POCKET+ General Presentation

David Evans, Ben Fischer

19/05/2017

POCKET+ AGENDA

Background:

Technical Explanation and Results:

Exploiting the design further

Conclusion

Background

Spacecraft communication links are a limited and shared resource.

So we allocate bandwidth carefully e.g. when choosing parameters we send (and their sample frequency) in housekeeping data. We design asynchronous events for parameters that we need to monitor but cannot fit them into the regular list.

However our work at ESOC has shown that spacecraft telemetry is 50% -90% full of bits that carry no information i.e. they could be removed by lossless compression.

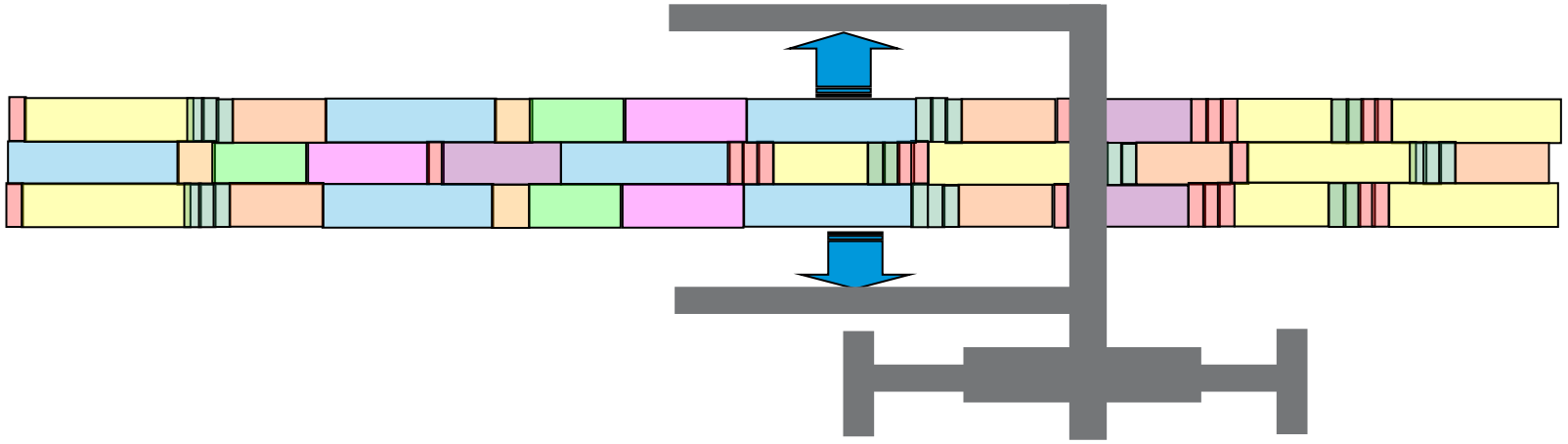
Every useless bit we send at the data level explodes to 2.5 useless bits when we add overheads and the channel coding required to close the microwave link.

Each useless bit we send means another information bit cannot be sent and/or we have less power to recover each information bit and/or we are receiving the real information bits later than necessary.

Can we make better use of this precious resource?

The problem with Housekeeping Data

Spacecraft housekeeping data is made up of sequential fixed length structures that contain values of parameters with different lengths, different encoding (e.g. an integer, followed by a Boolean, the a real etc.) All are sampled and grouped as they have something in common e.g. they belong to the same subsystem.



This mixed up data is hard to compress using the simple, efficient algorithms we use for spacecraft payload data compression, but it does compress well with complicated algorithms such as 7zip.

Project History

2009: ESA/ESOC US patent raised for a method that can compress stored HKTM telemetry

2011: ESA/ESOC international patent raised for a method to compress real-time and stored HKTM with a single process

- POCKET compression

2013: ARTES 5.1 study awarded to Spacebel SA, Belgium to investigate usefulness of POCKET for telecom satellites and produce an end to end demonstration

2017: ESA/ESOC international patent raised for a method to autonomously compress real-time and stored HKTM with a single process

- POCKET+ compression

2017: POCKET will fly on PROBA-2 - first time in orbit

What's special about real-time compression on spacecraft?

Spacecraft have very low power, low CPU processors with little memory. The communication is over microwave links that are not 100% reliable. Therefore in addition to lossless the compression process

- Cannot use much memory
- Cannot consume much CPU
- Be robust to occasional data loss

And still be efficient enough to compress data as it is generated..

Advantages if we are successful:

- Stored data compression is for free
- Can design the data structures for a single scenario
- We can easily control/regulate it per data structure type and instance
- There is a multiplication factor if we share the same link to send real-time and playback data (less data to send and more bandwidth to send it with)

POCKET+ AGENDA

Background:

Technical Explanation and Results:

Exploiting the design further

Conclusion:

POCKET+ WHY DOES IT WORK?

POCKET+ works because of the way we encode the parameter values:

1. For every parameter an extreme dynamic range for its value is established i.e. the extreme values it could possibly reach anytime
2. Then the "next largest standard type" in which this extreme range would fit is allocated, e.g. 8 bit integer or double real

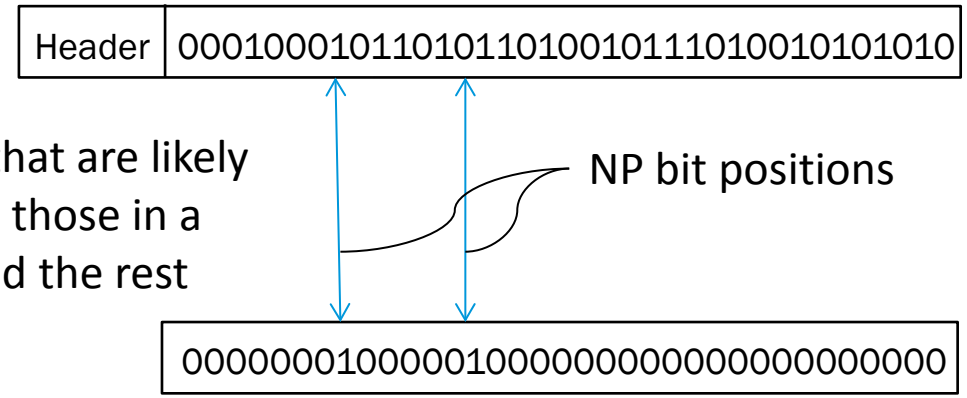
These elements each introduce information redundancy…

As regards the first point, POCKET+ tracks over short time periods, typically 20 iterations. Hence it effectively reduces the allocated dynamic range to that expected in the next tracking cycle rather than the entire mission. This is where the vast majority of the compression comes.

The same mechanism that predicts the next dynamic bit range also naturally removes any "extra bits" allocated by the standard type selection. It also removes any redundancy added by standard data type selection.

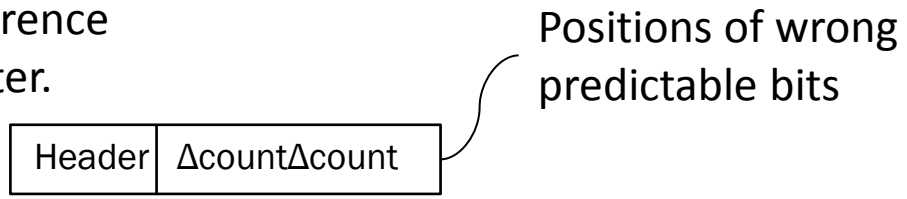
POCKET Receipt

Take a fixed length data structure

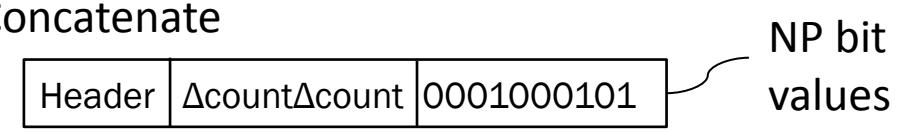


Determine all bit positions in that structure that are likely to contain bits that have an identical state to those in a reference data. Call these predictable bits and the rest non predictable (NP) bits.

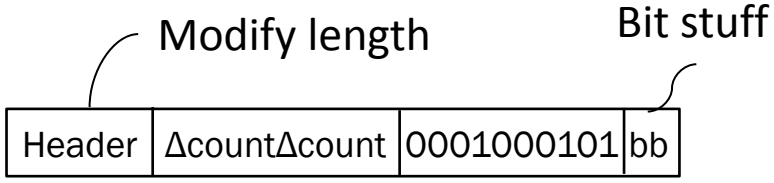
When a new instant of that data structure arrives, check that each predictable bit is the same as the reference data, if it is not then mark its position in a counter.



Read all the non predictable bits in the clear. Concatenate into a smaller but variable length structure.



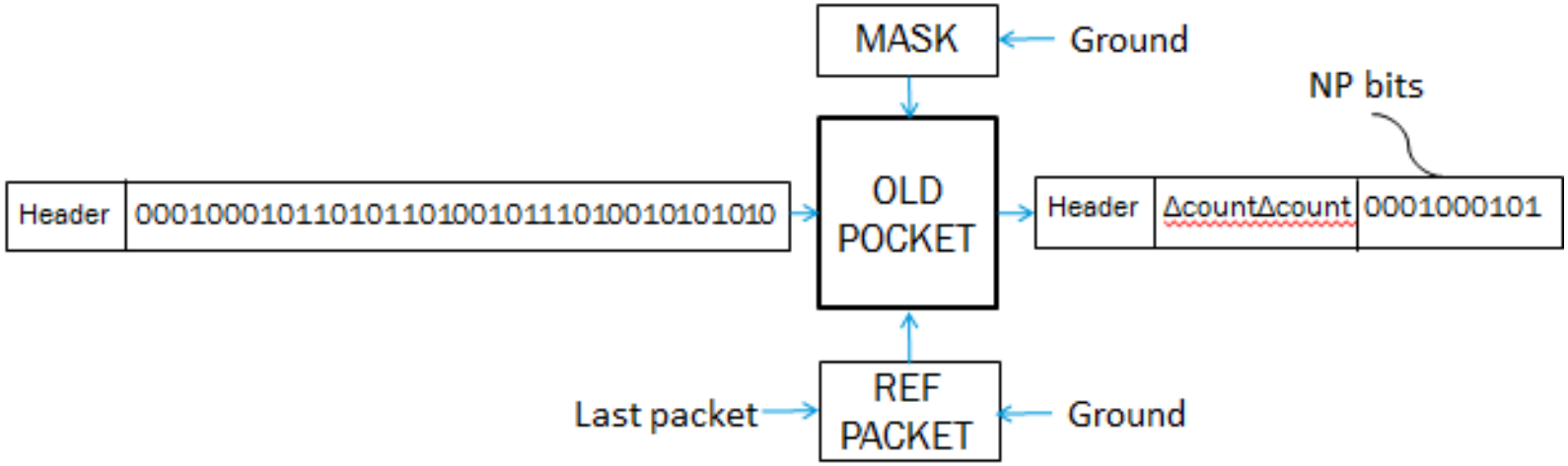
Modify the original header (or add one) indicating the original data structure ID and the length of the new variable length structure.



Old POCKET recap

The MASK is calculated on ground using data analytics techniques. It is loaded to the spacecraft and updated when necessary

The REFERENCE is calculated on ground using data analytics techniques.
It is loaded to the spacecraft and updated when necessary (Reference Mode).
Or the REFERENCE is the last data structure generated (Chain mode)



Results: Chain mode is best for performance and stable when the data behaviour changes but it is not robust to data loss. Reference mode is robust but is less stable and gives lower compression.

POCKET+ Design

GOALS

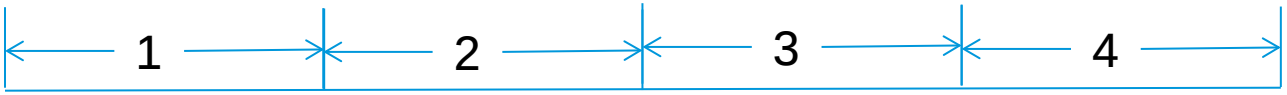
Remove the need for the ground to create/update the mask & reference data

Combine performance/stability of chain with the robustness of reference mode

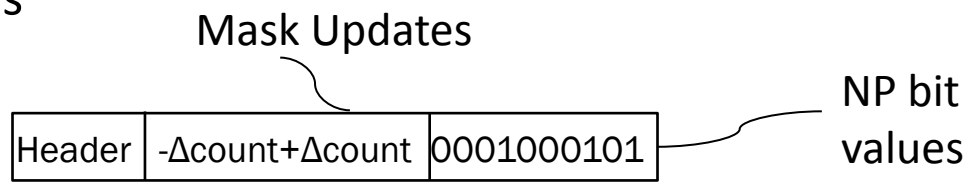
SOLUTION

Create/Update a mask automatically on-board following these rules

- Any predictable bit value that is not the same as the reference data value is immediately classed as non predictable. (Negative Mask Update)
- Any non predictable that has the same value as the reference data value over a tracking period will be classed as predictable in the next tracking period (Positive Mask Update)



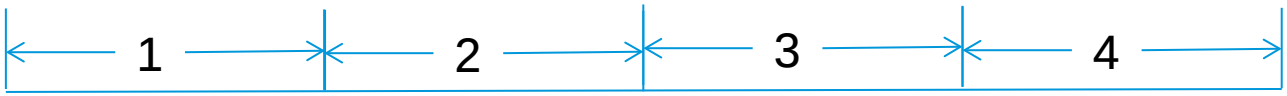
- Only send +/- Mask Updates to the ground in the form of counters, plus the corresponding NP bit values



Implications of MASK update rules

The mask is “perfect” i.e. all predictable bits are identical to the reference data.

The reference data can be any data structure instance of that type successfully received in the present or last tracking period e.g. in tracking period 4 any previous from period 4 or period 3 can be used

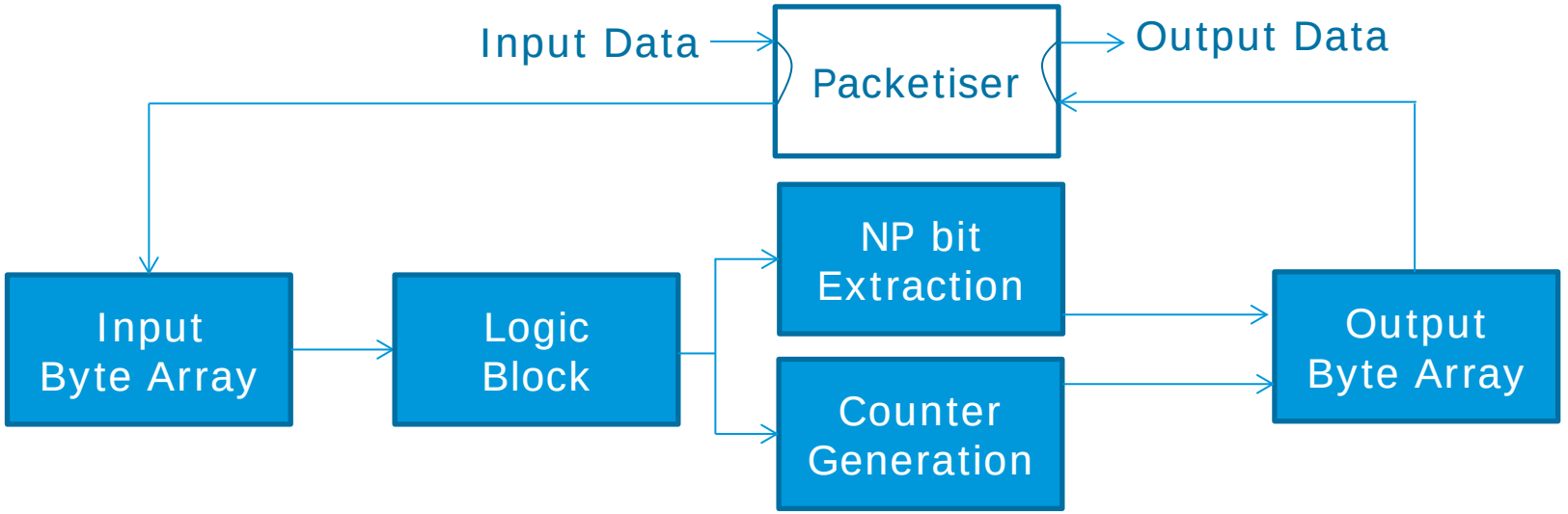


The mask is updated every iteration. Therefore it constantly tracks the data redundancy in a “conservative” manner described below

- Unexpected changes are dealt with immediately but it takes at least one tracking period of consistent positive results to declare a bit predictable again.

This conservative approach means that there are far less mask update counters than there were wrong counters. Therefore it is possible to send mask updates covering multiple iterations with very little overhead. This is used to increase the robustness of the system.

POCKET+ Processes



Packetiser: Handles the Data Level Specifics e.g. identifying data structure ID and length, setting a flag to indicate the data is compressed etc.

Logic Block: Updates the mask and the reference data, builds the mask for the next tracking period, creates inputs for Counter Generation and NP Bit Extraction

Counter Generation: Create counters to identify +/- mask updates

NP bit extraction: Routine to read non predictable bit values in the clear

Logic Block Memory

POCKET+ does all its processing on word level to speed up the data processing and limits memory usage.

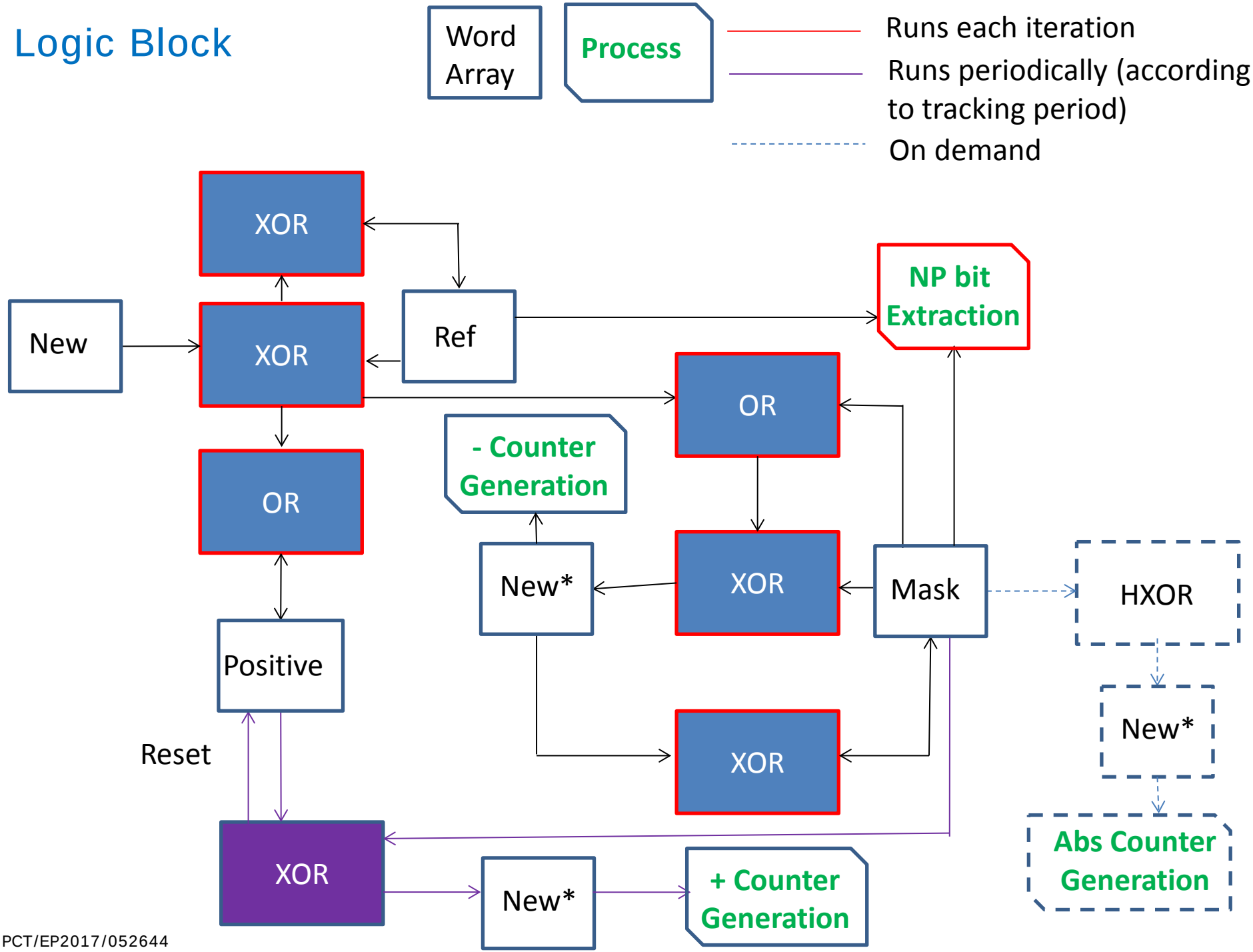
The algorithm requires the following four word arrays: New, Reference, Mask, Positive

1. New array is the input byte array is converted to a word array
2. Reference array is a copy of the New Array imported in the last iteration
3. Mask array is the Bit Mask
4. Positive array is the Mask to use in the next tracking cycle

The New* array is where the input for the NP Counter Generation and NP Bit Extraction processes is written. To save memory the New array is reused.

A reference, mask and positive array must be stored for each type of data structure one wants to compress.

Logic Block


Word Array
Process
Runs each iteration
Runs periodically (according to tracking period)
On demand
XOR
New
XOR
Ref
NP bit Extraction
OR
OR
- Counter Generation
New*
XOR
Mask
HXOR
Positive
New*
Reset
XOR
XOR
New*
+ Counter Generation
Abs Counter Generation

Logic Block Code

Runs every iteration for each word:

```
new[word] ^= ref[word];
ref[word] ^= new[word];
positive[word] |= new[word];
new[word] |= mask[word];
new[word] ^= mask[word];
mask[word] ^= new[word];
```

Runs once every tracking period (e.g. typically once every 20 iterations) for each word:

```
new[word] = positive[word] ^ mask[word];
mask[word] = new[word] ^ mask[word];
positive[word] ^= positive[word];
```

This is extremely efficient code!

Counter Generation Code

Reads input array New* backwards and processes each word as shown below

```
copy = New* [word]
While copy != 0 {
        x = copy & –copy; // produces a word with only the LSB set bit in copy set in x

        Bit position = Lookup[((uint32_t)(x* 0x077CB531U)) >> 27]
        Produce delta count (compared to last bit position)
         OPTIONAL: Exit if too many counters and send uncompressed data

        copy = copy – x; // removes the LSB set bit from copy
 }
```

Optimized for quickly finding a few set bits in long buffer of unset bits

Only native processor operations plus a de Bruijn lookup table and a multiplication are used

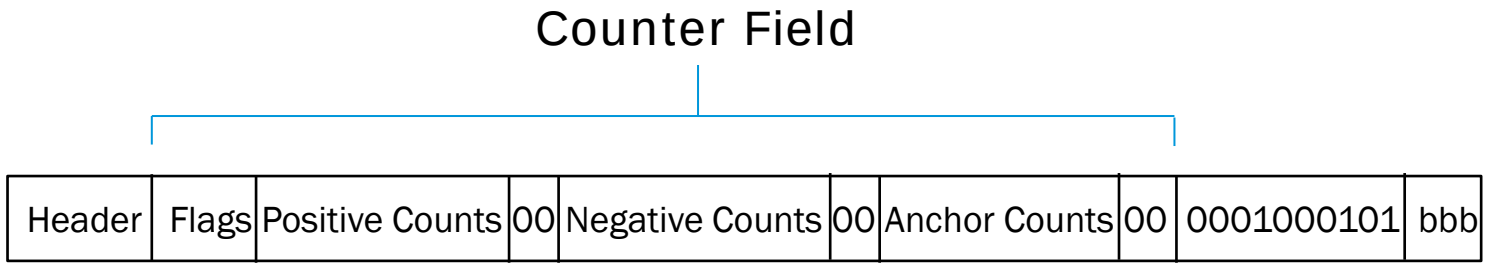
Counter Field Layout

Consists of up to three sets of delta counts (positive, negative or anchor). Each delta count ends with a zero count.

The counters can cover more than one mask update (referred to as protection level)

The protection level is the number of sequential mask updates that can be lost without the decompression algorithm requiring an absolute mask to be sent

If a sequential loss of data structure instances occur that is greater than the protection level then an absolute mask must be requested. The recorded compressed data can then be uncompressed.



NP Bit Extraction Code

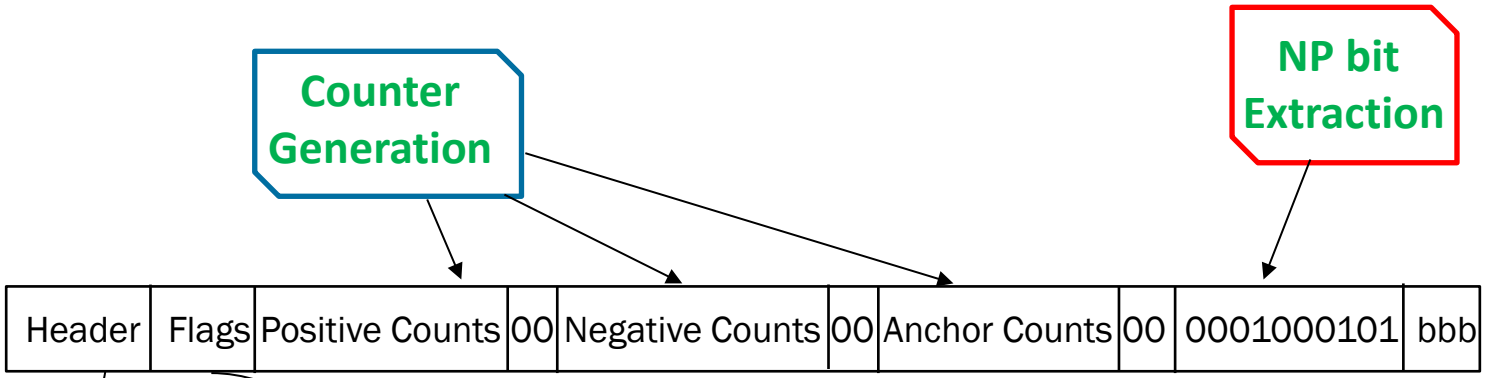
Read Mask array backwards and process each word as shown below

```
copy = Mask [word]
While copy != 0 {
        x = copy & –copy; // produces a word with only the LSB set bit in copy set in x

        IF (x & reference[word]) {set next bit in NP field}
        ELSE {move write pointer}
        OPTIONAL: Exit if too many NP bits and send uncompressed data
        OPTIONAL: Calculate/add error detection and correction fields

        copy = copy – x; // removes the LSB set bit from copy
 }
```

For NP extraction only native processor operations used – very efficient.

Since the process works on the bit level adding BCH or LRC can be done for very little performance or CPU overhead.

Output Byte Array Layout

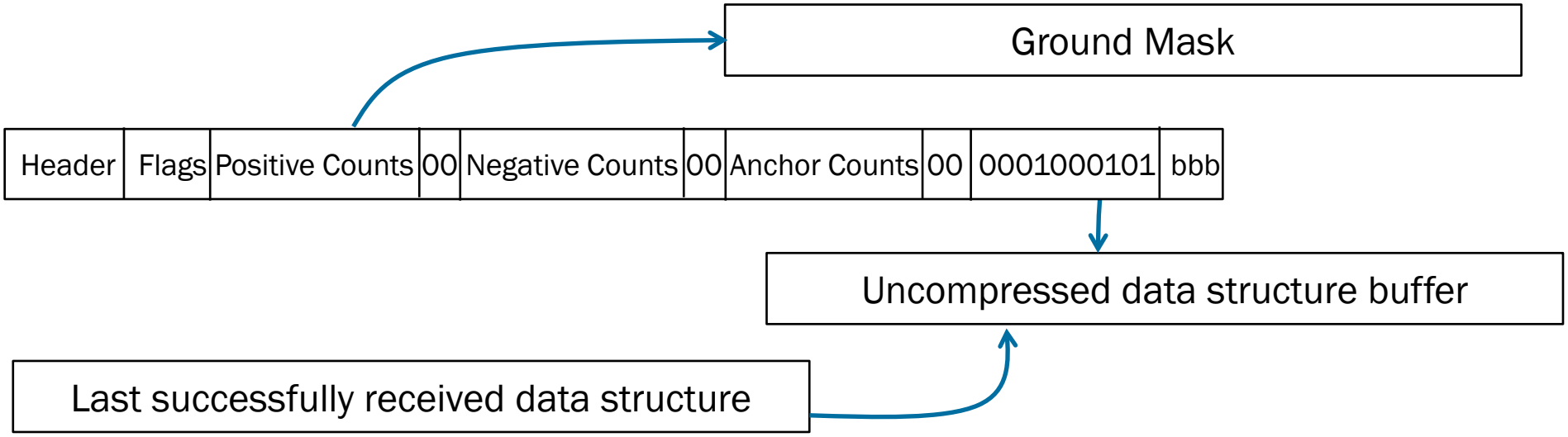


Maybe be done by the packetiser or at all.

Usually the header describes the byte length. ID s the data structure and indicates compressed data

3 bit flag describing types of counts in history

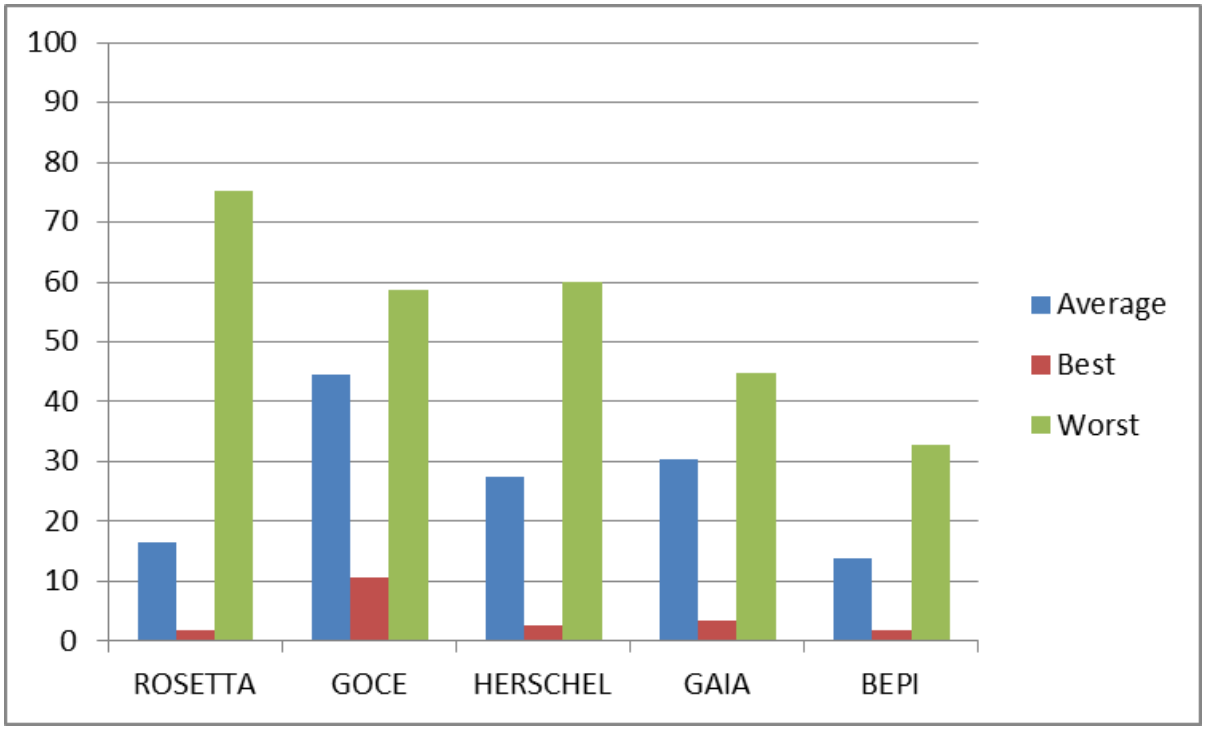
POCKET+ Decompression



1. Read the header and identify data structure type and if compressed
2. If compressed, read counts and reconstruct the mask changes
3. Apply the changes to the ground mask for that data structure “blindly”
4. Read the Non Predictable Bits
5. Take the last data structure that was successfully received and copy it into the uncompressed data structure buffer
6. Use the ground mask to place the recovered Non Predictable bits values into the correct positions in the uncompressed data structure buffer

POCKET+ RESULTS

The algorithm has been written in C and tested using various missions (at least a week of data, except Bepi which uses data collected in ground testing)

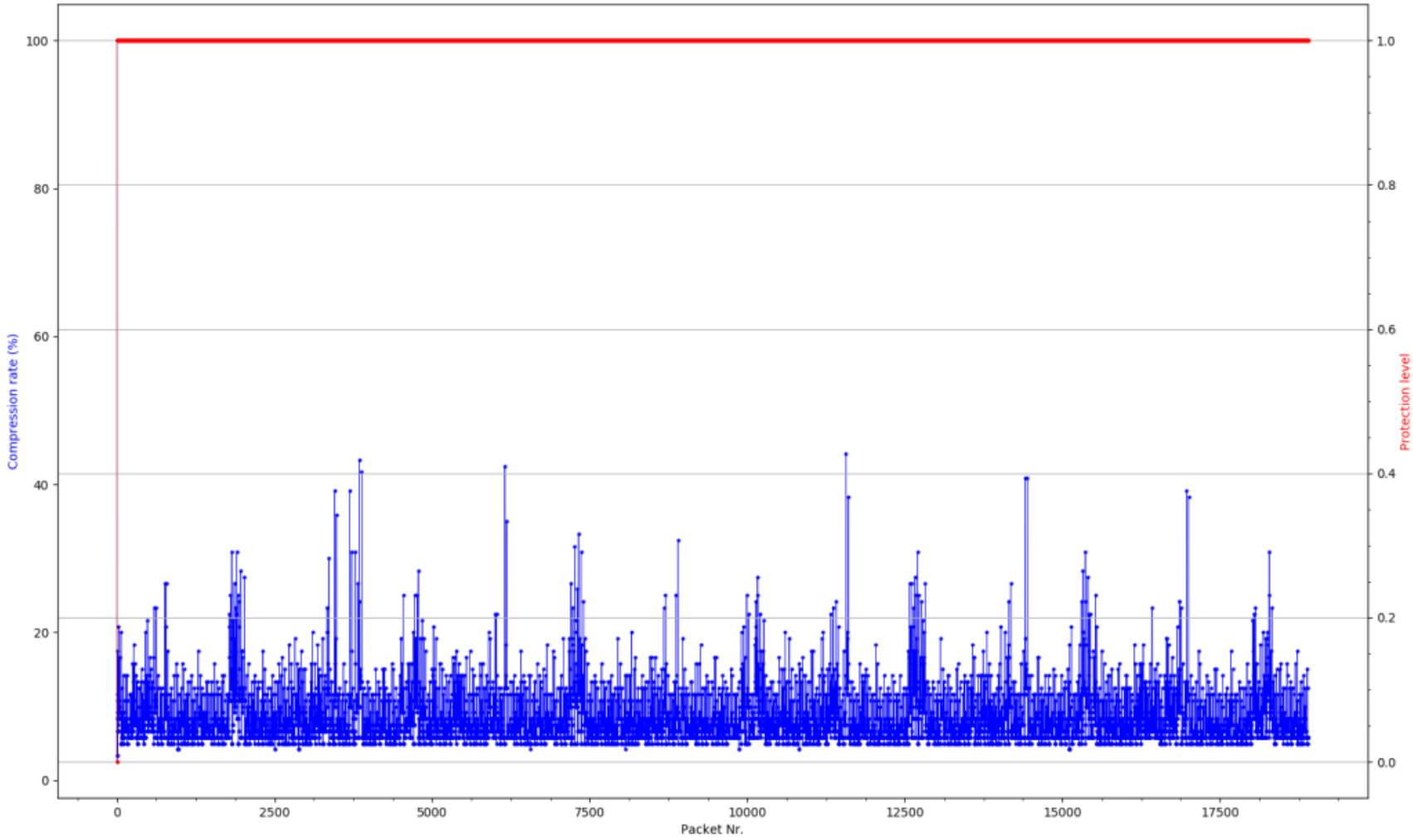


Note these results include the overhead due to using the original protocol (i.e. uncompressed headers)..

If one designs a system with POCKET+ in mind then performances will be much better (e.g. choosing a double INT in preference to a real)

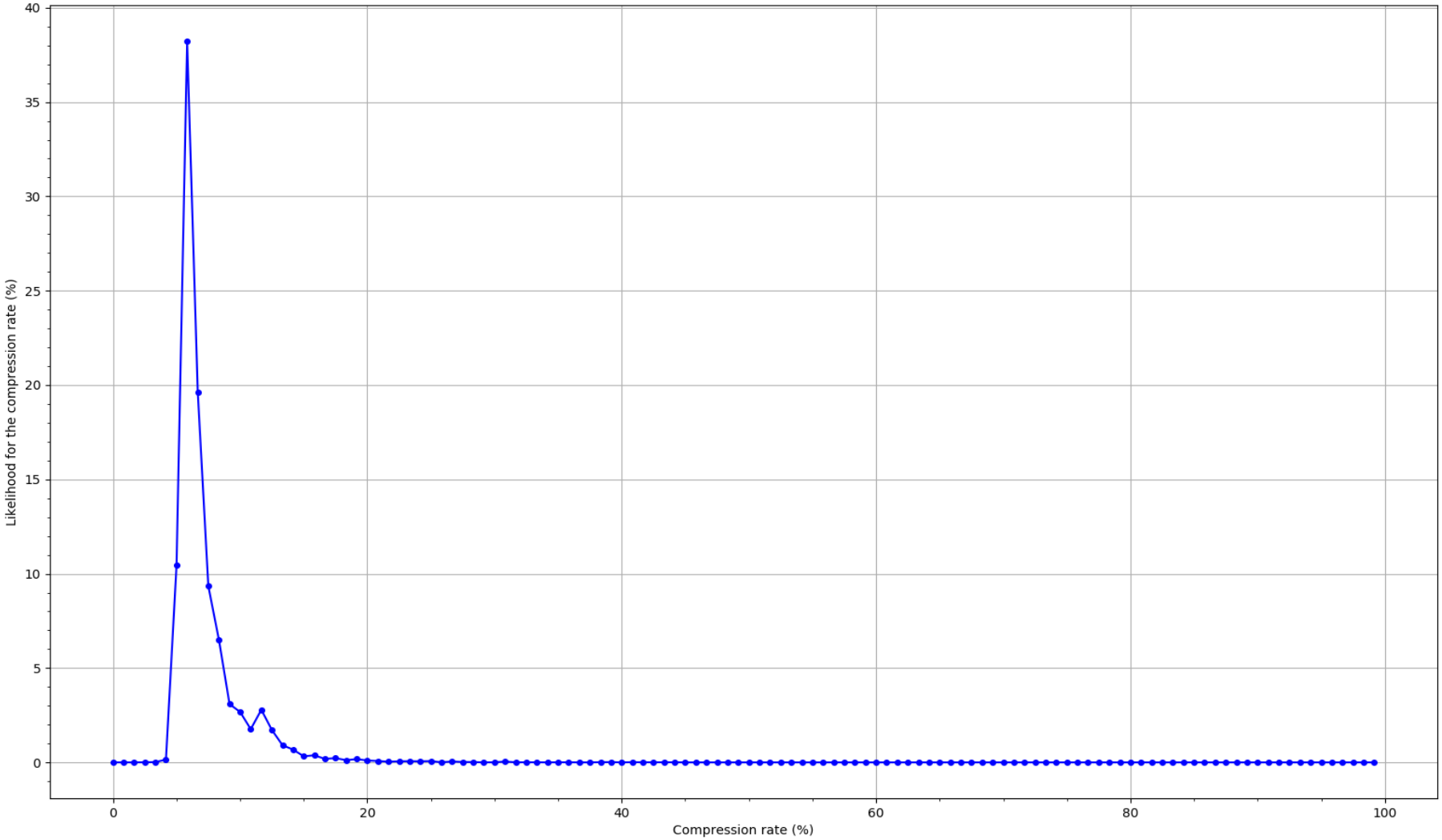
POCKET+ RESULTS

Compression rates are stable for the vast majority of data structures tested



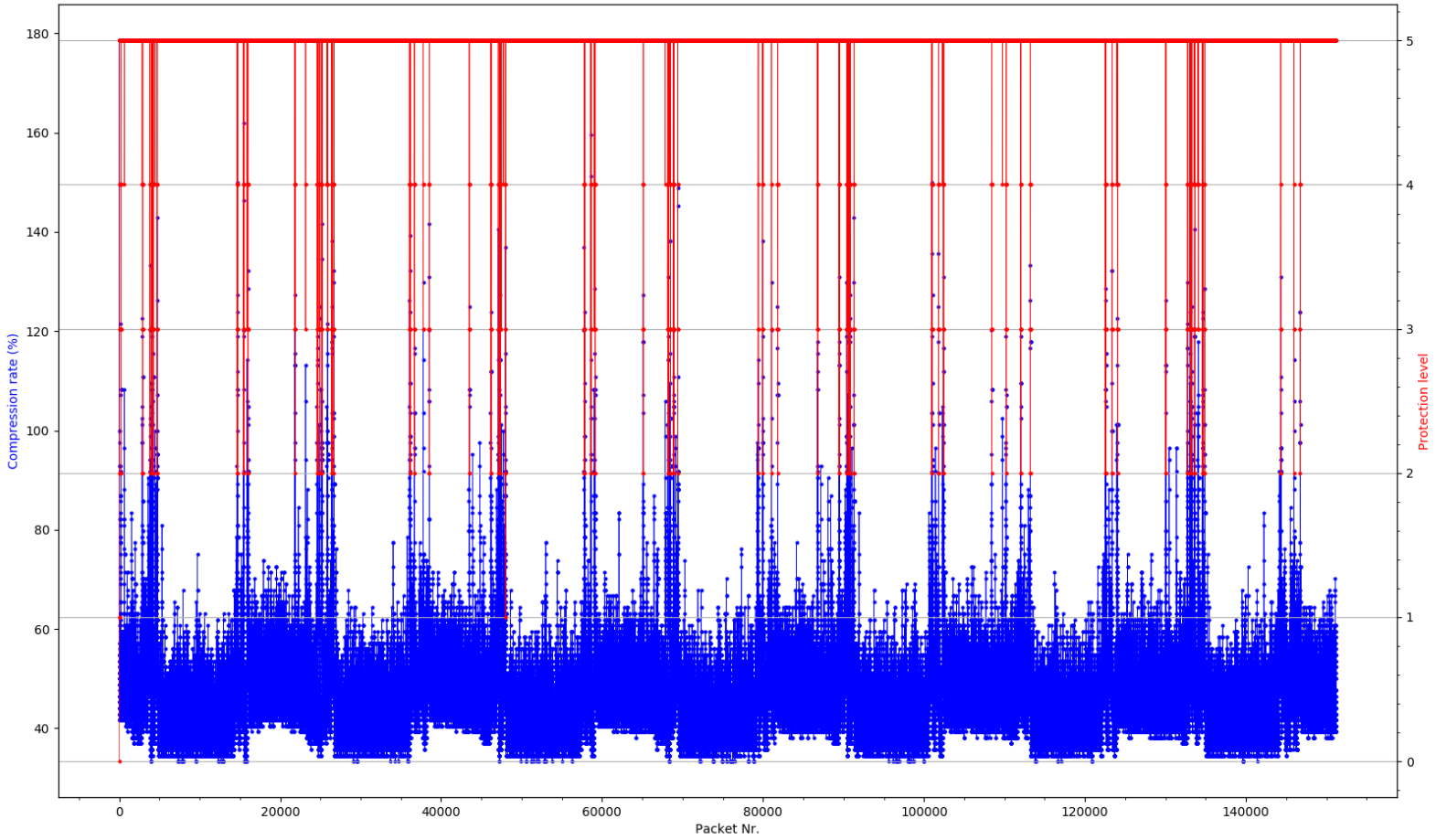
POCKET+ RESULTS

Very high probability for good compression rates. Sharp edge.



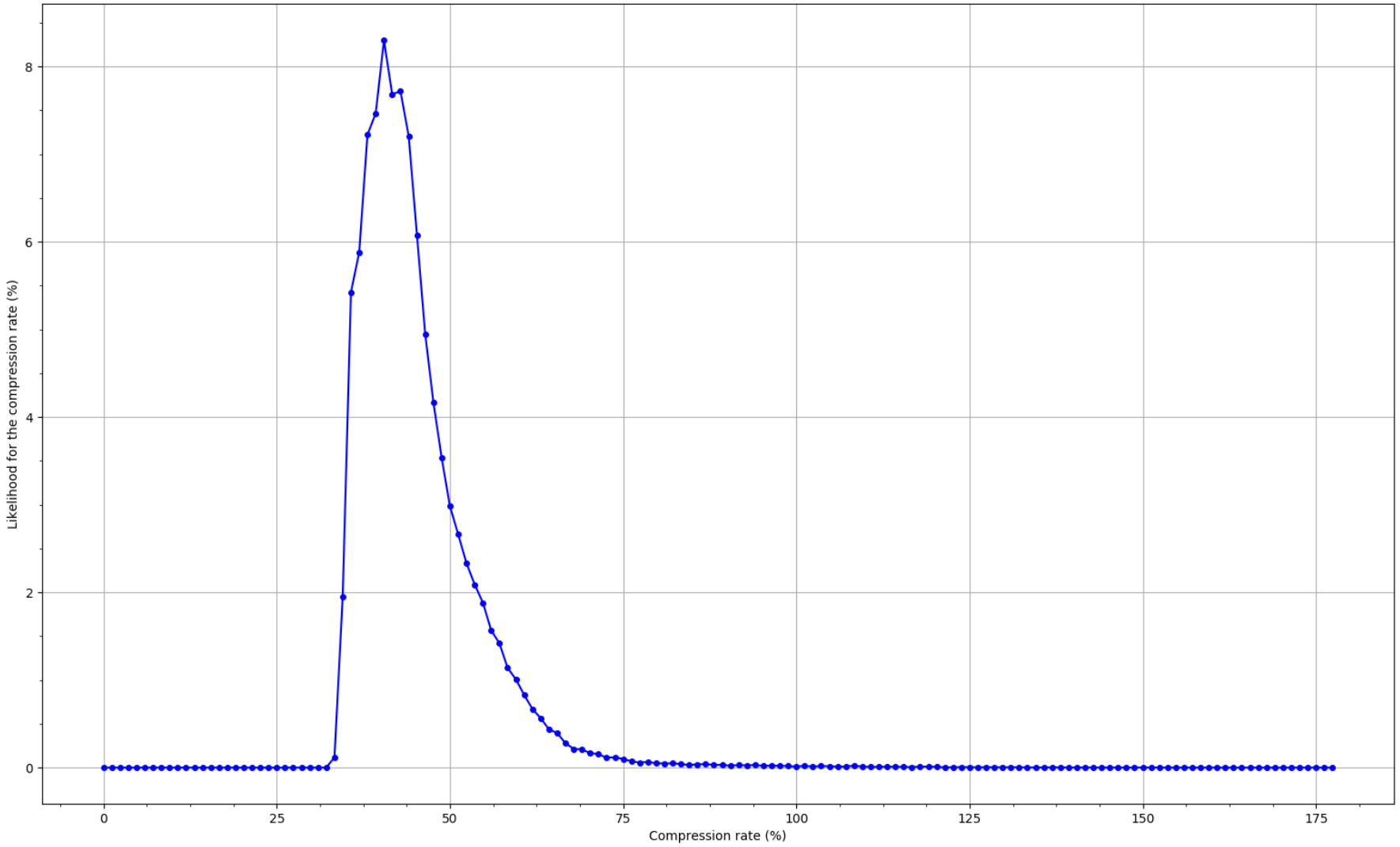
POCKET+ RESULTS

For data that does not compress well, output can occasionally be larger than the input. Pocket reduces the protection level or exits automatically as below.



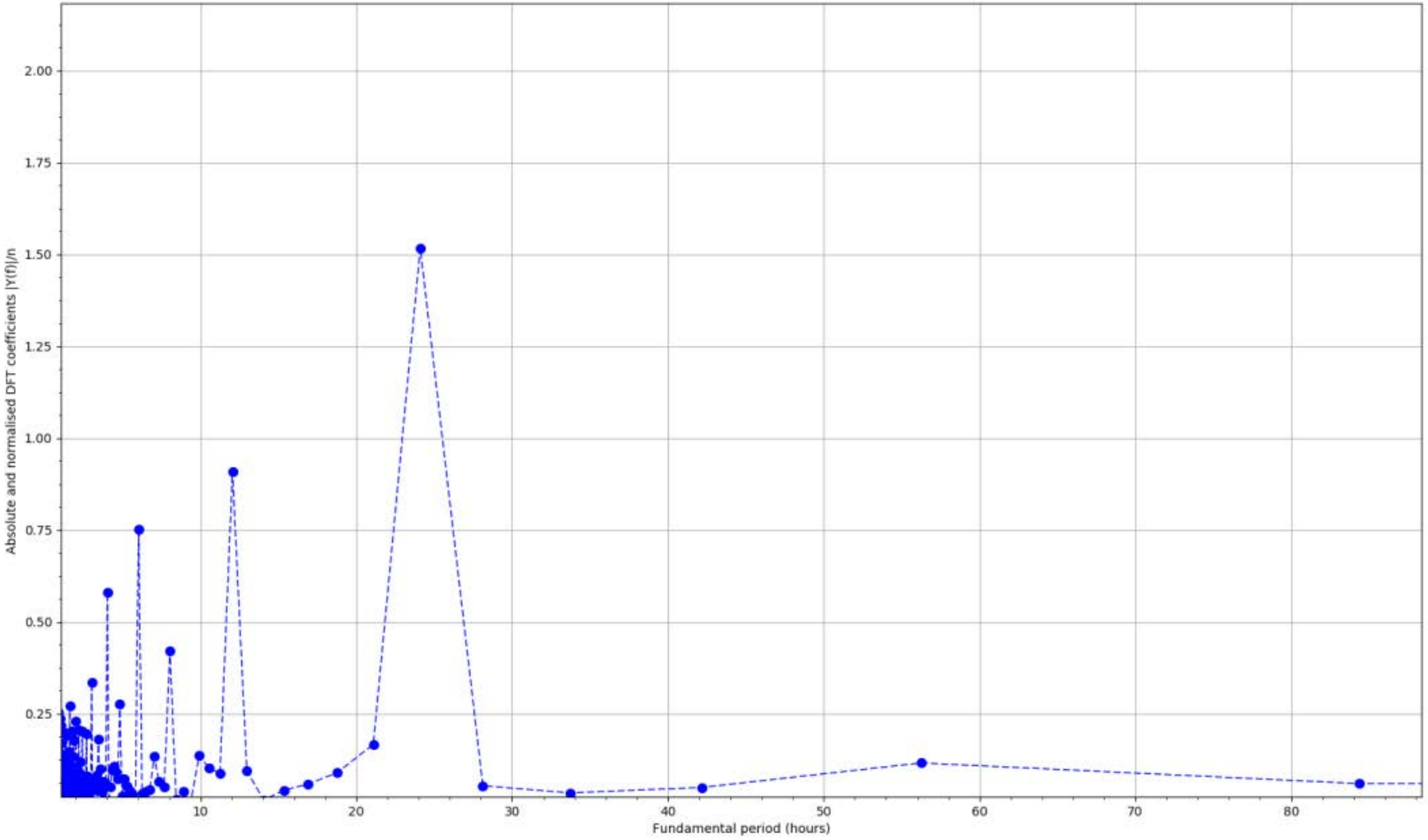
POCKET+ RESULTS

A wider probability spectrum can be seen for these data structures

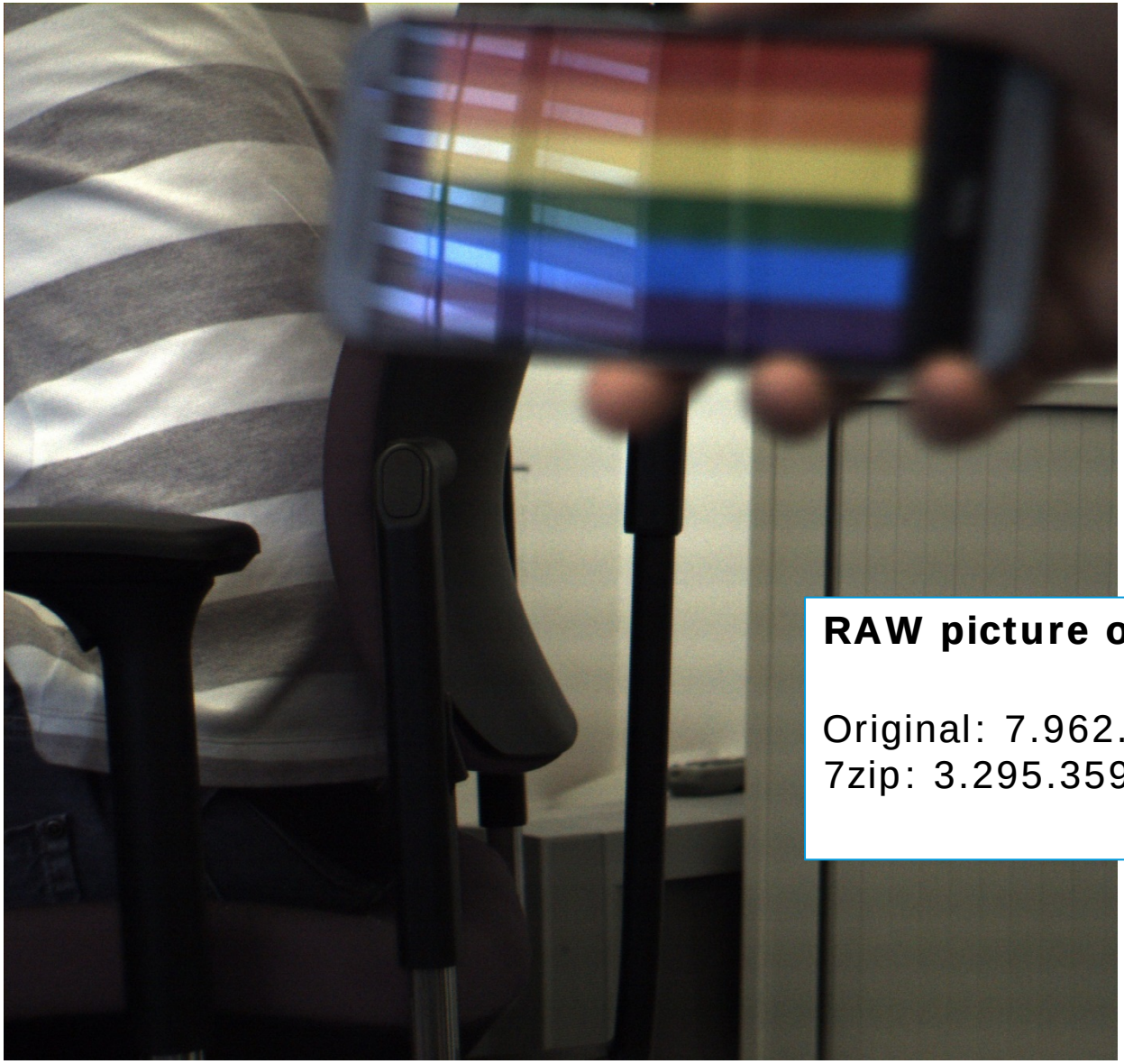


POCKET+ RESULTS

One can see operational changes in a frequency analysis of the performance



POCKET+ for non housekeeping like data



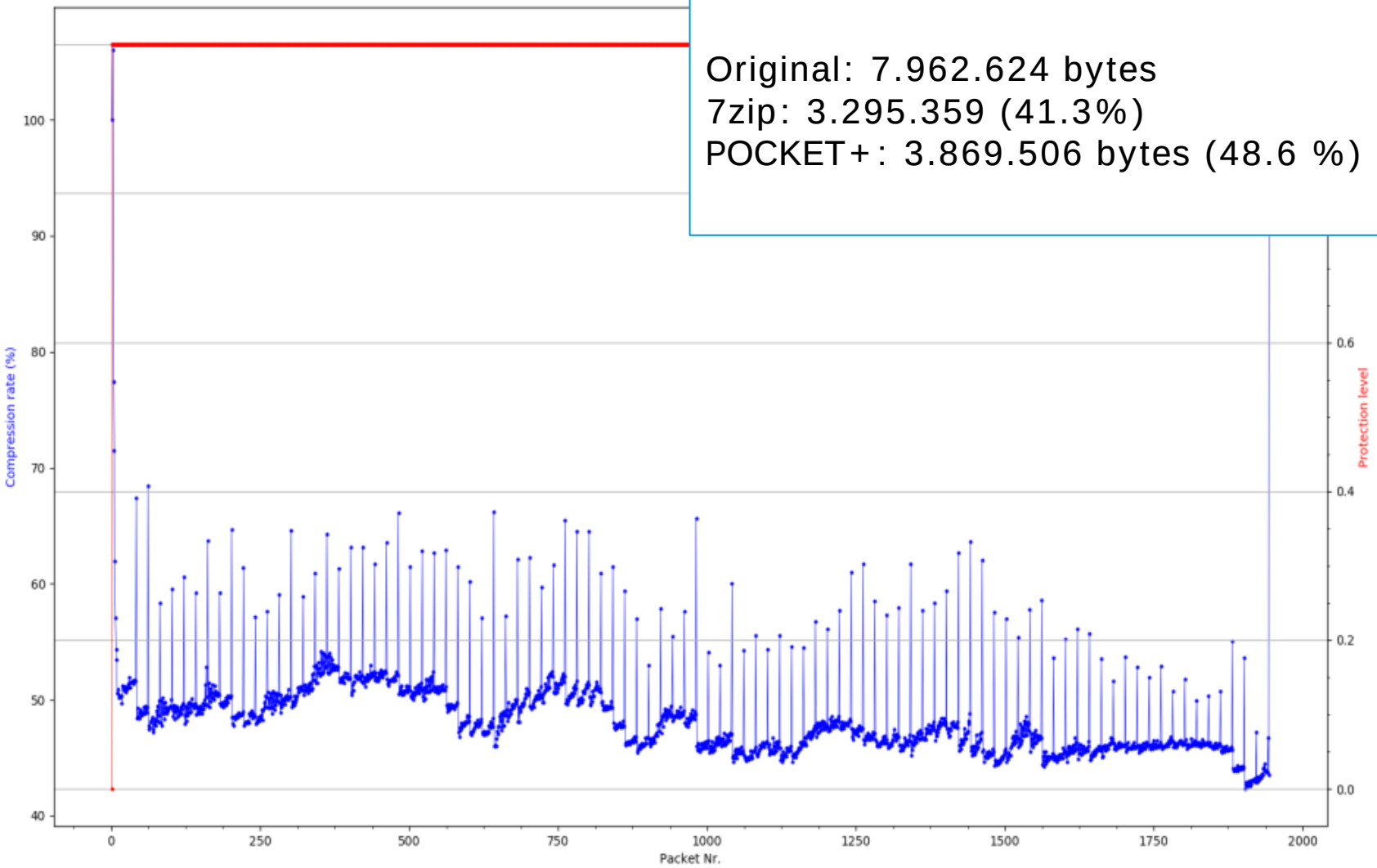
RAW picture of the OPS-SAT CAM

Original: 7.962.624 bytes
7zip: 3.295.359 (41.3%)

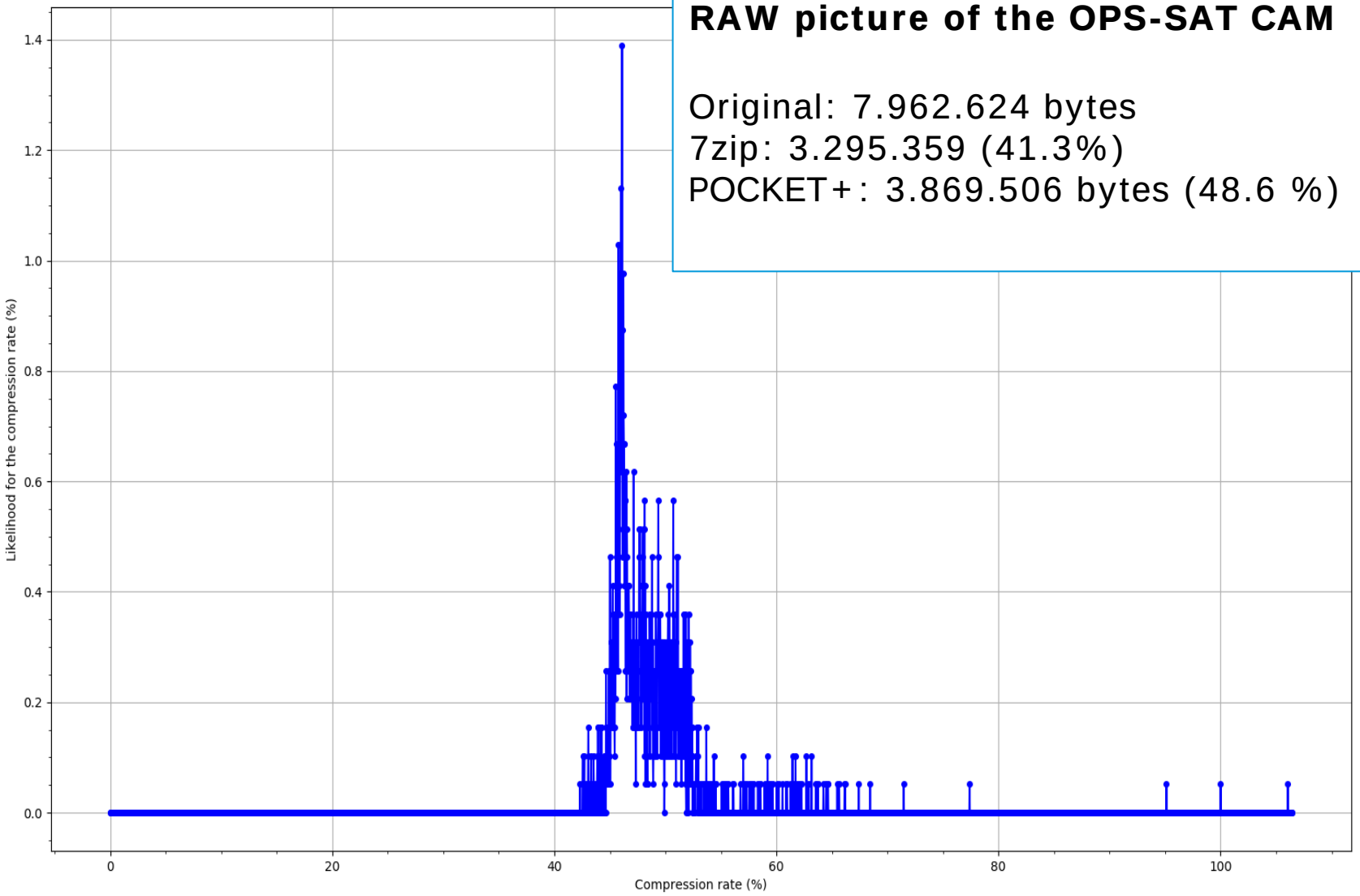
POCKET+ for non housekeeping like data

RAW picture of the OPS-SAT CAM

Original: 7.962.624 bytes
7zip: 3.295.359 (41.3%)
POCKET+: 3.869.506 bytes (48.6 %)



POCKET+ for non housekeeping like data



RAW picture of the OPS-SAT CAM

Original: 7.962.624 bytes
7zip: 3.295.359 (41.3%)
POCKET+: 3.869.506 bytes (48.6 %)

POCKET+ PERFORMANCE

On a standard ESA office PC (Intel® Core™ i5-3320M @ 2.60 GHz) running Windows with no special set-up, compression rates varied per data structure type between 150 Mbps and 1Gbps

Adding protection only had a very small impact on compression performance and speed

Preliminary tests have been run on the Avionics Test bench at ESTEC (RASTA), with the POCKET+ compression module deployed on a LEON-2 (80MHz) space qualified processor.

Different real Venus Express telemetry structures were compressed and the algorithm achieved speeds between 1000-1420 kbps.

Considering the typical housekeeping generation rate is 1-5 kbps there is no need to implement in hardware for the HKTM case (pictures might be different).

TEC-SW/EDD TESTS: TSIM & RASTA

Packet ID	Number of packets	Packet size (bytes)	Protection level	Average compression ratio	Throughput
1002	2.700	126	1	11,60 %	1,23 Mbit/s
1003	2.700	186	1	21,21 %	1,07 Mbit/s
1004	675	304	1	5,20 %	1,25 Mbit/s
1028	21.600	90	1	44,16 %	0,91 Mbit/s
1028	21.600	90	2	45,34 %	0,89 Mbit/s
1028	21.600	90	3	46,52 %	0,88 Mbit/s
1028	21.600	90	4	47,66 %	0,86 Mbit/s
1028	21.600	90	5	48,88 %	0,87 Mbit/s

- Tests on representative space hardware LEON-2 80 MHz
- Throughput depends slightly on the compression ratio
- Increase of the protection level leads to a small impact of the throughput and compression rate
- A throughput of about 1 Mbit/s on a 80 MHz processor indicates approximately 80 instructions per input bit are required

POCKET+ AGENDA

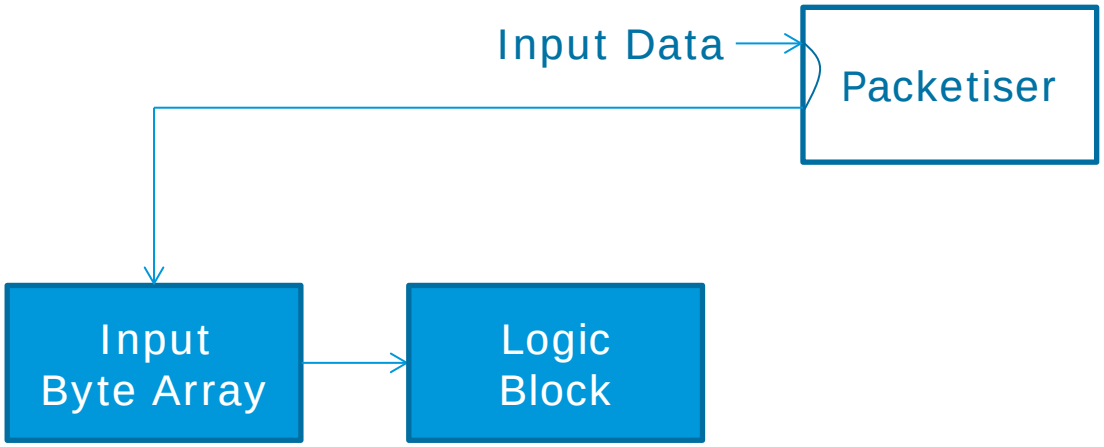
Background:

Technical Explanation and Results:

Exploiting the design further

Conclusions:

Exploiting the design: Background task



The Logic Block can be run alone a background task. In this case the algorithm actively tracks the data redundancy even when this is not being exploited. Tests showed this took 30-15% of the time compared to running the whole process for the different data structures.

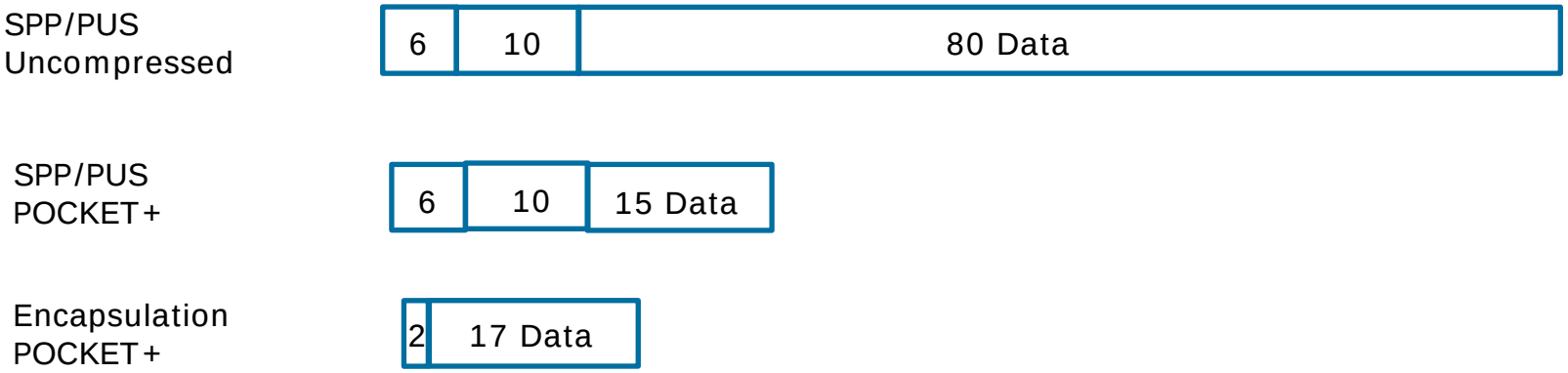
Advantage: When compression is requested the algorithm is already fully optimized and maximum compression is achieved immediately

Exploiting the design: More efficient protocols

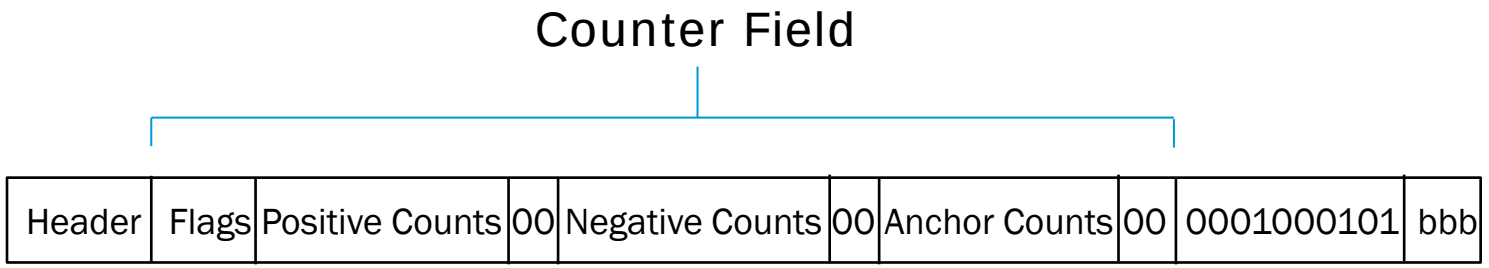
The packetiser can be configured to avoid impacting the data protocol e.g. identical headers used with the length field adjusted and a compression flag set.

However typically protocol headers compress very, very well and in principle we do not lose the information. Therefore we could choose a more efficient protocol for transport while the data structure is compressed.

Below shows the relative gain based on a compressing a typical CCSDS Venus Express Data Structure and encapsulating it all using another CCSDS protocol.



Exploiting the design: Recovering lost information



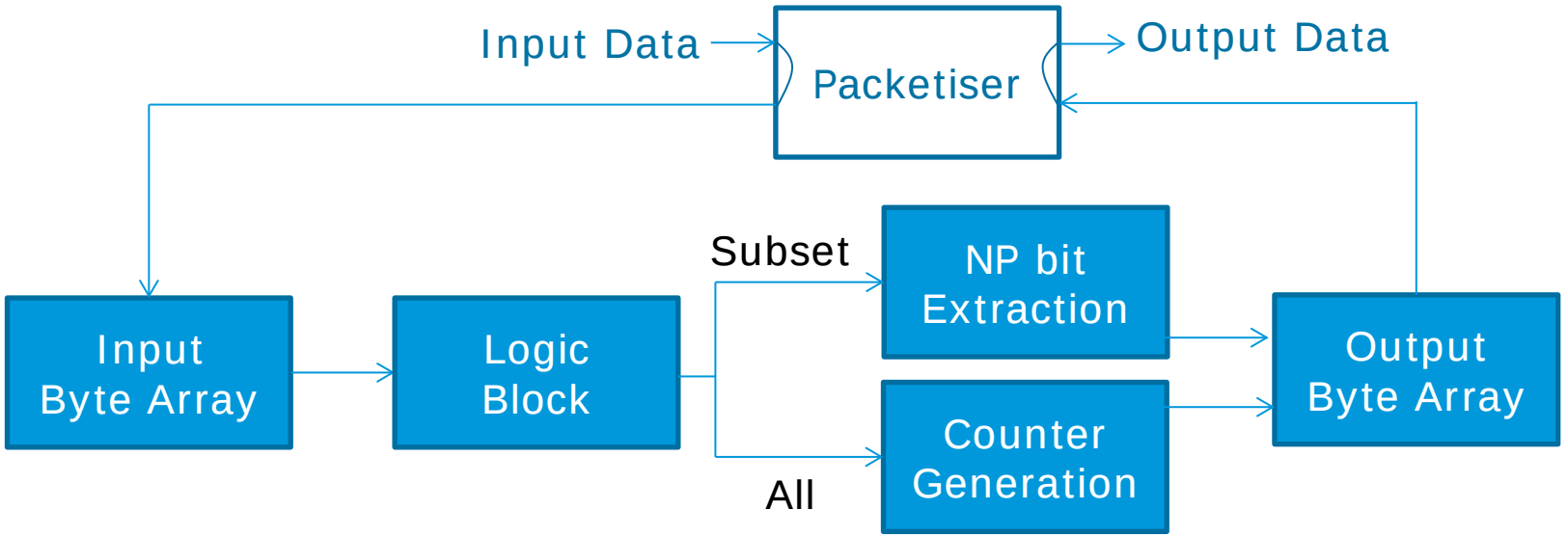
Recap that the counter field can cover more than one mask update. Therefore we are sending counter information multiple times in sequential data structure instances i.e. time redundancy

Even if an instance is lost completely we will still know a lot about its content from the subsequent counters

Advantage: The majority of the Non Predictable Bits tend to be the LSBs of parameters and the Predictable Bits are the MSBs (especially with integer encoding). This might be all the information we need for decision making..

The question is whether this information can be used operationally. How would we…. Display it?, Check it? Store it?

Exploiting the design: Increasing information density

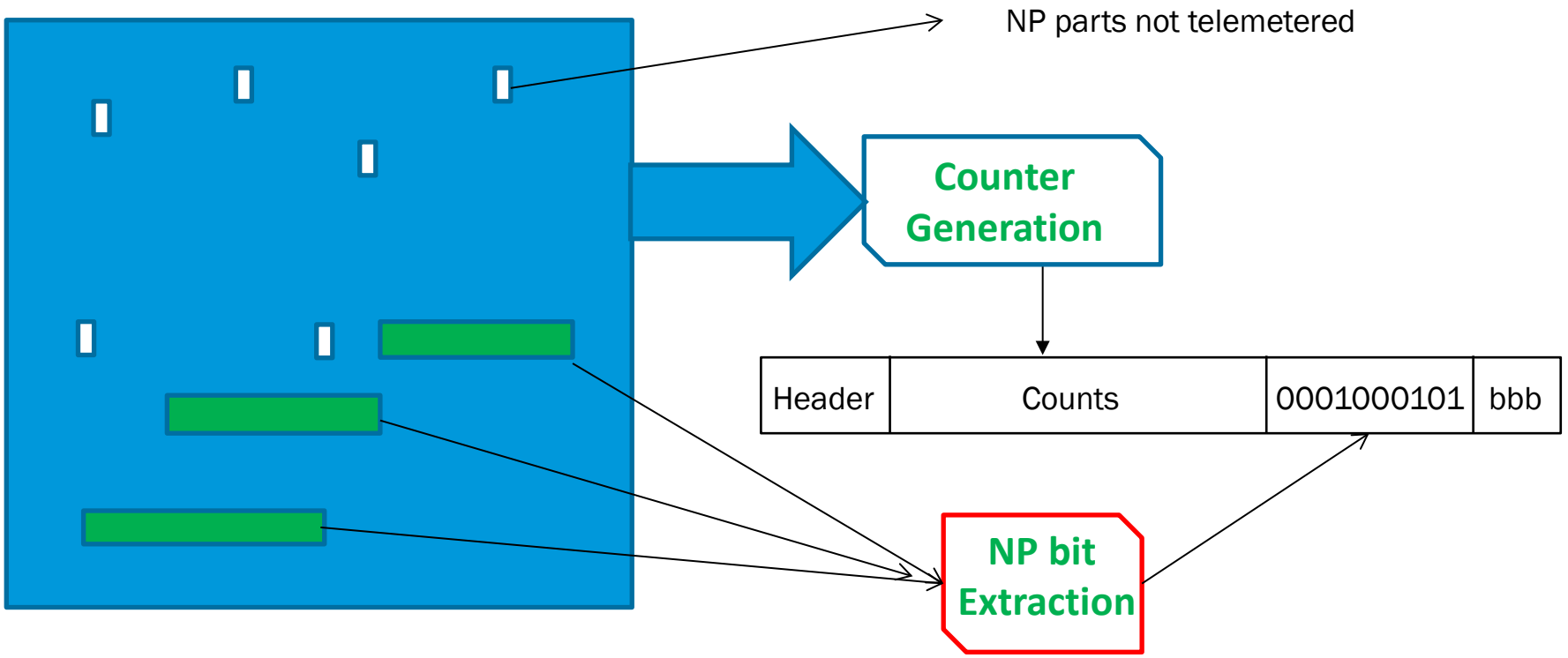


Let's exploit this idea further. Note that the NP bit extraction process can be run on a subset of the data structure compared to that used to generate the counters

Note the NP bit extraction process can consume up to 66% of the total time needed for compression.

Advantage: The NP bit extraction runs on that subset of the data structure carrying information we absolutely need. At the same time we can get much more information into the Output concerning the predictable bit statuses on a much larger data set - as the CPU usage is much less.

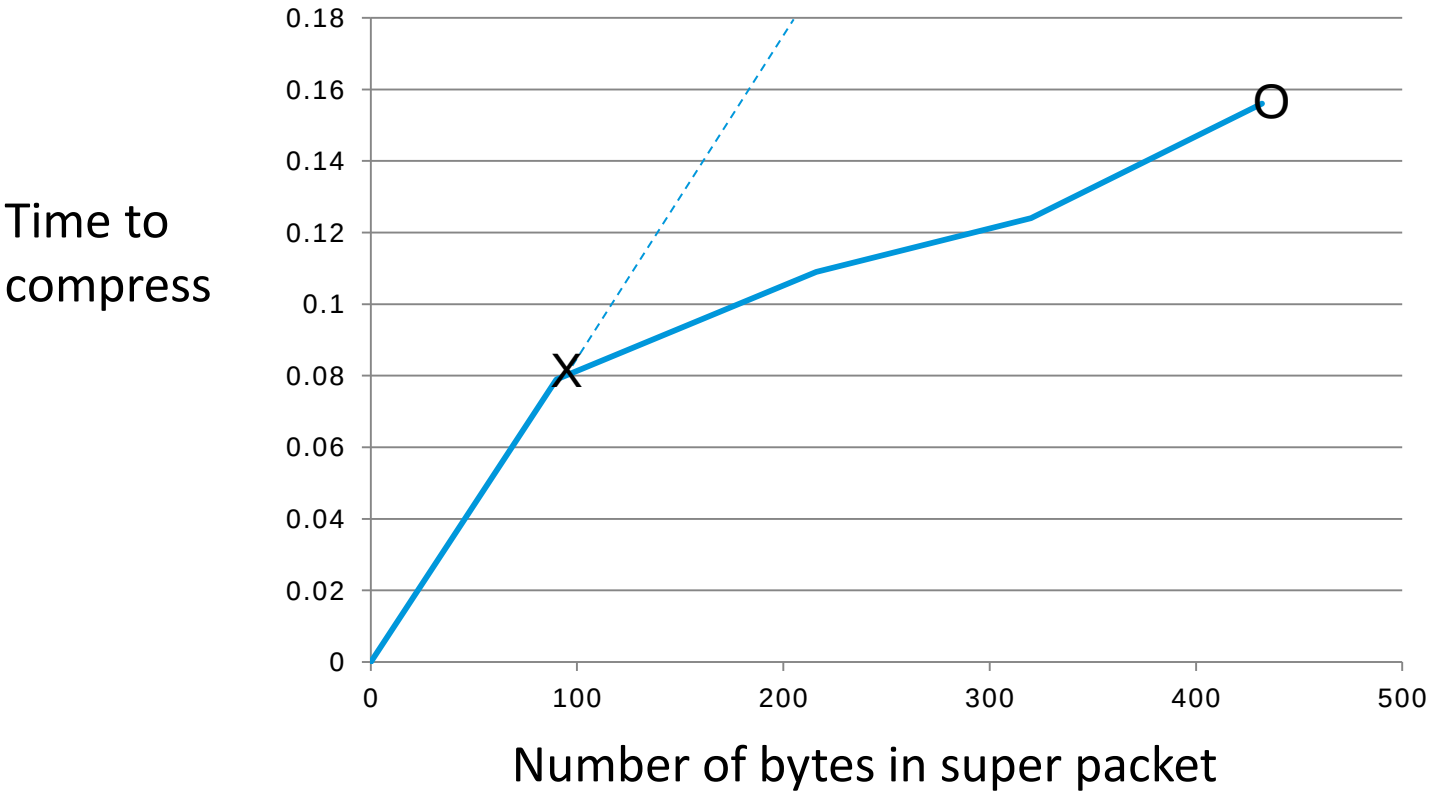
Exploiting the design: Increasing information density



Advantage: The majority of the Non Predictable Bits tend to be the LSBs of parameters and the Predictable Bits are the MSBs (especially with integer encoding)

Hence the we might get all the information we need for decision making in the counters

Exploiting the design: Increasing information density



Point X represents a packet X compressed using POCKET+ (typically factor 2).
Other packet types are added but only the Logic Block and Counter Generation run.
Point O represents a packet Y, which contains all the information of packet X plus the predictable bit status of significantly more bits than packet X (400% more)
The difference in time to compress packet X compared to Y is only double.

Exploiting the design: Introducing EDAC

The counter field can cover more than one mask update so we can check previously sent counters to check that they are consistent i.e. partial time redundancy.

The same mechanism can be used to potentially recover bad frames. By feeding back the 100% known value of certain bits back into the channel coding layer we can assist it to decode the frame successfully i.e. improve error correction.

Since the counter fields are sent multiple times we could add an internal block code (e.g. BCH) to the NP bits which are only sent once. This could help to recover bad frames. This has a low CPU cost since the NP bit extraction process already works at bit level i.e. improve error correction.

Instead of bit stuffing to reach an integer number of bytes we could calculate a Longitudinal Redundancy Code (LRC) for the NP bit field. This has a low CPU cost since the NP bit extraction process already works at bit level and we can work out in advance the length of the LRC field i.e. improve error detection.

Since data structures are much smaller we can send them multiple times still using less bandwidth than before i.e. full time redundancy.

POCKET+ AGENDA

Background:

Technical Explanation and Results:

Exploiting the design further

Conclusions:

POCKET+ CONCLUSION

1. POCKET+ compresses better than WinZiping a single file with entire week's data (compared to creating a million individual data packets as POCKET+ does).
2. It is simple, deterministic and very efficient. It can work on the real-time data stream and therefore one benefits from stored data compression for free, as well as the multiplication factor during playback.
3. It tracks data redundancy automatically so it is safe to use as behaviour changes.
4. Includes simple configurable exit criteria defining when it is better to send uncompressed data i.e. compression only takes place when it makes sense.
5. It has a low footprint. The compression module is 500 lines of C code and it requires the equivalent of one extra input buffer for each data structure type (assuming 32 bit processor).
6. It is highly configurable. One can turn it on and off for each individual data structure type or compress one out of every x iterations for example. One can adjust the tracking period and protection level autonomously or by command. The decompression module does not need to be synchronized with any configuration changes.

POCKET+ CONCLUSION

6. It is robust to occasional data loss and provides an easy way to introduce different error detection and correction schemes – improving the robustness of the data link.
7. The compression and decompression modules are generic with small differences in protocols being dealt with by a separate packetiser.
8. The packetiser can be configured to avoid impacting the data protocol e.g. identical headers used with the length field adjusted and a compression flag set.
9. Typically protocol headers compress very, very well and in principle we do not lose the information. Therefore we could choose a more efficient protocol for transport while the data structure is compressed.
10. The design introduces the possibility to recover the majority of information in instances that have been completely lost.
11. The design introduces the possibility to send compressible information for many more parameters (and partial information) than we could possibly download at the moment.


Thank
you