University of Pennsylvania
Department of Electrical and System Engineering
System-on-a-Chip Architecture

ESE532, Spring 2017 — HW2: Profiling — Wednesday, January 18

Due: Friday, January 27, 5:00PM

In this assignment, we will profile an MPEG-2 encoder implementation on the ARM core of the ZedBoard. The encoder source can be downloaded from the Eniac server.

Collaboration

Work with assigned partners.[1] Writeups should be individual. Partners may share code and profiling results and discuss analysis, but each writeup should be prepared independently.

Encoder Organization

The MPEG-2 encoder is a stripped-down version of the one in the MediaBench II benchmark suite. Understanding video encoding should not be necessary to profile the code. Nevertheless, we will introduce the video encoder in some detail for convenience. An MPEG-2 encoder takes a video sequence that consists of frames (pictures). Each frame consists of blocks of 16×16 pixels, the macroblocks. A pixel is represented by 3 values. One value (Y) represents the intensity (gray-level), which is also called luminosity (luma). The other 2 values (U and V) represent the color. They are also referred to as chroma. There are 2 types of macroblocks, intra and inter blocks. For inter blocks, we use information from a previous frame to make a prediction of the macroblock. Intra blocks have no prediction. In addition, we use 2 types of frames: I-frames and P-frames. I-frames contain only intra blocks, and P-frames may contain inter blocks too. Figure 1 shows a block diagram of a typical encoder implementation, which consists of the following components:

Original picture buffer A memory buffer in which the raw frames (pictures) of the video sequence are stored. In our implementation, this is an memory area pointed to by the global variable `allframes`. We load the frames into this area from the SD-card on startup using the `read_all_frames()` function.

Bi-directional motion estimator The motion estimator searches for every macroblock the most similar 16×16 block in the previous frame. The block's location is expressed as the difference vector between the positions of the two blocks, called a motion vector. In our encoder, the motion estimator is implemented as `motion_estimation()`. Note that our motion estimator is actually not bi-directional.

[1] To be posted under Files in canvas.

MC predictor / interpolator The motion-compensated (MC) predictor / interpolator makes a prediction of the macroblocks based on the motion vector obtained by the motion estimator, and a reconstruction of the last encoded frame. The prediction is made by retrieving the 16×16 block of the reconstructed frame pointed to by the motion vector. This is implemented in `predict()`.

Subtractor If the 16×16 block found in the previous frame by the motion estimator were exactly the same as the macroblock to be encoded, we would only need the motion vector in the decoder to reconstruct it. Unfortunately, that is rarely the case, so in the Subtractor, we compute the difference between the prediction and the macroblock. Ideally, the difference is small, and we don't need many bits to encode the difference.

DCT The eye is more perceptive for lower frequencies than for higher frequencies. The (forward) Discrete Cosine Transform (DCT) converts each macroblock difference to the frequency domain such that we can take advantage of the eye's senstivity profile. The forward DCT has been implemented in `fdct()`.

Q The quantizer (Q) discretizes the freqencies from the DCT. The quantization step size of a frequency component depends on the sensitivity of the eye for that frequency. Larger step sizes result in fewer used quantization levels. Each level results in a different code word, so larger quantization steps yield less code. In this step, we may lose information, which results in a lower picture quality. However, we choose the information that is lost in an intelligent way, such that the perceived quality is impaired as little as possible. Q is implemented as `quant_intra()` and `quant_non_intra()`.

VLC The variable-length coder (VLC) stores all information necessary to reconstruct the video sequence. This includes, among others, frame width and length, DCT frequency components, and motion vectors. In our encoder, functions that output (a part of) the encoding start with `put`, e.g., `putbits()` stores one value, and `putpict()` stores the encoding of a frame.

IDCT and IQ The inverse quantizer (IQ) and inverse DCT (IDCT) perform the inverse of the Q and DCT components in order to reconstruct the difference between the macroblock and the prediction. IQ is implemented as `iquant_intra()` and `iquant_non_intra()`. The IDCT is in `itransform()`.

Adder The adder makes the final reconstruction of the macroblock by adding the difference from the IDCT to the prediction from the motion compensation.

Buff This is the output buffer, in which the encoded frames are stored. In our implementation, this is `output_buf`. The encoded frames are written back to disk using `write_all_output()`.

The encoder has a few configuration parameters, which are stored in a file called `Config.par` at the root of the SD-card. It includes also a format specifier that selects the raw input files, and the output file.

Importing the Encoder

To import the video encoder into SDSoC, we have to follow these steps:

1. Launch SDSoC, and open the workspace that you used before. You can also create a new workspace.

2. Create a new *Board Support Package*. SDSoC will ask you whether you would like to create a *Hardware Platform Specification* as well, which you should confirm. Enter a name for the specification in the *New Hardware Project* dialog that pops up, point *Target Hardware Specification* to `platforms/zed/hardware/prebuilt/export/zed.hdf` in your Xilinx SDSoC root directory, and dismiss the dialog to complete the hardware platform specification.

3. In the *New Board Support Package Project* dialog, you should enter a name for the board support package (BSP). Make sure that the *OS* is set to *standalone*, and press *Finish*.

4. Check the box in front of `xilffs` in the *Board Support Package Settings* dialog, and dismiss it.

5. Change the value of the define `_USE_STRFUNC` in the header file `libsrc/xilffs_v3_3/src/include/ffconf.h` in the BSP to 1.

6. Build the BSP project.

7. Create an empty SDSoC project.

8. Import the code into SDSoC into the `src` subdirectory of your project. You can find how to import code in Eclipse manual. The easiest way is to select *Archive File* as import source in the import dialog.

9. Inform the compiler about the location of the header files and libraries in the BSP. You can do this in the *Paths and Symbols* section of the *Project Properties* dialog. Add the `ps7_cortexa9_0/include` subdirectory of the BSP as include directory, and add `/standalone_bsp_0/ps7_cortexa9_0/lib` as library path. Let SDSoC rebuild the index again when it asks you.

10. Download the video and configuration files from Eniac. Extract the files to the root of an SD-card. Place the SD-card in the card reader of the ZedBoard.

11. You can now build and run the code on the target as before.

Homework Submission

1. **Learning and navigating the tools.** Document all the problems you encountered in working the assignment and how you solved them. For each, include:
   (a) Step (if applicable)
   (b) OS you were running
   (c) Error message or unexpected behavior you encounted
   (d) How the problem was resolved
   (e) How you found the solution (e.g., how did you experiment or reason through an answer, which document had the answer, where on the web did you find an answer, which student or TA was able to point out the answer)
2. **Firmware and OS Infrastructure** What is `xilffs`, and why do you need to incorporate it into the build?
3. **Measure**
   (a) Report the throughput achieved (frames/second) compressing the provided video sample using the default single ARM Core mapping. For this, you will need to instrument the code. Do not include the time spent on loading / storing the configuration, quantization matrices, and video frames from / to disk in your estimate. You can find out how to do this in the SDSoC Environment User Guide. You can find the clock frequency in the `project.sdsoc` file.
   (b) Create an execution profile of the encoder using TCF profiler. Add the results for the first 10 bins to your report. You are allowed to make a screenshot of the TCF Profiler view. Profiling is described in the same section of the user guide as instrumentation.
4. **Analyze**
   (a) Using the TCF profiler output, which function consumes most execution time?
   (b) Use Amdahl's Law to determine the highest overall encoder performance improvement that you may obtain by optimizing the function that you found in the previous question.
   (c) Assuming you bring the input data in from one of the following sources, what frame rate could you possibly achieve (i.e., ignore the compute requirements for this, focus on I/O)? Will input from the source be a bottleneck compared to the computation above?
      i. SDcard
      ii. 512-MB DDR3 SDRAM
      iii. Gigabit Ethernet interface
      iv. USB

(d) Per input pixel, how much data is read by each of the following routines (on average, in total over an entire frame):

   i. `dist1` without interpolation and assuming `distlim`$= \infty$.

   ii. `fdct`

   iii. `putAC`

(e) Considering the first loop body of `dist1` (`if (!hx && !hy)`)

   i. estimate the total number of compute operations in one loop iteration of this loop body.

   ii. determine the critical path length for the same loop body. Exploit the associativity of addition.

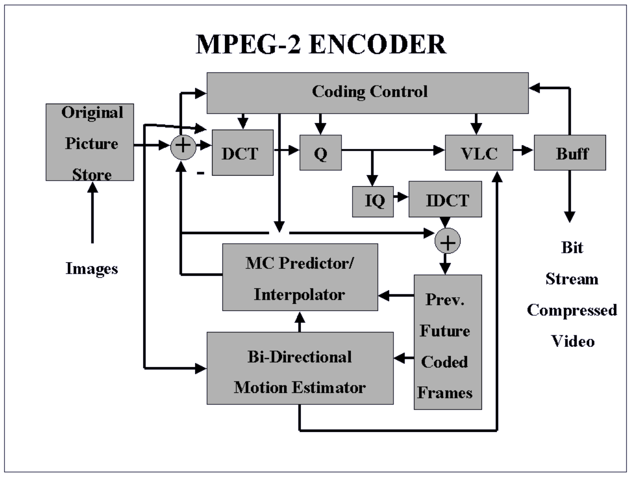
MPEG-2 ENCODER
Coding Control
Original Picture Store
DCT
Q
VLC
Buff
IQ
IDCT
-
Images
MC Predictor/ Interpolator
Prev. Future Coded Frames
Bi-Directional Motion Estimator
Bit Stream Compressed Video

Figure 1: MPEG-2 encoder block diagram