Investigation of HEVC/H.265 Encoder Decisions with HARP 2.0

Dominic Springer
dominic.springer@fau.de

Christian Herglotz
dominic.springer@fau.de

Andre Kaup
andre.kaup@fau.de

September 28, 2016

Abstract

The development of new HEVC/H.265 extensions remains a challenging task. New ideas are typically checked by time-consuming inspection and modification of the C++ HM source code. Also, encoder RDO decisions with their associated bits and distortions are hard to monitor as a whole due to recursion. Here, we present HARP v2.0, an open source BSD-licensed toolbox, which allows the inspection of CTUs, CUs, PUs, TUs, Pred/Resi/Reco YUVs and even RDO decision trees in native Python statements with unprecedented detail. This is essentially helpful for own prototype algorithms in the scope of the RDO, like higher-order motion search, interpolation techniques or new intra prediction schemes. HARP is freely available under www.lms.lnt.de/HARP.

1 Introduction And Motivation

The choice of C++ for the HM implementation can be seen as an optimal tradeoff between speed and code maintainability. In fact, a native Python or Matlab implementation would bring excessive encoding times, actually hindering scientific progress. Nevertheless, the choice against an interpreter-based language makes direct access to HM information difficult. HEVC analyzers like [1, 2, 3, 4] mitigate this problem on decoder side, but cannot export the huge CTU/CU/PU/TU tree structures. The encoder side is not supported at all, even though its RDO process is a major research field. Fig. 1 gives an overview over the basic HM encoder architecture:

- The Coding Unit (CU) is an elementary information structure and the backbone of the recursive HM implementation.
- CUs are processed in loops during rate-distortion optimization (RDO), during a first coeff/sideinfo encoding (POST), and during bitstream creation (FINAL).

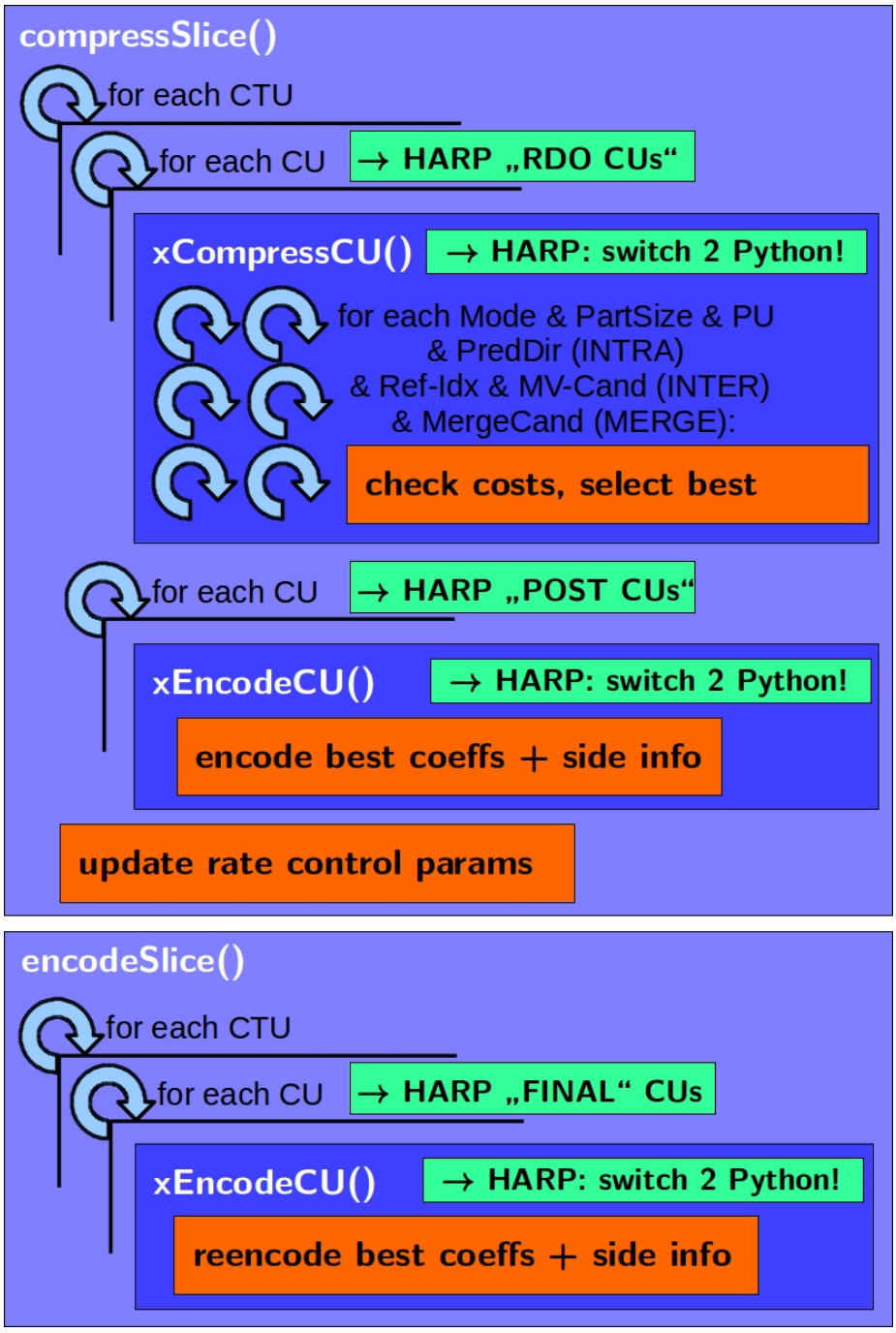

Figure 1: Outline of basic HM encoder architecture. HARP natively attaches to the HM encoder and provides detailed insight (green).

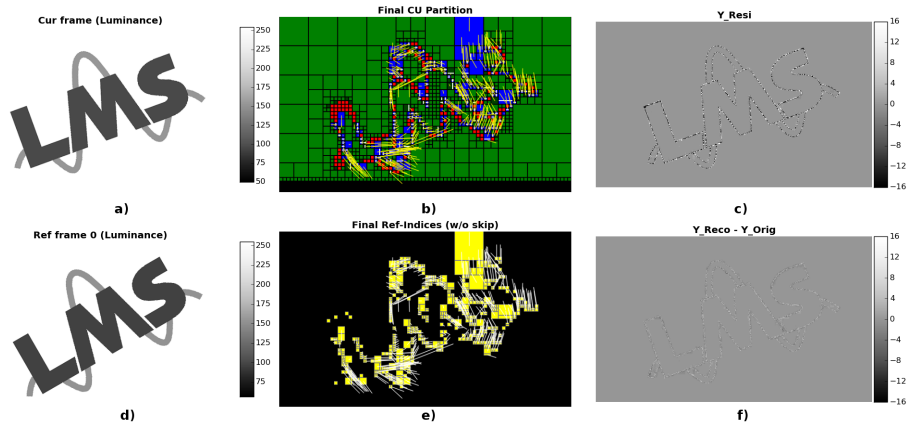

Figure 2: Example HEVC/H.265 encoding of a rotating frame (a) with LowDelayP and QP=20. All information was exported by HARP during runtime, creating a POC object, which can then be imported, analyzed and visualized in Python (b-f). Same is true for all following figures.

- The encoding of POST CUs is not identical to FINAL CUs, since rate control might further modify the FINAL CUs.
- The RDO- and POST-CUs are processed in the compressSlice() function, whereas the FINAL-CUs are processed in a successive call to encodeSlice().

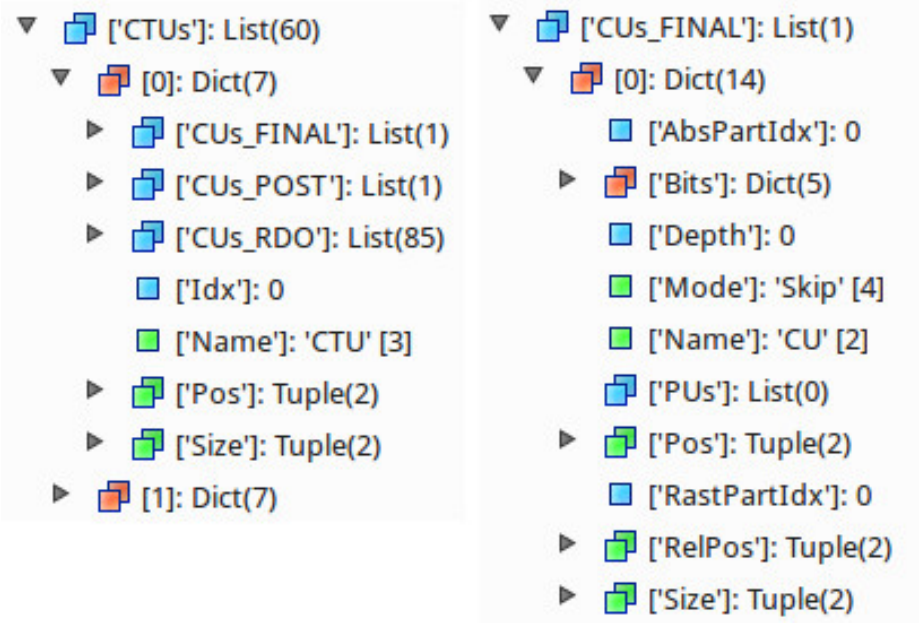

Figure 3: The first CTU in the CTUs list (left) and the first CU in the CUs-FINAL list (right). YUV buffers excluded for clarity.

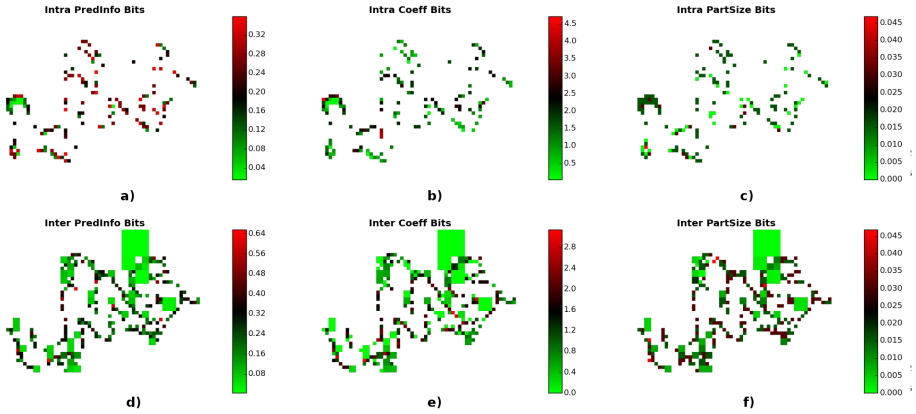

Figure 4: Analysis of HEVC/H.265 coefficient / side information bits for the rotating frame depicted in Fig. 2. Here, only POST-CUs were used for visualization. Top row shows intra modes, bottom row inter modes. Note the high number of bits required for transmitting the coefficients (center).

A simple quadtree-preserving logging of the above information would offer valuable insights for any researcher. However, this does not yet exist, because:

- All HM objects can only be accessed by complicated native C++ API calls and are hard to serialize as native trees.
- Much context and hierarchical information is discarded over CTU / CU / PU / TU / ref frame / neighborhood boundaries, just to keep HM complexity moderate.
- As a workaround, hierarchical information is often flattened to text files. YUV buffers at any time instance need to be exported and correlated with these text exports.
- This flattening severly impedes insight into the RDO, due to its countless recursive loops over depths, partitions, modes, predictor candidates, QP values, ref frames etc.

Version 1.0 of HARP [5] focused on detailed representation of FINAL CUs. It did not provide POST CUs, its RDO CU trees were simplified. Since we have extended HARP for fine-detailed RDO analysis and also received requests for guided integration of own HEVC/H.265 extensions with HARP (like new bitstream flags, new inter modes, new interpolation techniques etc.), we decided to put HARP v2.0 under open-source BSD license. Next to fine-detailed RDO logging, HARP v2.0 provides a new export mechanism for "POST CUs" (compare Fig. 1), with all associated YUV buffers, coefficients, distortion measures, side information, associated bit consumption etc. POST CU information is highly interesting when trying to correlate a final CU decision (as observable in the bitstream) with the vast field of conducted RDO checks. This is not trivial, since final CUs cannot be directly mapped to the RDO process and recursive

decomposition makes logging and correlation of associated checks difficult. Fig. 1 annotates HARPs new or extended capabilities in green color.

For HARP, we carefully focused on simple accessibility of the RDO, POST, and FINAL data structures. We hold this to be a gate opener for playing with new ideas and designing new HEVC/H.265 extensions directly in Python. Development can take place within a running encoder or offline. In the following, we will demonstrate examples and exemplary code snippets to demonstrate the simplicity of the approach. In fact, all shown figures can be directly reproduced by running "create_ICIP2016_Plots.py" from the HARP directory.

2 Inspecting Final Coding Units

The following examples are based on a YUV420 sample encoding, with the two first frames depicted in Fig. 2a+2d (note the rotation). The encoding (LowDelayP, QP=20) consumed just 12% more time than an unmodified encoder, including the creation of Python objects. With this at hand, we can simply access all POC, CTU, CU, PU and TU information. To iterate through them, simple for-loops are sufficient:

```
Partitions = getCanvas(POC)
  for CTU in POC["CTUs"]:
    for CU in CTU["CUs_FINAL"]:
      draw_CU(CU, Partitions)
```

The result is depicted in Fig. 2b (green=skip, red=intra, blue=inter). Fig. 3 gives an example of the structure of the CTU and CU objects. A more elaborate inspection of ref indices for inter PUs can by done by (see Fig. 2e):

```
RefIndices = getCanvas(POC)
for CTU in POC["CTUs"]:
  for CU in CTU["CUs_FINAL"]:
    for PU in CU["PUs"]:
      if PU["Mode"] != "Intra":
        RefIdx = PU["Motion"]["RefIdx"]
        draw_PU(PU, RefIndices, ColorMap1[RefIdx])
```

Note the simple iteration over CTUs, CUs and PUs in the POC. This simplicity extends to RDO trees likewise. Accessing YUV buffers (see Fig. 2c/2d/2f) is as simple as:

```
Y_Cur  = POC["YuvOrg"]["Y"]
Y_Ref0 = POC["List0_Refs"][0]["YuvOrg"]["Y"]
Y_Resi = get_FINAL_Buffer(POC, "Y_Resi")
```

3 Inspecting POST Coding Units

POST-CUs almost resemble the FINAL-CUs, with the subtle but essential difference that they represent the original recommendation of the RDO process.

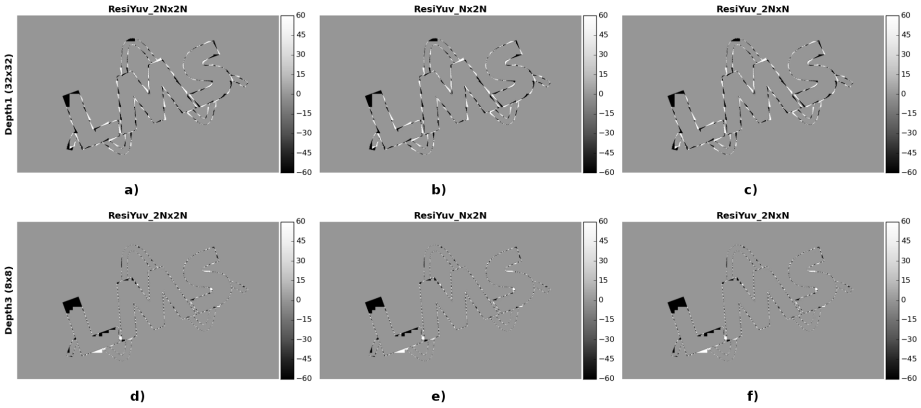

Figure 5: Analysis of residuals after HEVC/H.265 conducted 2Nx2N, Nx2N and 2NxN RDO inter checks for the rotating frame depicted in Fig. 2. For clarity, only depth 1 and 3 are shown, with the best MV and RefIdx found.

This recommendation may be changed by the rate control algorithm, targeting a specific rate / quality goal with QP decisions. In fact, FINAL-CUs are the outcome after rate control decision took place. Hence, they are typically not equal to the POST-CUs, even if they are submitted to the same xEncodeCU() function call (see Fig. 1). For prototype HM integrations with new modes, predictors, or interpolation techniques, it is typically necessary to check whether the final CU decisions visible in the bitstream were based on an educated RDO weighting of costs. For this, a correlation of POST CUs (not FINAL CUs) to the RDO CU checks is required. Fig. 4 gives an example for accessing coeff/sideinfo

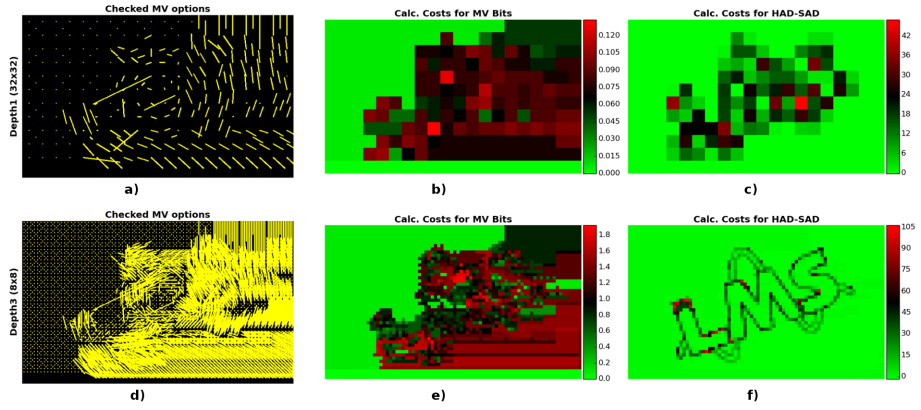

Figure 6: In-depth analysis of the RDO inter checks from Fig. 5, for depth 1 and 3. Left col: best MVs found among the available ref-frames. Center col: "cost" values estimated by HM for these MVs (costs values are basis for the final RDO decisions). Right col: achieved Hadamard-SAD values (encoder uses HAD-SAD metric for a quickly checking the achieved distortion in the fequency domain).

information in the POST CUs, which can be used for finding the corresponding RDO checks at certain depths:

```
Buffer = getCanvas(POC, np.float, NumCh=1)
Buffer = Buffer * np.NAN

for CTU in POC["CTUs"]:
  for CU in CTU["CUs_POST"]:
    if CU["Mode"] == Mode:
      Bits = CU["Bits"][Name]
      Idcs = get_Idcs_Y(CU)
      CU_Area = CU["Size"][0] * CU["Size"][1]
      Buffer[Idcs] = Bits / np.float(CU_Area)
```

4 Inspecting RDO Coding Units

A typically HM encoding of a small 640x360 inter frame leads to thousands of individual checks on modes, part-sizes, predictor candidates and many more. For HD and UHD sequences, the RDO decision tree reaches extensive sizes. The export of these trees with HARP is conducted using a fast C++ Python bridge, and can be controlled selectively. If the whole tree is exported (default), all checks from RDO are present and can be evaluated. For example, the residuals of all inter checks for a certain depth and PU partition size, with the best MV and RefIdx found, can be collected and inspected accordingly (see Fig. 5). Another interesting insight from the RDO tree might be the number of estimated bits for found motion vectors, as depicted in Fig. 6. The achieved distortion of the predicted luminance channel, as evident in the Hadamard-SAD (see 6c+f) is an interesting insight especially for improved estimation and interpolation techniques. Another example is the search for one specific "winning" PU during the checking of a specific depth at a specific partition size. It may be an inter or intra PU, and in case of inter, you may be interested in the specific reference index. See example below! Note the simplicity of harvesting data with simple Python "for" statements.

```
for CTU in POC["CTUs"]:
    for CU in subset(CTU["CUs_RDO"], "Depth", Depth):
        for CHECK in subset(CU["CHECKs"], "PartSize", PartSize):
            for PU in CHECK["PUs"]:

                # FIND THE "BEST" INTER-PU CHOICE
                BestIdx   = 0
                BestCosts = PU["CHECKs"][0]["Costs_Final"]
                for idx, PU_CHECK in enumerate(PU["CHECKs"]):
                    if PU_CHECK["Costs_Final"] < BestCosts:
                        BestCosts = PU_CHECK["Costs_Final"]
                        BestIdx = idx

                # WE FOUND THE "BEST" PU, GET ITS REFIDX FOR FUN!
                PU_BestCheck = PU["CHECKs"][BestIdx]
                RefIdx = PU_BestCheck["RefIdx"]
```

5 Conclusion

HARP v2.0 significantly extends the level of exported HM encoder details compared to its previous version [5]. It now provides access to the coeff/side information encoding step before rate control and offers an RDO tree of conducted checks with unprecedented detail. With this, the HM encoding process can easily be investigated in Python and extended with own prototype implementations. HARP is completely open source, and licensed under the industry-friendly BSD license. The authors know of no other software toolkit that provides such Python integration into the HM or detailed encoder-side RDO inspection. The exported structures can be used for seamless switching back and forth between C++ and Python, e.g. for passing of POC, CTU, CU, PU, TU, or YUV data to own Python prototypes during a live encoding.

References

[1] Codecian, "CodecVisa: HEVC and VP9 bitstream analyzer," www.codecian.com.

[2] Solveigmm, "Zond 265: HEVC bitstream analyzer," http://www.solveigmm.com/en/products/zond.

[3] Elecard, "Elecard HEVC bitstream analyzer," www.elecard.com.

[4] Sun Yat-sen University, "Gitl HEVC bitstream analyzer," www.github.com/lheric/GitlHEVCAnalyzer.

[5] D. Springer, W. Schnurrer, A. Weinlich, A. Heindel, J. Seiler, and A. Kaup, "Open Source HEVC Analyzer for Rapid Prototyping (HARP)," in *IEEE Int. Conf. on Image Processing (ICIP)*, Paris, France, October 2014.