

Unless otherwise specified, we disable the hardware retransmission in the Madwifi driver because the hardware retransmission may be redundant. 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 hardware retransmissions.

7.1.2 Experiment Setup and Methodology

We use one laptop acting as the sender and one laptop acting as the receiver. All our experiments are done in indoor environments. We randomly choose 40 sender and receiver locations; for each location, each driver runs for 60 seconds at 54 Mbps. The results of other data rates are not shown because they tend to have fewer partial packets than 54 Mbps at the locations we have chosen and basically duplicate the results of 54 Mbps at low partial packet ratio. The sender use Click [13] to generate 3000 UDP 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 to saturate the link.

Ideally, the comparison between the drivers should be carried out under the same 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 repeat 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 segments according to the channel condition in this second. As the wireless channel may fluctuate, the experiment in one sender and receiver location contributes to measurements under multiple channel conditions. We refer to the fractions of the erasure packets and the partial packets as the *erasure ratio* and the *partial ratio*, respectively, which are used as the metric of channel conditions in our evaluation. We show detailed analysis of the channels with erasure ratio no more than 0.1, because a higher erasure ratio represents a very

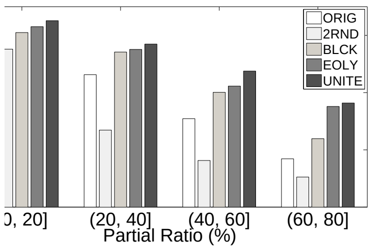


Fig. 7. The measured link throughput.

weak channel unlikely to be selected by the rate selection algorithm.

7.1.3 Throughput Comparison

Fig. 7 shows the average throughput for various partial ratio ranges. We can see that UNITE has higher throughput than other drivers at all partial ratio ranges. Employing better repair methods, the gain is higher when the partial ratio is higher, except for EOLY. EOLY has a close performance as UNITE because it also uses error correction code for packet recovery, while not under any CPU constraint.

Fig. 8 shows more detailed information about the throughput of the drivers, which compares the throughput of each one-second segment with regard to the partial ratio. For better visibility, each plot shows the performance of UNITE with one other 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 packet and must be retransmitted. 2RND only achieves good performance when the partial ratio is very small, e.g. around 5%; as the partial ratio increases, its performance drops sharply. The reason is that the 2RND receiver may spend an excessive amount of time in decoding the partial packets such that it cannot send the feedbacks in time; as a result, the sender cannot receive the feedbacks in time to clear packets in its queue and will be stalled when the queue is full. BLCK performs reasonably well when the partial ratio is low; however, as the partial ratio increases to more than 50%, its performance drops quickly. The reason is that with high partial radios, the retransmissions are more likely to fail. EOLY is better than these drivers but is still outperformed by UNITE in most cases.

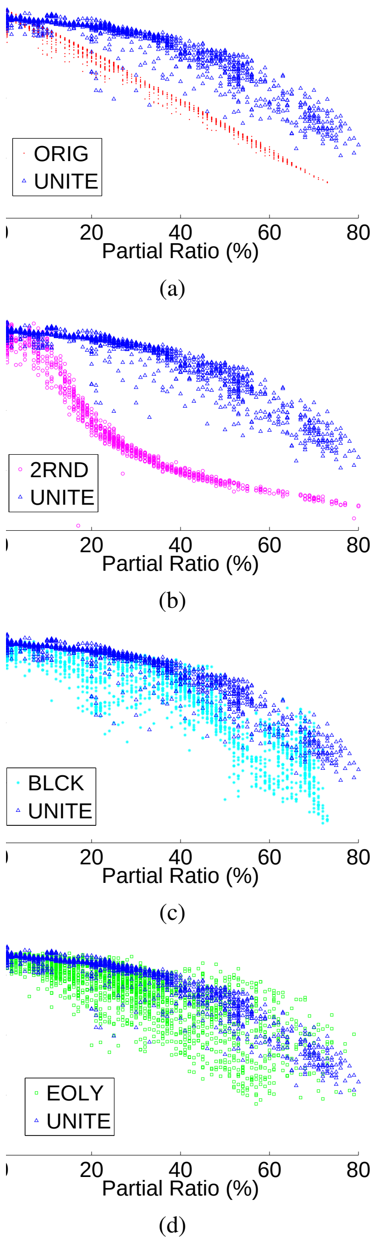


Fig. 8. Throughput comparison of UNITE with other drivers.

7.1.4 Throughput Gain Analysis

We give more detailed discussions of the throughput gain achieved by UNITE in the following.

(1) Sending Speed. One determining factor of the throughput is the activeness of the sender. We find that except ORIG, all other drivers may be forced to be idle for some time before being able to send packets again. The main reasons for the stall of the sender are the loss and delay of the feedback, because all drivers except ORIG have the sender queue and will not send packets if the sender queue is full, which may happen when the feedback cannot be received in time to clear the buffer. The other reason is that the sender may have to wait for the receiver to complete the transmission of the feedback frame. The straightforward metric for the level of activeness of the sender is the *sending speed*, defined as the average number of MAC layer frames transmitted per second, which is shown in Fig. 9. We can see that the sending speed of ORIG is relatively stable. The sending speeds of UNITE, ELOY, and BLCK drop gradually with the partial ratio, which is mainly because feedback loss increases as partial ratio increases. The sending speed of UNITE and ELOY are lower than BLCK, which is mainly because they require additional decoding time. The sending speed of 2RND, on the other hand, is pathological, because it drops very sharply when the partial ratio is more than 10% and is much lower than other drivers; the reason is that its repair parameters are not chosen optimally, leading to unnecessary

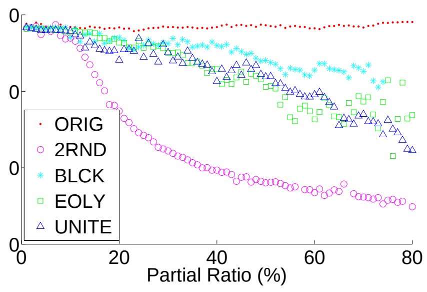


Fig. 9. The average sending speed.

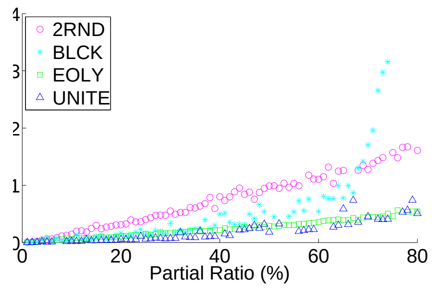


Fig. 10. The amount of repair data.

long decoding delays.

(2) Repair Data. The frames sent by the sender may contain new packets or repair data. Fig. 10 shows the fraction of time consumed in transmitting the repair data. We can see that in most cases, UNITE sends the minimum amount of repair data, which is because many packets can be repaired with TEC and the size of TEC repair data is very small. Also worth noting is that the repair data of BLCK increases sharply as the partial ratio increases, which is because the retransmission failures increase significantly, leading to more rounds of repair.

(3) Overhead. Except ORIG, all drivers incur protocol overhead, including the frame headers in the frames sent by the sender and the feedbacks sent by the receiver. Fig. 11 shows the fraction of the air time in sending the sender frame header. We can see that the frame header overhead is very small for all drivers and should have minimum impact on the throughput. 2RND has the largest sender frame header overhead because each sender frame in 2RND includes 100 pilot bits. Fig. 12 shows the fraction of the time in sending the feedbacks. UNITE has the largest feedback overhead because the feedback in UNITE includes both AMPS samples and block checksums. We note that the feedback overhead reduces the throughput by consuming the air time and reducing the sending speed; therefore, the measurement of the sending speed has included the impact of the feedback overhead on throughput.

(4) Delivery Ratio. We define the *delivery ratio* as the percentage of packets that can be delivered to the upper layer among all received packets. Some packet may never be delivered because it was corrupted but the repair eventually failed. Fig. 13 shows the delivery ratio with regard to the partial ratio. The delivery ratio of ORIG is basically the fraction of correct packets, therefore it drops almost linearly

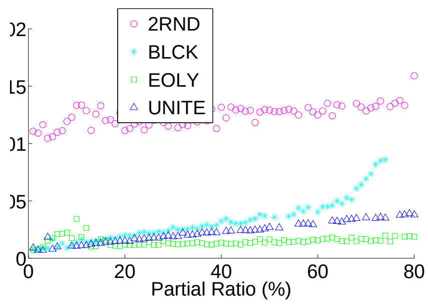


Fig. 11. The average sender frame header overhead.

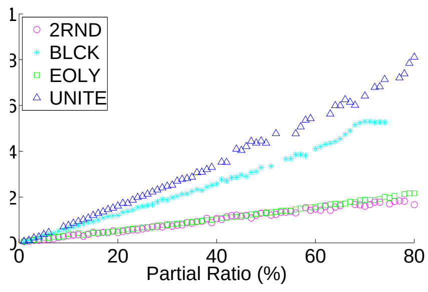


Fig. 12. The average feedback overhead.

with the partial ratio. The delivery ratios of the other drivers depend on the failure ratios of the adopted repair methods. As the partial ratio increases, the delivery ratio of BLCK drops quickly. UNITE has the highest delivery ratio, which is because the repair failure ratio of UNITE is low.

The above analysis explains why UNITE has higher throughput than other drivers. We note that roughly speaking, the throughput is proportional to the product of the sending speed, one minus the repair data fraction, and the delivery ratio. Clearly, the throughput of 2RND is limited by its low sending speed, and the throughput of ORIG is limited by its low delivery ratio. UNITE has lower sending speed than BLCK, but has much less repair data fraction and much higher delivery ratio. UNITE has similar sending speed and repair data fraction as ELOY, but has higher delivery ratio.

7.1.5 CPU Load Comparison

We measure the CPU load of the receiver for different drivers during the experiments with the Linux *vmstat* tool. Similar to ZipTx [14], 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 dose not consume significant

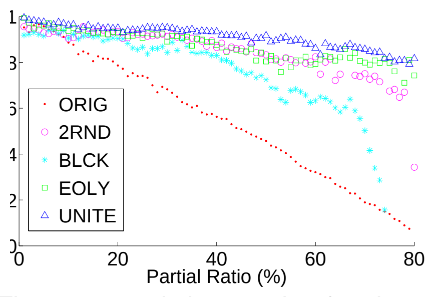


Fig. 13. The average delivery ratio of various drivers.

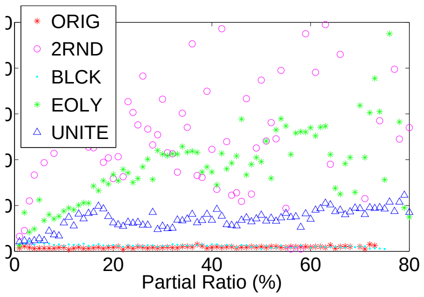


Fig. 14. Measured CPU load.

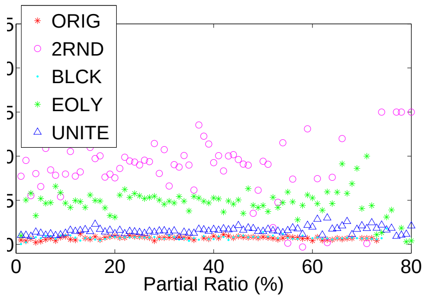


Fig. 15. Measured power consumption.

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.

Fig. 14 shows the average CPU load for various drivers with regard to the partial ratio. We can see that the CPU load of UNITE first increases as the partial ratio increases, then stabilizes at 20%. This confirms that UNITE is capable of exploiting the available CPU resources but will not over-consume the CPU. As expected, ORIG and BLCK have the smallest CPU load, both around 2% or 3%, which is because they do not require any software decoding. The CPU load of EOLY normally does not exceed 45% because excessively corrupted packets are not repaired by HEC. 2RND's CPU load, however, is much higher than other drivers, which is because its error correction parameters are not adjustable and tend to be overly conservative for the channel; it also confirms that the software decoding may consume high CPU resources in practical settings.

7.1.6 Power Consumption Comparison

We measure the power consumption of the receiver for different drivers during the experiments with the power meter *Wattsup* [3]. The machine has a baseline power consumption of 39 Watts even when there is no other running application.

Fig. 15 shows the average power consumption for various drivers with regard to the partial ratio. ORIG and BLCK have the smallest power consumption, both around 41 Watts, which is about 5% more than the baseline power consumption. The power consumption of UNITE is limited by its CPU time constraint. The power consumption of EOLY is normally around 45 Watts, which is about 15% more than the baseline power consumption. Similar to CPU usage, 2RND's power consumption is much higher than the rest of the drivers, which is because power consumption is heavily relevant to the CPU usage.

7.1.7 Delay Comparison

We also measure the *delivery latency* for different drivers during the experiments. We define the delivery latency as the time since a packet is realized by the receiver to the time the packet is correctly delivered to the upper layer. Note that the delivery latency is only for partial packets and erasure packets. Since ORIG uses the hardware retransmissions to recover packets that are not ACKed, we do not have a

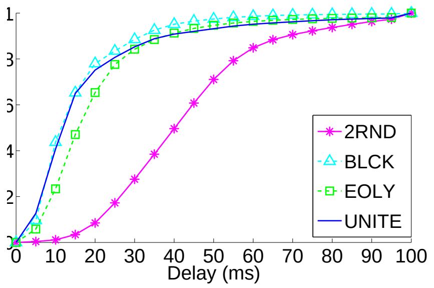


Fig. 16. CDF of average delivery latency.

method to measure its delivery latency. Therefore, we only consider the delivery latency of the other four drivers.

Fig. 16 shows the CDF of average delivery latency of various drivers. We can see that the latencies of UNITE, BLCK, and EOLY are close. As expected, 2RND has the largest delivery latency, which is because the average decoding time in 2RND is the largest.

7.1.8 Repair Methods Failure Ratio of UNITE

We measure the average repair failure ratio of the three repair methods in UNITE. Fig. 17 shows the average percentage of HEC, TEC and block-retran that failed on the first repair attempt in UNITE for various partial ratio ranges. TEC is the most efficient among all the three repair method; its failure ratio never goes beyond 5%.

7.2 UNITE under Different CPU Time Constraint

UNITE's performance is determined by the amount of system resources it is allowed to consume. We run a set of experiments in which the CPU constraint β is set to be 0.05, 0.1, 0.15, 0.2, and unlimited, at the same set of locations

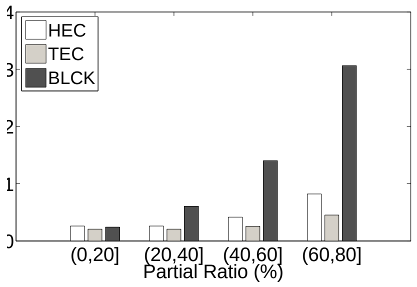


Fig. 17. The average percentage of HEC, TEC and block-retran in UNITE that failed on the first repair attempt.

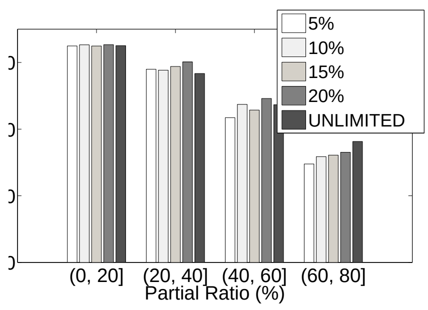


Fig. 18. Throughput when varying the CPU constraint.

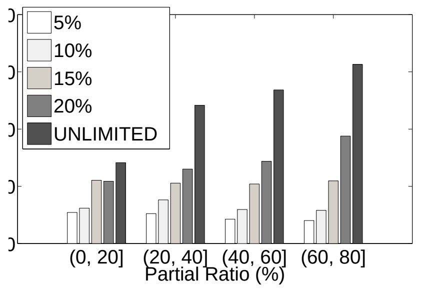


Fig. 19. CPU load when varying the CPU constraint.

as the experiments in Section 7.1.1 at 54 Mbps. When β is set to be unlimited, there is no scheduling algorithm; all packets qualified for TEC are repaired with TEC, other packets, if qualified for HEC, are repaired with HEC.

Fig. 18 and Fig. 19 show the average throughput and the average measured CPU load for different CPU constraints, respectively. We can see that when the channel is good, i.e., the partial ratio is no more than 20%, the throughputs are similar. However, when partial ratio is higher, a larger CPU time constraint generally leads to a larger throughput, which is mainly because with a larger CPU constraint, more packets are repaired by HEC than by block-retran while HEC is more efficient than block-retran. This confirms that UNITE is capable of capitalizing the allowed CPU resource for higher link throughput. The average measured CPU load does not exceed the constraint value, which confirms that UNITE is capable of limiting the CPU load. The CPU load of the unlimited is never higher than 40%, 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 limited.

One of the key features of UNITE is that it supports multiple repair methods and makes dynamic selections of the repair methods. Fig. 20 shows the number of packets repaired by different methods per second for various partial ratio ranges and various CPU constraints. We can see that a large portion of partial packets are repaired by TEC even with the lowest β. With a larger β, more partial packets are repaired with HEC which is more time consuming.

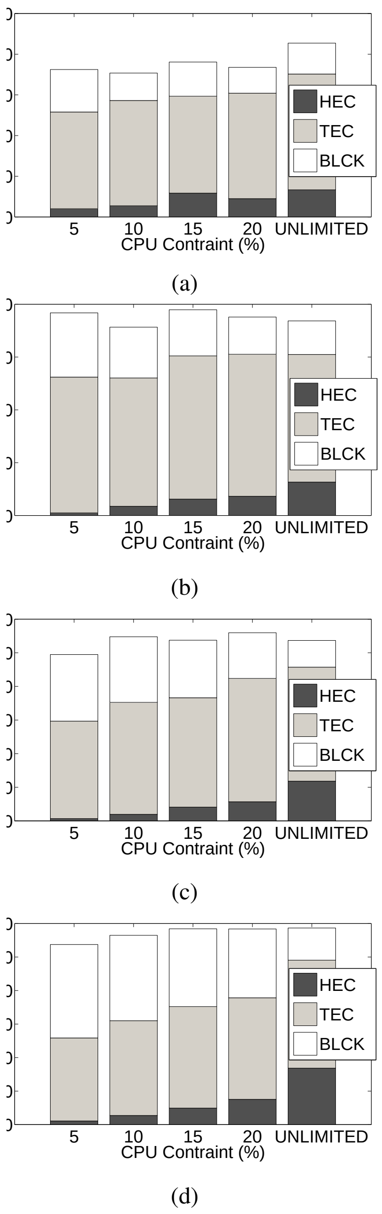


Fig. 20. The choice of repair methods in different partial ratio ranges. (a). Partial ratio (%) (0, 20]. (b). Partial ratio (%) (20, 40]. (c). Partial ratio (%) (40, 60]. (d). Partial ratio (%) (60, 80].

8 Discussions and Extensions

Currently, UNITE is implemented on the MadWifi driver. UNITE can also be implemented on other platforms as long as the hardware can deliver partial packets to the driver. The design of UNITE need not be changed; UNITE only needs to calibrate the CPU and power estimation models for the new platform. We note that the CPU model estimates the CPU usage based on the software decoding time, which can be measured by the driver when the driver is installed. The power consumption model is a linear model with multiple parameters. In this paper, we have learned the coefficients of the parameters with measurements on the power meters. Without a power meter, a possible approach is to build the power consumption models for various hardware configurations and choose a suitable model based on the configuration of the machine.

UNITE is a software solution largely independent of the physical layer. Currently, UNITE is implemented and tested on 802.11a. We note that UNITE will still run on faster links with 802.11n or the future 802.11ac. The main challenge with faster links is the increased decoding demand, as more packets are transmitted in the same amount of time and more may require decoding. We note that this will likely be addressed by exploiting additional computing capacity in multi-core processors which are widely used today. One possible scenario, for example, is to implement the decoding job in a thread and run it in one of the cores that may otherwise be idle.

9 Conclusions

In this paper, we study software partial packet recovery in 802.11 wireless LANs. Our main contributions include the following. First, we propose to employ both the EC-based and block-based repair methods, as well as a novel repair method, TEC, which is very efficient for recovering a large number of packets with very few errors. With multiple options, UNITE can use the best repair method to match the error condition of the packet under the system resource constraints. Second, we design an error estimator, AMPS, to estimate the number of errors in a partial packet to assist intelligent decisions on packet recovery. AMPS has an overhead of only 8 bytes per partial packet and can be implemented with simple table lookup. Third, we propose the CPU usage and power usage models for partial packet recovery. Fourth, we design an algorithm to make selections among the repair methods based on the error estimate and the system resource constraint, such that UNITE will not over-consume the CPU or power. We implement UNITE on the MadWifi open-source driver. Our experiments confirm that UNITE outperforms all existing schemes while consuming the system resources under the specified constraints.

References

[1] Cisco aironet 802.11a/b/g wireless cardbus adapter. http://www.cisco.com/.

[2] The madwifi project. http://madwifi-project.org/.

[3] The wattsup power meter. https://www.wattsupmeters.com/.

[4] B. Chen, Z. Zhou, Y. Zhao, and H. Yu. Efficient error estimating coding: feasibility and applications. In *ACM SIGCOMM*, pages 3–14, New Delhi, India, 2010.

[5] J. Cohen, P. Cohen, S.G. West, and L.S. Aiken. *Applied multiple regression/correlation analysis for the behavioral sciences*. Routledge Academic, 2002.

[6] H. Dubois-Ferrière, D. Estrin, and M. Vetterli. Packet combining in sensor networks. In *ACM SenSys*, pages 102–115, San Diego, CA, USA, 2005.

[7] M. R. Garey and D. S Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[8] 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 *USENIX NSDI*, San Jose, CA, USA, 2010.

[9] A.P. Iyer, G. Deshpande, E. Rozner, A. Bhartia, and L. Qiu. Fast resilient jumbo frames in wireless LANs. In *IEEE IWQoS*, Charleston, SC, USA, June 2009.

[10] K. Jamieson and H. Balakrishnan. PPR: Partial packet recovery for wireless networks. In *ACM SIGCOMM*, pages 409–420, Kyoto, Japan, 2007.

[11] P. Karn. DSP and FEC Library. http://www.ka9q.net/code/fec/.

[12] S. Katti, D. Katabi, H. Balakrishnan, and M. Medard. Symbol-level network coding for wireless mesh networks. In *ACM SIGCOMM*, pages 401–412, Seattle, WA, USA, 2008.

[13] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, August 2000.

[14] K. C.-J. Lin, N. Kushman, and D. Katabi. Ziptx: Harnessing partial packets in 802.11 networks. In *ACM MobiCom*, pages 351–362, San Francisco, CA, USA, 2008.

[15] S. Lin and D. J. Costello. *Error Control Coding: Fundamentals and Applications*. Prentice Hall, 2004.

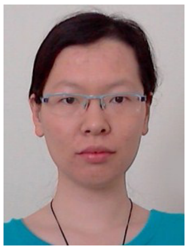
[16] M.-H. Lu, P. Steenkiste, and T. Chen. Design, implementation and evaluation of an efficient opportunistic retransmission protocol. In *ACM MobiCom*, pages 73–84, Beijing, China, 2009.

[17] A. Miu, H. Balakrishnan, and C. E. Koksal. Improving loss resilience with multi-radio diversity in wireless networks. In *ACM MobiCom*, pages 16–30, Cologne, Germany, 2005.

[18] G. R. Woo, P. Kheradpour, D. Shen, and D. Katabi. Beyond the bits: Cooperative packet recovery using physical layer information. In *ACM Mobicom*, pages 147–158, Montreal, QC, Canada, 2007. ACM.

[19] J. Xie. *Design, implementation, and Evaluation of an Efficient Software Partial Packet Recovery System in 802.11 Wireless LANs with Configurable Resource Usage*. PhD thesis, Florida State University, 2012.

[20] J. Xie, W. Hu, and Z. Zhang. Revisiting partial packet recovery in 802.11 wireless LANs. In *ACM MobiSys*, pages 281–292, Bethesda, MD, USA, 2011.



J in Xie received her B.E. degree in Computer Science and Technology from China University of Petroleum, Beijing, China, in 2005, and her M.S. degree in Applied Mathematics from Zhejiang University, China, in 2007. She received her Ph.D. degree in Computer Science from Florida State University in 2012. Her research interest is in wireless networks. Currently, she is a Software Engineer at Aruba Networks.

W ei Hu received his B.S. degree and M.Eng. degree from Huazhong University of Science and Technology in 2005 and 2007, respectively. He received his Ph.D. degree in Computer Science from Florida State University in 2012. His research interest is in wireless networks. Currently, he is a Software Engineer at Aruba Networks.

Zhenghao Zhang (M'02) received his B.Eng. and M.S. degrees in electrical engineering from Zhejiang University, Hangzhou, China, in 1996 and 1999, respectively. He received his Ph.D. degree in electrical engineering from the State University of New York at Stony Brook in 2006. From 1999 to 2001, he worked in industry as an embedded system Software Engineer. From 2006 to 2007, he was a Postdoctoral Researcher in the Computer Science Department at Carnegie Mellon University. Currently, he is an Assistant Professor in the Computer Science Department at Florida State University, Tallahassee, FL. His research interests include wireless networks, network security, and high speed optical networks.