Gr-WiFi Time Complexity

This report provides an extended analysis of the computational complexity and CPU load of GR-WiFi's receiver implementation.

1: Measurement via Performance Counter in GNU.

GNU Radio provides a built-in tool called Performance Counter [1] to provide low-level profiling of GNU Radio blocks. We now use Performance Counter to assess the computational cost of each processing block in GR-WiFi. For each block, we measure the average number of output samples produced per functional call and the average number of CPU clock ticks consumed per call. We then compute the average number of CPU clock ticks per output sample, which gives a direct measure of computational load.

Table 1: Clock ticks needed for each output item per block for SISO VHT and SU-MIMO VHT. Unit:clock cycles per output item per block

SISO VHT

MCS	0	1	2	3	4	5	6	7	8
Trigger	2.09	2.22	1.77	2.57	2.35	2.65	2.06	2.75	2.22
Sync	3.51	6.03	7.47	7.67	10.19	11.64	9.70	11.28	10.66
Signal	13.18	14.33	16.62	51.24	21.19	23.33	21.41	51.04	29.81
Demod	20.26	13.49	16.51	12.73	18.09	14.97	12.28	14.94	11.62
Decode	93.49	93.98	178.80	166.70	238.13	209.28	215.95	249.58	206.82

SU-MIMO VHT

MCS	0	1	2	3	4	5	6	7	8
Trigger	2.42	2.18	3.32	3.46	3.88	2.88	7.27	3.70	3.32
Sync	7.26	8.17	8.87	10.25	12.09	25.25	50.30	48.33	17.00
Signal2	22.19	25.19	26.77	28.92	32.25	38.66	35.10	35.73	41.37
Demod2	21.17	15.92	20.15	11.99	16.67	13.66	14.94	15.59	13.95
Decode	94.42	83.96	165.12	151.27	227.61	225.25	226.31	237.20	230.08

The measurement was done on a DELL OptiPlex-7090 computer running Ubuntu 22.04.05 LTS, with an Intel Core i7-11700 CPU and 16 GB RAM. A data burst of 100 packets, each packet having 500 Bytes of payload, is processed by the receiver. Table 1 presents the average number of CPU clock ticks per output item for each block in the receiver chain across various MCS levels. We observe that the decode block is the most computationally intensive block, suggesting that further optimization is demanded. We observe an increasing trend on the processing cost with higher MCS values.

2: Analysis on Signal Processing Operations

Since the receiver contributes to the primary computational bottleneck in the system, we focus complexity analysis on the receiver processing chain. A key feature of our implementation is that each packet, which may contain a varying number of symbols, is divided into multiple batches of input data for processing by each block. Each block invokes its work function when ready to process a new input batch. The time complexity analysis presented in Table 2 is for the computational cost per work function call. Each block operates a state machine, where different stages exhibit varying computational complexity depending on the current processing state.

We use the following definitions of math symbols:

N_{in} – the number of input items in work function calls.
N_{out} – the number of output items in work function calls.
L - correlation window length of LTF.
F - input size of FFT ($F = 64$ in the implementation).
V – the length of the Viterbi trellis of the signaling symbols.
V_s – the length of the Viterbi trellis of the data packet.
K - number of states in the Viterbi decoder (K = 64).
M - constellation size.
d_{SD} - data subcarriers per OFDM symbol.
d_{CBPS} - coded bit per OFDM symbol.

Table 2 highlights the following computational aspects:

- Synchronization operations (Trigger and Sync blocks) are relatively lightweight, with complexities such as $O(N_{\text{in}})$ for most operations, scaling linearly with input size.
- Signal detection (Signal and Signal2 blocks) involves more intensive computations, e.g., $O(3F\log(F)) + O(K \times V) + O(V)$ for demodulation, where $F = 64$ represents the FFT input size, highlighting logarithmic scaling due to FFT operations. Signal2 has two stream output only affects the S_COPY state where it copies two streams instead of one, but this doesn't change the core signal detection complexity.
- Demodulation (Demod and Demod2 blocks) computation complexity varies depending on the packet format (e.g., VHT, HT, Legacy). During the DEMOD_S_FORMAT stage, HT and Legacy packets require additional FFT operations for packet format classification. For multi-user packets (e.g., 2×2 MU-MIMO), the cost is higher because an extra symbol (VHT-SIG-B) must be processed. In the DEMOD_S_DEMOD stage, the computation cost depends on the selected MCS. Higher MCS modes have higher complexity since more bits are mapped to each OFDM symbol. Demod2 block costs more as two streams are processed in parallel, approximately doubling the processing time compared to single-stream demodulation.
- Decoding operations (Decode block) scale with Viterbi trellis V_s, which varies with the packet size. Larger payloads significantly increase the decoding cost, particularly at high MCS levels due to more coded bits per packet.

Table 2: Time Complexity of each work function call. The complexity depends on the current state of of the internal state machines when the module is called. The items in bold fonts are the name of the states in the source codes.

	Detail on each stage per block
Trigger	**Trigger:** $O(N_{\text{in}})$
Sync	**SYNC_S_IDLE:** $O(N_{\text{in}})$ **SYNC_S_SYNC:** $O(L)$
Signal/Signal2	**S_TRIGGER:** $O(N_{\text{in}})$ **S_DEMOD:** $O(3F\log(F)) + O(K \times V) + O(V)$ **S_COPY:** $O(N_{\text{out}})$ **S_PAD:** $O(1)$
Demod/Demod2	**DEMOD_S_RDTAG:** $O(1)$ **DEMOD_S_FORMAT:** Case VHT: $O(2F\log(F)) + O(K \times V) + O(V)$ Case HT: $O(2F\log(F)) + 2(O(K \times V) + O(V))$ Case LEGACY: $O(2F\log(F)) + 2(O(K \times V) + O(V))$ **DEMOD_S_VHT:** Demod: Case 2×2 MU-MIMO: $O(3F\log(F)) + O(K \times V) + O(V)$ Case SISO: $O(2F\log(F)) + O(K \times V) + O(V)$ Case NDP: $O(2F\log(F)) + O(K \times V) + O(V)$ Demod2: Case 2×2 SU-MIMO: $O(6F\log(F)) + O(K \times V) + O(V)$ Case SISO: $O(2F\log(F)) + O(K \times V) + O(V)$ **DEMOD_S_HT:** Demod: Case 2×2 MU-MIMO: $O(2F\log(F))$ Case SISO: $O(F\log(F))$ Case NDP: $O(F\log(F))$ Demod2: Case 2×2 SU-MIMO: $O(4F\log(F))$ Case SISO: $O(F\log(F))$ **DEMOD_S_LEGACY:** $O(1)$ **DEMOD_S_WRTAG:** $O(1)$ **DEMOD_S_DEMOD:** Demod: $O(F\log(F)) + O(d_{\text{SD}} \times \log_2(M)) + O(d_{\text{CBPS}})$ Demod2: Case 2×2 SU-MIMO: $O(2F\log(F)) + O(d_{\text{SD}} \times \log_2(C)) + O(d_{\text{CBPS}})$ **DEMOD_S_CLEAN:** $O(1)$
Decode	**DECODE_S_IDLE:** $O(K \times V_s)$ **DECODE_S_DECODE:**$O(K \times N_{\text{in}}) + O(V_s)$ **DECODE_S_CLEAN:**$O(1)$

3: Analysis of cross- and auto-correlation

Each correlation call operates on two vectors of 16 complex samples. Specifically, for complex vectors $a, b \in \mathbb{C}^{16}$, the normalized correlation is computed as:

$$\mathrm{Corr}(a, b) = \frac{\left| \sum_{i=0}^{15} a_i \cdot b_i^* \right|}{\sqrt{\sum_{i=0}^{15} |a_i|^2} \cdot \sqrt{\sum_{i=0}^{15} |b_i|^2}}.$$

We detail in Table 3 the FLOP count required for each call to the correlation function. Since both cross-correlation and auto-correlation operate on two 16-sample complex vectors and invoke the same subroutine, each call incurs the same computational cost, which is approximately 292 FLOPs.

Table 3: Detail FLOPs needed for the correlation function.

Operation	Count	FLOPs per op	Total FLOPs
Complex multiplications	16	6	96
Complex additions	15	2	30
Magnitude of inner product (abs)	1	4	4
Norm of first vector	16	4	64
Norm of second vector	16	4	64
Final sum of mags	15x2	1	30
Final sqrt on each norm	2	1	2
Final normalization (division)	1	2	2
Total count per call			292

Hence, the choice of auto- versus cross-correlation is not decided by the computational complexity, *rather by the ability to process the cyclic shift structure.*

Unfortunately, the Performance Counter cannot measure the clock ticks of the preprocessing block as it is an hierachical GUI module. This prepropssing block is the same as that in WIME package, and it does not appear to be the performance limiting factor for the WiFi receiver. We hence did not purse further optimization (e.g., FFT based cross correlation or sliding window based auto correlation methods) beyond the straightforward implementation.

Changes in the revised paper:

- We choose to include the measurement via Performance Counter in the revised paper, as the use of Performance Counter is popular in the GNU radio community.
- The analysis on the signal processing blocks are too detailed for a reader who might not want to check out the source codes. We choose not to add Table 2 in the revised paper, instead, we include it in the documentation directory of the software package in gitHub.
- Pre-Processing (correlation) is not a center part of the receiver, we choose not to include Table 3 in the main paper. We add a comment that the auto-correlation and cross-corelation in our implementation have the same computational complexity.

References

[1] T. W. Rondeau, T. O'Shea, N. Goergen, Inspecting GNU radio applications with controlport and performance counters, in: Proceedings of the Second Workshop on Software Radio Implementation Forum, SRIF '13, Association for Computing Machinery, New York, NY, USA, 2013, p. 65–70. doi:10.1145/2491246.2491259.
URL `https://doi.org/10.1145/2491246.2491259`