

### Highlights

- ◆ Accelerates open source JPEG reference code with minimal code modifications.
- ◆ Achieves 7x and 8x acceleration on focused inverse DCT and color conversion algorithms
- ◆ Achieves 2.4x overall cycle count reduction including Huffman decode.
- ◆ With code restructuring, 4x overall acceleration is achievable for a pipelined design.

This case study follows the implementation of a Cascade coprocessor for decoding of JPEG images. The starting point is open-source reference code from the Independent JPEG Group ([www.ijg.org](http://www.ijg.org)) and includes a requirement that the code not be significantly modified during the implementation.

### JPEG Decoder

JPEG is a popular image compression standard, representative of discrete cosine transform (DCT) methods used in many image and video compression algorithms. JPEG includes a variety of techniques, but baseline sequential color decoding is by far the most popular and is the technique considered in this implementation. JPEG can be compressed with different quality factors. As quality decreases, the visual quality of the recovered image suffers as shown in Figure 1.



*Figure 1: Sample JPEG Image with varying quality factors from left to right*

### JPEG Algorithm

In JPEG compression, pixels are usually represented using 8-bit luminance (Y) and red (Cr) and blue (Cb) chrominance component values. The basic unit of compression is an 8x8 set of component values. Because the human eye is most sensitive to luminance, it is typical to sample the Y value four times for every chrominance pair. The original image is divided into minimum-coded unit (MCU) regions, where each MCU includes all component blocks necessary to reconstruct that region. In the typical case with down sampled chrominance values, each MCU would contain four luminance blocks

and two chrominance blocks representing a 16x16 pixel region in the original image. MCUs are processed from left to right and top to bottom to reconstruct the image. When compressing, each block is DCT transformed into the frequency domain, and the frequency values are quantized to reduce the information, with many of the higher frequency coefficients becoming zero. The coefficient values are zigzag ordered to maximize runs of zeros, and the values are run-length and Huffman encoded.

The decode flow, shown in Figure 2, begins by reading the header information to determine the JPEG type and the number and composition of MCUs. Tables needed for decoding and de-quantization are read in. The remaining stream contains the compressed values for all the MCUs in left-right, top-bottom order. As the stream is read in, the values are Huffman decoded and expanded to recover each MCU's coefficients. Because of the Huffman codes, the amount of bits required to be read from the stream varies between MCUs.

```
read header, image characteristics, decode and quantization tables

for each MCU {
    decode and dequantize block frequency coefficients
    transform blocks using inverse DCT
    convert and upsample YCrCb values to RGB
}
```

*Figure 2: Baseline Sequential Color Decoder*

Each 8x8 block within the MCU is scaled by the appropriate array of quantization values, and then run through an inverse DCT transform to recover an approximation of the original Y, Cr, or Cb component values. Once all component blocks have been recovered, the chrominance values may need to be upsampled, and each YCrCb pixel is converted to RGB representation.

## Independent JPEG Group Reference Code

Several open source JPEG decoders are available. The open-source reference code from the Independent JPEG Group (<http://www.ijg.com>) is used as the starting point for this case study.

The IJG codec is written in C and is designed to process a wide range of image sizes while running well on personal computers with limited memory. The original code reads compressed JPEG data from files during decompression so that the full image need not be entirely contained in memory during the processing.

The decompressor is designed to decode and transform each MCU completely. Once a row of MCUs is assembled, color conversion can be performed on a pixel row (scan line) basis. The biggest impact is in the strong coupling between Huffman decoding (serialized bit flow) and inverse DCT (parallel byte flow).

Though written in C, the implementation emulates single inheritance by using indirect function calls through upcasted pointers. The indirect function calls simplify algorithm variation but greatly complicate code analysis.

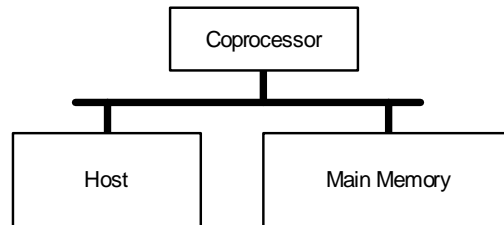
Error processing is handled with stack unwinding (`setjmp/longjmp`), which simplifies the code structure but complicates function offloading.

All data structures, even constant lookup tables, are dynamically allocated on the heap using a custom memory allocator.

JPEG defines a variety of strategies for image compression depending on the characteristics of the original image. This causes different flows through the decoder code depending on characteristics of the compressed data being processed, complicating testbench design.

## Target Implementation

The target implementation is an ARM7-class processor with a single coprocessor used for acceleration. The ARM7 host, memory, and coprocessor communicate through the ARM AHB system bus as shown in Figure 3:



*Figure 3: Single Coprocessor Implementation*

The implementation considers baseline sequential compression for both color and grayscale images. Both uniform and subsampled chrominance images are included.

Additionally, only minimal changes to the reference code are to be considered.

## Initial Decoder Implementation

For the initial implementation, the reference code is modified to remove the file handling dependency and testbench code is developed. An initial series of offload passes give a feel for expected results.

### Initial code modifications

The reference code is initially modified to remove all file handling calls during decompression. File handling I/O is typically implemented within the operating system, and this should be isolated from the decompression routines which will be accelerated. The indirect function call approach used in the original code is used to redirect the input processing to a memory buffer. The decoder output is similarly redirected to an image buffer. Decoder outputs to stderr for error and warning reporting are disabled, although the exception handling with `setjmp/longjmp` remain.

### Testbench Development

The main testbench challenge is identifying and stimulating the multiple paths through the code which depend on the dynamic characteristics of the images being decoded. With only baseline sequentially coded images to consider, the primary variation comes in color versus grayscale and the degree of chrominance subsampling.

The most popular JPEG format subsamples by 2 in both the Cr and Cb chrominance channels. Second popular is uniform YCrCb sampling. These cases also cover asymmetric subsampling modes. Both cases use interlaced MCUs. Grayscale processing is also considered and has only a single color component for decompression. It bypasses the color convertor and is not interlaced.



The processing is mostly independent of the size of the image. Images which are not integer multiples of MCU size require dummy block padding.

A testbench constructed with three small images- YCrCb with chrominance subsampling, YCrCb with uniform sampling, and grayscale, stimulate all paths of interest.

Name	cum%	self%	desc%	calls
jpeg_idct_islow	40.40%	40.40%	0.00%	0
-----				
decode_mcu	24.59%	19.36%	5.22%	0
jpeg_fill_bit_buffer		5.22%	0.00%	1641
-----				
ycc_rgb_convert	15.77%	15.77%	0.00%	0
-----				
jpeg_fill_bit_buffer	5.22%	5.22%	0.00%	1641
-----				
start_pass_huff_decoder	4.14%	0.04%	4.10%	0
jpeg_make_d_derived_tbl		3.55%	0.54%	14
-----				
jpeg_make_d_derived_tbl	4.10%	3.55%	0.54%	14
-----				
h2v2_fancy_upsample	3.88%	3.88%	0.00%	0
-----				
decompress_onepass	2.43%	2.40%	0.03%	0
jzero_far		0.00%	0.03%	144
-----				
jpeg_start_decompress	1.45%	0.01%	1.44%	3
jinit_master_decompress		0.51%	0.92%	3
-----				
jinit_master_decompress	1.44%	0.51%	0.92%	3
jinit_upsampler		0.02%	0.02%	3
jinit_d_main_controller		0.02%	0.00%	3
jpeg_calc_output_dimensions		0.02%	0.01%	3
jinit_inverse_dct		0.02%	0.00%	3
jinit_d_post_controller		0.00%	0.00%	3
jinit_d_coef_controller		0.01%	0.00%	3
jinit_color_deconverter		0.75%	0.00%	3
jinit_huff_decoder		0.01%	0.00%	3
-----				
read_markers	1.28%	1.15%	0.12%	0
get_sof		0.04%	0.00%	3
jpeg_alloc_huff_table		0.00%	0.00%	10
jpeg_alloc_quant_table		0.00%	0.00%	5
-----				
sep_upsample	1.17%	1.17%	0.00%	0
-----				
cb_decode_scan	0.87%	0.39%	0.47%	3
jpeg_read_scanlines		0.47%	0.00%	192

Figure 4: Testbench execution profile

Running the testbench on the RealView ARM7 ISS yields an overall instruction cycle count of 2,738,272 cycles.

## Offload Identification

Using RealView tools, a profile of the testbench code is shown in Figure 4:

The use of indirect function calls complicates the analysis because they do not show up as descendents in the profile calculations. In actuality, the `cb_decode_scan()` function is called once per image. It calls `jpeg_read_scanlines()` to decode one or more scanlines, which in turn calls the main workload functions of `decode_mcu()`, `jpeg_idct_islow()`, and `ycc_rgb_convert()`.

Since the `cb_decode_scan()` function is called once per image decode and does not include the initial header processing and setup, it is a good high level choice for offload.

Within the testbench, the 3 calls to `cb_decode_scan()` consume 2,485,308 of the original 2,738,272 cycles.

For code which contains direct function calls, Cascade can automatically determine the entire call tree for offloading. Once again though, the indirect function calls complicate the offload functions. To determine which functions are called, the code can be dynamically traced. However, any indirect calls which are missed will not be offloaded during synthesis and will result in early coprocessor exits. Also, it may not be enough to trace only the testbench execution since it may not catch all indirect function calls. Code coverage and manual inspection should be used to determine any missed calls. Fortunately, any calls which are missed will not be in the critical path of the algorithm and will not impact coprocessor design. Missed functions may be offloaded after the coprocessor is designed through a simple reprogramming step without compromising the overall system acceleration.

## Error Recovery Mechanism

The JPEG reference code uses `setjmp/longjmp` to unwind the stack when an unrecoverable error is found. This mechanism is quite convenient for the programmer since it removes the tedious checking of function return codes, and the code has been structured to avoid memory leaks in the presence of errors.

Unfortunately, stack unwinding cannot be offloaded to Cascade, so any `longjmp()` calls will cause an early coprocessor exit. Rather than rework the code, a shared status variable is introduced which is updated upon normal coprocessor exit. This variable is checked as `cb_decode_scan()` returns, and error recovery proceeds normally.

Since error recovery is an exceptional condition, this practical solution has no impact on code acceleration and is implemented with minimal code modification.

## Initial Offload

The initial offload function call set includes the entry function `cb_decode_scan()` plus 11 additional functions indirectly called and some additional functions directly called by these functions as shown in the right column of Figure 5.

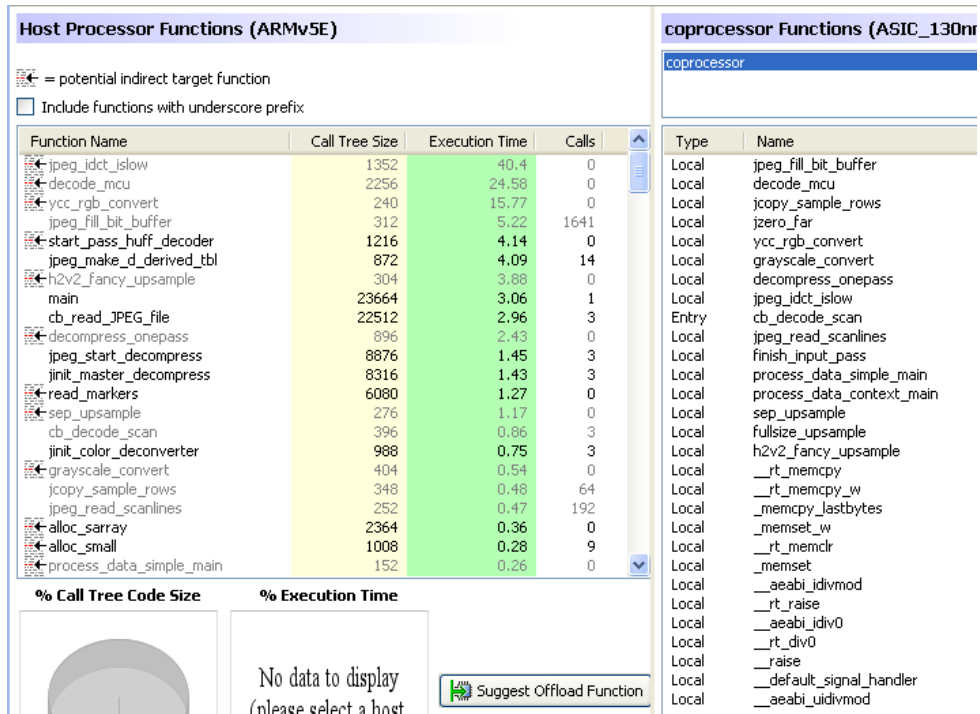
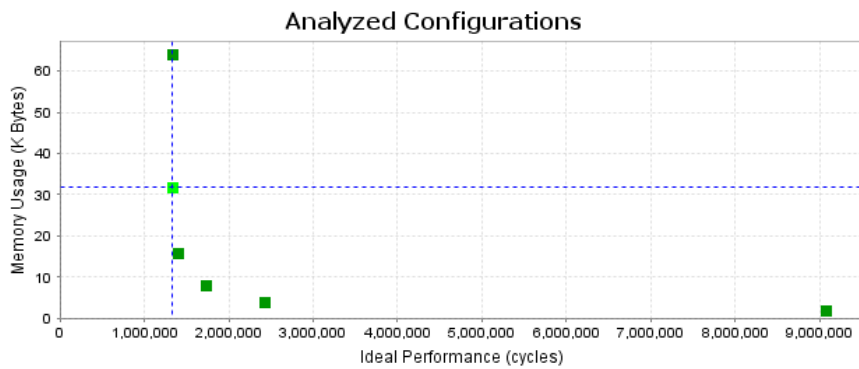


Figure 5: Offloaded functions

Generating data caches with default choices yields the memory profile shown in Figure 6. For an initial run, a large cache is chosen to reduce the impact of cache misses on subsequent analysis. An initial single port associative cache of 128 by 256 byte lines yields the candidates shown in Figure 7.

Average memory accesses per cycle: 0.393



Highlighted Cache Configuration Description				
Access Unit	Type	Size	Banks	Description
Cache0	access_assoc_1	32768	1	1xR/W Associative Cache

Figure 6: Data cache selection

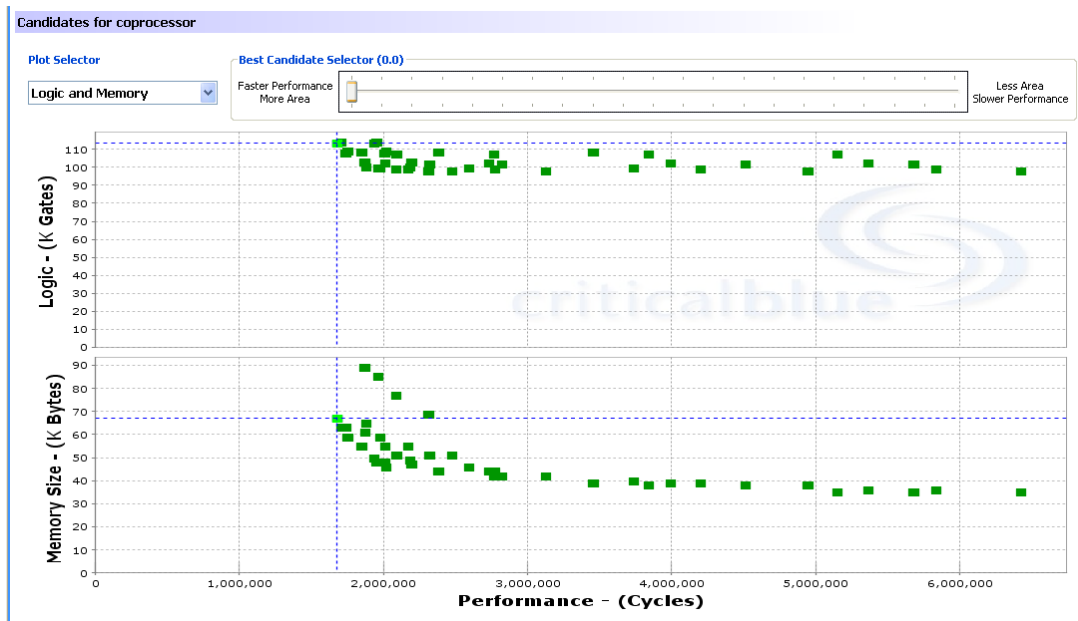


Figure 7: Candidate results

Selecting the fastest candidate and generating the microcode yields an initial cycle count of 1,603,476 cycles for an acceleration of 1.5x the original ARM7 code, shown in Figure 8. The performance of the most significant offload functions is also shown in the figure.

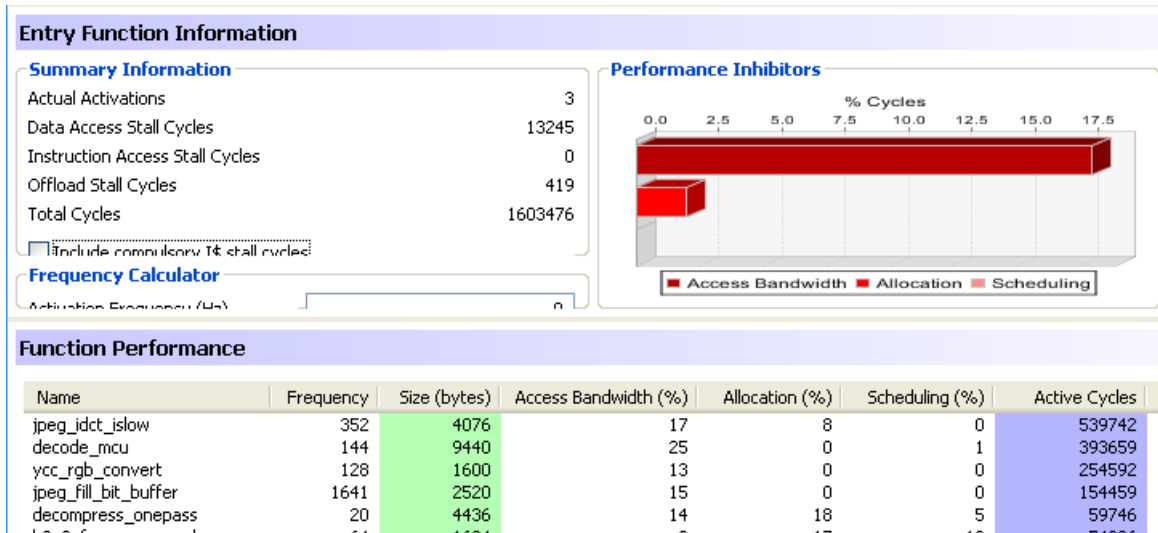


Figure 8: Initial candidate

The ARM profile data can be used to compare the results for the significant functions. The `jpeg_idct_slow()` function was originally around 1.1M cycles, and is 540k cycles (2x) in the initial design. The `decode_mcu()` function, which includes `jpeg_fill_bit_buffer()`, is originally around 670k cycles in the original while still around 550k cycles (1.2x) on the initial coprocessor. This



reflects the expected difficulty in accelerating the Huffman decode, which is strongly serialized and bit intensive.

Surprisingly, the access bandwidth, a measure of the amount of cycles lost because of a lack of bandwidth between the data cache and the execution units, is moderate (under 25%) in all the units.

## Relaxing Aliasing

As a next step, Cascade is rerun with relaxed aliasing enabled. With relaxed aliasing, pointers in hot spot regions which do not alias during testbench functional simulation are considered disjoint. This dynamic optimization improves the amount of code motion allowed during synthesis.

Since sequential mode JPEG decompression is a single pass operation, with good coding practices, pointer aliasing should not be expected. Rerunning Cascade yields the results shown in figure 9.

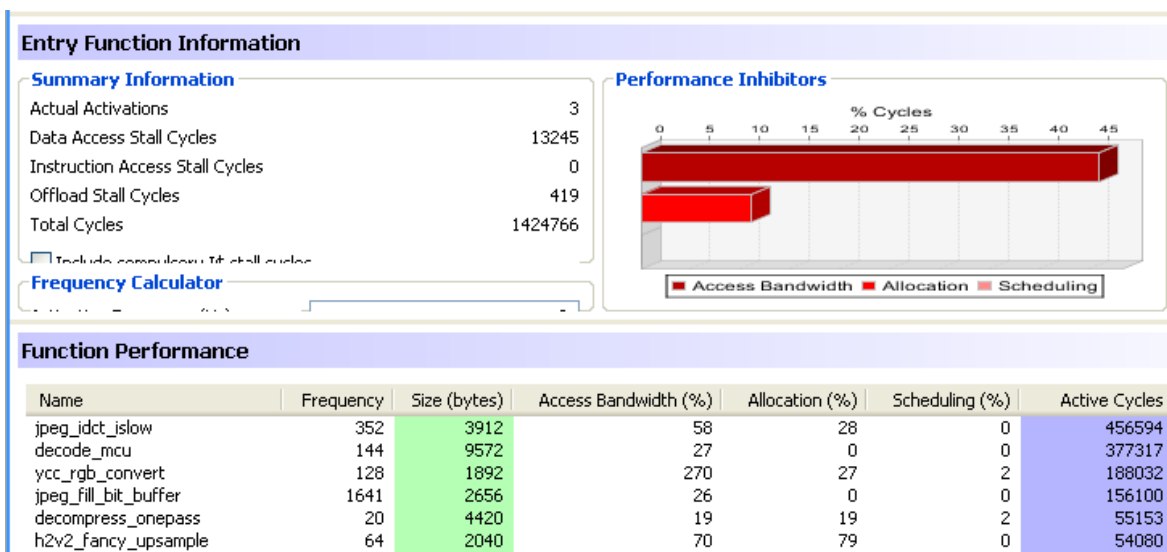


Figure 9: Relaxed aliasing results

The offloaded functions have been reduced to 1,424,766 cycles which is now 1.75x faster than the original.

Almost all the data structures are allocated on the heap and accessed through pointers, so this improvement is not unexpected. Significantly, there is now significant access bandwidth pressure in the inverse DCT and RGB conversion functions, which suggests increasing the number of data ports between cache and the execution units.

## Memory-Execution Unit Bandwidth

An additional read port can be added between the associative data cache and the execution units by rerunning data cache generation with multi-port data caches considered, yielding the new memory-ideal cycle profile shown in figure 10.

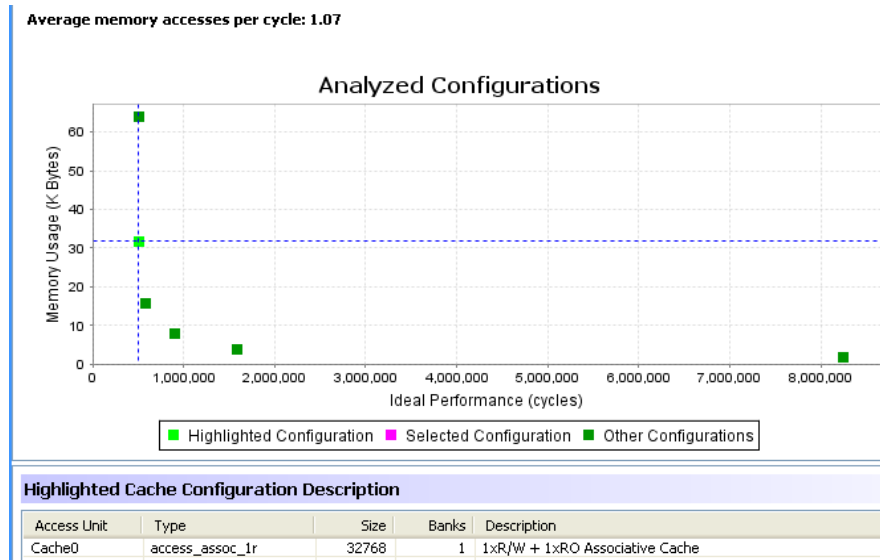


Figure 10: Multiport data cache selection

Note that the ideal average accesses per cycle have increased from .393 to 1.07. Selecting the same 32k cache size and generating candidates and microcode yields results shown in figure 11.

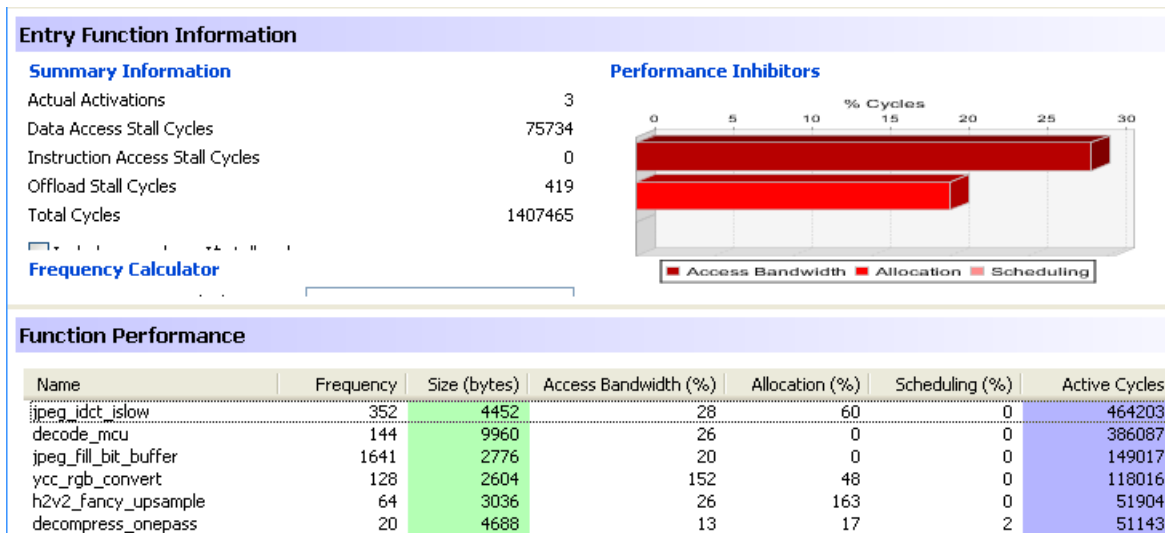


Figure 11: Multiport Results

Compared to the relaxed aliasing results, the decode and inverse DCT functions are virtually unchanged. The RGB conversion function has improved from 188k to 118k cycles and still has significant access bandwidth pressure. The other functions have much less access bandwidth pressure, but there is now a 5% data cache miss rate for the same sized cache. Allocation pressure is zero for the decode functions, suggesting that little additional performance improvement can be expected in these functions. There is still room for improvement in the overall memory system and inverse DCT and RGB conversion functions.

## Inverse Discrete Cosine Transform (IDCT)

The inverse DCT is a two dimensional calculation on an 8x8 array of 8-bit coefficient values. The calculation is usually factored into two 1-D passes. The default integer implementation in the reference code unrolls the two DCT computations in one dimension.

A modified testbench is created to isolate the inverse DCT so that it's performance can be more easily analyzed. Running on ARM7, the modified testbench spends 4,079,752 cycles in the `jpeg_idct_islow()` function.

Offloading this function to a coprocessor with an 8k dual port associative cache yields the initial results shown in Figure 12.

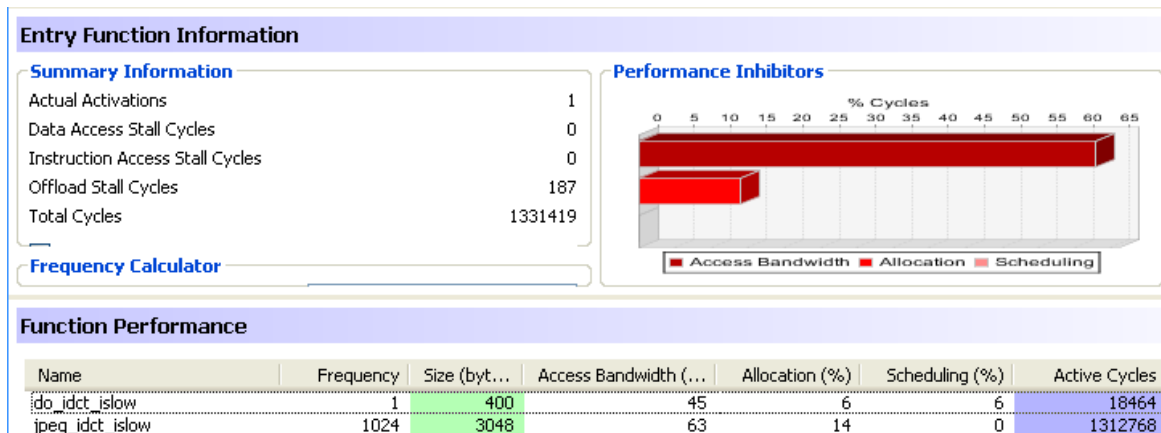


Figure 12: Original inverse DCT acceleration

A focused synthesis of the inverse DCT achieves a 3.1x reduction to 1,331,419 cycles. Access bandwidth pressure is now fairly high.

## Improved Unrolling

The initial inverse DCT function is a row and column computation which is manually unrolled in one dimension. Examining the initial offload reveals that the inverse DCT is not unrolled in the second dimension.

The code, partially shown in Figure 13, has an if clause designed to shortcut the computation for rows of all zeros. The computation requires anding eight values and is only beneficial if a significant percentage of rows are all zeros, which is frequently true for the first stage of the computation. Assuming that Cascade will parallelize may of the additions, the value of the if clause is marginal, and removing it will make it easier to unroll the loop.

The results of removing the if clause are shown in figure 14. The outermost loops are now partially unrolled 3 times, and the cycles are further reduced to 1,049,825 cycles for a 3.9x acceleration over the original algorithm running on ARM7.

```

for (ctr = DCTSIZE; ctr > 0; ctr--) {
#   if !defined(USE_NO_IDCT_ZERO_CHECKS)
       if (inptr[DCTSIZE*1] == 0 && inptr[DCTSIZE*2] == 0 &&
           inptr[DCTSIZE*3] == 0 && inptr[DCTSIZE*4] == 0 &&
           inptr[DCTSIZE*5] == 0 && inptr[DCTSIZE*6] == 0 &&
           inptr[DCTSIZE*7] == 0) { /* AC terms all zero */

           int dcval = DEQUANTIZE(inptr[0], quantptr[0]) << PASS1_BITS;

           wsptr[DCTSIZE*0] = dcval;
           wsptr[DCTSIZE*1] = dcval;
           wsptr[DCTSIZE*2] = dcval;
           wsptr[DCTSIZE*3] = dcval;
           wsptr[DCTSIZE*4] = dcval;
           wsptr[DCTSIZE*5] = dcval;
           wsptr[DCTSIZE*6] = dcval;
           wsptr[DCTSIZE*7] = dcval;

           inptr++;           /* advance pointers to next column */
           quantptr++;
           wsptr++;
           continue;
       }
#   endif

       z2 = DEQUANTIZE(inptr[DCTSIZE*2], quantptr[DCTSIZE*2]);
       z3 = DEQUANTIZE(inptr[DCTSIZE*6], quantptr[DCTSIZE*6]);

       z1 = MULTIPLY(z2 + z3, FIX_0_541196100);
       tmp2 = z1 + MULTIPLY(z3, - FIX_1_847759065);
       tmp3 = z1 + MULTIPLY(z2, FIX_0_765366865);

       ...

```

Figure 13: Short-circuit IDCT clause

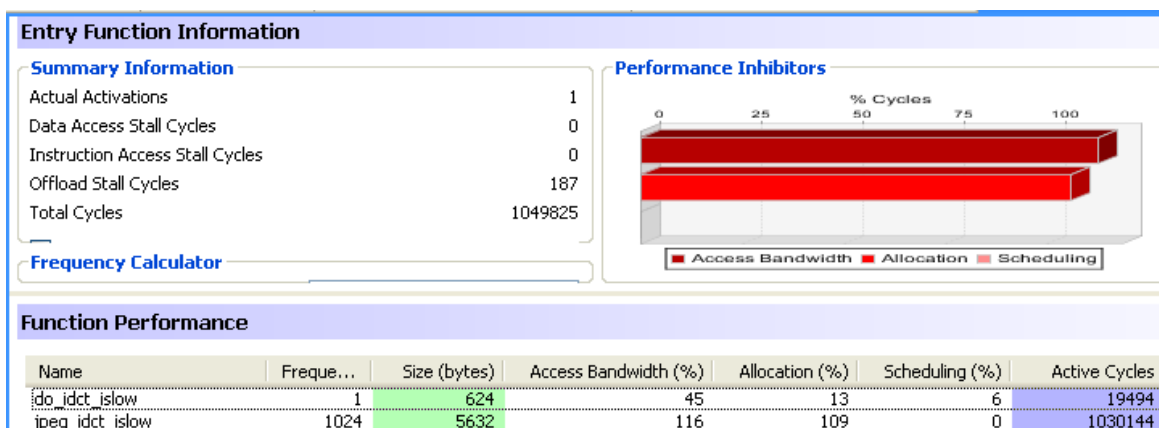


Figure 14: Partially unrolled results

## Fast IDCT

The JPEG reference code comes with several inverse DCT algorithms. A fast integer DCT algorithm is provided which runs faster but is susceptible to larger rounding errors. It is quite suitable for smaller image sizes. On an ARM7, this implementation requires 35% fewer instructions.

When this algorithm is directly substituted, the resulting Cascade synthesis results are shown in Figure 15 and show a 4.2x acceleration over the original algorithm running on ARM7.

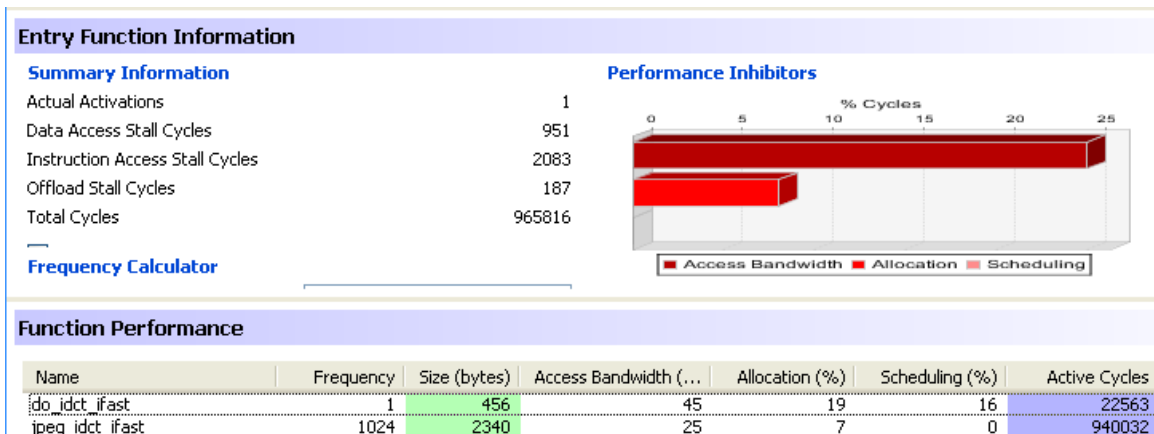


Figure 15: Fast IDCT acceleration

## Static Tables

There is still access bandwidth pressure. The IDCT uses lookup tables for scaling quantization values and to clamp out of range results. If the memory allocation is modified, these tables can be placed in static memory, and Cascade can relocate them to an additional static cache, providing additional memory bandwidth to the core execution units.

Reducing the associative cache to 4K bytes and adding a 4K static cache doubles the bandwidth to the execution units. Adding a static cache and synthesizing a new design in Cascade yields the results shown in Figure 16.

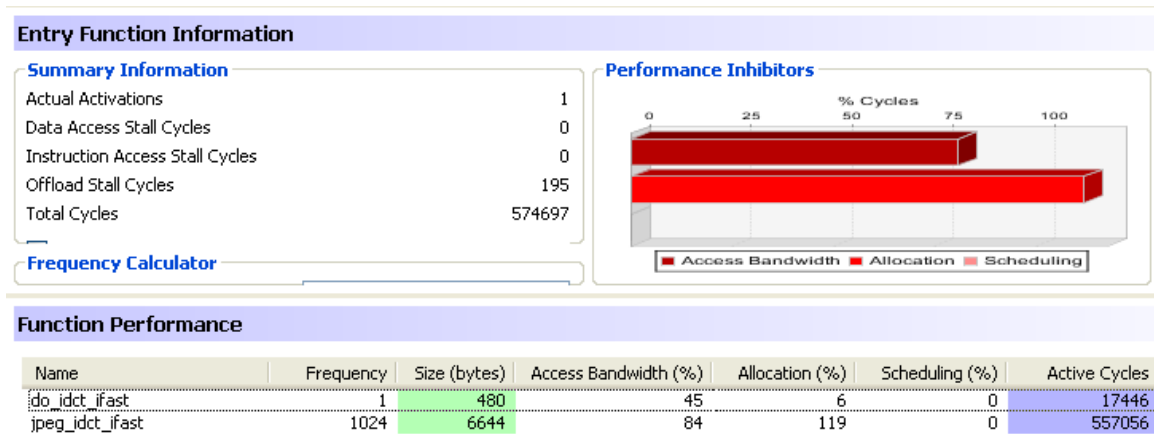


Figure 16: Final IDCT acceleration

The instruction cache line width was also increased to 64 bytes to avoid a large instruction cache stall. The final IDCT coprocessor implementation is 7.1x faster than the original ARM7 implementation.

## RGB Conversion

The RGB conversion function converts YCrCb pixel values to their equivalent RGB values. The equations are shown in Figure 17.

$$\begin{aligned}
 R &= Y + 1.403 * Cr \\
 G &= Y - 0.714 * Cr - 0.344 * Cb \\
 B &= Y + 1.770 * Cb
 \end{aligned}$$

Figure 17: Conversion functions

The reference code implementation shifts and scales the computations to be appropriate for integer arithmetic. The multiplication operations are replaced with lookup tables and shifts, which accounts for the high access bandwidth pressure.

Similar to the inverse DCT, a modified testbench is developed to isolate the conversion operations. On ARM7, time spent in the conversion functions is 3,504,286 cycles.

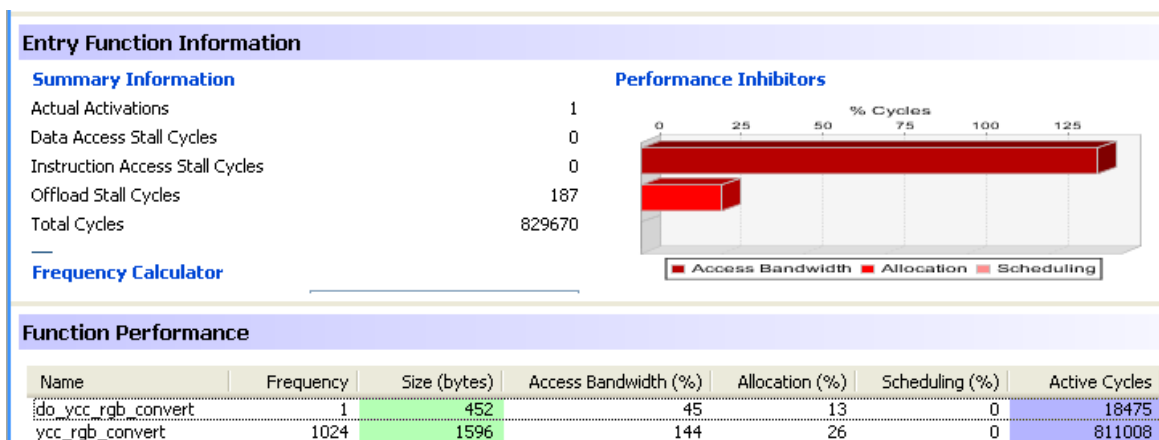


Figure 18: Initial RGB coprocessor implementation

Using an 8k multiported associative cache, results in a reduction of 4.2x to 829670 cycles as shown in figure 18.

## Static Tables

That is effective acceleration, but the same static table approach used for inverse DCT can be employed with this function. This time, in addition to a 4k associative cache, two 4k static caches are used, one to hold the color lookup tables and a second to hold the stack along with the range limiting tables used for clamping.

The final results are shown in figure 19 and demonstrate an 8.2x acceleration over the original ARM7 implementation.

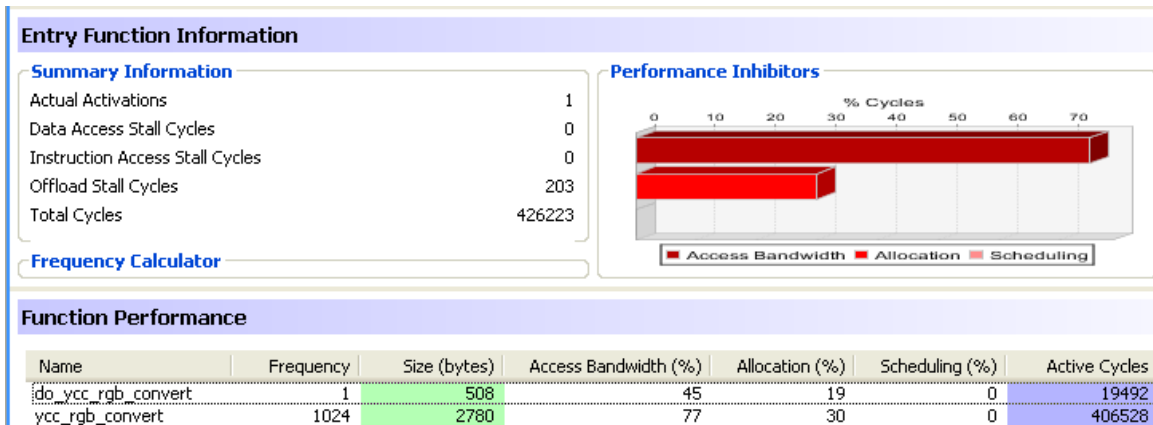


Figure 19: Final RGB coprocessor implementation

## Integrated Decoder Implementation

The next step is to integrate the lessons learned in the inverse DCT and RGB conversion optimizations to the integrated JPEG decoder offload. Using a 16k associative cache with 8k and 4k static caches, and including the inverse DCT and RGB conversion optimizations yields the results shown in figure 20.

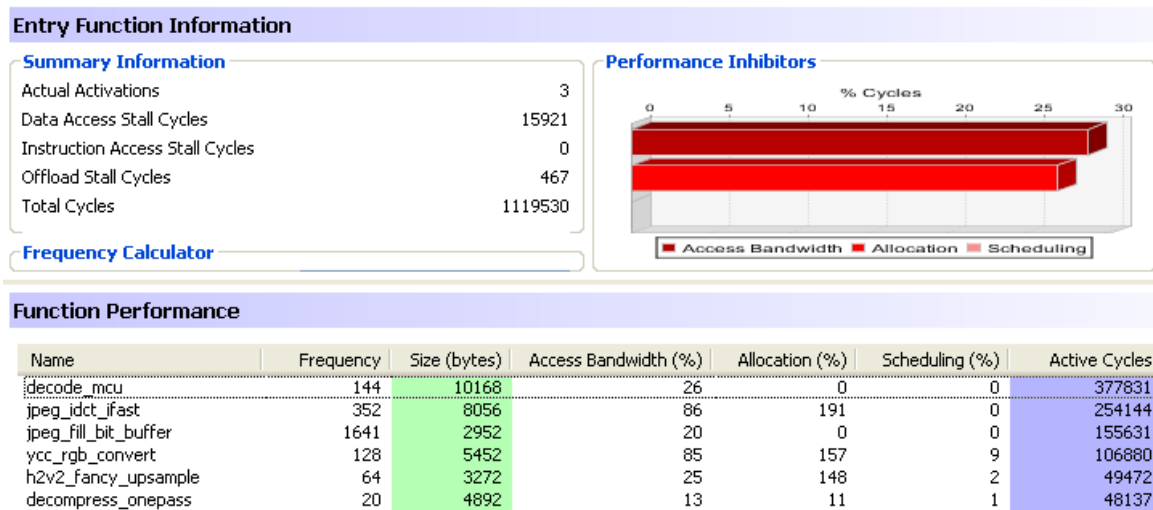


Figure 20: Integrated decoder implementation

The overall acceleration has increased from 1.75x to 2.2x.

## Huffman Decode Limitations

The Huffman decode limits the amount of acceleration achievable. The decode processing is inherently serial, and the amount of bit processing is dependent on previously processed bits. Additional code modifications can be made, but they have negligible impact.

## Multirrow Scan

In the default implementation, the pixel rows are normally processed one scan line at a time. The code can be operated to process multiple rows at a time, but the number of rows is limited to 2-4 at a time. Adjusting the code and rerunning Cascade gives the candidates shown in Figure 21 with the final results shown in Figure 22.

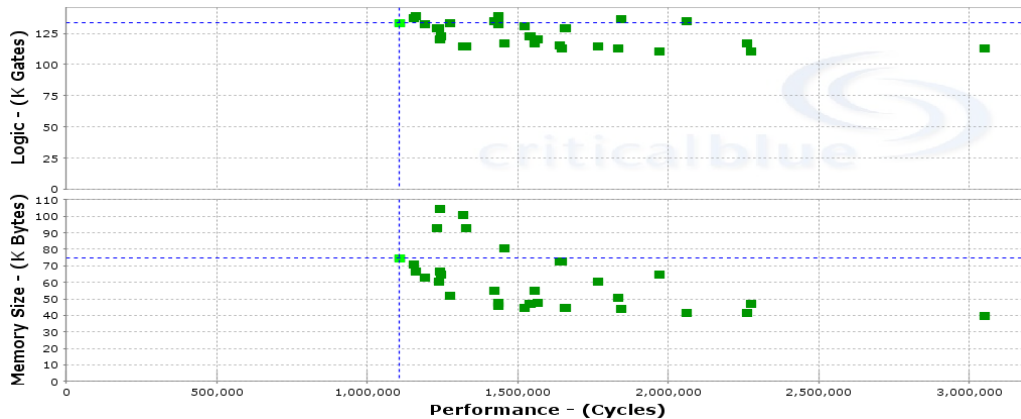


Figure 21: Integrated decoder candidates

The implementation shows a final result of 1,047,401 cycles with an overall cycle count reduction of 2.4x versus the original ARM7 implementation.

Variations of Cascade parameters with only minor code modifications do not improve this result significantly.

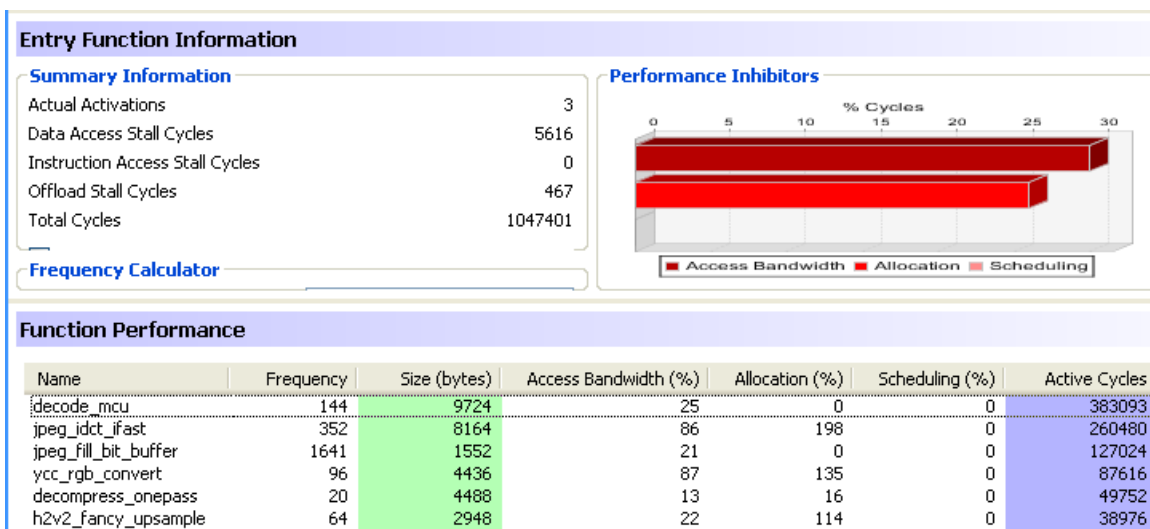


Figure 22: Integrated decoder implementation

## Going Further

The Huffman decoding function is the limiting factor in greater acceleration. Several approaches can be used to further improve the decoding performance.

### Custom Function Units

Custom function units could be added to speed up Huffman decoding. Using function units requires some code modification, but the impact of the custom function units can be measured within Cascade before the actual hardware is designed. The approach begins by conducting detailed profiling within the `decode_mcu()` function to identify control and datflow hotspots withing Cascade. Figure 23 shows a code analysis view inside Cascade which can be used to analyze detailed traces looking for custom acceleration functions. Once candidate sequences are identified, they can be refactored in the original code and Cascade can be rerun presuming these custom function units are available. Custom units can be added in this way until desired acceleration is reached, and only then do the custom function units need to be designed in RTL.

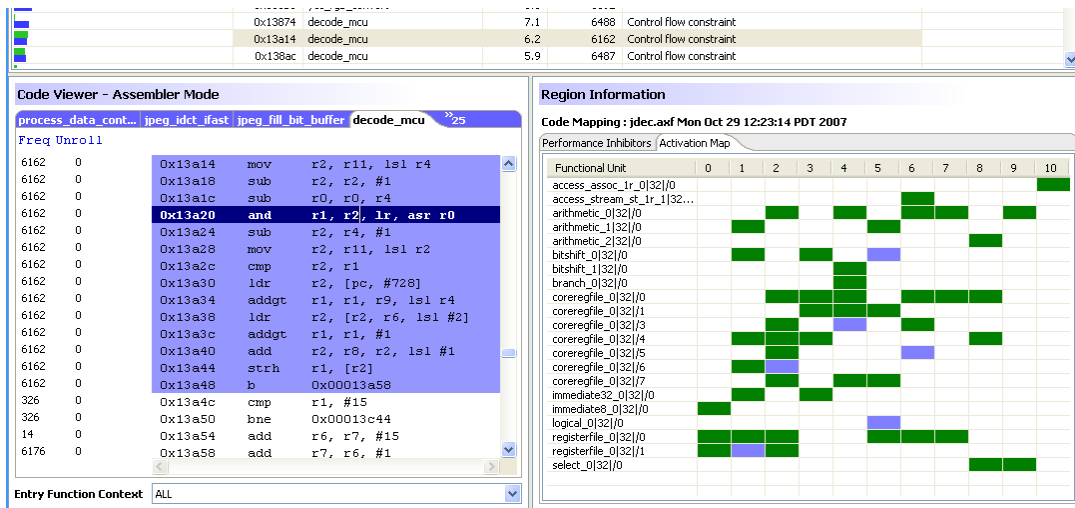


Figure 23: Code analysis view

### Pipelined JPEG Decoder Implementation

All previous implementations operate the coprocessor as a slave to the ARM7 processor. The slave interface simplifies the implementation, but the ARM core stalls while the coprocessor is operating. Additional cycles can be gained by operating the coprocessor as a master in parallel with the ARM.

A straightforward approach would pipeline MCU processing through Huffman decode, inverse DCT and color conversion operations. Since the Huffman decode is the limiting operation, an optimal pipeline balance will split the Huffman decode between the ARM and coprocessor. The Huffman decode is partitioned so that the time spent on the ARM in the front end of the Huffman decode matches the time spent in the back end Huffman and the accelerated inverse DCT and color conversion operations to create a balanced two stage pipeline.

This implementation requires significant rework of the reference code, and as such is outside the intent of this case study, but experiments suggest that a 4x speedup is achievable with this approach.

## Conclusions

Starting from open-source reference software, a series of Cascade runs developed a coprocessor which accelerated the decoder implementation by 2.4x over the original ARM7 implementation, or looked at another way, the coprocessor executes the same decompression using only 40% of the cycles required by the original ARM7 implementation.

Acceleration of over 7x was achieved on focused implementations of the inverse DCT and RGB conversion operations.

The overall JPEG decoder acceleration was limited to 2.4x by the serial nature of the Huffman decoder. With restructuring of the original open-source code, a pipelined implementation could increase the acceleration to over 4x.

*Code used in this case study is available by contacting CriticalBlue.*