pMORE: Exploiting Partial Packets in Opportunistic Routing

Wei Hu, Jin Xie, Zhenghao Zhang
Computer Science Department
Florida State University
Tallahassee, FL 32306, USA

***Abstract*—Opportunistic routing improves the performance of multi-hop wireless networks by exploiting the overhearing opportunities. However, without special hardware support, the current opportunistic routing protocols cannot exploit partial packets, i.e., packets with errors but still containing much useful information. In this paper, we present pMORE, a software-only opportunistic routing protocol capable of exploiting partial packets. pMORE performs random network coding at the granularity of blocks and uses block checksum test to efficiently locate correct information in a partial packet and avoid the risk of propagating errors. We design an algorithm to assemble packets with partial packets to add maximum fresh information while maintaining a low overhead. The experiment results from a 12-node indoor wireless testbed show that pMORE achieves higher throughput than MORE with a median throughput gain of $1.34\times$.**

I. Introduction

Opportunistic routing [1], [2], [4] has been proposed in recent years to improve the performance of wireless multi-hop networks by exploiting the broadcast nature of the wireless medium: when a node transmits, many nodes can opportunistically overhear, even the relatively far nodes. Existing opportunistic routing protocols such as ExOR [1] and MORE [2], however, do not exploit the other property of the wireless medium, namely the existence of partial packets. Partial packets are packets with errors but still contain much correct information, which can be seen very often in wireless transmissions [5], [15], [11]. The percentage of partial packets is higher for the longer links with higher loss ratios, which opportunistic routing protocols attempt to exploit. Currently, protocols such as ExOR [1] and MORE [2] discard partial packets. As recent study [11] show that many corrupted packets just contain very few errors, it is clear that higher performance can be achieved by exploiting partial packets in opportunistic routing. Indeed, MIXIT [4] has been proposed as an opportunistic routing protocol that uses partial packets which has shown significant gain over MORE [2]. However, MIXIT [4] is built on special hardware where each node can report the physical layer information, i.e., the confidence of a physical layer symbol, to the upper layer, and is not applicable to many of the existing hardware platforms.

In this paper, we introduce pMORE, a *software-only* solution that exploits partial packets in opportunistic routing without requiring special hardware. pMORE is built as an non-trivial extension of MORE [2], and inherits many of the features of MORE [2] including random network coding. The first challenge is that random network coding requires the random mixing of packets and the errors in one partial packet can potentiality spread to all packets which renders the system useless. MIXIT [4] solves this problem by relying on physical layer hints and only mixing the symbols that are likely to be correct. pMORE, however, does not have access to the physical layer and adopts a new solution. Basically, we divides a packet into *blocks* and adds a checksum for each block; only the blocks that pass the checksum test can be mixed with other blocks. In this way we ensure that all data involved in the random network coding is correct and prevent error propagation. The second challenge is that by using partial packets, we also introduce more overhead per packet as different blocks in a packet may use different vectors for network coding where the size of a vector is non-trivial. To this end, we develop a heuristic algorithm that attempts to minimize the overhead while exploiting as many available blocks as possible. We implement pMORE on top of the code base of MORE at [10] and test pMORE with extensive experiments on a 12-node indoor wireless testbed. We find that pMORE achieves higher performance than MORE with a median throughput gain of $1.34\times$.

The rest of the paper is organized as follows. Section II describes the design of pMORE . Section III describes our experimental evaluation. Section IV discusses related works. Section V concludes the paper.

II. pMORE Design

We discuss the design of pMORE in this section.

A. Random Network Coding

pMORE is based on MORE [2] and uses *random network coding*, briefly described as follows. Basically, with random network coding, a node sends random linear combinations of the received packets. The coefficients used in the linear combination is called the *code vector* and are transmitted

TABLE 1
DEFINITIONS USED IN THE PAPER.

Term	Definition
Data Block	A data packet is divided into L data blocks.
Coded Block	A transmitted packet contains L coded blocks. Also referred to simply as block.
Code Vector	The vector of coefficients.
Segment	A segment contains one or more blocks, which share the same vector. The blocks in a segment do not need to be adjacent.
Full Segment	A segment with all correct L blocks.
Partial Segment	A segment with less than L blocks.

along with the packet. Other nodes, after received a sufficient number of such linear mixes, may recover the original packets by solving linear equations. Roughly speaking, the main advantage of random network coding in multi-hop wireless networks is that an upstream node does not need to know what packets its downstream nodes have received. By sending random mixes, the packets sent by the upstream node contain useful information with high probability.

B. Overview of pMORE

As mentioned earlier, the key difference between pMORE and MORE [2] is that pMORE accepts partial packets while MORE discards such packets. The main challenge in processing partial packets for random network coding is that a packet may be mixed with many packets and an error may propagate to many packets. To solve this problem, pMORE divides packets into blocks and sends checksums of the blocks along with the packet. A block fails the checksum test is discarded, thus pMORE does not propagate any errors. This, however, may come at a cost of a higher overhead. We note at because pMORE's network coding is at the granularity of blocks, every block may potentially need a separate code vector, the size of which is non-trivial. To address this challenge, we design a packet assembling algorithm which attempts to use the same code vector for as many blocks as possible in a packet, while sending as much useful information as possible. The algorithm is inspired by the greedy algorithm for *Set Cover*.

C. Preliminaries

pMORE forwards packets from the *source* to the *destination* with the assistance of a set of *forwarders*. Table 1 gives the definitions of the terms used in the paper. We begin with the source.

1) Source: The source divides data into *batches* of K packets and transmits batch by batch. Each data packet is divided into L *data blocks* of equal size, padded if necessary. pMORE's network code works at the granularity of blocks. The i^{th} *coded block* in an outgoing packet is denoted as B'_i and is generated by a linear combination of the i^{th} data blocks in the K data packets in the same batch:

$$B'_i = \sum_{j=1}^{K} c_j B_{ji},$$

where B_{ji} is the i^{th} data block in the j^{th} data packet and c_j is a random coefficient chosen for B_{ji}. We call $(c_1, ..., c_K)$ the *code vector* of the block. A checksum is added for each coded block and then attached to the pMORE header. All coded blocks sharing the same code vector belong to a *segment*. If all L coded blocks in a packet belong to one segment, this segment is called a *full segment*. The packets transmitted by source always contain one full segment. The source calculates a list of forwarders for the destination, and adds the list to the pMORE header.

2) Forwarders: A forwarder listens to the channel. Whenever it overhears a packet, including a partial packet, it checks the pMORE header to determine whether it is in the forwarder list. If true, it checks the integrity of each coded block by comparing the checksum in the pMORE header with the calculated checksum based on the received data. A coded block that fails the checksum test is removed. Thus, a full segment could become a *partial segment*. The forwarder then performs *innovativeness check*, i.e., checks the code vector of each segment to determine whether it contains new information according to the policies described in Section II-D. The forwarder stores the *innovative segments* which passed the innovativeness check, and discards others. The forwarder also uses a heuristic algorithm described in Section II-E to assemble an outgoing coded packet by creating a combination of the buffered segments, and broadcasts it, when the MAC layer permits. The packets sent by the forwarder will contain multiple segments if it used partial segments when assembling the packet.

3) Destination: For each packet it receives, the destination filters out the faulty coded blocks and does the innovativeness check for every correctly received coded block. For the i^{th} data blocks in the batch, if it has received K linearly independent coded blocks, it can recover the data blocks by solving K linear equations. Once the destination decodes all data blocks, it finishes decoding the whole batch. As long as it received enough information to decode a data block, it will start to decode. After the destination decodes the batch, it immediately sends an ACK to the source. When the source receives the ACK, it will clean its buffer and move to the next batch.

D. Innovativeness Check

In the following, for simplicity, we sometimes refer to coded blocks simply as blocks; data blocks are still always referred to as data blocks. When a forwarder receives a new packet, it performs innovativeness check. Since each segment in a packet has its own code vectors, innovativeness check is done per segment. In pMORE, for simplicity, we consider a segment innovative if its code vector is linearly independent of the code vectors of the previously received full segments. We apply Gaussian elimination [13] to check the linear independence. Only the full segments are checked against, because the number of full segments is bounded by the batch size, while the number of partial segments can be potentially very large and may lead to excessive CPU usage.

All the innovative segments, full or partial, will be stored in the buffer. As full segments are always preferable over partial segments as they lead to less overhead, we also implement a replacement policy. Basically, when the forwarder receives a new innovative full segment, it will remove all the partial segments from its buffer whose code vectors are no longer linearly independent of the code vectors of the full segments. Thus, when a forwarder received K innovative full segments, the forwarder will no longer need to store the partial segments.

E. Packet Assembling

One of the major challenges of pMORE is how to assemble the packets. On one hand, we want to send as much new information as possible, which means that we want to use as many innovative segments as possible when assembling the packet. On the other hand, the segments are often not on the same block locations and mixing the segments almost always leads to a packet with many segments and an increased amount of overhead as each segment has a unique code vector of K bytes that must be transmitted in the pMORE header. The algorithm, therefore, must achieve a tradeoff between innovativeness and overhead. This is further complicated by the fact that a forwarder does not know which blocks have been received at the downstream, and the fact that the algorithm must be light weight.

We therefore adopt a heuristic algorithm. Basically, we first use *only* the full segments when assembling the packet. All such packets will still be a full-segment packet. This guarantees that we never introduce more overhead than the original MORE when we do not have to, and avoids increasing the number of segments in the outgoing packets. If the number of full segment is F, a total of F full-segment packets can be transmitted. After F full-segment packets has been transmitted, when a new packet needs to be assembled, we look into the partial segments and attempt to find a set of partial segments that covers all blocks yet leads to a minimum number of segments in the transmitted packet. The packet generated by the partial segments is still combined with a random combination of the full segments, as this does not incur additional overhead and yet can carry the information in the full segments.

To be more specific, we maintain a counter W, the number of full-segment packets that can be transmitted. Whenever received a full segment, W is incremented by 1; whenever a full-segment packet is transmitted, W is decremented by 1. We use R to denote the number of partial segments and use PS_i to denote partial segment i. All partial segments in the buffer is associated with a bitmap indicating the blocks in this segment. Whenever the node needs to assemble a packet, the algorithm described in Algorithm 1 is called which returns P_{out}. In the algorithm,

- $\mid x \mid$ denotes the number of '1's in binary vector x,
- $bitmask$ denotes the current bitmask,
- π denotes the set of partial segments found in the current iteration of the *while* loop,
- T denotes a subset of bits in $bitmask$ which is obtained by a "bitwise and" of $bitmask$ with the bitmap of the partial segment which has the most number of common bits with $bitmask$.

A partial segment is *flagged* if it has been added to π in the first iteration of the *while* loop when calling Algorithm 1, and is maintained in the subsequent calls. It can be seen that in each iteration of the *while* loop, we basically attempt to find a set of partial segments that share the same set of blocks indicated by the bit pattern of T while attempting to find a T with maximum number of '1's. This is inspired by the greedy algorithm for Set Cover, because we attempt to use a minimum number of partial segments to cover all blocks; also, multiple partial segments can all be used without increasing the number of segments in the packet if they cover the same set of blocks. A flagged partial segment is not evaluated in the first iteration of the *while* loop because such segments may have been used in earlier calls to Algorithm 1 and without this heuristic check the algorithm will produce the same packet over and over again.

Algorithm 1 Assembling an outgoing packet at a forwarder.

1: $P_{out} \leftarrow$ a random combination of the full segments
2: **if** $W \leq 0$ **then**
3: $bitmask \leftarrow 2^L - 1$
4: **while** $bitmask \neq 0$ **do**
5: $\pi \leftarrow \emptyset$
6: $T \leftarrow 0$
7: **for** $i = 1$ to R **do**
8: **if** (PS_i is flagged) and ($bitmask == 2^L - 1$) **then**
9: **continue**
10: **end if**
11: **if** $\mid bitmap(PS_i) \& bitmask \mid > \mid T \mid$ **then**
12: $\pi \leftarrow \emptyset$.
13: $T \leftarrow bitmap(PS_i) \& bitmask$.
14: **end if**
15: **if** $bitmap(PS_i) \& bitmask == T$ **then**
16: $\pi \leftarrow \pi \cup PS_i$.
17: **end if**
18: **end for**
19: **if** $T == 0$ **then**
20: $bitmask \leftarrow 0$
21: **else**
22: $P_{out} \leftarrow P_{out} \oplus$ a random combination of the segments in π for blocks within bitmap T
23: **if** $bitmask == 2^L - 1$ **then**
24: set all segments in π flagged
25: **end if**
26: $bitmask \leftarrow bitmask \& \neg T$
27: **end if**
28: **end while**
29: **end if**
30: **return** P_{out}

The outer loop is executed at most L times and the inner

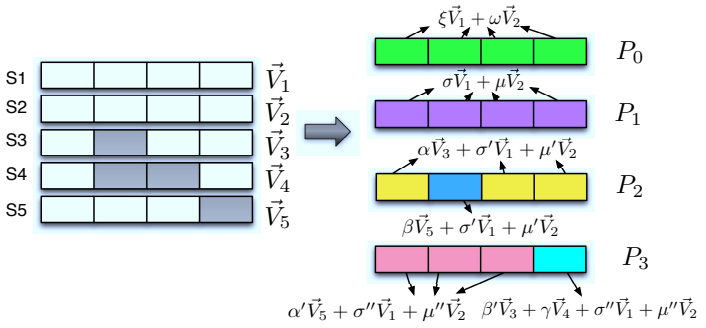

Fig. 1. Assembling coded packets with partial segments: the forwarder received 5 segments with code vector $\vec{V}_1$ to $\vec{V}_5$ respectively. Last 3 segments contain corrupted blocks shown as shaded areas.

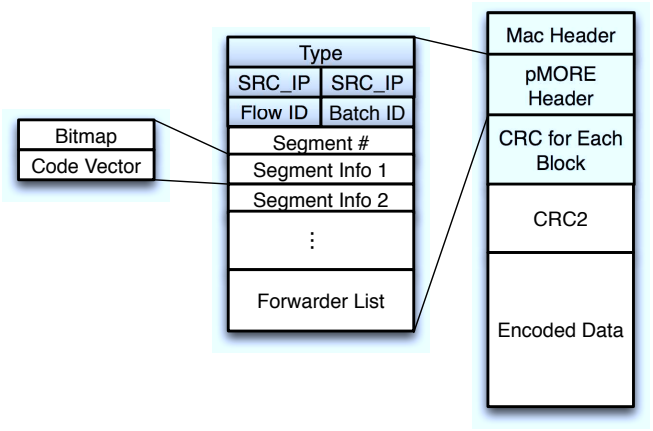

Fig. 2. Packet format. The shaded fields are mandatory, others are optional.

loop is executed R times. Therefore, Algorithm 1 requires $\mathcal{O}(LR)$ operations.

1) An Example of Packet Assembly: Consider a simple example in Figure 1 where $L = 4$. A node has received 5 segments among which 2 are full segments. At first, the node will assemble two packets P_0 and P_1 with the full segments. After that, it has to create a packet P_2 with partial segments. A naive solution which combines all partial segments will result in a packet with 4 segments. The heuristic algorithm, however, will select S_3 in the first iteration and generate the first segment with a code vector of $\alpha\vec{V}_3+\sigma'\vec{V}_1+\mu'\vec{V}_2$, where α, σ' and μ' are random coefficients. This segment covers blocks B_1, B_3, and B_4. Meanwhile, S_3 will be flagged and S_5 will be selected in the second iteration. The algorithm ignores blocks B_1 and B_3 in S_5; the second segment has one block, B_2, with $\beta\vec{V}_5+\sigma'\vec{V}_1+\mu'\vec{V}_2$ as the code vector. As a consequence, the assembled packet will have 2 segments. If the node wants to create another coded packet P_3, in the first iteration of the *while* loop, the algorithm does not evaluate S_3 because it has been flagged, and chooses S_5 instead. S_5 covers B_1, B_2 and B_3, and in the next iteration, B_4 in *both* S_3 and S_4 will be used by the algorithm. The assembled packet still has only 2 segments and their code vectors are $\alpha'\vec{V}_5+\sigma''\vec{V}_1+\mu''\vec{V}_2$ and $\beta'\vec{V}_3+\gamma\vec{V}_4+\sigma''\vec{V}_1+\mu''\vec{V}_2$, where α', β', γ, σ'', and μ'' are random coefficients.

F. Interleaving

pMORE also adopts *interleaving*. Basically, every block in the outgoing packet is relocated to a random block location according to a random permutation of blocks, and every received packet will undergo *deinterleaving* in which every block is mapped to its original location according to the reverse of the random permutation. This is because bits in a data packet do not share the same fate [11]; blocks near the header of the packet are less likely to be corrupted than blocks towards the end. Without interleaving, it could happen that all blocks are decoded except the last block. The interleaving procedure spreads the corrupted blocks evenly in each packet and significantly reduces the probability of such events.

G. Packet Format

pMORE inserts a variable length header in each packet, as shown in Figure 2. After the MAC header are the pMORE header, CRC for each block, CRC2 and the Encoded Data. In the pMORE header, the type field indicates whether the packet contains encoded data. It is followed by the source and destination IP addresses, the flow ID, and the batch ID. Each segment information field contains the code vector of the segment and a bitmap indicating the block locations of the segment. CRC2 is the checksum of the pMORE header. A packet failed CRC2 will be discarded.

III. Implementation and Evaluation

We implemented pMORE with the code base at [10] and ran experiments to evaluate its performance.

A. Testbed

We employ a 12-node indoor testbed. The nodes are distributed in several locations in an indoor environment. Each node uses the Cisco Aironet 802.11a/b/g wireless cardbus adapter [9] and runs the open source Madwifi driver [8] at 1 Mbps[1]. We set the Madwifi driver in the monitor mode to allow the raw data frames to be delivered to pMORE. We modified the link measurement code of MORE to measure the *block loss ratio*, and use it as the input to the path calculation algorithm of MORE. The forwarder list contains 1 to 6 nodes and the average loss ratio is 48%. In all the experiments, the batch size is 32 for both MORE and pMORE and the packet size is 1500 bytes.

B. Throughput Comparison and Analysis

We wish to verify whether pMORE improves throughput by exploiting partial packets. We first run the link measurement program and calculate the routes for both MORE and pMORE. In each experiment, we randomly choose source and

[1]The reason why we choose 1 Mbps is because we have several slow machines in our testbed and their CPU speeds cannot catch up with the higher data rates in the wireless link when running random network coding.

destination pairs in our testbed and run each protocol for 60 seconds. We repeat the experiment three times and obtain the average throughput.

Figure 3 shows the scattered plot of the throughputs achieved by pMORE and MORE for the same source and destination pair. Figure 4 shows the CDF of the throughput taken over 124 source and destination pairs. The results reveal that pMORE does outperform MORE: pMORE's median throughput is 34% higher than MORE.

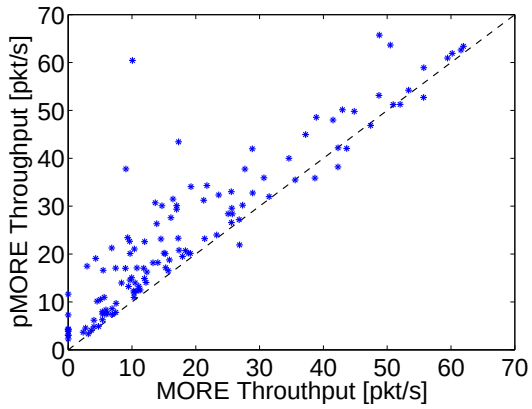

Fig. 3. Scatter plot of overall throughput. Each point represents the throughput of a particular source destination pair for MORE and pMORE. A point in the 45 degree line represents a source destination pair for which the two protocols have the same throughput.

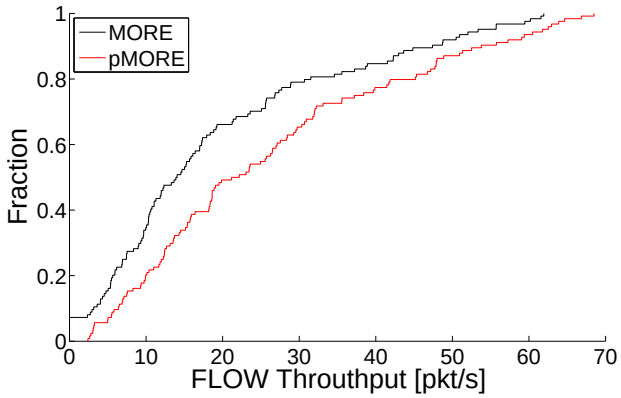

Fig. 4. CDF of the throughput of MORE and pMORE.

To verify the source of the gain, we show in Figure 5 the relation of the gain over MORE and the partial packet ratio, where the partial packet ratio is the average partial packet ratio of all nodes participate in packet forwarding. We can see that a larger partial packet ratio leads to a larger gain over MORE, which verifies that pMORE achieves a better performance by exploiting partial packets. The gain flattens as the partial packet ratio further increases, which is likely because when the partial packet ratio is too high, many packets are erased and cannot be exploited.

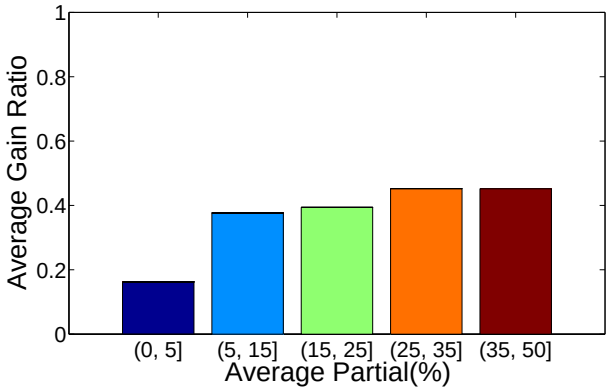

Fig. 5. The percentage of gain v.s. the average partial packet ratio.

C. Overhead

pMORE's overhead is largely determined by the number of segments in a packet. As packets travel further down the path, more packets will be corrupted and more segments may be introduced. pMORE relies on Algorithm 1 to suppress the number of segments. We collect data to verify how the number of segments increases as packets get further away from the source. Figure 6 shows the average number of segments in a transmitted packet at each hop. The result suggests that the average number of segments increases mildly.

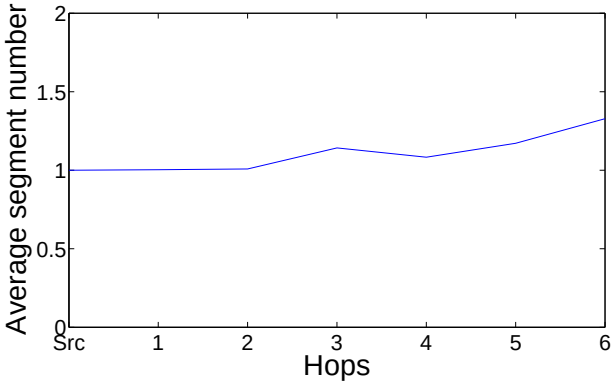

Fig. 6. The average number of segments in a transmitted packet at each hop along the path.

IV. Related Works

ExOR [1] first introduced the opportunistic multi-hop routing for wireless networks. Since several nodes may receive the same packet, ExOR attempts to arrive at an agreement on choosing the "best" node to forward the packet. ExOR sends packets based on batches; for each batch, the source adds a candidate forwarder list in front of each packet. After the nodes receive the batch of packets, they will wait for their turns to send packets which have not been acknowledged by the higher priority nodes. Although this strict scheduling can reduce the duplicate transmission, it discourages spatial reuse. MORE [2]

is a MAC-independent opportunistic routing protocol. It improves ExOR [1] by utilizing random network coding to reduce packet duplication and to enhance spatial reuse. MORE also sends packets in batches, for each batch, it mixes the packets randomly before transmission. The randomness removes the need of the scheduling because every packet contains useful information with high probability. pMORE builds on MORE's foundation but introduces methods to exploit partial packets.

MIXIT [4] realizes that opportunistic routing cannot exploit long, lossy links without being able to exploit partial packets. It proposes a new opportunistic routing architecture which is based on the correctly received symbols. MIXIT uses the physical layer information to distinguish the correct symbols to broadcast clean symbols. For each decoded symbol, the physical layer computes a confidence value based on which a symbol is marked clean or faulty. However, it could still happen that a faulty symbol is marked as clean and which leads to error propagation. Using the physical layer information also needs a collaboration of physical layer and upper layer, which limits its application to other platforms. On the other hand, instead of using symbol level information, pMORE divides a packet into blocks, and sends packet based on correctly received blocks. pMORE does not need physical layer information and is a software-only approach, therefore it can be applied directly on top of any platform.

Exploiting partial packets has been studied extensively in recent years. For example, ZipTx [5], Maranello [15] and FRJ [16] have been proposed for single-hop wireless LANs. We note that pMORE is different because it is a multi-hop network protocol. Seda [14] has been implemented for wireless sensor networks which recovers errors by retransmitting only the corrupted blocks instead of the entire packet. We note that although also identifies correct information with block checksums, pMORE is fundamentally different from Seda because pMORE forwards packet according to random network coding while Seda forwards packets according to the traditional protocol without exploiting overhearing in wireless networks.

V. Conclusions

In this paper, we propose pMORE, a software-only opportunistic routing protocol capable of exploiting partial packets. Our main contributions include the following. First, we propose to use block checksum test to efficiently locate correct information in a partial packet and use them free of the risk of propagating errors. Second, we design protocol and algorithm to efficiently assemble packets from full or partial segments of received data, maximizing the fresh information sent to the downstream while maintaining low overhead. We implement pMORE and test it on a 12-node indoor wireless testbed. Our experiments show that pMORE achieves a significant better performance than the original MORE.

References

[1] S. Biswas and R. Morris. Opportunistic routing in multi-hop wireless networks. In *SIGCOMM*, 2005.

[2] S. Chachulski, M. Jennings, S. Katti, D. Katabi. Trading structure for randomness in wireless opportunistic routing. In *SIGCOMM*, 2007.

[3] S. Katti and D. Katabi. MIXIT: The network meets the wireless channel. In *HotNets*, 2007.

[4] S. Katti, D. Katabi, H. Balakrishnan and M. Medard. Symbol-level network coding for wireless mesh networks. In *SIGCOMM, 2008*.

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

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

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

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

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

[10] *http://people.csail.mit.edu/szym/more/README.html*.

[11] B. Han, L. Ji, S. Lee, B. Bhattacharjee, and R. Miller. All bits are not equal A study of IEEE 802.11 communication bit errors. In *INFOCOM, 2008*.

[12] D. S. J. De Couto, D. Aguayo, J. Bicket, and R. Morris. A high-throughput path metric for multi-hop wireless routing. In *MOBICOM, 2003*.

[13] R. Koetter and M. Medard. An algebraic approach to network coding. In *IEEE/ACM Trans. on Networking, 2003*.

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

[15] 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 *NSDI, 2010*.

[16] 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.