

INTERNATIONAL ORGANISATION FOR STANDARDISATION
ORGANISATION INTERNATIONALE DE NORMALISATION
ISO/IEC JTC1/SC29/WG11
CODING OF MOVING PICTURES AND AUDIO

ISO/IEC JTC1/SC29/WG11 MPEG/m51024
October 2019, Geneva, Switzerland

Source: Apple Inc.

Status: Input document

Title: G-PCC low latency bypass bin coding

Author(s): David Flynn
Khaled Mammou

davidflynn@apple.com
kmammou@apple.com

Abstract

The current G-PCC specification includes two methods for formatting coded symbols. The first uses an arithmetic codec to encode all symbols and produces a single data stream that forms the data unit/payload body. The second [1, M47827] splits the symbols into two sub-streams, one coded with the arithmetic codec and the other as a string of bypass bins in reverse order. The main motivation for this second encoding is to avoid using the more expensive arithmetic codec to code incompressible symbols. The cost of this is that the first bypass symbol to decoded is represented using the lass bit of the payload.

This contribution proposes an alternative approach based on a chunk-interleaved representation of the two sub-streams. It aims to balance the benefit of not arithmetically coding bypass bins, the ability to transmit and receive the bitstream as a whole in the forward order, with the chunk signalling overhead. Based on a 256-byte chunk size, the overhead is 0.4% of bitrate.

Chunking

Data from the two symbol streams are multiplexed to form chunks, as depicted in Figure 1. Each 256 byte chunk is formed of:

- a one byte header that indicates the number of arithmetically coded bytes present in the chunk, n ,
- n bytes of arithmetically coded data, and
- $255 - n$ bytes of (non arithmetically coded) bypass data.

In the event that a low end-to-end latency mode is enabled, the last byte of bypass data (if present) contains

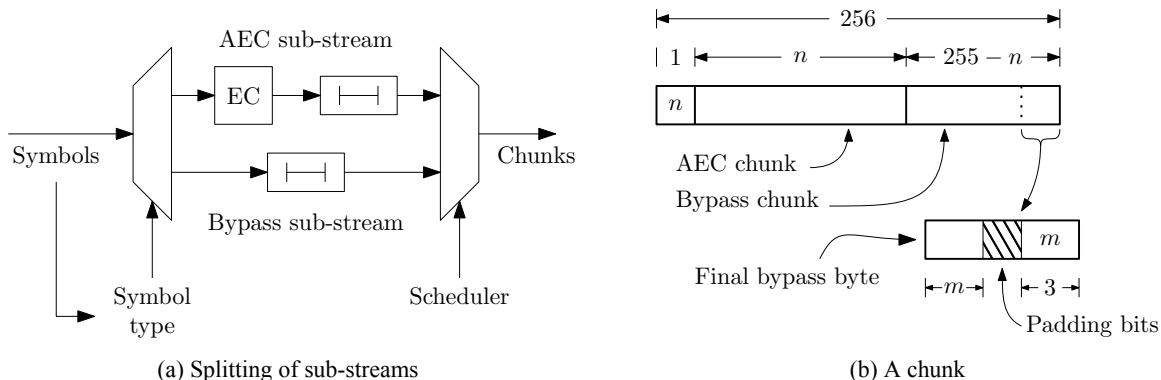


Figure 1 – Construction of a multiplexed sub-stream chunk

a three-bit value indicating the number of bits present in the last byte.

The chunk syntax permits a chunk with no bypass data ($n = 255$), and similarly it permits a chunk consisting only of bypass data ($n = 0$).

The chunk size of 256 bytes is chosen since it allows simple implementation when combined with bitwise operation of the arithmetic codec.

A sequence of consecutive chunks are used to represent the entirety of the two symbol streams.

In order to avoid inefficiencies associated with the combined length of the symbol streams not being a multiple of 255 bytes, the last chunk may be truncated. While the truncation point may be determined by external means (such as a payload length indication), this is not necessary for decoding operations, since the number of arithmetically coded bytes is known and the bypass bytes follow immediately. An actual decoder implementation should either pad the buffer to the full chunk size, or otherwise test to avoid buffer overflow conditions.

Overhead

Each chunk of 255 bytes of coded symbol data incurs an overhead of one byte (low end-to-end latency mode disabled), equivalent to 0.4%.

Simulations using the common test conditions [2, 3] show that the implementation performance matches the predicted results.

Latency

Considering only the chunk syntax, for a given set of coded symbols there exist multiple chunked representations. In extrema, an encoder could choose to write chunks consisting only of arithmetic coded data followed by chunks of bypass coded data, or vice versa.

In order to bound the resources required by the decoder, the chunk construction is semantically constrained such that a decoder need maintain only two chunk buffers and a read pointer for each of the respective coded symbol sub-streams.

With this construction, while it is possible to either bound the resources required by the decoder, or alternatively, those of the encoder, placing a bound on the end-to-end delay is not possible. This may, however, be achieved by allowing an encoder to produce incomplete chunks. First, consider extremes of a pair of coded symbol streams consisting of m AEC bytes and a single bypass bit that can be coded in one of two ways:

- The first chunk contains 254 AEC bytes and one bypass byte, with all subsequent chunks containing only AEC bytes.
- The last chunk containing fewer than 255 AEC bytes and one bypass byte, with all preceding chunks containing only AEC bytes.

In the first case, the encoder cannot write out the first chunk until it is determined that there are no more bypass bits, which is only known at the end of the stream. However, a decoder can decode chunk-by-chunk.

In the second case, the encoder can write out each chunk on-the-fly, but if the single bypass bin were required at the start of decoding, a decoder must buffer the entire stream before continuing.

The total end-to-end latency, as well as the encoder and decoder resources can be bounded by allowing the final bypass byte of a chunk to be incomplete. An encoder may, at its discretion, flush a chunk without waiting a complete bypass byte.

Results

Table 1 – Compression performance of chunked coding with bitwise occupancy coding compared to TMC13v7.0 using octree geometry and LoD attribute coding

Condition	Class	BPP Ratio [%]			D1	D2	BD-Rate [Δ %]			R	Avg. of ratio maxrssk [%]		Ratio of avg. runtime [%]	
		Geometry	Colour	Refl			Y	Cb	Cr		Encoder	Decoder	Encoder	Decoder
C1_ai	cat1-A						0.4	0.4	0.4		100	100	101	100
C1_ai	cat3-fused						0.4	0.4	0.4	0.4	100	100	97	93
C1_ai	cat3-frame									0.5	100	100	99	96
C1_ai	overall						0.4	0.4	0.4	0.5	100	100	100	98
C2_ai	cat1-A				0.4!	0.4!	0.4!	0.4!	0.4!		100	100	100	97
C2_ai	cat1-B				0.4!	0.4!					100	100	94	
C2_ai	cat3-fused				0.6!	0.6!	0.4!	0.6!	0.6!	0.5!	100	100	99	92
C2_ai	cat3-frame				0.4	0.4				0.7	100	100	97	98
C2_ai	overall				0.4!	0.4!	0.4!	0.5!	0.5!	0.6!	100	100	100	
CW_ai	cat1-A	100.4	100.4								100	100	100	101
CW_ai	cat1-B	100.4									100	100	99	92
CW_ai	cat3-fused	100.4	100.4	100.4							100	100	104	105
CW_ai	cat3-frame	100.4		100.4							100	100	99	95
CW_ai	overall	100.4	100.4!	100.4							100	100	99	96
CY_ai	cat1-A						0.4	0.4	0.4		100	100	100	100
CY_ai	cat3-fused						0.4	0.4	0.4	0.4	100	100	102	98
CY_ai	cat3-frame									0.4	100	100	107	107
CY_ai	overall						0.4	0.4	0.4	0.4	100	100	102	101

NOTE — Condition CY metrics reported using Hausdorff PSNR.

Table 2 – Compression performance of chunked coding with bitwise occupancy coding compared to TMC13v7.0 using octree geometry and RAHT attribute coding

Condition	Class	BPP Ratio [%]			D1	D2	BD-Rate [Δ %]			R	Avg. of ratio maxrssk [%]		Ratio of avg. runtime [%]	
		Geometry	Colour	Refl			Y	Cb	Cr		Encoder	Decoder	Encoder	Decoder
C1_ai	cat1-A						0.4	0.4	0.4		100	100	101	101
C1_ai	cat1-B						0.4	0.4	0.4		100	100	99	98
C1_ai	cat3-fused						0.4	0.4	0.4	0.4	100	100	99	99
C1_ai	cat3-frame									0.5	100	100	104	105
C1_ai	overall						0.4	0.4	0.4	0.5	100	100	100	100
C2_ai	cat1-A				0.4	0.4	0.5	0.5	0.5		100	100	99	99
C2_ai	cat1-B				0.4	0.4	0.4	0.4	0.5		100	100	100	101
C2_ai	cat3-fused				0.4	0.4	0.4	0.4	0.4	0.4	100	100	99	103
C2_ai	cat3-frame				0.4	0.4				0.9	100	100	105	109
C2_ai	overall				0.4	0.4	0.5	0.4	0.5	0.8	100	100	100	101

Table 3 – Compression performance of chunked coding with bitwise occupancy coding compared to TMC13v7.0 using octree geometry and LoD attribute coding

Condition	Class	BPP Ratio [%]			D1	D2	BD-Rate [Δ %]			R	Avg. of ratio maxrssk [%]		Ratio of avg. runtime [%]	
		Geometry	Colour	Refl			Y	Cb	Cr		Encoder	Decoder	Encoder	Decoder
C1_ai	cat1-A						0.4	0.4	0.4		100	100	97	97
C1_ai	cat3-fused						0.4	0.4	0.4	0.4	100	100	91	88
C1_ai	cat3-frame									0.5	100	100	108	104
C1_ai	overall						0.4	0.4	0.4	0.5	100	100	99	98
C2_ai	cat1-A				0.4!	0.4!	0.4!	0.4!	0.4!		100	102	92	93
C2_ai	cat1-B				0.4!	0.4!					100	101	84	
C2_ai	cat3-fused				0.6!	0.6!	0.4!	0.6!	0.6!	0.5!	100	100	93	92
C2_ai	cat3-frame				0.4	0.4				0.7	100	102	101	92
C2_ai	overall				0.4!	0.4!	0.4!	0.5!	0.5!	0.6!	100	101	92	
CW_ai	cat1-A	100.4	100.4								100	100	92	95
CW_ai	cat1-B	100.4									100	100	97	93
CW_ai	cat3-fused	100.4	100.4	100.4							100	100	107	101
CW_ai	cat3-frame	100.4		100.4							100	100	103	97
CW_ai	overall	100.4	100.4!	100.4							100	100	96	95
CY_ai	cat1-A						0.4	0.4	0.4		100	100	96	96
CY_ai	cat3-fused						0.4	0.4	0.4	0.4	100	100	97	94
CY_ai	cat3-frame									0.4	100	100	112	109
CY_ai	overall						0.4	0.4	0.4	0.4	100	100	99	98

NOTE — Condition CY metrics reported using Hausdorff PSNR.

Bitstream syntax

```
ae_chunk() {
    chunk_num_ae_bytes = u(8);
    for (i = 0; i < num_ae_bytes; i++)
        chunk_ae_byte[i] = u(8);
    for (j = 0; i < 255; j++, i++) {
```

```

    if (chunk_padding_enabled_flag && i == 254) {
        chunk_bypass_5bits = u(5);
        chunk_bypass_num_flushed_bits = u(3);
    } else
        chunk_bypass_byte[j] = u(8);
}
}

```

chunk_padding_enabled_flag equal to 1 indicates that the last byte of a chunk containing fewer than 254 **chunk_ae_byte** elements contains a variable number of padding bits. **chunk_flush_enabled_flag** equal to 0 indicates that **chunk_bypass_5bits** and **chunk_bypass_pad** are not present in the bitstream.

chunk_num_ae_bytes indicates the number of **chunk_ae_byte** and **chunk_bypass_byte** elements present in a chunk.

The variable NumChunkBypassBytes is derived as follows:

$$\text{Max}(0, (\text{chunk_padding_enabled_flag} ? 253 : 254) - \text{chunk_num_ae_bytes})$$

chunk_ae_byte[i] specifies the i-th byte of the arithmetically encoded symbol sub-stream of the current chunk. Each **chunk_ae_byte[i]** is appended to the AeByteStream array as follows:

```

for (i = 0; i < chunk_num_ae_bytes; i++)
    AeByteStream[AeStreamLen++] = chunk_ae_byte[i]

```

chunk_bypass_byte[j] specifies the j-th byte of the bypass symbol sub-stream of the current chunk. Each **chunk_bypass_byte** is appended to the BypassBitStream array as follows:

```

for (j = 0; j < NumChunkBypassBytes; j++)
    for (b = 7; b >= 0; b--)
        BypassBitStream[BypassBitStreamLen++] = (chunk_bypass_byte[j] >> b) & 1

```

chunk_bypass_num_flushed_bits specifies the number of bypass bits contained in **chunk_bypass_5bits**.

chunk_bypass_5bits specifies the values of up to five bypass bits at the end of the bypass symbol sub-stream of the current chunk. Each bit is appended to the BypassBitStream array as follows:

```

for (b = 0; b < chunk_bypass_num_flushed_bits; b++)
    BypassBitStream[BypassBitStreamLen++] = (chunk_bypass_5bits >> (4 - b)) & 1

```

References

- [1] D. Flynn and S. Lasserre, “G-PCC Bypass coding of bypass bins,” ISO/IEC JTC1/SC29/WG11, 126th meeting, Geneva, Tech. Rep. m47827, Mar. 2019.
- [2] 3DG, “Common Test Conditions for PCC,” ISO/IEC JTC1/SC29/WG11, 127th meeting, Gothenburg, Tech. Rep. w18665, Jul. 2019.
- [3] —, “G-PCC performance evaluation and anchor results,” ISO/IEC JTC1/SC29/WG11, 127th meeting, Gothenburg, Tech. Rep. w18667, Jul. 2019.