Coralie Saysset

H.264 Decoding Performance on a DSP Architecture

Turku Centre *for* Computer Science

TUCS Technical Report
No 1089, September 2013

H.264 Decoding Performance on a DSP Architecture

Coralie Saysset
Turku Centre for Computer Science, Embedded Systems Laboratory
Joukahaisenkatu 3-5 B, FIN-20520 Turku, Finland
coralie.saysset@abo.fi

TUCS Technical Report
No 1089, September 2013

Abstract

This technical report evaluates a H264 reference decoder performance. Our studies are focused on DSP profiling results, but PandaBoard and Raspberry Pi architectures are also used as reference platforms. The DSP is 120% slower than the Raspberry Pi and 640% slower than the PandaBoard. However, the C6000 DSP presents some interesting behaviors. The last part of the decoder, the loop filter, is more efficient on the DSP. The TI platform needs half of the time (in percentage of total decoding time) spent by the other devices in the same function, and the real time spent in this function is 80% of the time spent on the Raspberry Pi.

Keywords: H.264, decoder, DSP, profiling, MPEG4, Raspberry Pi, PandaBoard

TUCS Laboratory
Embedded Systems Laboratory

Contents

1 Introduction

1.1 H.264 and DSP

Video teleconferencing, High definition television, streaming Internet video content, everyday we receive and exchange more and more data. Video manipulation becomes each day a more challenging process. This problem is complex : first because of the diversity of the video content available but also because of the different capacities of all the devices requiring these videos.

H.264 is a widely used video standard compression. Also called Advanced Video Coding (AVC) or MPEG-4, H.264 offers significantly better compression than previous standards developed. The main goal of this report is to evaluate different algorithms used in H.264 video processing applications and assess. The researches were restricted to the decoder part of the process. We will pay a specific attention to determinate the efficiency of a Texas Instruments TMS320C6713 of Digital Signal Processors (DSP). A normal x86 personal computer will be considered as a primary reference platform for the DSP, then ARM architectures with a Raspberry Pi and a PandaBoard will be used. More than presenting different profiling results, our objective is to highlight parts of the decoder interesting to decode on a DSP.

1.2 Report structure

For a better understanding of the topic, the next chapter is a presentation of the H.264 video compression technology. In this second chapter the different steps to encode and decode a video with H.264 codec are detailed and explained.

The following chapter is more practical and focused on a reference software implementation of the standard H.264. During the experiments we met different problems in the implementation, this chapter presents these undefined behaviors or bugs and explains how we have solved them.

The fourth chapter presents the profiling concept and introduces a profiling tool : gprof: In this chapter we discuss its advantages and limits.

Before working on any other platform, we try to run the code with a desktop computer, because the environment is mastered. We decode first videos and get first profiling results easily. Having an OS, we have several tools to get time measurement. This chapter concludes that x86 architecture is too fast to get accurate profiling results, more over the total decoding time for a given video is more than 50 time slower on a DSP. After this fifth chapter, a desktop computer will never be considered as a good reference platform for the DSP.

The sixth chapter introduces a new reference board: a PandaBoard. After a quick presentation of its architecture, the decoder's performance on this platform is analyzed. The call tree is built from the profiling and commented.

The seventh chapter of this report is one of the most important. It presents

the experiments done on the DSP. YAs the decoder can not be run on the DSP directly, this chapter starts by presenting needed adjustments (pseudo file system, time measurement). The first decoding times were very slow, but the DSP could be very faster with the proper options and initialization. Moving the stack in the internal memory improved a lot the decoding time, this point made us think that the slow results could come from memory problem. We studied deeper the memory architecture of the platform, and this knowledge is summarized in this chapter. Thanks to this research, we learn that L2 cache is disable by default, enabling it gives better results, closer to the PandaBoard performance. The chapter ends with an exhaustive list of building options, with the effect of each on the decoding time.

The next chapter is focused on a second reference platform. A Raspberry Pi is a good second reference. This board is situated between the performance of the C6713 (the slowest device) and the PandaBoard (the fastest). Raspberry Pi architecture is presented and the pipeline system which is a bit different from the DSP system. Having an OS on the board could be both an advantage and a disadvantage and we will explain why. Consequently after experiments on the Raspberry Pi we try to run the program on the board without any OS. Because of a lack of time, we don't go further than running a software on the Raspberry baremetal. Section 9, resumes our experiments on this device.

The next section is a comparison between all the profiling results, highlighting things in common but also their difference. We try to identify which part of the decoder is the most time consuming when run on a DSP. We conclude this chapter by the theoretical interest of running the deblock filter on the TMS320C6713. The following section is a more detailed description of the Deblocking filter process.

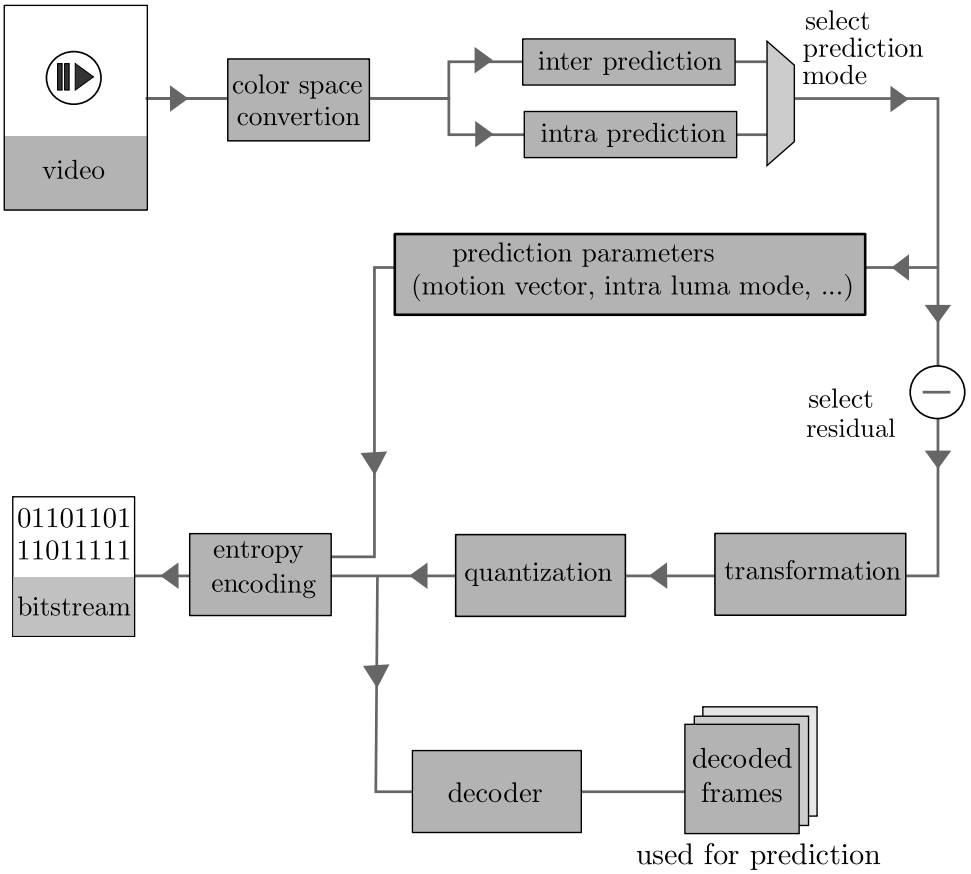
2 A theoretical approach of H.264 decoding

2.1 Introduction of decoding concept

This chapter contains a generic description of H.264s basic concepts. It tries to describe the algorithms at an intermediate level, more detailed than a pure H.264 feature description, but not as complete as the official standard description. As much as possible, when an algorithm is presented, we try to cite the associated file in JM decoder where it is implemented. This chapter is not a replacement for a full-fledged specification as detailed information such as precise syntax specifications are missing[1]. This section has the objective to be a good introduction to the concept for beginners.

The H.264 standard defines a lot of features. The encoder can use different algorithms resulting in better compression, faster decoding time or better visual quality. Devices are limited in computation power and memory and enabling a feature on the decoder has effect on the code complexity. The H.264 standard

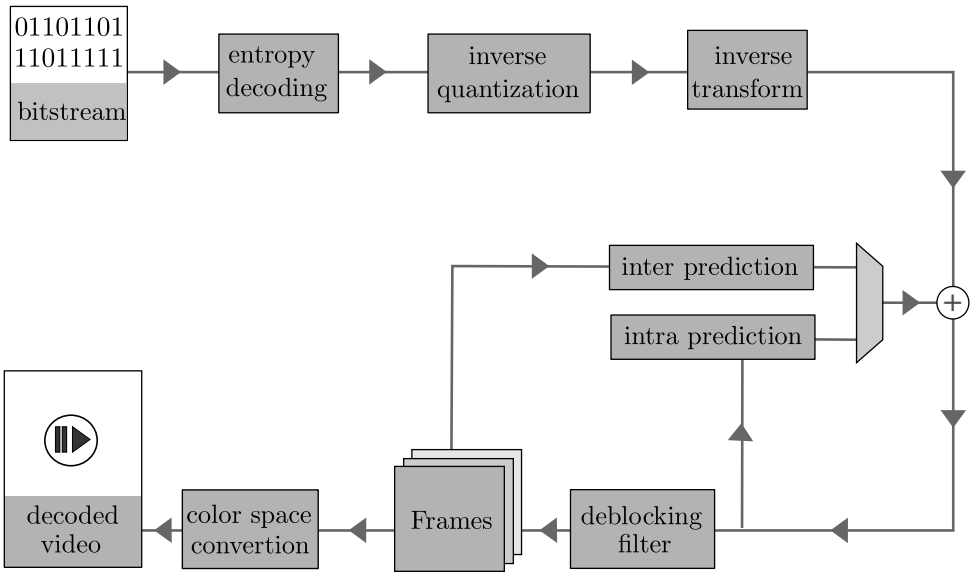
Figure 1: H.264 Encoding process:



defines different profiles. Features are enable or disable according to the profile limitation. A device, has specific capabilities and a maximum supported profile. The device is able to decode a video encoded with all lower profiles. The max profile supported depends on the memory and the max resolution available. 21 profiles are officially defined, but the most known and used are (from the most basic to the most complex): Baseline, Extended, Main, High. Baseline target device with low processing power such as smartphones, whereas High profile is used for high-definition television applications and Blue Ray disc storage. The profile only indirectly influences the quality. Some features of higher profiles allow more advanced compression techniques to create a video file that has the same quality as the Baseline profile, but with smaller size. The following presentation of the decoding process present in detail features supported by different profiles.

The traditional definition of decoding is: “reverse of encoding”. This is why understanding the encoding objectives and different steps of process is essential before studying the decoding part. Encoding is the action to transform informa-

Figure 2: H.264 Decoding process:



tion from one format into an other one, and decoding try to rebuild the original information from the transformed one. On the following section the encoding process is detailed chronologically, and for each step of the process, the inverse (the decoding process associated) is explained.

Figures 1 and 2 are two block diagrams presenting the encoding and decoding processes. On the encoder, you have first the prediction, then the transformation, then the quantization and finally the entropy coding. The decoder does exactly the inverse : entropy decoding, inverse quantization, inverse transformation, prediction, reconstruction and a last step is deblock picture. All these step are explained in detail on the following subsection.

2.2 Color consideration

Color are usually represented with RGB scheme. Each pixel of the screen is represented by three numbers indicating the red, blue and green proportions. All colors can be decomposed with these three components. Human vision is more sensitive to luminance (brightness) than colors themselves, video encoding uses this fact to obtain a most efficient representation of color. Y is the luminance and can be expressed like a weighted average of Red Blue and Green :

$$Y = k_r R + k_g G + k_b B$$

We could also write the color difference or chrominance component:

$$B - Y = C_b$$

$$G - Y = C_g$$

$$R - Y = C_r$$

However the chrominance component sum is constant. It means that with only two chrominance components, the third can be found from the others. H.264 uses a Y:Cr:Cb representation for the color space. Both RGB and YCrCb use three components, but the advantage of the YCrCb is that the chrominance components can be represented in a lower resolution without having an obvious effect on visual quality.

Figure 3: Chrominance subsampling patterns:

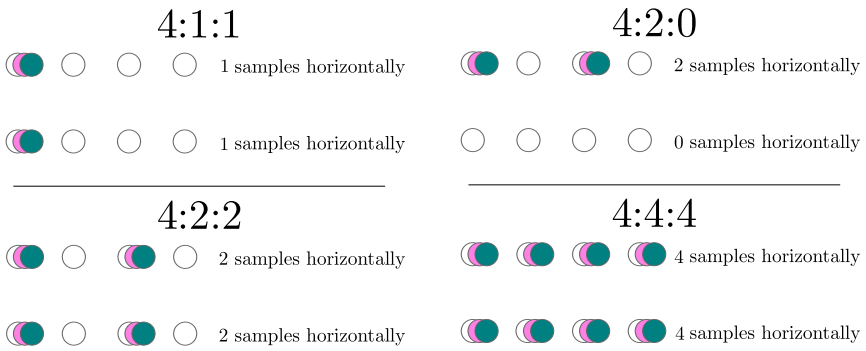


Figure 3 shows 4 popular patterns for "sub-sampling". 4:4:4 indicate the rate of each component in horizontal direction, it means for every 4 luminance sample, you have 4 Cr and Cb on the first horizontal line, and 4 on the second. This format preserves chrominance component. The most popular sampling pattern in video compression is 4:2:0. A video using this chroma sample requires half as many sample as 4:4:4 video and the drop of quality is almost imperceptible. Most of H.264 standard profiles use 4:2:0 but few higher profiles are able to work with 4:2:2 and 4:4:4 videos. 4:0:0 is a monochrome format, allowed from the High profile.

2.3 Picture, Frame or Slice ?

A coded video is a sequence of several coded picture. Often the term *picture* is a too general notion as it could refer to one frame or one field. Usually a frame is composed by two different slices, one on the odd rows, the second on the even rows. If the two fields of a same frames capture two different time instant, the frame is an interlaced frame, otherwise it is an progressive frame.

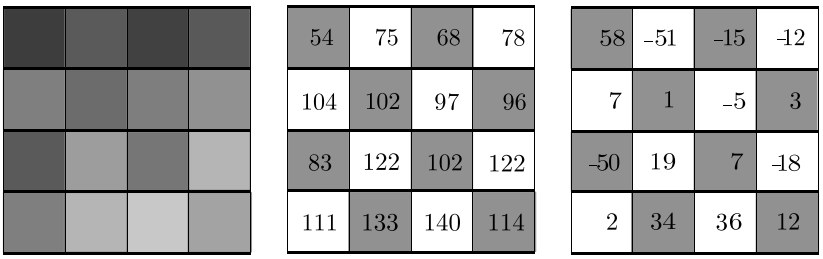
A frame is divided in fixed size macroblock (MB). A 16x16 square of pixel in a frame is represented by a 16x16 sample of the luma component (luma MB),

and two 8x8 sample of chroma (Cr MB and Cb MB). This partition is designed to respect the 4:2:0 sampling pattern (cf subsection on color consideration).

A picture is composed of one single slice or more. A slice is a sequence of macroblock and can be decoded independently of other slices. A slice could contained the first marcoblocks in the order of a raster scan, but Flexible Macrocblock Ordering could also create other region partition. The FMO avoids to spread out errors and limits the error to one slice. For example, on video conferencing application an useful slice division of a frame is two slice creating a checkerboard pattern. The two slice are sent in different packets and if one of the slice is lost during transmission, the frame could still be reconstructed. Only the Baseline and Extended profiles allow the FMO utilization.

2.4 Transformation and inverse transform

Figure 4: Original luma picture-Coefficients- Transformed coefficients:



A traditional image is difficult to compress because neighboring pixels are highly correlated and energy is distributed across the image. The transformation objective is to compact the energy into small significant numbers and decrease the data correlation. The whole process is reversible and lossless. H.264 works with square of 16*16 pixels called MacroBlock .

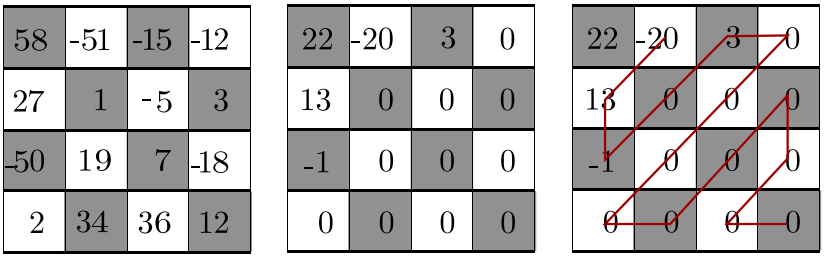
Other video standards, like MPEG-2 use 8x8 Discrete Cosinus Transformation (short in DCT). H.264 uses 4x4 integer transformations witch are originated from DCT but are lower complexity with little degradation on performance. An other advantage of these transformations is they use only integer arithmetic, this avoid mismatch problems during inverse transformation. Figure 4 presents a block of 4x4 pixel, first with the real pixel color, then with the associated coefficient and finally after the transformation.

The High Profile (and above) could also work with 8x8 blocks of luminance. This profile uses adaptive transform that decides on the fly if 4x4 or 8x8-pixel blocks should be used. Typically, 4x4 blocks are used for the parts of the picture that are dense with details, while parts that have little details are transformed using 8x8 blocks. Chroma blocks are all represented with 4x4 blocks.

The functions needed for the transformation are implemented in ldecod/src/-transform8x8.c and lcommon/src/tranform.c files of the JM software.

2.5 Quantization and inverse

Figure 5: Transformed coefficients Quantized coefficients- “Zigzag” pattern:



The objective of encoding is to have an image of quality with few coefficients. The transformation is not really a compression, it just represents the data in an other domain in witch the compression will be more efficient. The quantization is this compressing process. The coefficient matrix is divided by a quantization matrix and the result is rescaled. Rescaled, the important component can not be re-found, so quantization is lossy. After this step, main values are in the top left corner of the matrix and other values are zero. For a better compression, it is interesting to reorder this matrix in an array where values are ordered as much as possible. The interest of having these values ordered comes from the entropy encoding process, and will be detailed in section 2.9. Figure 5 is an illustration of the quantization process on a 4x4 block.

The standard allows the encoder to use custom quantization matrix, in this case the encoder has to explicit the matrix used to the decoder, but the standard also defines four default quantization matrix: inter 4x4, inter 8x8, intra 4x4 and intra 8x8. This matrix are stored in the quant.c file of the ldecod folder of the JM implementation . Inter and Intra refer to intra and inter prediction, process presented in the following subsection.

2.6 Prediction and compensation

In a frame, a pixel is often correlated to his neighbors (spacial correlation) and to the same pixel in previous and following frames (temporal correlation). It is possible to take advantage to this correlation for a better compression. The encoder will create a prediction of the current block and compare the prediction with the regular block, the difference is the block to encode. If the prediction is good, the residual block contains very few data, these data will be encoded (transformation

+ quantization) in a very small size. On the decoder side, the received block is decoded and added to the local prediction of the block. Encoder and decoder have the same predictor.

2.6.1 Intra prediction

Intra prediction uses spacial correlation to create his predictor. A block of data will depend on adjacent blocks. There is one prediction for the luma and two for the chroma. There are different modes of prediction, chroma can use 4 modes of 8 x 8 blocks prediction to be represented. An 8 x 8 block is a quarter of a full marco block. The luma can be represented with 4 different modes of intra prediction on a macroblock , for part of the picture with few details. If the MB is dense in detail it could be broken in 4 x 4 block and predicted with 9 modes of prediction. Figure 6 presents some intra prediction methods on 4x4 block.

Intra Chroma Prediction'modes for 8 x 8 blocks are :

- Mode 0 : vertical prediction
- Mode 1 : horizontal prediction
- Mode 2 : DC prediction (mean value)
- Mode 4: Plane prediction

Intra Luma prediction uses the same modes but prediction is done on 16 x 16 block (one macro block).

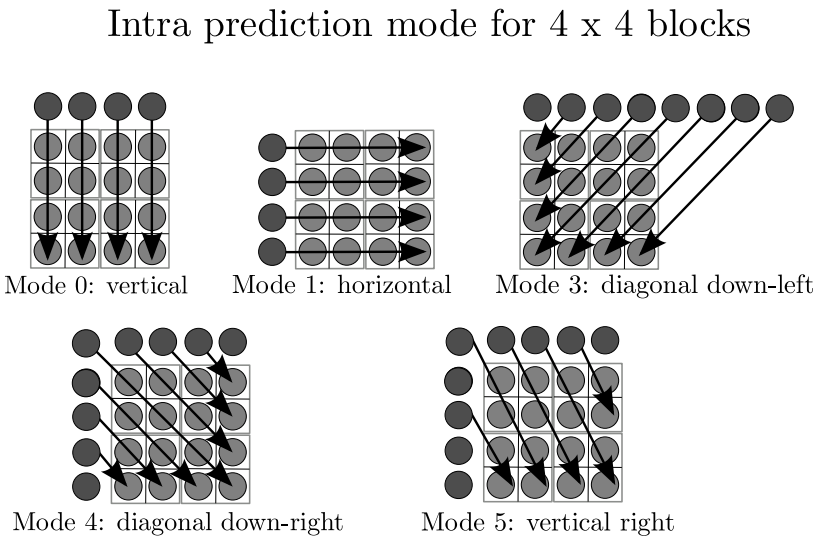


Figure 6: Illustration of luma prediction:

Intra Luma Prediction's modes for 4 x 4 blocks are :

- Mode 0 : vertical prediction
- Mode 1 : horizontal prediction
- Mode 2 : DC prediction (mean value)
- Mode 3 : Diagonal down left prediction
- Mode 4 : Diagonal down right prediction
- Mode 5 : Vertical right prediction
- Mode 6 : Horizontal down prediction
- Mode 7 : Vertical left prediction
- Mode 9 : Horizontal up prediction

2.6.2 Inter prediction

The fact of finding matching pixel in inter frame coding is called inter prediction ,motion estimation or motion compensated prediction. The JM decoder uses both intra prediction and MC prediction to refer to this process. Two following frames have a great chance to be similar, so instead of recoding the block of pixels, it could be copied from a previous (or a following) similar block in an other frame .

On the encoded frame, for each macroblock partition (from 16 x 16 to 4 x 4), an algorithm researches the best match in different reference frames (previous or future frames). This is why the inter prediction is computationally expensive on the encoder. B frames that allow both reference to previous frames and following frames, increase again this complexity, this is why they are only used in Main profile and above. Different methods exist on the encoder part to find the good reference pixel: diamond search,hexagon search, Simplified Uneven Multihexagon search (UMH), etc. But we are interested in the decoding part, so we only have to know that a reference block of pixel is found.

With the best match found, the motion vectors are created, then the encoder try to reconstruct the current frame with only the previous frames and the motion vectors. The reconstructed frame is different from the current one, the residual difference is the frame we have to encode and send to the decoder. The decoder will receive the motion vectors, make a prediction and add the decoded residual difference frame to the result.

Adjacent blocks tend to move in the same directions. This is why the motion vectors are also encoded using prediction. When the motion vectors are found, this same motion vectors are calculated with previous motion vectors, and the encoder send the difference between the real motion vector and the predicted one. mv_prediction.c is the dedicated file for the motion vector estimation.

2.6.3 Skip/Direct mode

Often used with B frame, in this mode the encoder doesn't send a residual error or a motion vector to correct the future prediction of the decoder. The encoder only send the information : this macroblock is in skip mode. They are two ways

to construct a motion vector. The first mode is the temporal mode, it uses the motion vector of the block of a previous frame and one from a following frame. The spatial mode predicts the motion vector using the motion vectors neighboring blocks. This mode used only few bit to encode a macroblock, but it can not be used in every situation.

2.6.4 Mode Decision

Each prediction mode (inter or intra) has a Rate Distortion cost. The mode decision on the encoder selects the best mode of prediction, the mode with as little rate distortion code as possible. This process is also computationally expensive.

2.7 Adaptive Frame/Field coding operation

This feature concerns only video with interlaced frames. When an object moves on a interlaced frames, the two fields representing 2 different moments in the time are more different , so more difficult to compress. Statically it reduces spacial correlation,and leads to a less effective intra prediction. The H.264 proposes 3 solutions to this problem:

- Two fields are not that different, and continue the encoding process like usual. The two fields are coded together as one single frame. It is the frame mode.
- Consider that two fields are too different to be coded together, and coding them separately. It is the Field mode.
- Compress and code the field together as one single frame but on the macroblocks level the encoder can decide to de interlace the two vertically adjacent macroblock, the upper MB coding the first field, the lower coding the second field.

The choice between these 3 option is done for each frame, and each MB of a frame in a video. Deciding if a picture should be consider as a frame or two separate fields is named the picture-adaptive frame/field or PAFF. This choice on the macroblock level is referred to as macroblock-adaptive frame/field or MBAFF

2.8 Deblocking Picture

As its name suggests, the Deblocking filter tries to improve visual quality and prediction performance by smoothing the sharp edges which can form between macroblocks because of the used block coding techniques.

H.264 deblocking filter algorithm performs one dimensional filtering in horizontal direction, then one in vertical direction. The filtering could be strong

(strengh 4), standard (strength 1, 2, 3) or not performed (strength 0). The deblocking filter can be configured to be switched off or the strength could be fixed at a certain ranged. Otherwise the strength is determined by coding modes of adjacent blocks (intra, inter prediction), quantization step size, and the steepness of the luminance gradient between blocks. Files associated are loopFilter.c, loop_filter_mbaff.c and loop_filter_normal.c, all situated in the ldecod folder as the deblocking filter is not the inverse of a transformation in the encoder, but a specific step to the decoder.

2.9 Entropy encoding and decoding

Entropy encoding is a lossless data compression scheme. Huffman coding and arithmetic coding are the most known techniques. H.264 uses 3 different entropy coding algorithms.

Context Adaptive Binary Arithmetic Coding (CABAC) is based on arithmetic coding. It provides very good results in data compression compared to other entropy coding algorithms used in video encoding. On the other side the decoding process needs also much more processing resources. This is why CABAC is not used for every H.264 profiles, but only used from Main to the highest profiles. The CABAC decoding is a linear process very difficult to parallelize.
Context Adaptive Variable-Length Coding (CAVLC) is less effective than CABAC in compression, but is still a high quality solution. A stream encoded with CALVC requires less processing to be decoded than one encoded with CABAC.
CABAC and CAVLC take advantage of recurring data to reduce the bit rate. If a sequence is statically more present its codeword's size will decrease. In video decoding they have to transform square blocks containing mostly zeros.

$$\begin{pmatrix} 14 & 1 & 0 \\ 2 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \underset{VLC}{\longrightarrow} 14, 1, 2, 6 \times 0 \underset{CABAC}{\longrightarrow} 1$$

H.264 uses a last entropy coding : Exponential-Golomb coding with k parameterized at zero. Although it is a variable length coding, codeword are logically defined as followed:

$$0 \to 0,\ 1 \to +1,\ 2 \to -1,\ 3 \to 2,\ ...$$

Exp-Golomb code is used to encode data that are not the frame itself like : image type, prediction modes, motion vector,...

3 H.264 reference software

For all the future experiments we have used the JVT JM H.264/AVC reference software [10].

We have used the last version, which is version 18.5. The objective was to have a representative reference software to run on several platforms and being able to compare their behavior on the same code.

3.1 An undefined behavior on different platforms

Executing the program on a standard desktop machine or under Linux on ARM (panda board and raspberry pi) worked perfectly, but we met problem with running the program on bare metal and on the DSP. It crashed on the ARM and the decoding was incorrect on the DSP.
After decoding a video, the program compares the decoded video with a reference and prints the SNR for luma and chroma. These values are supposed to be zero. Running JM 18.5 on the DSP results in a first frame perfectly decoded and several errors on the following frames, like shown on table 1.

Table 1: Standard output with the 3 frames video on a DSP with incorrect Snr values:

Frame	POC	Pic#	QP	SnrY	SnrU	SnrV	Y:U:V	Time(ms)
00000(IDR)	0	0	28	0.0000	0.0000	0.0000	4:2:0	3305
00002(P)	4	1	28	58.6381	63.2264	63.2709	4:2:0	2720
00001(b)	2	2	30	56.6368	60.9770	61.4360	4:2:0	2277

3.2 Casting byte pointers

This undefined behavior comes from loopfilter files :ldecod/src/loopFilter.c, /ldecod/src/loop_filter_mbaff.c and /ldecod/src/loop_filter_normal.c. The loop filter does a large number of casts from byte pointers to int and int64 pointers. It relies on ints containing four bytes in little-endian order, and unaligned reads and writes being possible. This is not the case on several platforms.

3.2.1 Comparing several bytes

In many places, the code casts a byte pointer to an int pointer in order to check four bytes in parallel, as in listing 1. This does not work if an int does not contain exactly four bytes. This seems to be the case at least on the TI C64+ DSP platform, where this will cause incorrect decoding. There are a few similar cases with casts to int64, to process eight bytes in parallel.

Listing 1: four byte comparison example in loop_filter_normal.c

```
// Previous code
```

```
// only if one of the first 4 Strength bytes is != 0
// if ((*((int *) Strength)))

//Solution proposed
if ( Strength[0] != 0 || Strength[1] != 0 ||
Strength[2] != 0 || Strength[3] != 0 )
```

3.2.2 Writing an array of byte

In a few places, the code casts a byte pointer to an int pointer in order to write four bytes at a time. Listing 2 presents a real example in the source code.This also does not work under the same circumstances as above.

Listing 2: writing four byte example in loop_filter_mbaff.c

```
//Previous code
// *(int*)(Strength+idx) = StrValue * 0x01010101;

//Solution proposed
Strength[idx] = StrValue;
Strength[idx + 1] = StrValue;
Strength[idx + 2] = StrValue;
Strength[idx + 3] = StrValue;
```

3.2.3 Single byte writing

And in a few other places, the parallel writing code mentioned above has apparently been incorrectly copied into places where only a single byte needs to be written. This causes the three following bytes to be zeroed, which will write outside the bounds of the buffer. It will also not work on a big-endian platform, where it will just fill the buffer with zeroes. Furthermore, it will do unaligned 32-bit writes, which ARM does not support, causing a crash. Listing 3 presents an example of an incorrect writing on one single byte.

Listing 3: casting pointers single byte writing example in loop_filter_mbaff.c

```
//Previous code
// *(int*)(Strength + idx) = StrValue;

//Solution proposed
Strength[idx] = StrValue;
```

This crash seems not to happen when running under Linux on ARM, but only when we tried running on the bare metal. It may be that Linux is catching the

exceptions caused by unaligned reads and emulating the reads in software. This would be very, very slow, however.

This bug has been reported on mantis bug tracker [4], with a diff file that removed all pointer casts. The patch we proposed was for the decoder only, but we supposed same work had to be done for the encoder.The report has been accepted, and this problem is now solved on the last implementation on both encoder and decoder.

4 GNU profiler

4.1 Why profiling

To estimate the performance of the DSP on a decoding work, it is important to have another reference platform. First, this reference platform could give us a comparison point for the DSP results, but there is another advantage. To profile a function on a DSP, you have to add a get time function handwritten before and after each call. We can't guess which functions are important or not. We need to find a device where it is easy to get an accurate profiling result. Once we know which functions are the most interesting, we can measure them on the DSP.

4.2 Profiling with gprof

The GNU profiler [2] is a tool designed to know where a program spent its time. It answers the questions: how many times is this function called? Which functions are its children, its parents? How much time did the program spent in each function? This tool is thought to help programmer to optimize their code spotting them slow candidate functions for rewriting. Sometimes it could be used during the debugging process, if functions are not called as many times as they are supposed to.

To use this profiler, you are supposed to add a *-pg* option to your compiling and linking options. The program built is slower but while running it generates a profile data file, named gmon.out. This binary file could be read with this command:
gprof options [executable-file [profile-data-files...]] [> outfile].

An interesting tool in addition of gprof is gprof2dot [5]. This is a Python script to convert the output from many profilers (not only gprof) into a dot graph.

4.3 Counting functions's call

The -pg option changes how every function is called, before calling your function, the profiler stashes away some information on this cal in a reported file. So

the number of call, and call tree (Which function calls which one) are totally accurate. However a high level of optimization during compilation could modify the call tree. Code and data that are accessed closely together in time should be placed close together in memory to increase spatial locality of reference. Often the compiler can replace a call to a function by a copy of the function body. That's why an important function could be hidden under another function name. So if the number of call written on the Flat profile is correct, the function could represent more (or less) that its name suggests.

4.4 Profiling by sampling

If the number of call is accurate, it is not the case for the time measurement. When the program built with -pg option runs, N time per second of run time the profiler looks at the program and saves information about the program state. The problem with this sampling process is the statistical inaccuracy of the result. A very short function could not be catch by sampling, or if catched but only once, the profiler could say it finds this function zero times or twice. The sampling period was 100 times per second of run time for each platform used in this report, but this number may vary from system to system.

To obtain an accurate result, the sampling period has to be considerably smaller than the run time. It is also possible to calculate an expected error int the profiling, like explained in [2].

> The actual amount of error is usually more than one sampling period. In fact, if a value is n times the sampling period, the expected error in it is the square-root of n sampling periods. If the sampling period is 0.01 seconds and foo's run-time is 1 second, the expected error in foo's run-time is 0.1 seconds. It is likely to vary this much on the average from one profiling run to the next. (Sometimes it will vary more.)

One way to get more accuracy is to add profile data from several runs.

4.5 Estimating children time

Another inaccuracy is reported in the gprof man [3]:

> We assume that the time for each execution of a function can be expressed by the total time for the function divided by the number of times the function is called. Thus the time propagated along the call graph arcs to the function's parents is directly proportional to the number of times that arc is traversed.

It means that if a function named A(int param) returns quickly with zero as argument, and this function is called as many time by B() and C(), and B function only

calls A(0) , but C does'nt , in this case the time spent in A() with B as parent is not the same that time spent in A with C as parent. And gprof doesn't know that and will just split fairly the total time spent in A between B and C. As a conclusion, gprof has the potential to be a powerful tool, but it has its limit. On our experiment we will pay a specific attention to these two inaccuracy point.

5 A first try for a reference platform

5.1 Input video stream

A foreman test video is used as input for our experiments. YUV 4:2:0 pixel format is used, and the resolution is 176 x 144 (qcif resolution). This video was encoded with the H.264/AVC reference software. The last version available at this moment was the JM18.5 and excepting the correction we have presented on the above section we used this software. The encoder was configured to encode the first 3 pictures of the video, then the first 50 frames, and finally the whole video (300 frames).

We used the encoder configuration file in the JM decoder corresponding to a High profile and a 4.0 level. In this configuration Slice mode is off (Not interlacing, the video is progressive), the stream is encoded using adaptive CABAC. 7 B frames are allowed, and their quantization parameter is 30. I frames and P frames are using a 28 parameter. Transformation on 8x8 block is allowed. The Motion Estimation uses the EPZS algorithm.

5.2 First experiments on a desktop personal computer

On the following section the machine used is a **Intel(R) Core(TM)2 Duo CPU E6750 @ 2.66GHz**.
Compiling and running the program was easy, and quickly we had the first run results. While decoding, the software prints in the console several information on the decoding process: time, signal to noise ratio, frame type (I Frame, B Frame ,...). Table 2 is an illustration of a software standard output.

Table 2: Standard output of the JM decoder :

Frame	POC	Pic#	QP	SnrY	SnrU	SnrV	Y:U:V	Time(ms)
00000(IDR)	0	0	28	0.0000	0.0000	0.0000	4:2:0	3
00008(P)	16	1	28	0.0000	0.0000	0.0000	4:2:0	2
00004(B)	8	2	30	0.0000	0.0000	0.0000	4:2:0	1
00002(B)	4	3	31	0.0000	0.0000	0.0000	4:2:0	1
00001(b)	2	4	32	0.0000	0.0000	0.0000	4:2:0	0
00003(b)	6	4	32	0.0000	0.0000	0.0000	4:2:0	0

The program gives itself a decoding time for each frame and their sum is used as total decoding time (Initialization of the decoder or its closing is not supposed to be counted in this time). Each run gives a sensible different total decoding time, but a 300 frame video decoding needs around 0,3 sec (results between 0,210 and 0,387). It implies about 1 ms for each frame, and often the decoding time for a frame written is "zero millisecond". Gprof gives a total decoding time between 0,31 and 0,45 seconds.

Table 3: gprof profiling, 300 frames video

% time	cumulative seconds	self seconds	calls	self µs/call	total µs/call	name
30.00	0.09	0.09	144274	0.62	0.62	get_block_luma
10.00	0.12	0.03	87939	0.34	1.48	perform_mc
10.00	0.15	0.03	300	100.00	133.33	find_snr
6.67	0.17	0.02	895777	0.02	0.02	biari_decode_symbol
6.67	0.19	0.02	27182	0.74	0.74	edge_loop_luma_ver

Each sample counts as 0.01 seconds.

Table 3 presents the result of a gprof profiling on a 300 frames video run once. We try to calculate the error of the values found with the method presented in 4.4. 0,09 seconds are spent get_block_luma, it is the most important function in time and represents 30% of the total. The error on this value is

$$\sqrt{\frac{0,01}{0,09}} \approx 0,33$$

30% of error is not acceptable and using a standard desktop computer will not be considered again as a good reference. However we can comment the results of this profiling, to have a first idea of the JM behavior.

get_block_luma and perform_mc are function used for decoding a macroblock (inverse transform, prediction, inverse quantization...). An interesting point is perform_mc calls get_block_luma . Both have a µs/call rather important (comparing to other functions profiled which need 0,00 µs according to gprof. get_block_luma is called about twice more than perform_mc and is also twice more slower.

find_snr represents 10% of the decoding process (on this run and with the percentage of error seen before). When a frame is decoded, this function compares it with the reference decoded video. This function is called once per picture, and the gprof confirms these 300 calls. The snr value found is written on the standard output. This value should be equal to zero. But, this step is not really a part of the decoding process itself but a feature of the decoder. In the future we will not add the reference decoded video to avoid spending time in this function.

biari_decode_symbol is a leaf function called for decoding cabac code. This function is very fast 0,02 µs but also called a lot which could explain its presence in this top 5 time consuming function. edge_loop_luma_ver is part of the deblocking picture step.

6 A new reference platform, the PandaBoard

The board used on the following section is a PandaBorad ES. It is composed of a Dual-core ARM Cortex A9MP Core at 1,2GHz each but can be run at 350MHz and 1GB RAM.

The PandaBoard is a single-board computer development platform based on a Texas Instruments OMAP4460 SoC. This platform is low power and can be run at 350MHZ which is rather close to the DSP frequency (225MHz). Moreover we can have a better control on task running at the same time than the decoder on this platform than we had on the Intel computer. For these raisons the PandaBoard is more similar to the DSP than the desktop computer was. Consequently this board will be used as the new reference for the following work and profiling on the desktop computer will be abandoned. The DSP memory has a 16MB SDRAM and this size doesn't allow to load a 300 frame video and decode it, the maximum size supported is a 50 frame video. This is why, in the future, experiments will be done with a trunked video of the foreman.

6.1 First results

The PandaBoard is slower than the personal computer used in the previous section. We wish having a better accuracy with it. The total decoding time is different for each run but around 0,5 seconds for 50 frames. It represents an average of 10 millisecond per frame but like shown an extract of the standard output on the PandaBoard (table 4), some frames need 45 milliseconds and others "0". On this platform a sampling period on gprof is counting at 10 milliseconds, it is still too fast to have a good accuracy. To improve the quality of our result we run the decoder with the 50 frame video as input several time. Table 5 is an extract of the first lines of the profiling data with 30 runs.

Gprof gives an average decoding time for a 50 frames video of 0,527 second. Let's calculate the error rate for the most important function :

- check_motion_vector_range $\sqrt{\frac{0,01}{3,82}} \approx 5\%$ of error
- get_block_chroma $\sqrt{\frac{0,01}{1,20}} \approx 9\%$ of error
- perform_mc $\sqrt{\frac{0,01}{1,19}} \approx 9\%$ of error

Table 4: Standard output with the 50 frames video on a PandaBoard 350 MHz:

Frame	POC	Pic#	QP	SnrY	SnrU	SnrV	Y:U:V	Time(ms)
00000(IDR)	0	0	28	0.0000	0.0000	0.0000	4:2:0	24
00008(P)	16	1	28	0.0000	0.0000	0.0000	4:2:0	18
00004(B)	8	2	30	0.0000	0.0000	0.0000	4:2:0	0
00002(B)	4	3	31	0.0000	0.0000	0.0000	4:2:0	0
00001(b)	2	4	32	0.0000	0.0000	0.0000	4:2:0	45
00003(b)	6	4	32	0.0000	0.0000	0.0000	4:2:0	0

Total decoding time: **0.512 sec**

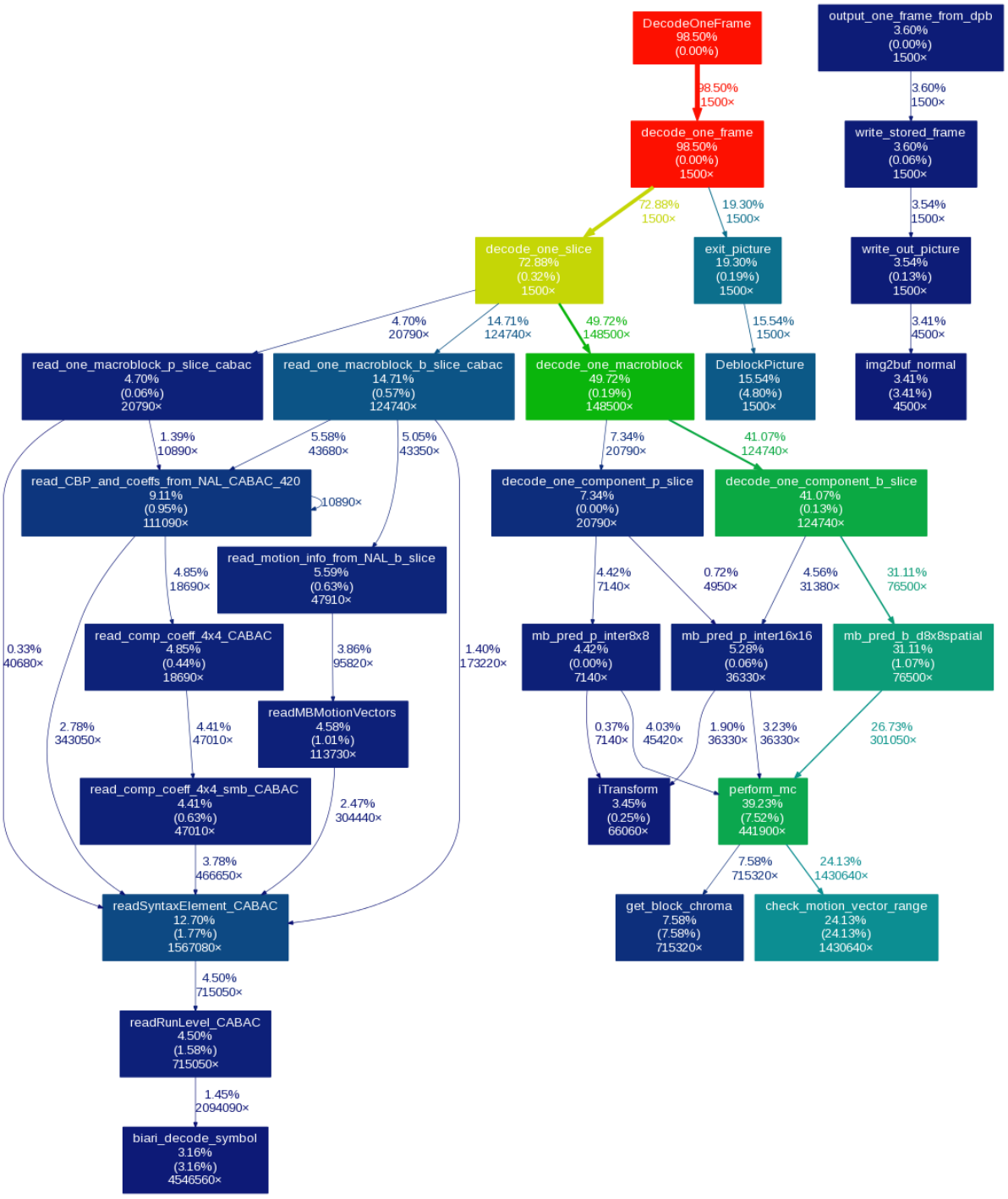
Table 5: gprof profiling, 50 frames video decoded on PandaBoard 30 times

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
24.13	3.82	3.82	1430640	0.00	0.00	check_motion_vector_range
7.58	5.02	1.20	715320	0.00	0.00	get_block_chroma
7.52	6.21	1.19	441900	0.00	0.01	perform_mc
4.80	6.97	0.76	1500	0.51	1.64	DeblockPicture
3.41	7.51	0.54	4500	0.12	0.12	img2buf_normal
3.16	8.01	0.50	4546560	0.00	0.00	biari_decode_symbol
2.27	8.37	0.36	143610	0.00	0.00	edge_loop_luma_ver

- DeblockPicture $\sqrt{\frac{0,01}{0,76}} \approx 12\%$ of error
- img2buf_normal $\sqrt{\frac{0,01}{0,54}} \approx 14\%$ of error

It is still not perfect but really better than the previous error rate. On this profile we observe functions we have ever seen on the previous profiling : perform_mc, edge_loop_luma_ver and biari_decode_symbol are still important functions. The get_block_luma function which was very present on the previous profiling is now absent and two new functions replace it. check_motion_vector_range and get_block_chroma representing 30% of the total decoding time are perform_mc children like get_block_luma was. Buffer function is at the 5th place of the most time consuming function. This function was not spot on the previous profiling but it's probably more because the memory is smaller and slower on this board than because the profiler has given wrong results.

Figure 7: Call graph on a PandaBoard 350 MHz:



6.2 Call graph

Figure 7 is the call graph built with gprof2dot.py from profiling information given by gprof. This call graph represents a decoder decoding a 50 frames video 30 times. Nodes below 3% of the total decoding time are not represented. The color of the nodes and edges varies according to the total time % value. It uses a temperature-like color-map, functions where most time is spent (hot-spots) are marked as saturated red, and functions where little time is spent are marked as dark blue.

The main function of the decoder calls DecodeOneFrame, itself calling decode_one_frame. decode_one_frame calls decode_one_slice. Here there is only one slice per frame so decode_one_slice is calledas many times as the number of frames but if the video was interlaced it would not be the case (see section 2.6.3 Skip/Direct mode).

The decoding slice function does two things: it calls a read_one_macroblock function to get the syntax elements from the NAL, then it calls decode_one_macroblock, and repeat these two operations until the end of the slice.

There is not really a read_one_macroblock function but six. There are different ways to read a macroblock depending on the slice type (I slice, B slice, P slice) and the encoding method used by the encoder (CABAC or CAVLC). The six functions are named respecting this convention:

$$read\_one\_macroblock\_SliceType\_EntropyCodingMode$$

The video is coded using CABAC so the CAVLC functions are not called, but if the video was encoded with baseline profile, CAVLC functions would replace the CABAC functions.The reading I slice macroblock doesn't appear on the graph but it is only because the graph is limited to important function.

When the entropy code is decoded, the decode_one_slice function calls decode_one_macroblock. A macroclock has three components Y U and V. The decode_one_macroblock function calls for each component the decoding_one_component function according to the slice type (I B or P slice). The decoding_one_component function has the objective to call the correct macroblock prediction function. The function are named like this :

$$mb\_pred\_SliceType\_PredictionMode\_BlockSize$$

SliceType can be B I or P. PredictionMode can be direct (short in "d"),inter or intra. BlockSize is from 4x4 to 16x16. If the prediction mode is the direct mode, a last precision is added at the end of the function name: temporal or spatial. On this video decoding the three most important functions are inter frame prediction on 8x8 blocks on p slice (4,42%), then inter frame prediction on 16x16 blocks on p slice (5,28%) and spacial direct prediction on 8x8 block on B slices(31,11%).

The two inter prediction functions call iTransform as shown on the graph. We can also observe that the direct prediction doesn't call this function performing the

inverse transformation. It is coherent with the fact that slices predicted with direct mode are constructed with only a prediction: no compensation implies no inverse transform or quantization. A further analyze of the call graph show that iTranform is called by mb_pred_b_d8x8spatial, a direct mode. But the implementation of iTransform in block.c file show that in this case the function don't perform inverse tranform but only do different copies. It is a perfect illustration of gprof inaccuracy with children time 4.5. Even if mb_pred_b_d8x8spatial and mb_pred_b_inter8x8 are both calling iTransform, the function has different behavior if the caller is a direct mode prediction or not. Dividing the total time spent in iTransform between it's parent is incorrect. However, the case where the direct function is the caller represents only 240 call on 6606 (less than 0,4%). This little number of call explains the absence of arrow between the two functions on the limited graph.

A very important function is perform_mc. MC is for Motion compensation. Its 39,23 % of the total decoding time come from the 7,53% spent in the function itself and two important children get_block_chroma and check_motion_vector_range (resp 7,58% and 24,13%). These functions are called only by perform_mc so we can wish having a rather accurate total time spent in perform_mc. An analyze of the gprof file show an other children for perform_mc : CheckVertMV but there is no mention to a get_block_luma function which was very present on the previous profiling. Hypotheses: -O3 profiling could hide the function
After a profiling a -O2 optimization program on the PandaBoard we have found the get_block_luma function. A short comparison between -O1 and -O2 gprof profiling program show us that on O2 get_block_luma is called 51186 times and on O1 23844.
On the raspberry profiling 16 we can see that the perform_mc calls get_block_luma, get_block_chroma and check_motion_vector_range as many times. On the PandaBoard check_motion_vector_range is called twice and get_block_luma is not called. We guess that the luma function is hidden behind the check_motion_vector_range label.

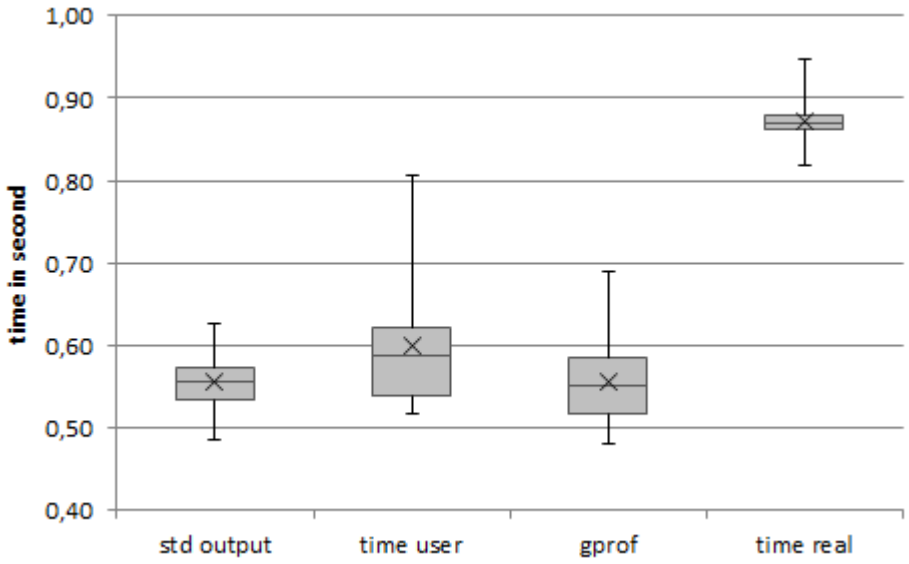
On the top right of figure refcallgraph we have a branch separated of the call tree. The leaf is img2buf_normal, this function converts image plane to temporary buffer for file writing. This function is called 4500 times (3 times per frame). This function is called only by write_out_picture, itself called by write_stored_frame. output_one_frame_from_dpb is the root of this branch according to the graph, but this graph is limited to important node and the gprof profilling data shows this function called 510 times by flush_dpb and 990 times by store_picture_in_dpb. The Store function is called by the exit_picture function, a function shown on the call tree figure. Flush_dpb is called by activate_sps 60 times (probably 2 times per video decoding), FinitDecoder 60 times (probably 2 times per video decoding) and 30 times per idr_memory_management (probably oncer per video decoding). These functions are also linked to the main call tree, but are not essential part of the decoding process.

6.3 Different ways to measure time

The JM software write out on the standard output few information on the decoding process, including the time spent in every frame, and after closing the decoder write out the total time (the sum of the time spent in each frame). Deactivating this feature doesn't decrease the total time written at the end of the process. But we don't have a good precision on the total decoding time, because at each run we receive a sensible different result. Time printed at the end of the total decoding time varies from less than 0,5 sec to more than 0,6. Moreover, when other process are running on the machine, the decoding time is affected, and the total decoding time could double or more.

We have experiment the behavior of 5 measuring time method. Just reading the standard output, using the command time ./ldecod.exe (user time and total time), a gprof profiling, and with the getrusage function called at the beginning and the end of the program. The results using rusage struct are not different from the result using time command, only the results from time command measures will be presented. We use twenty time each time measuring method, and we present the result in box plot on figure 8. On this Figure the bottom and top of the box are the first and third quartiles, and the line inside the box is the second quartile (the median). The ends of the whiskers can represent several possible alternative values, but we have decided to use the minimum and maximum values. The crosses represent the average value.

Figure 8: Measuring time on a PandaBoard



- Real time given by time command is wall clock time - time from start to finish of the call. It includes time slices used by other processes and time

the process spends blocked (for example if it is waiting for I/O to complete). The box plot is very small and the median and the average value are very similar. The extreme values are 0,07 second different from the median. Statically it is the most reliable way to measure time. But this one gives results much more important than the others (the median is 160% of the std out median). SD card reads, ssh connection, Linux threads and other process increase this time.

- User time given by the time command is the amount of CPU time spent in user-mode code, i.e. the actual CPU time used only in executing the process. . This is only actual CPU time used in executing the process. It does not include other process or time the process spent blocked. However, a process interrupted several times by the kernel will never have the same time in user mode than the same process not interrupted. This method returns results a bit more important than others. It's plot box is the biggest. Based on the obtained maximal value, we should discard some runs because this method could return non pertinent results.

- Std output : The JM software measuring time system gives quite good results. The box plot is the smallest (after the real time 's box) and the median and the average value are very similar. This counter is supposed to count the time spent decoding frame, it doesn't include the initialization of the decoder, or the closing decoder functions. Run times measured with these methods are supposed to be lower than the other method measuring the total time spent in JM. Experiments show it is not really he case.

- The details about grof profilling methods have been discussed in section 4.4. The box is bigger than the stdout's but it is still acceptable. Moreover, the median and average are the same than the values given by stdout.

After this analysis we can conclude that both stdout and gprof could be acceptable ways to measure time. It implies several runs to be sure we are not in the extreme case and trying as much as possible to keep the JM program the only process running.

7 Experiments with a DSP

The experiments described in this chapter are performed on a C6713 DSP board. The initial DSP frequency is 225MHz and has 8 pipelines. It could work at 1.35 giga-floating-point operations per second (GFLOPS). It brings the highest level of performance in the C6000 family of floating point DSP.

7.1 Measuring time on a DSP

Before profiling, the main question is how to measure time. On the panda board there is an operating system able to do this measurement. On a DSP without these system calls, a get_time function has to be programed as showed bellow in listing 4. A register exists where a counter is increment with clock 0. The counter starts at 0, is incremented and restart counting at 0 after about 80 seconds. At the beginning of the program we put a value in the control time register that restart the counter. The program don't run more thanone minute, so it should'nt have a register time overflow.

Listing 4: time functions

```
static void InitTimer()
{
volatile unsigned int *ctl =
(volatile unsigned int *) 0x01940000u;
volatile unsigned int *prd =
(volatile unsigned int *) 0x01940004u;

*ctl = 0x280;
*prd = 0xffffffff;
}

void gettime(TIME_T *time)
{
//initialize counter
volatile unsigned int *cnt0 =
(volatile unsigned int *) 0x01940008u;
unsigned int a = *cnt0;
a /= 225000/4;
*time = a;
}
```

Table 6: Standard output with the 3 frames video on a DSP with gettime and before any optimization:

Frame	POC	Pic#	QP	SnrY	SnrU	SnrV	Y:U:V	Time(ms)
00000(IDR)	0	0	28	0.0000	0.0000	0.0000	4:2:0	3305
00002(P)	4	1	28	0.0000	0.0000	0.0000	4:2:0	2720
00001(b)	2	2	30	0.0000	0.0000	0.0000	4:2:0	2277

Total decoding time : **8,302 sec**

The results (table 6) are very slow, but the code code is running without any optimization and with default configuration.
By default all the code is stored by code composer studio in external memory (CE0). The memory architecture of the DSP is explained in more details in section 7.2, but the important point is that external memory is very slow. Putting the stack on internal memory and enabling the third level of optimization on the compiler (-o3) gives better results. From more than 8 seconds for decoding 3 frames, it needs now 2,7 seconds. But these results are still unsatisfactory, for comparison the panda board can decode about 260 frames in 2,7 sec. For a better understanding of the DSP platform we have to study in details its architecture.

7.2 C6713 Memory architecture

Figure 9: C6713 Block diagram

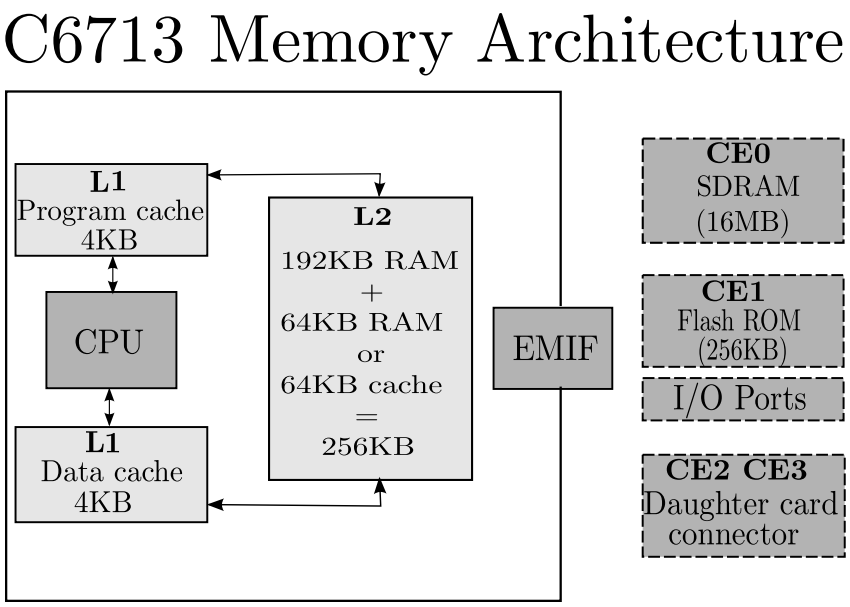


Figure 9 presents the TMS320C6713 memory architecture. This architecture is organized as describe bellow: The C6713 has a two levels architecture memory.

- **Level 1 memory** is always used for cache. There are 2 caches : one for the data, one for the program. Each one is 4KB.The two level 1 caches are always available and are single cycle access. If the data is not on the L1 cache, L2 is accessed.
- **Level 2 memory** is 256KB. 64KB can be made cache, the other 192KB are always RAM (data or program).

The External Memory interface (EMIF) is broken in four 128 Mbytes external range, each one has a dedicated strobe (CEx) The DSK uses 4 external Memory regions (CEx).

- CE0 is 16MB and dedicated to SDRAM.
- CE1 is 256KB and used for Flash memory and I/O Ports
- CE2 and CE3 are available via daughter card connector.

The L2 memory is controlled by several register. The Cache Configuration Register (CCFG) is a 32b register. Its three lowest bits are interesting in our case:

L2MODE	Value	Description	Cache	SRAM
DEFAULT	0h	L2 Cache is disable	00KB	256KB
	1h	1way cache	16KB	240KB
	2h	2way cache	32KB	224KB
	3h	3way cache	48KB	208KB
	4h-6h	reserved		
	7h	4way cache	64KB	192KB

Each external memory address space of 16 Mbytes is controlled by one Memory Attribute Register bit (0: noncacheable, 1:cacheable). 16 MAR registers control the memory caheability. MAR0 at address range 0184 8200 controls CE0 range from 8000 0000 to 80FF FFFF. By default the memory is not cachable. This is why the decoding process was so slow. As soon as possible in the program, L2 cache is configured.

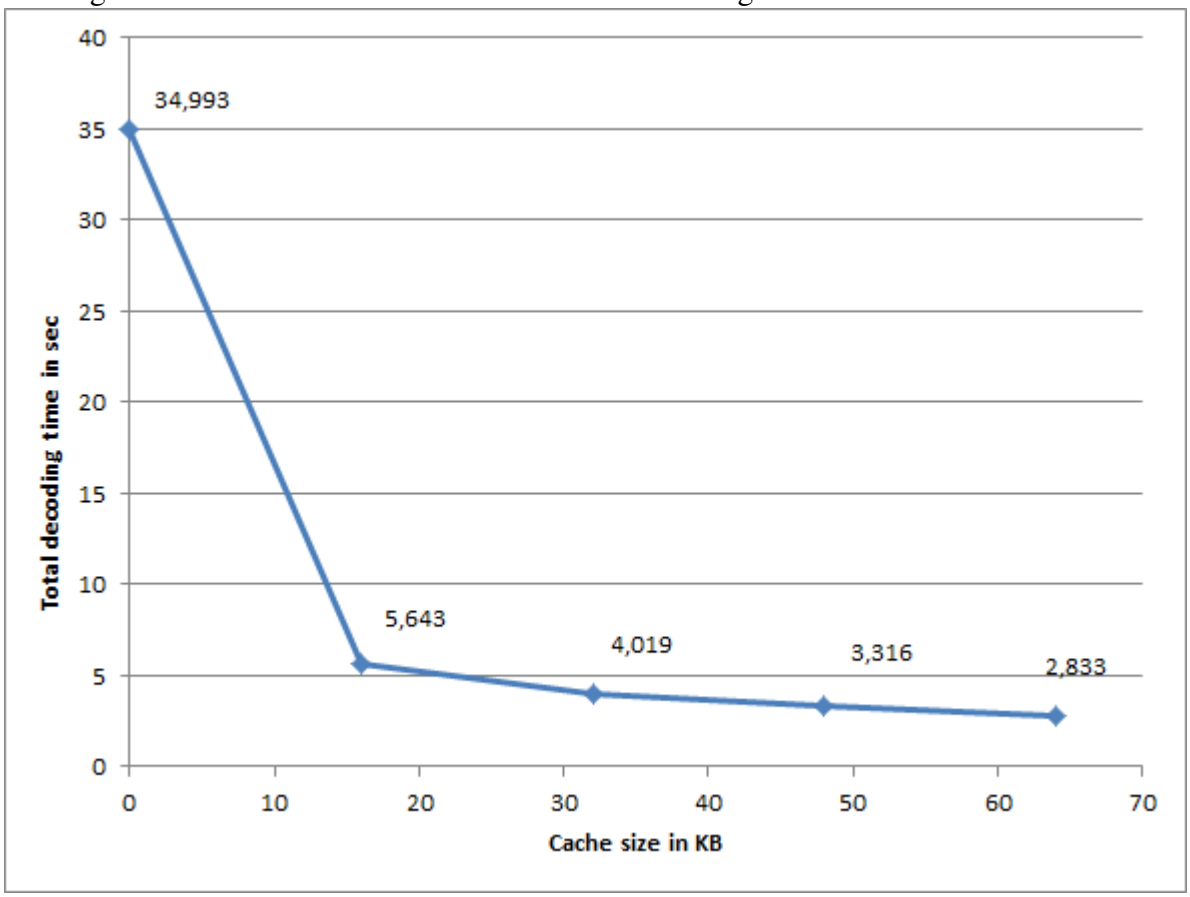
Listing 5: configuring L2 cache

```
void enableL2cache()
{
//CCFG
*(volatile unsigned int*)(0x01840000) =
*(volatile unsigned int*)(0x01840000) | 0x7;
//MAR0
*(volatile unsigned int*)(0x01848200) =
*(volatile unsigned int*)(0x01848200) | 0x1;
}
```

The enableL2cache function as written above allow to use 64KB of cache. Figure 10 shows the influence of the cache size on the software speed. Tests are done with no cache, 16 KB cache, 32 KB, then 48 KB and finally 64KB.

For more information on the C6713 memory management, there is a Texas Instrument datasheet center on this specific point [6].

Figure 10: Cache influence: time needed for decoding a 50frames video



7.3 C6713 Pipeline

All the C6000 DSP family is based on VLIW architecture. The C6713, the last one in the family, is designed to perform floating operation at a high level of performance. The objective of such architecture is to take advantage of instruction level parallelism. In traditional approach of parallelism, all decisions are made internally and the processor has to schedule itself instructions in its pipeline. It implies higher cost, energy consumption and complex hardware. The VLIW approach is totally different. The following section is an introduction on DSP pipeline, a more detailed presentation is available on Texas Instrument user guide [8]

It is not the processor but the program that decides if instructions will be executed simultaneously and how. Consequently the hardware is simpler and the compiler more complex. On the C6713 DSP platform an instruction is 32 bit, and the Program bus is 256 bits wide. It means that each clock cycle, the C6713 fetches 8 instructions. The 256 bits fetched forms a Very Long instruction Bit Word (VLIW). The CC6713 has 2 data path A and B, and both have 4 functional units:

- D for loading storing and arithmetic operations
- L for Logical and arithmetic operations
- M for Multiply operations
- S for Branch, bit manipulations and arithmetic operations

Both of them can perform one operation per cycle. At 225MHz and with 8 units it means 1800 million instructions per second (MIPS). C6713 is a floating point processor. The L M and S units can perform one float operation per cycle, the platform can perform 1.35 giga float operation per second (GFLOPS)

Ideally we want all the 8 instructions executed in parallel, even it's often not possible. For example if we want to do 3 multiplications or if an instruction needs the result of an other operation and has to wait for it. Every instruction follows these steps: fetching, decoding then executing. Fetch is divided in 4 operations:

- PG : Program address Generate
- PS : Program address Send
- PW : Program address ready Wait
- PR : Program fetch packet Receive

Decode is divided in 2:

- DP: Instruction DisPatch
- DC: Instruction DeCode

Execute has 10 phases labeled E1-E10. All instructions don't need these 10 phases to complete.
Figure 11 presents a pipeline. A first instruction is stored on the first line at clock 1. At clock 2 this first instruction is still processing (step Program address send)

Figure 11: An ideal and non realist pipeline

					clock cycle						
1	2	3	4	5	6	7	8	9	10	11	12
PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5	E6
	PG	PS	PW	PR	DP	DC	E1	E2	E3	E4	E5
		PG	PS	PW	PR	DP	DC	E1	E2	E3	E4
			PG	PS	PW	PR	DP	DC	E1	E2	E3
				PG	PS	PW	PR	DP	DC	E1	E2
					PG	PS	PW	PR	DP	DC	E1
						PG	PS	PW	PR	DP	DC

but a new instruction can be fetched in parallel. At clock 11 the pipeline is full because all eleven instructions are proceeding through the various phases E1 -E10. Most instruction tie up their functional unit for only one phase (E1). This process has limitation and the pipeline presented in figure 11 is not realist. In a real, instruction flow bubbles are introduced in our pipeline process. These bubbles represent extra idle cycles.

Figure 12 gives a good example, if an instruction needs the result of a previous instruction; the instruction will wait the end of the execution of the previous one. We start with a subtraction operation. The subsequent instruction uses the result of this subtraction as an input parameter. The addition instruction execution cycle is supposed to start at 8 but an extra cycle is added to be sure the ADD will read the result of the subtraction and not the result before the addition occur.

Figure 12: A more "actual" pipelining operation

clock cycle								
1	2	3	4	5	6	7	8	9
PG	PS	PW	PR	DP	DC	SUB	.	.
	PG	PS	PW	PR	DP	DC	Bubble	ADD

The two most problematic events for effective pipeline are interruption and branching. A branch instruction doesn't know its target address before the end of its execution stage. The following instructions have already been fetched when the return of the branch is known and that can cause problems. The solution is to flush the subsequent instruction in the pipeline. The pipeline is stalled and the processor waits until the end of the branch execution. That results in a bubble on the pipeline. Branches are very expensive on a DSP. Interruption's effects are very similar to Branch instruction's and Interruption can be seen as a branch to an interrupt subroutine .

7.4 Code Composer Studio : Build properties

A first option, necessary for building the project is increasing the stack and the heap size. Running the command *valgring ./ldecode.exe* give a heap summary information.

==9752== HEAP SUMMARY:
==9752== in use at exit: 0 bytes in 0 blocks
==9752== total heap usage: 1,383 allocs, 1,383 frees, 132,601,784 bytes allocated
==9752==
==9752== All heap blocks were freed – no leaks are possible

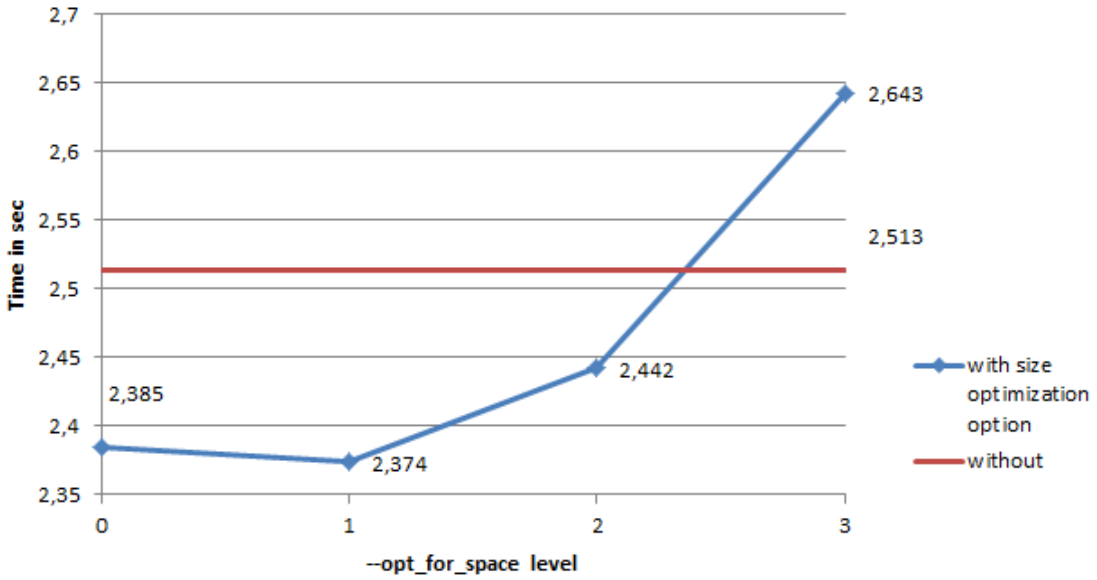
We decide to increase the stack size until the limit of the internal memory. The heap size is also fixed to be as big as possible. On the following experiment, the

fixed size are:
–stack_size 0xfe00
–heap_size 0x800000

7.4.1 optimization level

When -O3 is specified, the compiler optimizes primarily for performance. You can adjust the priorities between performance and code size by using the code size flag –opt_for_space (short in ms). The -ms0, -ms1, -ms2 and -ms3 options increasingly favor code size over performance as seen on Figure 13.

Figure 13: Influence of –opt_for_space option on decoding time



7.4.2 Pointer aliasing

Aliasing occurs when you can access a memory location in more than one way. For example when two pointers point to the same object. Using aliasing implies a worse optimization level, because any indirect reference can refer to another object. Fortran says Aliasing is not allowed and the compiler optimize the code ignoring possible aliasing. C code allows Aliasing and it is one of the reasons why Fortran generate often a more efficient code.

Using the *-mt* option indicates how aliases are used in the code.

- The -mt option indicates that indirect references on two pointers, P and Q, are not aliases if P and Q are distinct parameters of the same function activated by the same call at run time.

- The -mt option indicates that a pointer to a character type does not point to an object of another type.

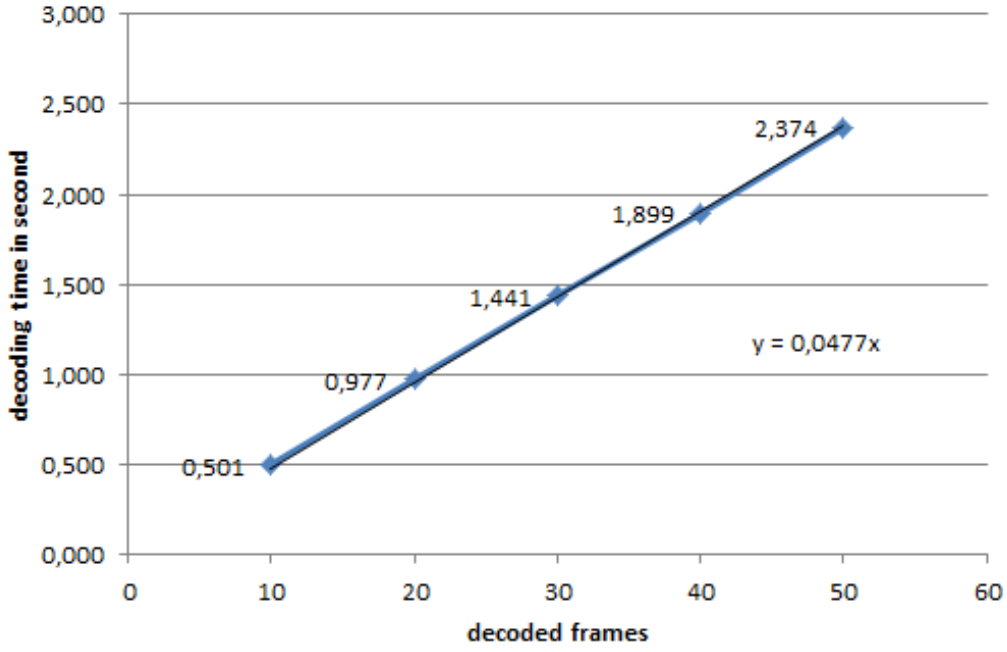
- The -mt option indicates that each subscript expression in an array reference A[E1]..[En] evaluates to a nonnegative value that is less than the corresponding declared array bound.

- The -mt option indicates that loop-invariant counter increments and decrements of loop counters are non-zero. Loop invariant means a value of an expression does not change within the loop.

The reference code respects all these points, so it is possible to use the -mt option. More information about this option are available in [7] Texas Instrument's user guide for C6000 Optimizing Compiler (pay specific attention to pages 69 and bellow).

7.5 Decoding process in an optimized environment

Figure 14: Decoding time, a linear function of frame number



With this new knowledge, we came back in the laboratory for new measurements. The decoding time printed is the same for each run. With the reference foreman video the first 10 frames are decoded, then the 20 first, ... Over 50, there is not enough memory for loading the encoded video and the decoded one. The trending line is linear and gives 47,7 ms for decoding one frame (figure 14).

7.6 Improvements

Table 7: Optimizations tabular

	first result	-o3	stack in the internal memory	cache size 16KB	cache size 32KB
time per frame in ms	2767	2143	699	113	80
percentage of improvement		129%	395%	2448%	3458%

	cache size 48KB	cache size 64KB	–opt_for_space =1	-mt option
time per frame in ms	66	57	50	48
percentage of improvement	4192%	4854%	5534%	5764%

Table 7 concludes this section by a recapitulation of all modifications made to find the final result : 48 ms to decode one frame.

8 Raspberry Pi experiment

Because a board was available and having a second reference platform can improve this analyze, we have decided to work also with a Raspberry Pi. A disadvantage of the DSP is its little memory whereas the Panda Board has an efficient one. 64KB L2 cache on the C6713 and 1 MB of L2 cache on the PandaBoard. The Raspberry Pi offers a good alternative, and we wish a slower decoding time and a behavior more similar to the C6714 ‘s. Another point is the CPU frequency of the PandaBoard could not be slow under 350 MHz and the DSP in running at 225MHZ. The Raspberry Pi can operate at 225MHz.

8.1 Raspberry architecture

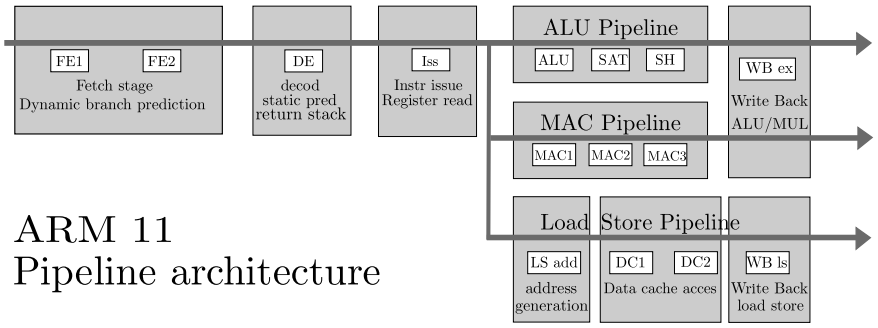
The Raspberry Pi used is a Model B. Its CPU is a ARM1176 that operates at 700MHz by default but could run at 225MHz. The board has a two levels cache, the level 1 cache is 32KB and the level 2 cache is 128KB. The L2 cache is primarily used by the GPU, not the CPU. The SDRAM is 512 MB

8.2 ARM11 pipeline

The ARM 11 architecture pipeline consists of 8 stages. It allows 8 different processing stages to be carried out at the same time, as seen on Figure 15

- Step 1 : Fetching stage. Including fetching and dynamic branch prediction.
- Step 2 : Decoding stage. Including return stack and static branch prediction
- Step 3 : Instruction stage Including Instruction issue and register read.
- Step 4 : the pipeline can use 3 different ways: Alu pipeline, Multiply and ACcumulate pipeline (MAC) and a Load/store pipeline

Figure 15: Processor micro architecture



This pipeline architecture has been design to deliver high performance efficiently. Branch instruction could interrupt the instruction flow and create latency. And we have seen on the DSP that branches are very expensive. On the ARM11 architecture's first pipeline stages there is a branch prediction step. (2 stages one for dynamic and one for static branch prediction respectively on stage 2 and 3) This architecture is supposed to avoid a loose of performance while branch occurs.

The branch predictor try to guess what is the most likely result of the branch before the branch itself is executed. The processor fetches these most likely next instructions and adds them in the pipeline. These instructions are speculatively executed. Later, if the prediction was good everything is OK and a branch don't take more time than another instruction. If the prediction was wrong, the speculative executed instructions are discarded, the pipeline is flushed and restart at the

correct branch with a delay. The time wasted in case of a branch misprediction is equal to the number of stages in the pipeline. A long pipeline implies a good predictor. Branch prediction is not used in DSP architecture.

8.3 Raspberry Pi Performance

To decrease the CPU frequency on the board, in the /boot/config.txt file we add "arm_freq=225" and reboot the system. While the system is running three commands can inform us about the system frequency:
sudo cat /sys/devices/system/cpu/cpu0/cpufreq/cpuinfo_max_freq or the command *cpuinfo_min_freq* and *cpuinfo_cur_freq*. All of them return the value 225000. Decoding a given video on the Raspberry Pi gives a different total decoding time at each run. But the average of 20 gprof profiling gives a total decoding time of 1,931 second. It is slower than the PandaBoard (0,527 sec) but still faster than the DSP (2,374 sec).

Table 8: Standard output with the 50 frames video on a Raspberry Pi 225 MHz:

Frame	POC	Pic#	QP	SnrY	SnrU	SnrV	Y:U:V	Time(ms)
00000(IDR)	0	0	28	0.0000	0.0000	0.0000	4:2:0	71
00008(P)	16	1	28	0.0000	0.0000	0.0000	4:2:0	58
00004(B)	8	2	30	0.0000	0.0000	0.0000	4:2:0	45
00002(B)	4	3	31	0.0000	0.0000	0.0000	4:2:0	41
00001(b)	2	4	32	0.0000	0.0000	0.0000	4:2:0	29
00003(b)	6	4	32	0.0000	0.0000	0.0000	4:2:0	31

Total decoding time : **2,001 sec**

Table 8 shows the first six frames decoded by the Raspberry Pi decoder. At least a frame need 23 ms to be decoded, whereas on the PandaBoard some frames needed 0 ms. We can hope having a better profiling with this architecture. The first lines of gprof flat have this error rate :

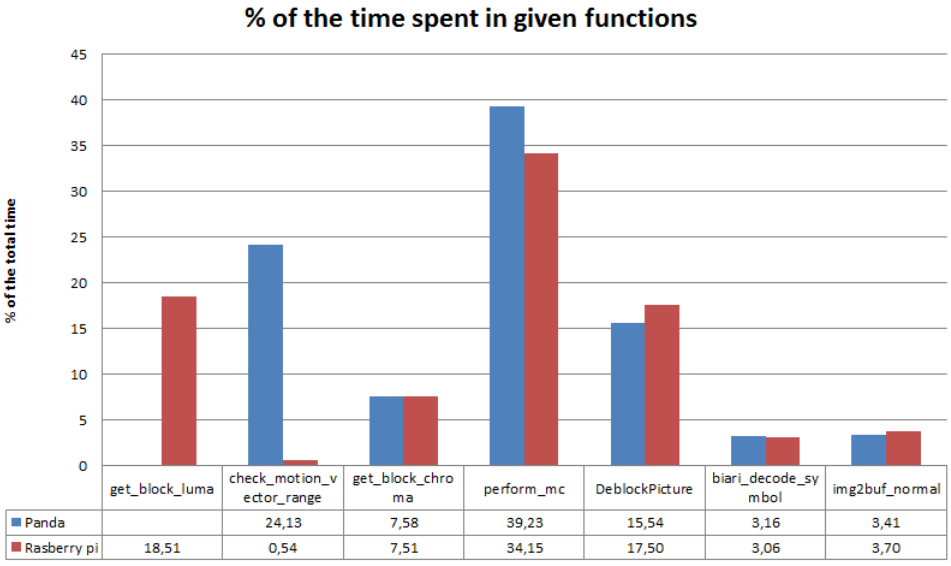
- get_block_luma $\sqrt{\frac{0,01}{18.51}} \approx 0,05\%$ of error
- perform_mc $\sqrt{\frac{0,01}{7.51}} \approx 0,13\%$ of error
- get_block_chroma $\sqrt{\frac{0,01}{7.51}} \approx 0,13\%$ of error
- DeblockPicture $\sqrt{\frac{0,01}{4.70}} \approx 0,21\%$ of error
- img2buf_normal $\sqrt{\frac{0,01}{3.70}} \approx 0,27\%$ of error

These error rates are low, but the time used for these calculations comes from the sum of several run. Errors existed and are probably much higher than 0,05%.

8.4 Comparison with the PandaBoard behavior

Even if their run times are different, we can ask the question of their general behavior. Are these platforms spending their time in the same functions? The Figure 16 presents a beginning of answer. From gprof profiling results, we get 7 important functions. get_block_luma, get_block_chroma, perform_mc, DeblockPicture, img2buf_normal, biari_decode_symbol and check_motion_vector_range. The section 6.2 details the objectives of these functions. On this graph, the time spent in each function and its children (given by the gprof call tree) is divided by the total decoding time on each platform. The resulting percentages are presented on the graph. The board have very similar behavior on img2buf_normal, biari_decode_symbol and get_block_chroma functions. Considering the error, we can even imagine they spent proportionally the same time in these functions. Perfom_mc and Deblock Picture have respectively 2 and 5 percent of difference. We have ever seen that get_block_luma function does not appear on PandaBoard profiling with -O3 optimization, but we have guessed it was hidden behind the check_motion_vector_range name. But if we use the sum of these two functions and compare the result on both platforms, the PandaBoard takes still much more time in this sum of function. Perform_mc function is the parent of these functions, and we can see this difference of behavior, even if this difference is less important.

Figure 16: % of the total decoding time spent in given function



9 Raspberry Pi baremetal

9.1 The boot process

When you power up the Raspberry Pi, the GPU core starts working. The GPU access the SD card. This one has to be formated in fat32 and contained different files used to boot the system.

- First the **bootcode.bin** is loaded by the GPU core. The GPU binary is loaded on L2 cache and enable SDRAM
- The **loader.bin** contains code able to load the next file, an elf binaries.
- The **start.elf** is the last bootloader stage (still a GPU binary). If a file called config.txt is present in the SD root then the start.elf file will load it configuring the system according to the specified options. It could configure from the video resolution to the frequency of the ARM core. Then, it loads a file called kernel.img in memory at the SDRAM address 0x8000 (or the address specified by the configuration file). Finally, the start.elf loader resets the ARM core and the ARM processor start execution from the loaded kernel starting address.
- The **kernel.img** file is an ARM executable build from the reference software code with the arm-eabi-gcc compiler.

9.2 Initialization

The initialization is writing in the system control register SCTLR. This is a 32 bit register. The bit 11 enables program flow prediction if set to 1. Then bit 0 is set to 1 to enable the MMU and bit 2 is set to 1 to enable data cache and the bit 12 to enable instruction cache.
Then it initialize the console : color, font,...Initialize frame buffer.

9.3 Unexpected results

Table 9 presents the total decoding time on the board for different frequency. The results are very slow. At 225MHz the Raspberry Pi with Linux needed 1,91 seconds to decode the same video. At the same frequency the baremetal is 786% time slower than with an OS. Such results are unexpected. We first thought, printing information on the output were the reason of these poor performance but after disabling printing, the results were still as bad as before. Because a lack of time we have not had the possibility to investigate more to explain them.

Hypothesis that could explain these results:

Table 9: Decoding time on Raspberry Pi baremetal

frequency in MHz	decoding time for a 50 frames video in second
225	15,026
350	11,940
500	09,238
700	7,592

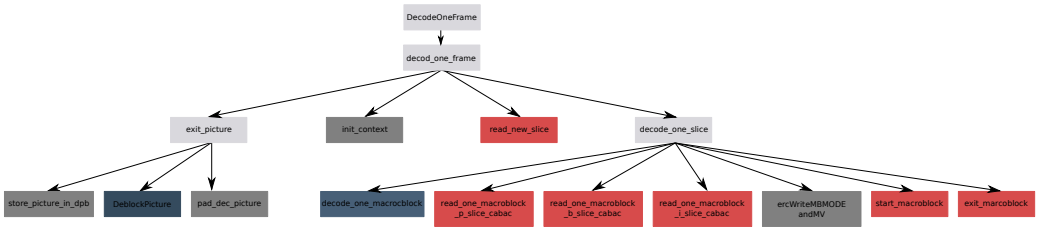
- Are we sure about our time measurement methods on the baremetal, on the linux
- Are we sure that the frequency written is the real frequency ? On the baremetal, on the linux ?
- Is the memory properly initialized on the baremetal ?
- Is Linux allowing optimization that could explain its speed?

A git repository reporting different bare metal works [11] can help to solve this problem.

10 Comparison

10.1 A detailed analyze of the call tree

Figure 17: H264 Decoding process:



To compare the different decoder, we have different elements: the total decoding time and time spent in specific function. We want now to split this total program in different discrete pieces of software that are logical representation of a part of a decoder. These pieces of program should be as decoupled as possible. To find these parts we study the call trees furnished by the grpof profiles and the information from the JM documentation and the source code itself. decode_one_frame

has 11 children. But four of them represent 98 % of the total decoding time on the raspberry pi and on the panda board.

- decode_one_slice called once per slice, this function and its children are parsing the stream but also decoding the entropy code, decoding the slice, performing prediction, ... The decode_one_slice represents more than 70% of the total decoding time on panda board and raspberry pi. It has eight children, the more important are :
  - decode_one_macrobloc: this function and its children perform the inter and intra prediction, the inverse transform, inverse quantization, the frame reconstruction.
  - start_macroblock performs different initialization for the macroblock: checks availability of neighbor, initializes the motion vector prediction, ...
  - exit_macroblock performs coordination for the next macroblock and check if it is the slice's end.
  - ercWriteMBMODEandMV writes the encoding mode and motion vectors of current MB to the buffer of the error concealment module.
  - read_one_macroblock_b_slice_cabac, read_one_macroblock_p_slice_cabac and read_one_macroblock_i_slice_cabac are decoding the entropy code (cabac) for each type of frame (B P or I).
- exit_picture: called once per picture (and not per frame), most of the time is spent in its children performing the deblock filter. But the Deblock function is not the only children of the function, there are:
  - find_snr, when a comparison with a reference picture is done, the comparison of snr values are done there.
  - store_picture_in_dpb store the picture in Decoded Pictures Buffer named dpb. The picture is saved here if it is needed later for temporal prediction.
  - pad_dec_picture is calling a pad_buf function that do some memory actions.
- read_new_slice reads new slice from bit stream dec
- init context: is used for CABAC context initialization.

From the call tree we have highlight and isolated 6 functions: DeblockPicture, read_new_slice, decode_one_macrobloc, read_one_macroblock_p_slice_cabac , read_one_macroblock_i_slice_cabac and read_one_macroblock_b_slice_cabac. These

functions are at the root of the program but there children are not the same (excepted for the read function, but they will be treated as one group) and they represent a specific moment of the decoding process. It will help us to see where the program spent its time: decoding? parsing? initialization? And if one of these macro functions need much more less time on the dsp, we will know we have to look further in this function and children to perform optimization.

10.2 Time partition

Figure 18: Time repartition on the panda board

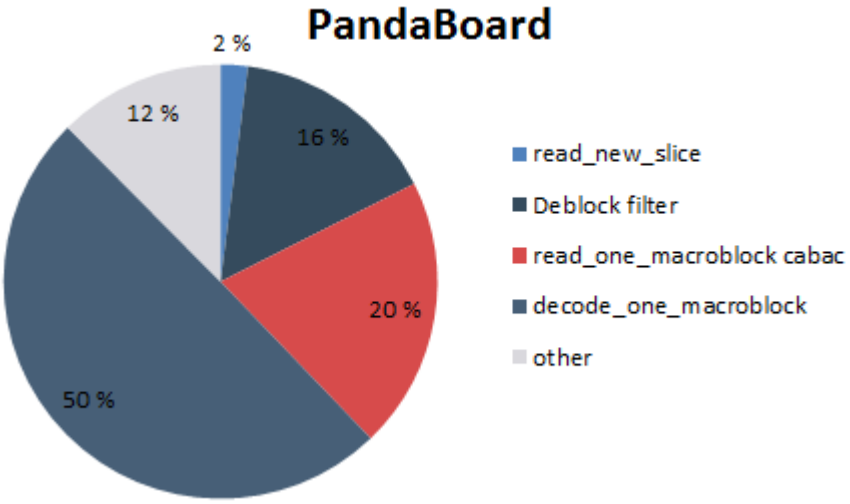
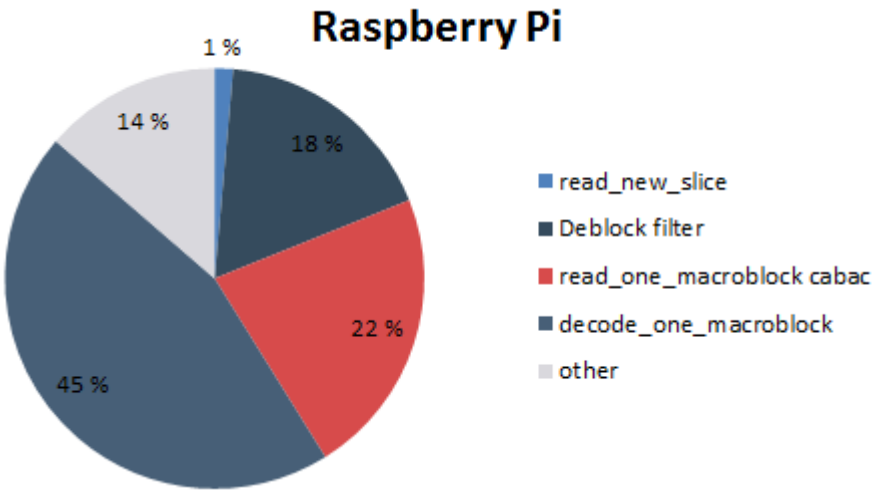


Figure 19: Time repartition on the Raspberri pi

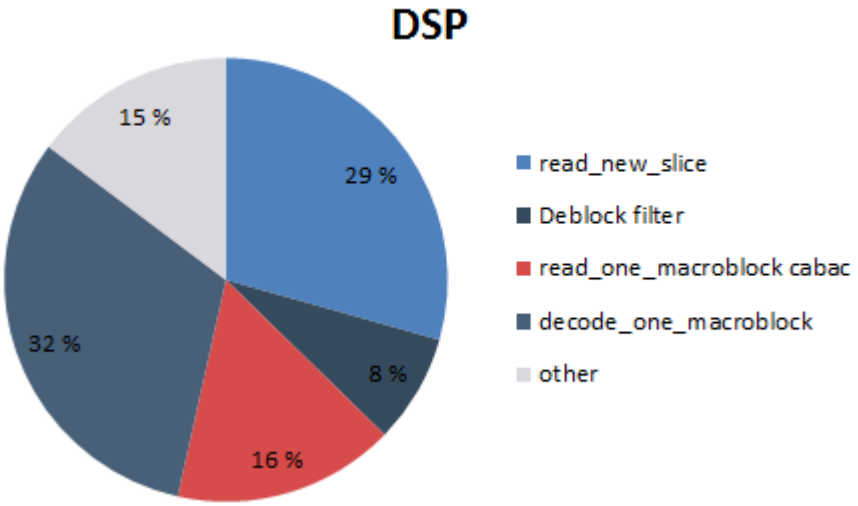


The result presented comes from a grpof profiling of a 50 frame video. The decoder was used 30 times on the panda board, and 20 times on the raspberry pi. The values come from the call tree information: the sum of the columns self and children. According to the gprof profiler, self is the total amount of time spent in this function. And children is the total amount of time propagated into this function by its children.

On the two different architectures we can observe similar behavior as seen on Figure 19 and 18. Half of the total decoding time is spent on the decoding branch. This branch does all the predictions and transformation. Then about twenty percent of the time is spent for reading the cabac code. This result is constructed as the sum of the time spent in the three read_one_macroclock_X_slice_cabac functions. Then, there is the Deblock filter part, and then others functions. Finaly the read_new slice takes only one or two percent of the total decoding time.

10.3 DSP time repartition

Figure 20: Time repartition on the C6713



The time is measured with the gettime function presented in listing 4. Before calling the decode_one_frame, this function checks the time for a first time, and at the return of the function the time is also checked. But this interval includes an I/O function reporting the details on the decoded frame. The reporting lines are the last of the exit_picture function and are commented during the experiment. This method for measuring the total decoding time give a more important result than the total decoding time written by the decoder on the standard output.

The first evident comment about this repartition is the importance of the read_new_slice function as seen on Figure 10.3. But this function is doing initialization and a lot

Table 10: Time spent in chosen function in ms

Function	Pandaboard (350MHz)	Raspberry Pi (225MHz)	C6713 (225MHz)
read_new_slice	11	27,5	985
Deblock filter	82	333,5	269
read_one_macroblock cabac	107	429,0	549
decode_one_macroblock	262	872,5	1076
Total decoding time	527	1931,0	3389

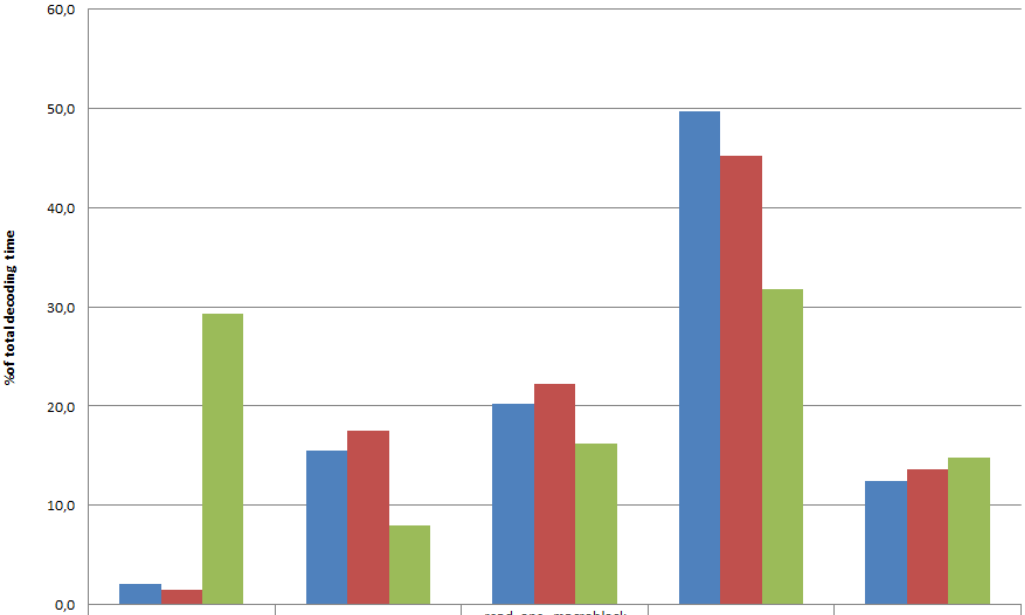
of memory allocations. And DSP are not known for their fast memory manipulations. We should keep in mind the volume of data used on each branch of the function. If we highlight a function faster on a dsp, we could migrate this function on a dsp dedicated to this task but memory allocations costs could compensate the gain. Then, we should remember that these figures are presenting percentage. The time repartition is different and you can think branches of the code are faster on the DSP like read_one_macroblock_X_cabac or decode_one_macroblock, but if you look at the table 10, you can see it is not the case. Moreover, these parts of the decoder are manipulating a lot of data, so there are not that interesting. The last and most interesting step of the decoder is Deblockfilter. This code is executed faster on the TMS320C6713 than on the Raspberry pi (330ms on the RPI and 269ms for the DSP as shown on figure 21). To deblock one frame, this process needs the frame itself and need to know how the frame was encoded because the strength of the filter depends on the encoding method used. Following and previous frame are not needed. It doesn't represent a very big amount of data, or at least an amount less important than on other part of the decoder.

11 Deblock Filter

The AVC standard is block oriented designed. One particular characteristic of this codec is the accidental production of visible block structures. Block interior pixels have a better accuracy than the edges of the block. The objective of this filter is to reduce this block effect by smoothing the edges. There are two different approaches of filters in video codecs: loop filter and post filter. Post filter implies that the filter is applied at the end of the process and can even be avoided. The MPEG4 uses the other method. The filter is integrated to the process, and filtered frames are used as reference frames for motion compensation. It implies that all the encoders have to perform identical filtering in order to stay synchronized with the decoder.

The H.264 filter is adaptive on three levels. First on the block edge level, filtering strength depends on prediction mode, motion difference and the presence of coded residual on the two blocks filtered. At this level, a boundary strength is

Figure 21: Time repartition in % on different platforms



	read_new_slice	Deblock filter	read_one_macroblock cabac	decode_one_macroblock	other
panda	2,0	15,5	20,3	49,7	12,4
raspberry	1,4	17,5	22,2	45,2	13,7
dsp	29,4	7,9	16,2	31,7	14,8

affected. It is represented by a number between 0 and 4. If it is 0, there is no filter, if it is 1, 2 or 3 a normal filter is applied and if it is 4 a special strong filter is used. Figure 23 presents the percentage of strong, normal and no filtering operations in a 50 frames video encoded with high profile. In practice luma blocks use more often strong filtering than chroma and less no filtering.

The second level is sample. It measures the difference between the edge and its neighbor, if its important, the filter is applied. If the difference is too big, this difference is the reflection of the picture behavior and the filter is avoided. It preserves the true edge of the picture unfiltered and the image's sharpness is preserved. The last level is slice level. The global filtering strength can be adjusted to the video sequence. A more detailed description of the adaptive deblocking filter can be found in [9].

Figure 22 represents the call tree of the filter part of the decoder for our reference 50 frames video. The DeblockPicture calls two functions get_db_strengh which calculates the filtering strength and perfom_db which applies the filter. get_db_stregth uses two sub functions, depending if the filter operation performed is vertical or horizontal. The perform_db function has 4 children called according to the situation luma or chroma block, vertical or horizontal filter. It is interesting to notify that the luma functions call two sub functions depending on the filtering strength whereas the chroma functions handle these two cases inline.

Figure 24 presents the profiling results on the TMS320C6713 DSP. 30% of

Figure 22: Deblock call tree:

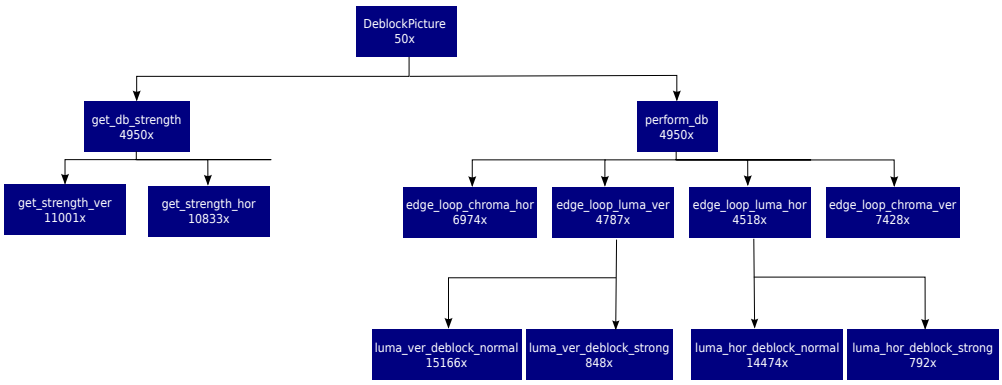
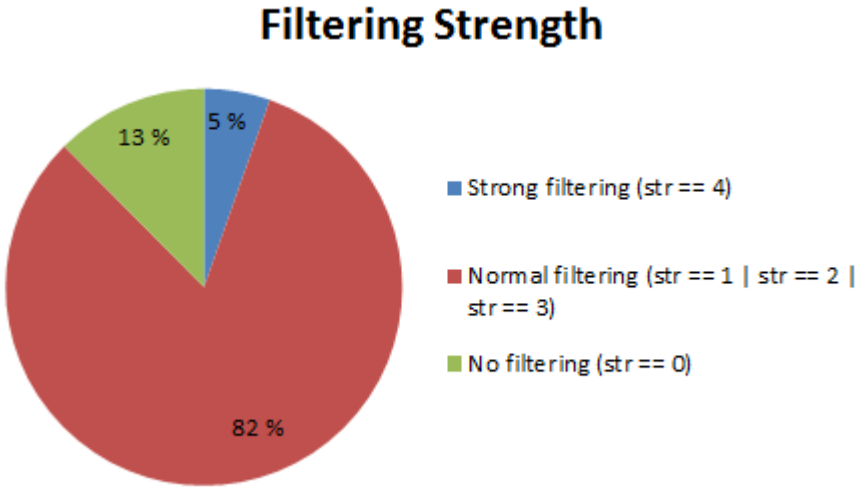


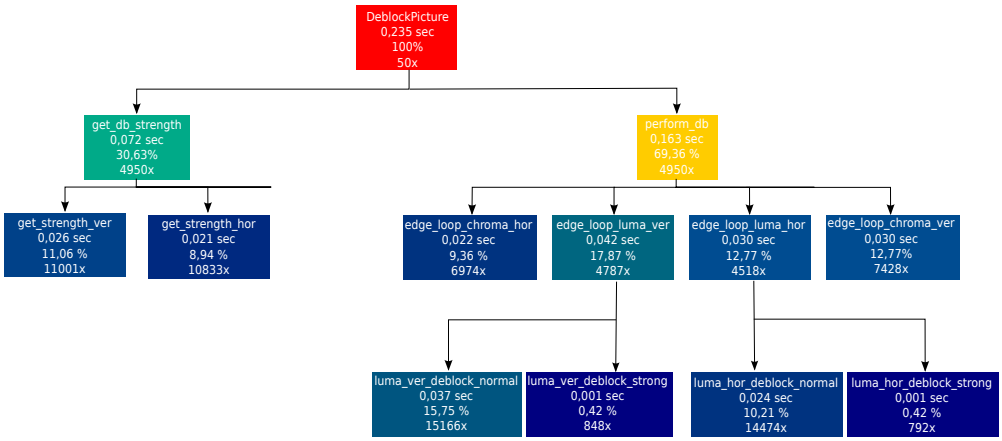
Figure 23: Different filtering strengths :



the time is spent to find the filtering strength and 70% to perform the filter itself. More time is spent to decode luma than chroma, even if the chorma filter functions are called more often (6974 times for edge_loop_chroma_hor and 4518 times for edge_loop_luma_hor). More time is spent to perform vertical filtering than horizontal filtering. This fact can be easily explained. For horizontal filtering, the neighbor blocks are the block on the left and on the right. These blocks are easy to access (index_current_block + 1, index_current_block - 1), and we can assume they are stored close in the memory, consequently if the current block is cached, its neighbor are also there. For vertical filtering, it is not the case, and the upper and the lower blocks are more difficult to calculate and access.

If we observe the four edge_loop functions, they have the same parameters *static void edge_loop_luma_ver(ColorPlane pl, imgpel** Img, byte *Strength, Mac-*

Figure 24: Filtering in the DSP:



*roblock *MbQ, int edge)*. These functions call deblock function (strong or normal, depending to their strength). But, the deblock functions have different parameters. The vertical functions use *imgpel **cur_img, int pos_x1* to send the blocks concerned by the filtering, and horizontal functions use *imgpel *imgP, imgpel *imgQ, int width*. It could be interesting to discover the reason of this difference, which version is the more time optimized and if this method could be used everywhere.

On the loop_filter_normal function, a comment mention *Cristina Gomila cristina.gomila@thomson.net: Simplification of the chroma deblocking*. Further work could consist in finding these simplifications and see if they can be applied on the luma functions.

12 Conclusion

This paper provides an overview of what H.264 is and how the reference decoder implementation of H.264 is executed on three different platforms. Decoding a video on a DSP is still 6,4 time slower than on a PandaBoard and 1,75 time slower than a Raspberry Pi. The DSP needs now 0,048 seconds to decode a frame from a video encoded with high profile. Considering that the first results indicated 2,76 second for the same work, it represents a 5 764% improvement, but further work may still improve this performance. Profiling results have highlighted the Deblocking part of the decoder takes twice the time (in percentage of total decoding time) on the ARM platforms than on the DSP. Moreover the Deblocking filter functions are executed faster on the DSP than on the Raspberry Pi : 333,5 ms on the Raspberry and 269 ms on the C6713.

References

[1] Download h.264 standard http://www.itu.int.

[2] manual page of gprof http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html#sec1.

[3] result of the man command for gprof, also available at http://unixhelp.ed.ac.uk/cgi/man-cgi?gprof, http://unixhelp.ed.ac.uk/CGI/man-cgi?gprof.

[4] Dag Agren. Bug reported on mantis bug tracker, august 2013.

[5] José R Fonseca. man page and source code of gprof2dot http://code.google.com/p/jrfonseca/wiki/gprof2dot, 2013.

[6] Texas Instruments. TMS320C621x/C671x DSP Two-Level Internal Memory Reference Guide, June 2004. Literature Number: SPRU609B.

[7] Texas Instruments. TMS320C6000 Optimizing Compiler v 6.0, July 2005.

[8] Texas Instruments. TMS320C67x/C67x+ DSP CPU and Instruction Set Reference Guide, November 2006. Literature Number: SPRU733A.

[9] Peter List, Anthony Joch, Jani Lainema, Gisle Bjøntegaard, and Marta Karczewicz. Adaptive deblocking filter. November 2006. IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS FOR VIDEO TECHNOLOGY, VOL. 13, NO. 7, JULY 2003.

[10] Karsten Sühring. JM H.264/AVC Reference Software, may 2013.

[11] Davyd Welch. Raspberry Pi ARM based bare metal examples https://github.com/dwelch67/raspberrypi, july 2013.

Turku
Centre *for*
Computer
Science

Joukahaisenkatu 3-5 A, 20520 TURKU, Finland | www.tucs.fi

University of Turku
Faculty of Mathematics and Natural Sciences
- Department of Information Technology
- Department of Mathematics

Turku School of Economics
- Institute of Information Systems Sciences

Åbo Akademi University
- Department of Information Technologies
- Institute for Advanced Management Systems Research

ISBN 978-952-12-2944-2
ISSN 1239-1891