

Revisiting Partial Packet Recovery in 802.11 Wireless LANs

Jin Xie, Wei Hu, Zhenghao Zhang
Florida State University



Wireless Transmission

- In a wireless link, a packet may be
 - Fully received
 - Erased
 - Partially received



Packet Recovery

- Erased Packets
 - Sender has to retransmit the entire packet.
- Partial Packets
 - Current 802.11 will retransmit!
 - But partial packets often has only few errors. Whole packet retransmission is not efficient.

Partial Packet Recovery

- Block based approach
 - Example: Maranello [NSDI 2010]
 - Divide the packet into blocks. Retransmit the corrupted block.
 - A block is found to be corrupted if it fails the checksum test

Partial Packet Recovery

- Error Correction based approach
 - Example: ZipTx [Mobicom 2008]
 - Divide the packet into blocks and encode into codewords. If there are errors, send parity bytes

Motivations to Revisit this Problem

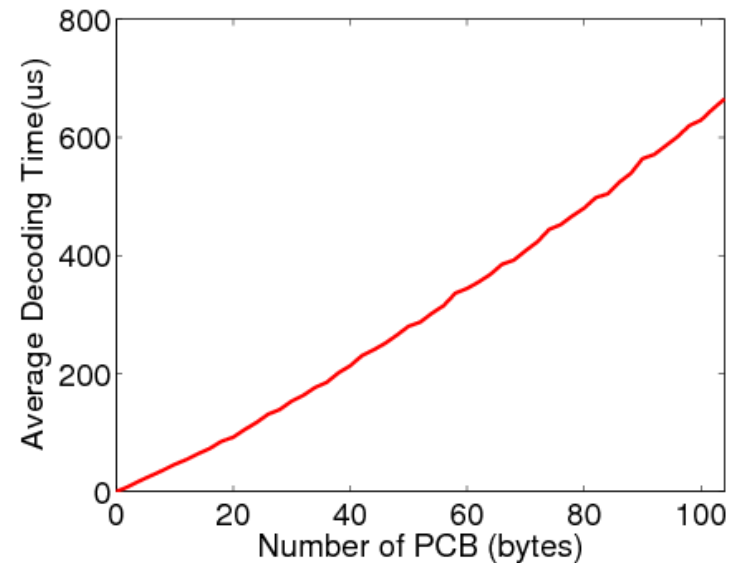
- Combining the two repair methods
 - They are not mutually exclusive

Motivations to Revisit this Problem


- Using error estimator
 - Determine the optimal repair method and repair parameters
 - Error estimators were not available to earlier schemes

Motivations to Revisit this Problem

- Observing the CPU constraint
 - Make sure that packet recovery does not consume too much CPU time.
 - Software decoding can be time consuming



Our Contribution

- Combine block-based and EC based and use three repair methods:
 - Block-retran: Traditional block retransmission.
 - HEC: Holistic Error Correction. Traditional error correction.
 - TEC: Target Error Correction. 
- Select the optimal repair method that
 - Minimizes the number of bytes sent
 - Abide by the CPU time constraint β

Combining Block-retran and Error Correction

- Block-retran and Error Correction have pros and cons:
 - Block-retran
 - does not need much CPU time
 - but retransmitted blocks may be corrupted and may transmit more data
 - Error Correction
 - need CPU time
 - but is more resilient to errors and need less data

Combining Block-retran and Error Correction

- When CPU time is a constraint, it makes sense to take advantage of both:
 - For more corrupted packets, use block-retran
 - For more lightly corrupted packets, use HEC
- *The number of errors in a packet is estimated by the estimator.*

Code block, interleaving, and checksum block

- Code Block:
 - Divide a packet into blocks, encode each into a codeword
- Interleaving:
 - Simply a random permutation
 - To spread the errors evenly in the code blocks
 - So we send the same number of parity bytes for all code blocks.
- Checksum block:
 - Divide the packet into (smaller) blocks
 - It is **after** interleaving because when retransmitting we prefer errors in clusters

Feedback of a Partial Packet

- The checksum is not transmitted with the data.
 - Only when the packet is a partial packet, the checksum is calculated at the receiver and sent as a feedback to the sender.
- Also in the feedback is the estimated number of errors
 - With which the number of parity bytes for each code block can be determined

TEC

- In addition we have TEC:
 - A partial packet often has only few error bytes.
 - For such packets,
 - block-retran will waste in transmitting correct bytes.
 - EC will waste in decoding correct code blocks.
 - Why not find the error checksum block (by checksum test) then send parity bytes for this checksum block only?

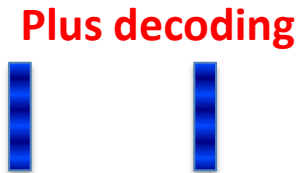
A partial packet:



Block-retran:



Error Correction:



TEC:



HEC and TEC

- Why not use TEC for all packets and not worry about HEC at all?
- The problem is that we do not know the number of errors in each checksum block, so we actually do not know how many parity bytes to send if there are many error blocks.

HEC and TEC

- If a packet has very few errors (no more than 3 in our implementation), it is qualified for TEC.
 - There are usually very few error checksum blocks and we fit them into one codeword.
 - We send 10 (no more than 1 error) or 20 (no more than 3 errors) parity bytes.
- If a packet has more errors, we have to rely on interleaving to spread the errors evenly in all code blocks.

How do we know how corrupted the packet is?

- Knowing which block is corrupted is easy – failed CRC.
- Knowing how many error bytes is difficult – one error byte and 100 error bytes both result in a CRC failure.
- Have to use error estimator.

AMPS

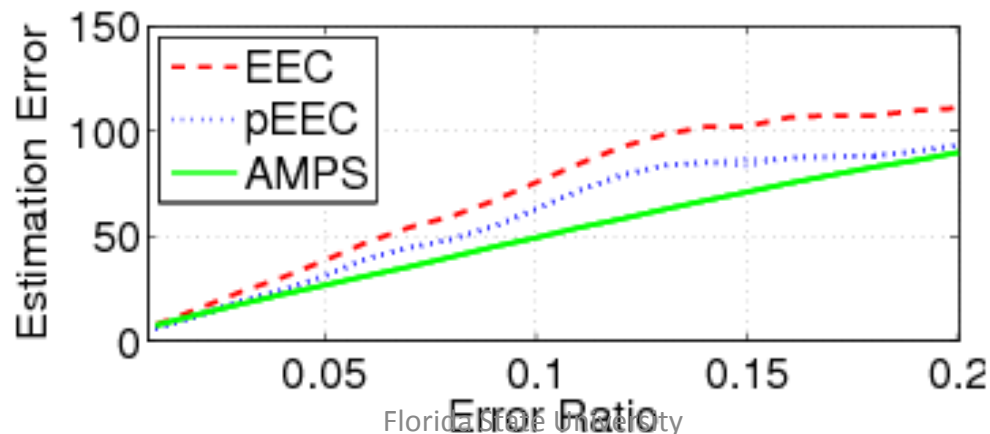
- We are aware of two estimators, EEC [Sigcomm 2010] and AMPS.
- All details about AMPS, including the table lookup implementation, getting the prior $P(Y)$ are in:
“Employing coded relay in multihop wireless networks”, <http://arxiv.org/abs/1012.4136>

AMPS

- AMPS estimates the number of errors in a packet based on the Maximum A Posteriori (MAP) criterion.
 - The parity bit of multiple bytes is a sample. Multiple samples are sent in the packet header. The receiver compares the received samples and the locally calculated samples. The number of mismatches (X) reveals the error conditions (Y).
 - We pick Y such that $P(Y/X)$ is maximized.

AMPS

- Comparing with EEC [Sigcomm 2010]
 - 1500-bit packet for EEC and 1500-byte packet for AMPS.
 - EEC uses all 10 levels with overhead 40 bytes. AMPS has overhead 8 bytes.
 - pEEC removes the results when EEC cannot estimate.
- Main reason
 - EEC is a heuristic with a bound, AMPS follows the MAP optimality criterion.
 - For example, EEC has a hard threshold which often leads to underestimation.



Selection of Repair Method

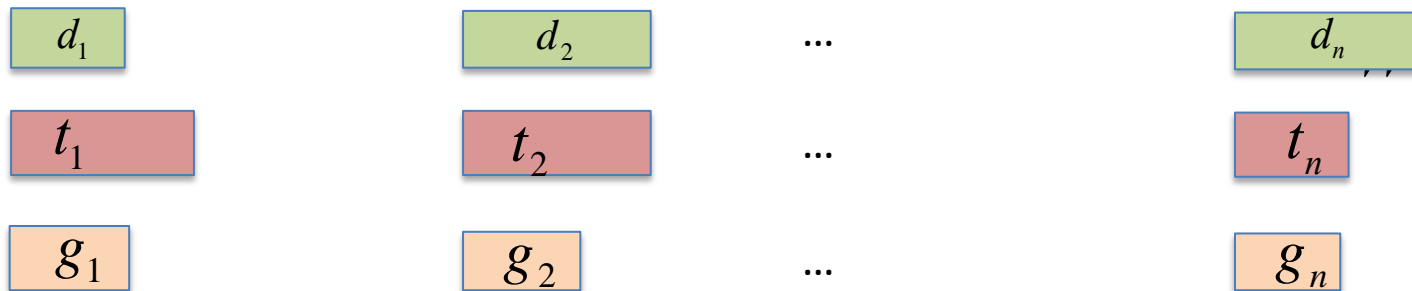
- A node may receive the feedback of multiple packets from a feedback frame.
 - Which repair method to apply for each packet?

Selection of Repair Method – Some simple rules first

- If a packet is qualified for TEC, don't use HEC
 - Means that we are always choosing between block-retran and one error correction method. Binary.
- If the more repair data has to be sent for error correction than with block-retran, use block-retran
 - Does not happen often but could happen

Selection of Repair Method

- For each partial packet, we have
 - d_i : the decoding time
 - t_i : the transmission time for block-retran
 - g_i : the transmission time for required parity bytes



- We are given the decoding time budget



Selection of Repair Method

- For packet P_i , we define a *value* v_i and a *weight* w_i
 - $v_i = t_i - g_i$
 - $w_i = d_i$
- For packet P_i we define binary variable x_i :
 - $x_i=0$ means should use block-retran
 - $x_i=1$ means should use error correction.

Selection of Repair Method

- 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, and we adopt the standard greedy algorithm:

Select the packet that has the largest ratio of value over weight for error correction.

The Decoding Time Budget

- At the Receiver
 - γ : the average time that the receiver encounters a partial packet
 - β : the CPU time constraint
 - N : the total number of partial packets in this feedback

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

Implementation

- We implemented our scheme within the Madwifi open source driver.
- We call it Unite.
- We use a link layer protocol similar to ZipTx:
 - Receiver aggregates the feedback of several packets into one feedback frame to reduce the overhead
 - Sender has a timeout mechanism to retransmit, in case all feedbacks are lost
 - A partial packet is repaired twice

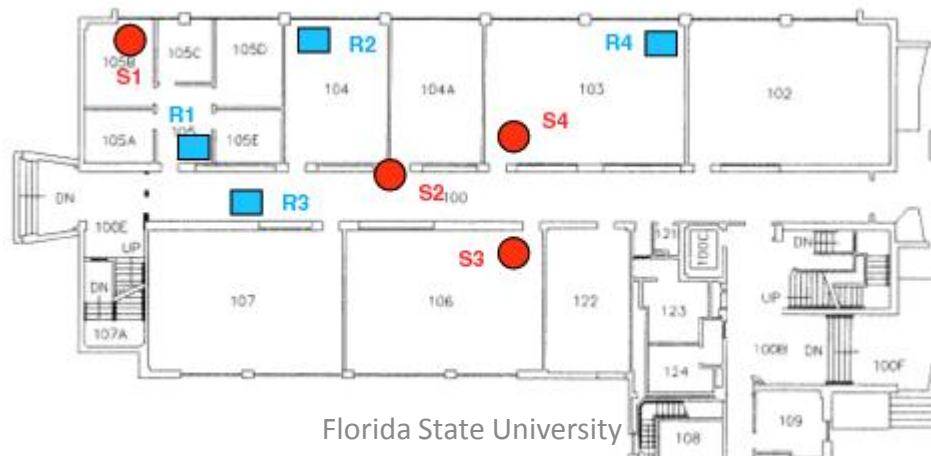
Experiments

Comparing with Other Drivers

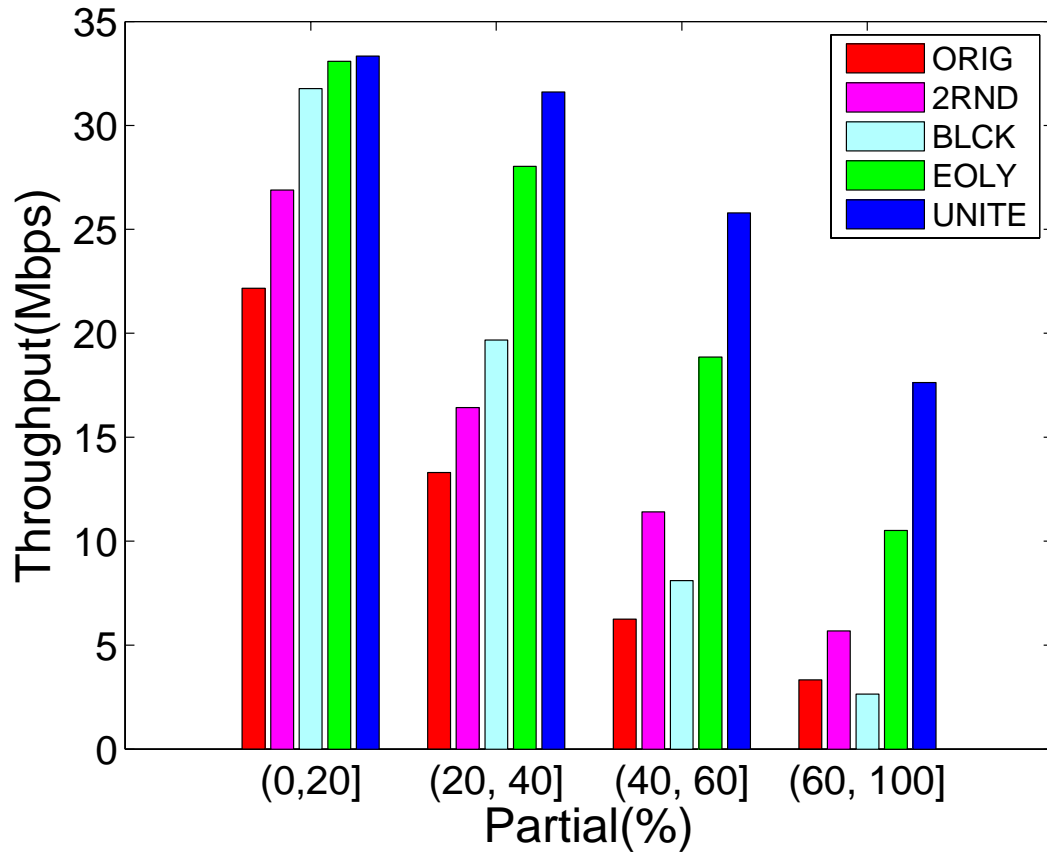
- **ORIG**
 - Original Madwifi driver
- **BLCK**
 - Original Madwifi driver enhanced with block-retran
- **EOLY**
 - Original Madwifi driver enhanced with HEC and AMPS under CPU time constraint
- **2RND**
 - Original Madwifi driver enhanced with a 2 round, fixed packet repair schedule according to ZipTx

Experiment Setup

- Randomly choose 60 sender and receiver locations
- Each driver runs for 45 seconds at 54 Mbps
- We collect per second data.
- $b = 0.2$.



Throughput Comparison

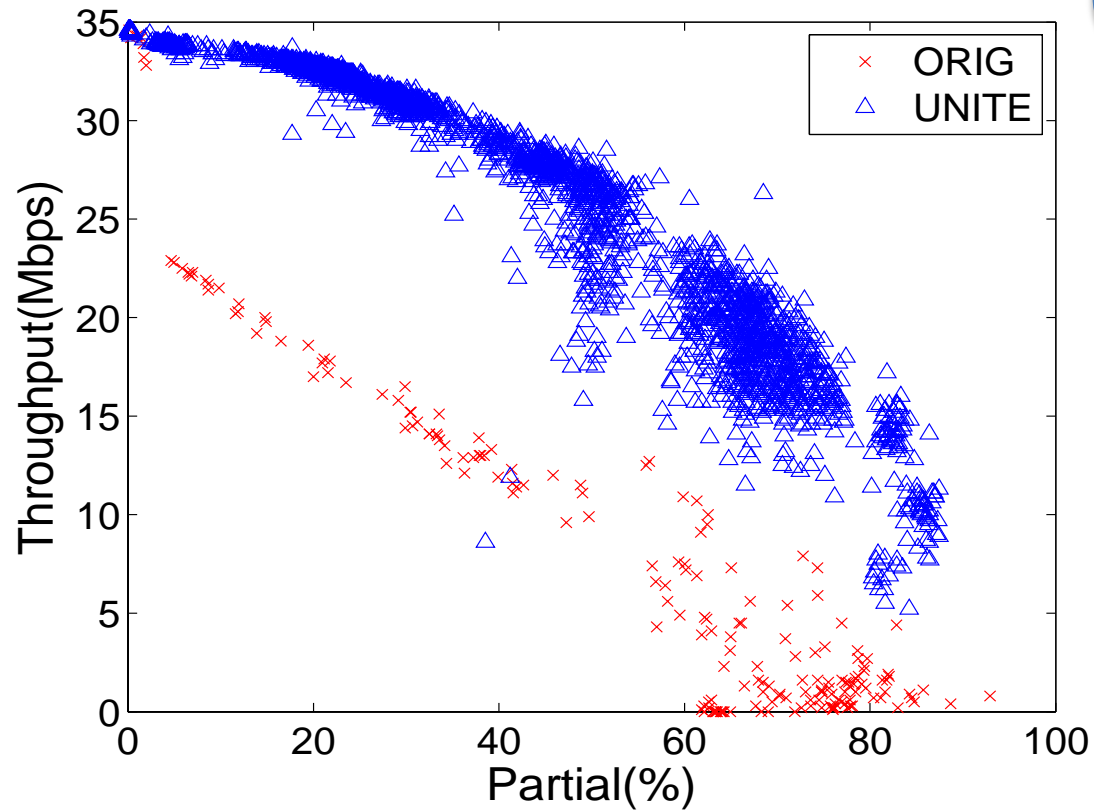


Unite achieves higher throughput than other schemes

Fine-grained Throughput Comparison

- Comparison with ORIG

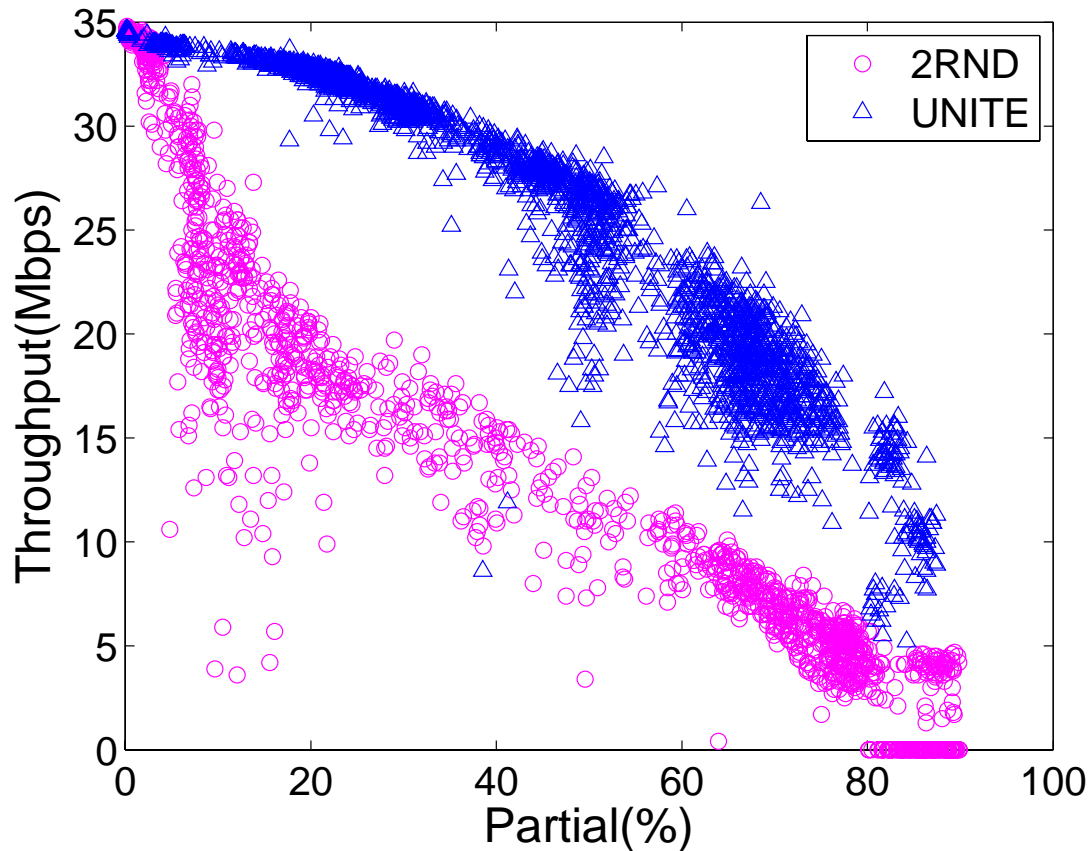
ORIG
performance
drops linearly



Fine-grained Throughput Comparison

- Comparison with 2RND

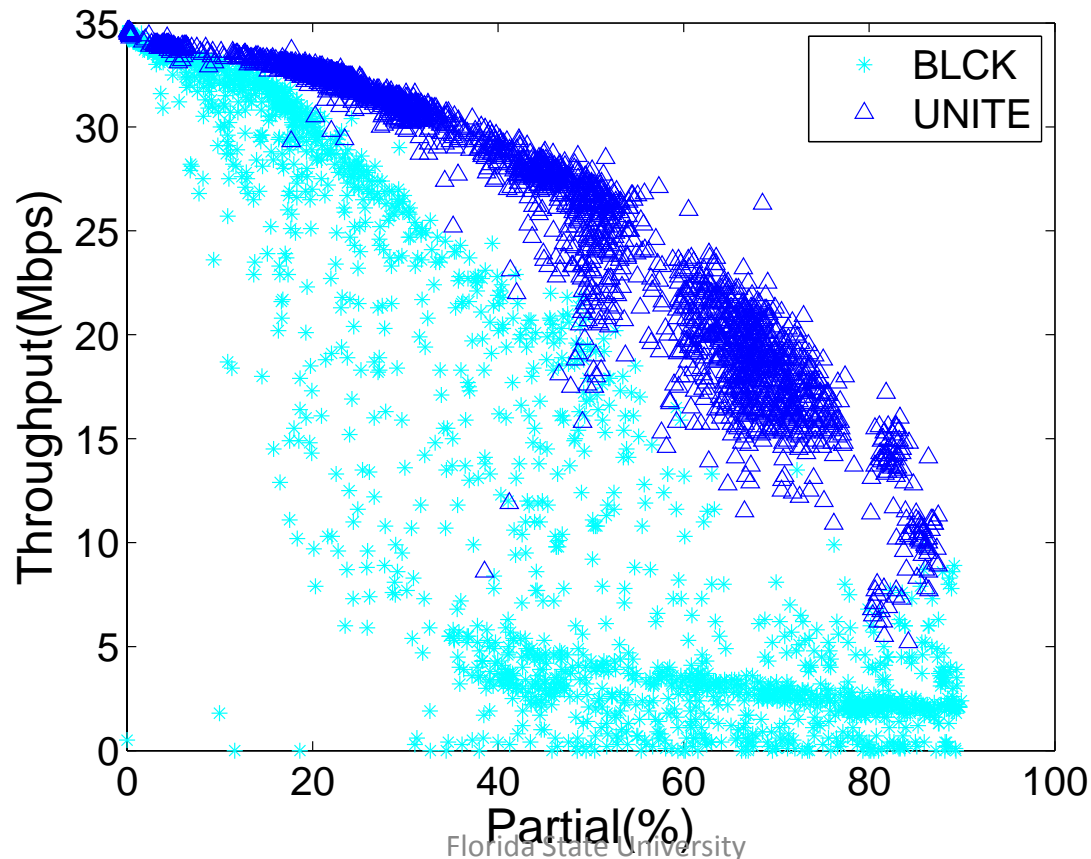
2RND is good when error is low, but drops fast as error goes high



Fine-grained Throughput Comparison

- Comparison with BLCK

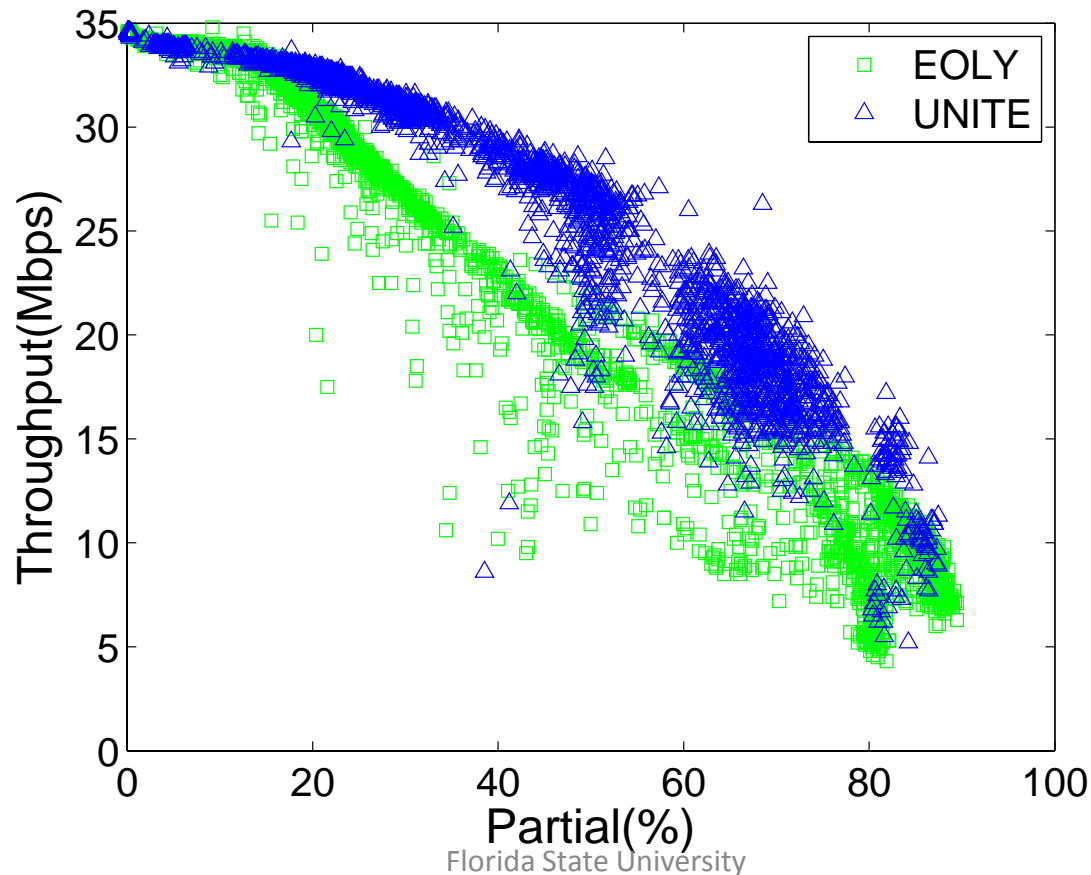
BLCK is good when error is low, but drops quickly when error is high



Fine-grained Throughput Comparison

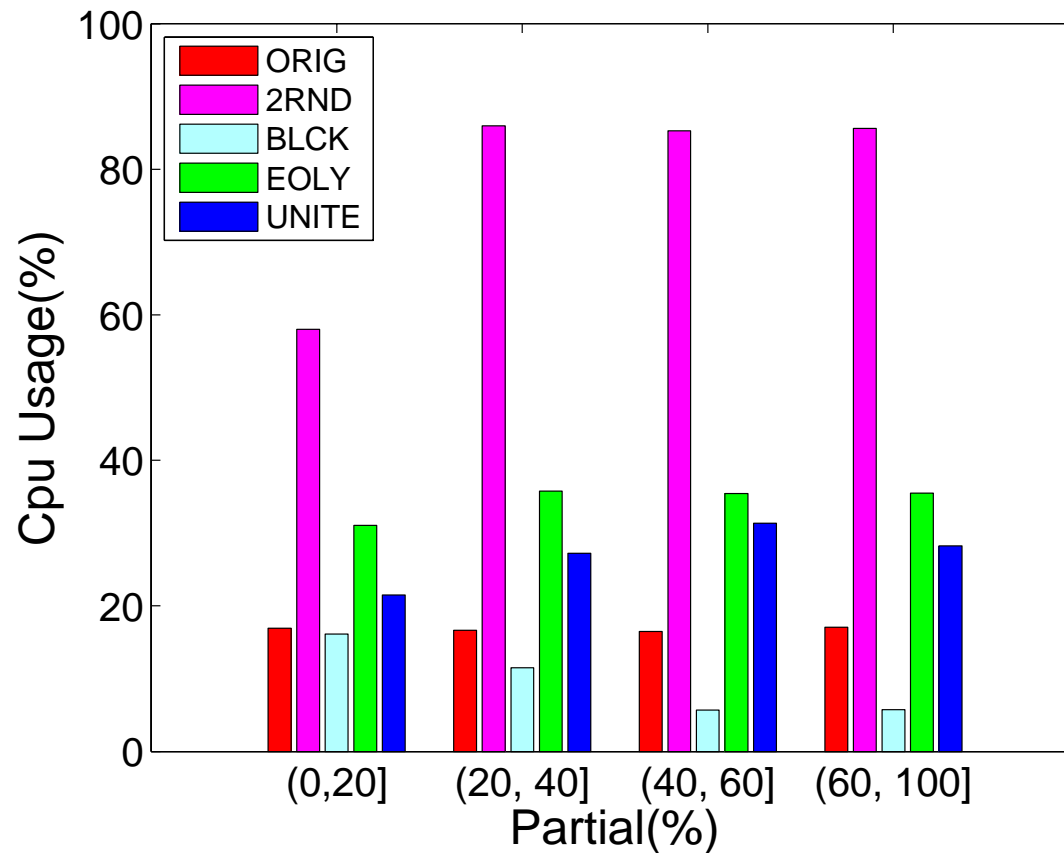
- Comparison with EOLY

EOLY follows most closely with Unite but is still worse

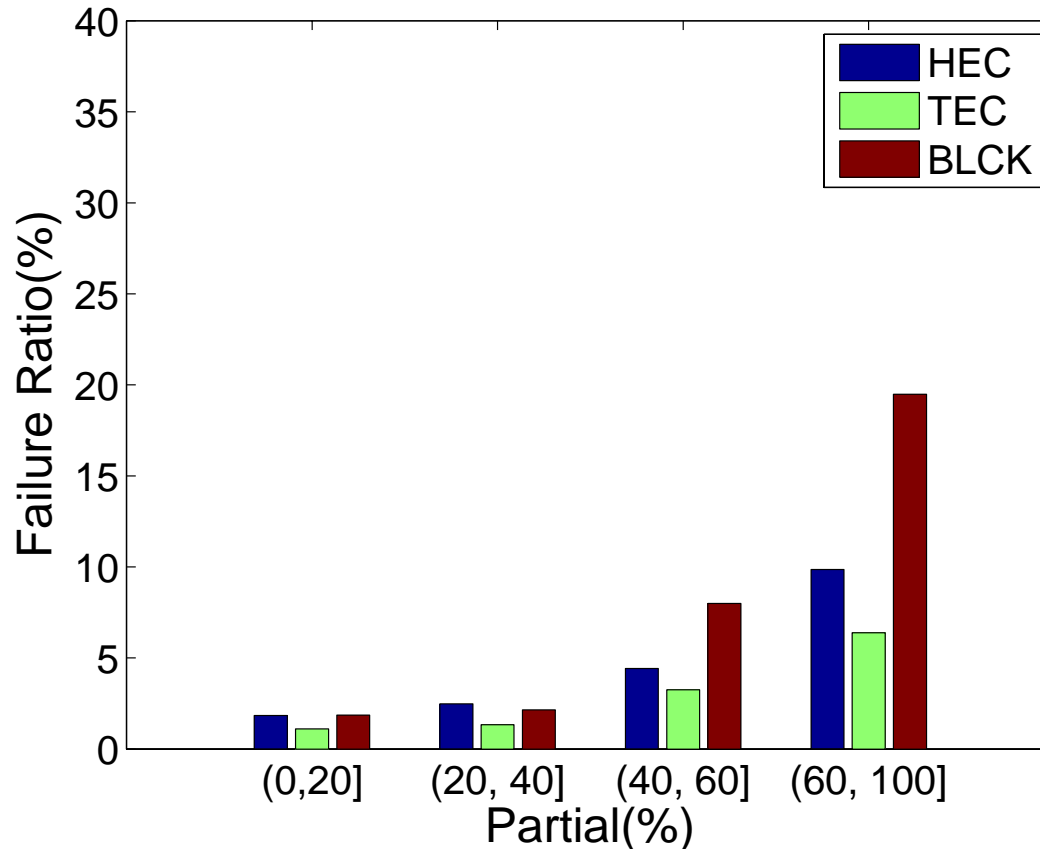


CPU Load Comparison

Unite does not over consume CPU time

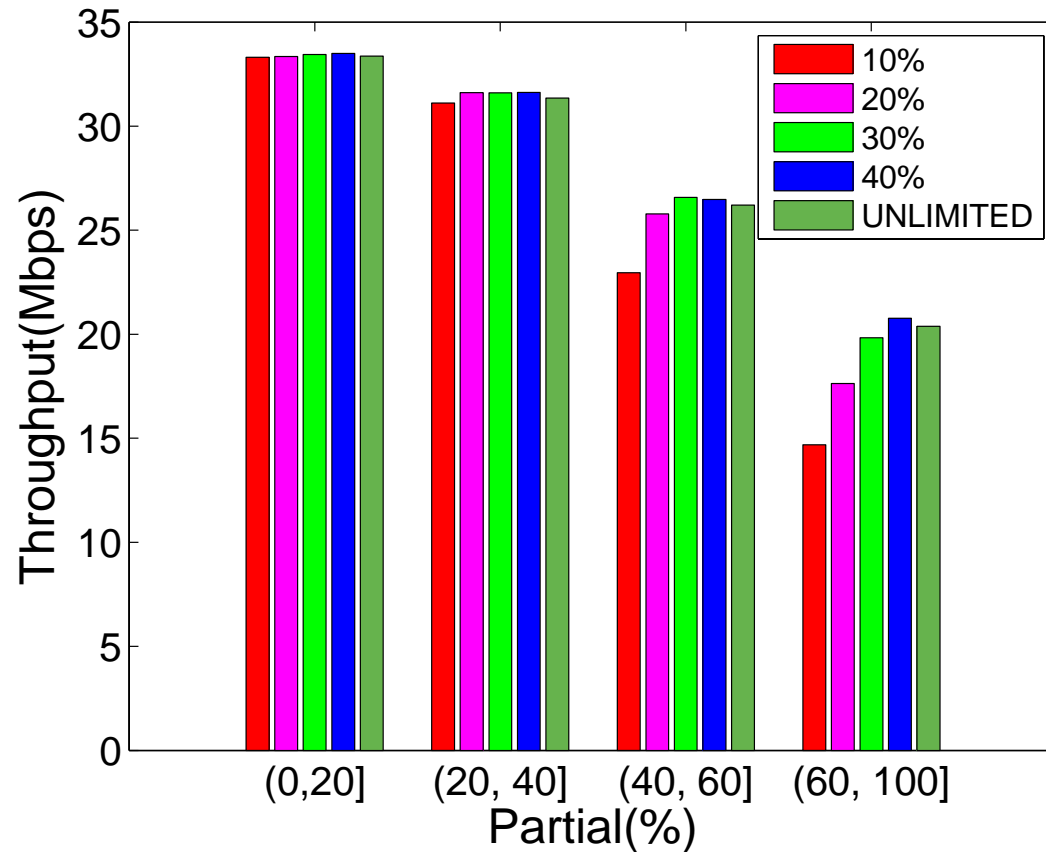


Repair Method Failure Ratio of Unite



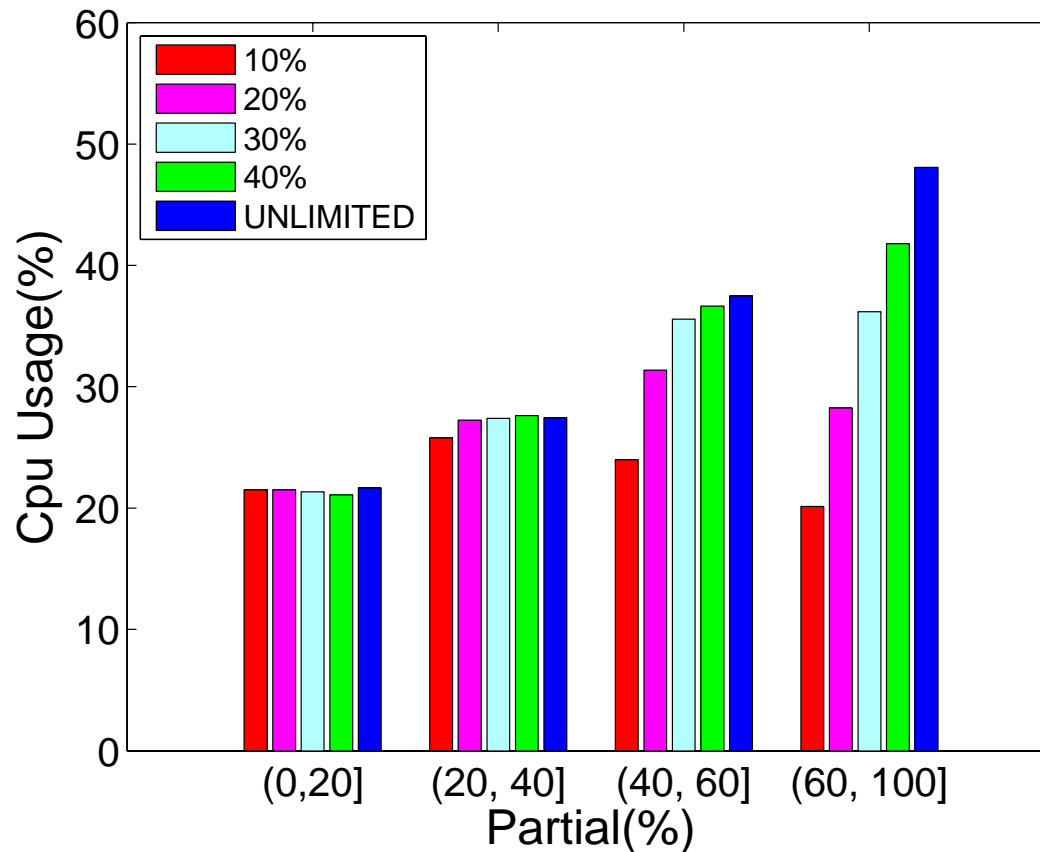
Repair may fail, but Unite's failure ratio is not high. TEC and HEC's failure ratio is less than 0.1, meaning AMPS is doing a good job

Throughput under Different CPU Time Constraint



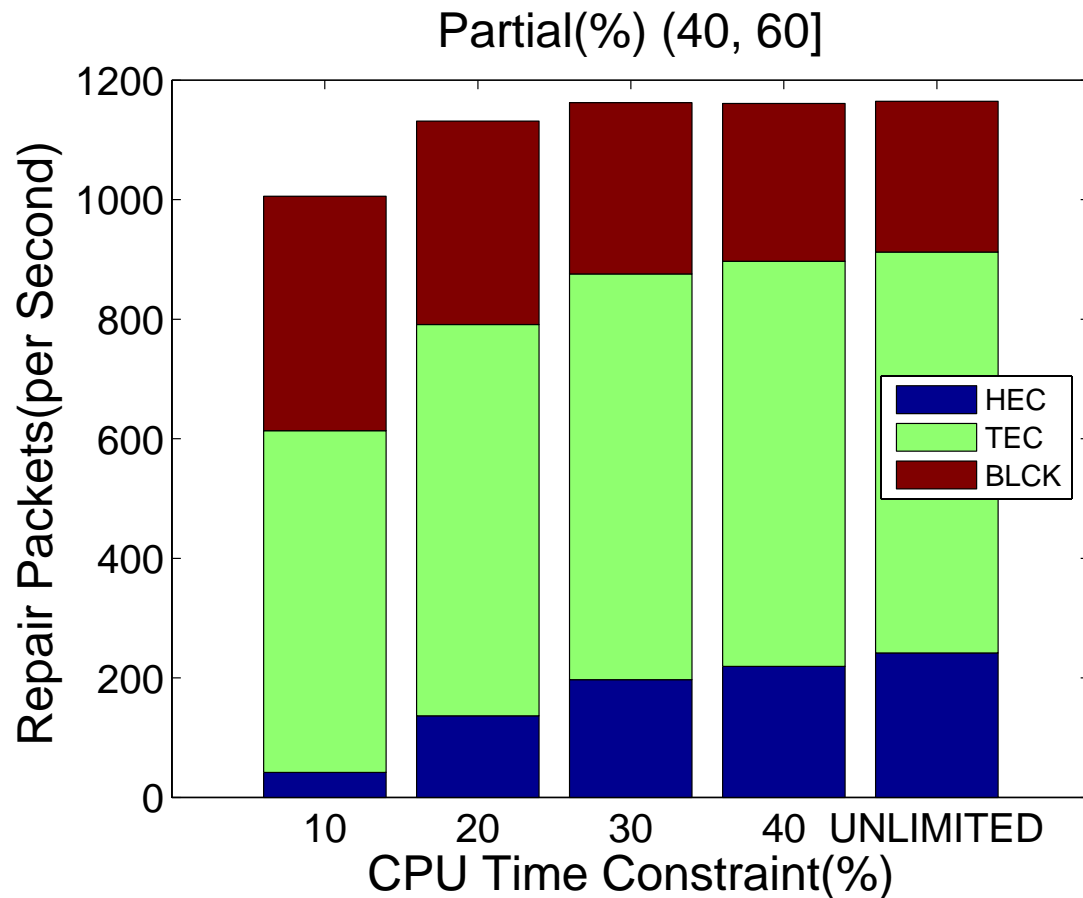
Throughput increases with more CPU time allowed

Measured CPU Load under Different CPU Constraint



CPU usage increases with more CPU time allowed; never above 50% because heavily corrupted packets are not repaired with error correction

The Choice of Repair Method Under Different CPU Constraint



- Majority of packets repaired with TEC.
- More allowed CPU time, more HEC.



Questions?