ABSTRACT

BADRAN-LOUCA, SERENA. Low Power Techniques for Video Codec Motion Compensation. (Under the direction of Dr. Rhett Davis).

A major milestone in the evolution of video coding standards is the well-known H.264/MPEG-4 Advanced Video Coding (AVC). In January 2013 that was followed by H.265/MPEG-5 High-Efficiency Video Coding (HEVC). Both of these standards achieve substantial improvement in bit-rate efficiency compared to their predecessor and are the standard of choice in many application standards such as HD, DVD, HD-DTV.

With the proliferation of handheld devices, emphasis on reducing power consumption and increasing battery life is growing, thus making the improvement of power efficiency the main goal of any decoder implementation. However, most of the current power solutions focus solely on reducing memory accesses, the largest power drain. Other aspects that are not getting as much attention include memory access efficiency, memory power down, and pipeline efficiency. Power for these aspects can be reduced with new architectures for Memory Access Arbitration and Reference Data Scheduling. As shown in this work, using these techniques results in 87.67% off chip memory power reduction and reduces the number of memory accesses by 86.9% when the memory is power up.
Low Power Techniques for Video Codec Motion Compensation

by
Serena Badran-Louca

A dissertation submitted to the Graduate Faculty of North Carolina State University in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Computer Engineering

Raleigh, North Carolina
2015

APPROVED BY:

_______________________________  ______________________________
Rhett Davis                                    Paul Franzon
Committee Chair

__________________________  ____________________________
James Tuck                                  Harry Perros
DEDICATION

To Sana, Kassem, Tony, Stefano and Rafaella. You are my everything.

To Grandma Koubayssi, forever in my heart.
BIOGRAPHY

Serena Badran-Louca was born in Beirut, Lebanon. She earned her Bachelor’s degree in Electrical Engineering from the American University of Beirut in June 1997. She started a coop position in March 1998 with ABB Electric Systems Technology Institute on the Centennial Campus of North Carolina State University and consequently obtained a Masters Degree in Computer Engineering from NCSU in Dec 2000. She started her career as an ASIC Engineer with AMCC in Jan 2001 until July 2002. After that she returned to NCSU and started taking Non-Degree courses in the ECE department until officially enrolling in the Ph.D program in March 2004. At the same time, she started a full time position at Qualcomm Incorp as a Hardware Engineer. She continues to work at Qualcomm as a logic designer in the processor group.

Serena is married to Tony Louca, a fellow AUB/NCSU alumni. They have a four and half year old son, Stefano and a two year old daughter, Rafaella.
ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor Dr. Rhett Davis for his technical and personal mentoring. His positive and encouraging attitude really helped push me against all the odds. His constructive supervision was a source of encouragement and helped me not lose sight of my goals. I would also like to thank all my professors at NC State especially during my Masters degree. Their expertise and passion for teaching is very inspiring.

I would also like to thank my coworkers and managers at Qualcomm Inc. for their support, especially during the early years of my PhD. I am proud to work at a company that values the continuing education of their employees so much.

To my parents, I thank you for all the sacrifices you made to make sure I had a good education. You set a perfect example for working hard and being professional and passionate about what you do. Thanks for all your love and support throughout the years, without it I wouldn’t have been able to accomplish much.

Last and certainly not least, to my amazing husband Tony. This dissertation would not have been possible without your help and support. It has been a long and frustrating road, but you were with me every step of the way. From brainstorming for ideas, to helping me in my research and providing inspiring discussions, you were always there even after long days at work. You believed in me and supported me unconditionally and for that I am eternally indebted to you.
LIST OF TABLES ................................................................................................................... vi
LIST OF FIGURES ................................................................................................................ vii

CHAPTER 1: Introduction ...................................................................................................... 1
  1.1. Background .................................................................................................................... 1
  1.2. Motivation ...................................................................................................................... 5
  1.3. Review of Present Status ............................................................................................... 8
    1.3.1. Reducing reference data accesses .............................................................................. 8
    1.3.2. Pipeline Efficiency ..................................................................................................... 9
  1.4. Proposed Techniques ..................................................................................................... 9
    1.4.1. Memory Access Arbitration ..................................................................................... 10
    1.4.2. Reference Data Scheduling ...................................................................................... 10

CHAPTER 2: Power Reduction Techniques ......................................................................... 11
  2.1. Memory Access Arbitration ........................................................................................... 11
  2.2. Reference Data Scheduling .......................................................................................... 19
    2.2.1. Reference Pixel Re-use ............................................................................................ 19
    2.2.2. Reference Data Prefetching ...................................................................................... 21

CHAPTER 3: System Model ................................................................................................. 24
  3.1. Model Architecture ...................................................................................................... 24
    3.1.1. Motion Compensator .............................................................................................. 25
    3.1.2. Deblocking filter .................................................................................................... 26
    3.1.3. Prefetcher ............................................................................................................... 27
    3.1.4. Arbiter .................................................................................................................... 28
    3.1.5. Cache Manager ......................................................................................................... 34
    3.1.6. Pixel Cache ............................................................................................................... 36
    3.1.7. Memory .................................................................................................................... 37
  3.2. Model Interfaces .......................................................................................................... 37
    3.2.1. Arbiter .................................................................................................................... 38
    3.2.2. Cache Manager ......................................................................................................... 38
    Table 4: Cache Manager Interfaces ........................................................................................ 39
    3.2.3. Pixel Cache ............................................................................................................... 39

CHAPTER 4: Experimental Results ...................................................................................... 40
  4.1. Model Assumptions ..................................................................................................... 40
  4.2. Memory Access Arbitration ........................................................................................... 44
  4.3. Reference Data Scheduling .......................................................................................... 47

CHAPTER 6: Conclusion ...................................................................................................... 53

III. REFERENCES ............................................................................................................ 54
IV. APPENDICES ............................................................................................................. 57
LIST OF TABLES

Table 1: Memory access ratio of H.264 decoder modules (W: frame width, H: frame height) 5
Table 2: Interpolation filters and reference data size of one MxN Luma partition .......... 19
Table 3: Arbiter Interfaces ............................................................................................. 38
Table 4: Cache Manager Interfaces ................................................................................ 39
Table 5: Pixel Cache Interfaces ...................................................................................... 39
Table 6: Coding methods for syntax elements ............................................................... 60
LIST OF FIGURES

Figure 1: Partitioning of macroblock & sub-macroblock for motion compensated prediction 2
Figure 2: H.264 Encoder ........................................................................................................... 3
Figure 3: H.264 Decoder ......................................................................................................... 3
Figure 4: Power profile of H.264 system .................................................................................. 5
Figure 5: Full pixel samples (shaded and with upper case letters) and sub-pixel sample positions (unshaded with lower case letters) for quarter sample luma interpolation .......... 19
Figure 6: Horizontal and vertical data reuse across 4x4 sub-partitions .................................. 20
Figure 7: Read pending queue. ............................................................................................... 22
Figure 8: System model top level diagram ............................................................................ 24
Figure 9: Arbiter details .......................................................................................................... 29
Figure 10: Foreman frame showing partition choices. ........................................................... 41
Figure 11: Memory Access Arbitration Results ..................................................................... 46
Figure 12: Motion Compensator Hits/Misses in the Cache vs. Cache size ............................ 47
Figure 13: Motion Compensator hit rate in the Cache vs. Cache Size. ................................. 48
Figure 14: System Power Reduction vs. Cache Size ............................................................ 49
Figure 15: Cache Cost vs. Size ............................................................................................... 50
Figure 16: System Cost vs. Cache Size .................................................................................. 51
Figure 17: Motion Compensator Hits in the Cache ................................................................. 51
Figure 18: Matrices of the three different transforms applied in H.264 ................................. 61
Figure 19: Partitioning of macroblock and sub-macroblock for motion compensated prediction. ............................................................................................................................... 62
CHAPTER 1: Introduction

Nearly every mobile computing device today contains the ability to display streaming video. Specialized video CODECs (enC Oder/DECoder pair) are an expected component in most media processors, and the power consumption of these CODECs have a large impact on the battery life and customer satisfaction with their smart-phones and tablets. This work seeks to develop new techniques to reduce the power consumption of video CODECs and extend the battery life of these mobile devices by focusing on the largest contributor off-chip memory power.

1.1. Background

The key innovations presented in this work involve better organization of how memory is accessed during video decoding. These innovations will be discussed using the standard terminology for video memory organization. A digitized video signal consists of a periodical sequence of images called frame. Each frame consists of a two dimensional array of pixels. H.264 is a block based coding standard that divides a frame into 16x16 pixels macroblocks (MBs) and uses variable block size motion estimation (VBSME) which allows blocks to be estimated independently using different motion vectors. Figure 1 shows the block hierarchy of the H.264 standard where MBs could be divided into partitions (16x16, 16x8, 8x16, and 8x8) that in turn could be divided into sub-partitions (8x8, 8x4, 4x8, and 4x4).
Figure 1: Partitioning of macroblock & sub-macroblock for motion compensated prediction

Figure 2 is the block diagram of the H.264 encoder. In the encoder, a decoder path like the one in Figure 3 is used. Like previous ISO/IEC video coding standards, only the decoder and not the CODEC (enCOder/DECoder pair) is standardized. By restricting the bit stream and syntax, and by defining the decoding process, all standard decoders will produce similar output given an encoded bit stream conforming to the standard constraints. Such a limitation of the scope permits implementation freedom in a manner appropriate to the application.
The input to the decoder is a compressed bit stream delivered via the NAL (Network Abstraction Layer). The stream is entropy decoded then reordered to produce a set of quantization coefficients. The coefficients are then rescaled and inverse transformed and this value is added to the prediction (inter or intra) based on previously encoded frames.
There are two types of prediction P, intra prediction and inter prediction. Intra prediction exploits the spatial redundancy among neighboring blocks in a video frame whereas inter prediction exploits the temporal redundancy of a video sequence. Finally a filter is used to eliminate the blocking artifact that is a disadvantage of block-based video coding. This deblocking filter helps generate a smoother picture.

Appendix A provides a more detailed description of each sub-block of the H.264 decoder.
1.2. Motivation

The following figure shows the power profile of the H.264 decoder. The motion compensation engine and the off chip memory, with the highest power percentages, are clearly the ideal candidates to focus on for a low power solution.

![Figure 4: Power profile of H.264 system [5]](image)

Table 1 shows the breakdown of memory accesses among various blocks in the H.264 decoder. Again, the motion compensation engine is responsible for the biggest portion of memory accesses.

<table>
<thead>
<tr>
<th>Module</th>
<th>Max memory access bytes</th>
<th>Ratio (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ref. picture store</td>
<td>(W<em>H + 2</em>(W/2)*(H/2))</td>
<td>10%</td>
</tr>
<tr>
<td>De-blocking</td>
<td>((W/16)<em>(H/16-1)</em>(16<em>4+4</em>4*2)*2)</td>
<td>5%</td>
</tr>
<tr>
<td>Display feeder</td>
<td>(W<em>H + 2</em>(W/2)*(H/2))</td>
<td>10%</td>
</tr>
<tr>
<td>Motion compensation</td>
<td>((W/16)<em>(H/16)<em>16</em>(9</em>9+2<em>3</em>3)*2)</td>
<td>75%</td>
</tr>
</tbody>
</table>
One interesting observation from Table 1 is that, even though the de-blocking filter and the motion compensator are touching the same number of unique pixels (same frame size), motion compensation has 15X the memory accesses of the de-blocking filter. This is because the interpolation filter reads the same reference pixel multiple times: when the pixel is interpolated and when the pixel is in the neighborhood of another pixel being interpolated.

Current techniques being used to reduce memory accesses highlighted above include:

- Only read the neighboring pixels needed for interpolation based on the motion vector [7].
- Cache neighboring reference pixels locally to prevent reading them again while interpolating the next sub-partition [9].

These techniques, while reducing the number of memory accesses, do not take advantage of some opportunities that would offer additional power savings.

- Memory power down: When the required memory bandwidth is less than the available bandwidth, memory devices are under-utilized. When a memory device like SDRAM is not being accessed, it could be put into a power down mode like self-refresh where the external clock and IOs (input sensors and output drivers) are turned off and internal clock is shut off to most of the logic. Only a small portion of the logic will be running to preserve the data stored in memory. Self-refresh mode can use < 1% power than the standard full ON mode [18]. There is a latency cost, however, for self-refresh entry and exit [19] mainly due to external clock startup and stabilization and waiting for a specified time before the first page activation. This requires memory accesses to be organized in bursts with relatively long stretches of idle periods. An arbitration scheme would bunch up the requests from various clients to achieve this goal.
• Reference Pixel Reuse: Depending on the sub-partition motion vector, the motion compensator can require up to 6 neighboring pixels to estimate one pixel in the new frame. Using a local cache and allowing reference pixel caching across MBs boundary we can make sure that every reference pixel is not read more than once from memory thus bounding the number of memory accesses to the reference frame size. According to Table 1 this would reduce the motion compensator memory reads by up to a factor of 15.

• Prefetching reference data: Different sub-partitions estimated by the motion compensator could have different motion vectors requiring a different interpolation filter to be used. The filter pipeline can be made to re-configure on the fly so that it does have to flush the previous sub-partition data before starting on the new one. For this to be effective, reference data for the new sub-partition has to be available right when the last reference pixel of the previous sub-partition goes into the filter pipeline. This could be achieved by prefetching reference data from the next sub-partition ahead of time. Also, by doing this, external memory could be put in power down mode for a longer period before the motion compensator runs out of reference data.

The absence of solutions that take advantage of these opportunities is the key motivation behind this research. Subsequent sections will describe in details techniques to lower off-chip memory power by exploring the three opportunities listed above. Next, a cycle accurate SystemC model that implements these techniques will be presented. Finally, the results from the model will illustrate the power savings from these techniques.
1.3. Review of Present Status

Researchers have attempted to tackle the problem of reducing memory accesses using approaches which can be categorized as follows:

1.3.1. Reducing reference data accesses

According to the H.264 specification, the sub-pixel filter in the motion compensation requires, based on the motion vector, up to six reference pixels to be read in order to produce one sub-pixel. These reference pixels might come from neighboring sub-partitions which also need to be read from off chip memory. By default, all neighboring reference pixels are read to perform interpolation, a total of \((M+5) \times (N+5)\) pixels. However, the interpolated sub-pixel type, and therefore the interpolation filter used, can be determined from the motion vector. [7], [13] and [14] show how this information can be used to only read the neighboring pixels needed by the interpolation filter in question.

Even though this technique reduces the number of memory accesses it does not provide the opportunity of idling memory long enough to be powered down. In other words, even though the number of requests is reduced they might still be sparse enough to require memory to remain powered on. Alternatively, a dedicated memory arbiter could bunch up requests allowing for memory to be powered down between bursts. This is the premise of the first technique in this research; Memory Access Arbitration.
1.3.2. Pipeline Efficiency

The interpolation of pixels in a sub-partition requires the use of pixels from neighboring sub-partitions. These pixels will then be re-used when interpolating these neighboring sub-partitions. To minimize memory accesses, researchers use a cache to store the neighboring pixels locally to prevent reading them while interpolating the next sub-partition as shown in [9]. Two new observations in this work extend the concept from [9] to increase memory and pipeline efficiency:

- The caching concept may be extended beyond the macroblock boundary. The idea is that once you move from one macroblock to another you could use cached reference pixels just as you would within a macroblock.

- When transitioning from one sub-partition to another, we could eliminate pipeline stalls waiting for the filter to finish with the current sub-partition until starting on the next one. This could be achieved by prefetching reference data for the next sub-partition ahead of time and switching the filter pipeline on the fly to start processing the next sub-partition. Note that the next sub-partition could potentially have a different motion vector which means that the filter pipeline needs to be modified to handle different motion vectors simultaneously.

The above extensions would allow reference pixel data to be always readily available to the interpolator and increase pipeline efficiency. This is the essence of the second technique in this research; Reference Data Scheduling.

1.4. Proposed Techniques

The increasing demand in wireless and handheld multimedia devices has put the power footprint of any decoder in the forefront. Consequently, the main goal of this research is to reduce the power dissipation in an H.264/H.265 decoder. The following two objectives work toward achieving this goal:
1.4.1. Memory Access Arbitration

Reducing memory accesses reduces memory utilization and creates an opportunity to power down memory devices when not in use. By bunching up memory requests, active memory request scheduling guarantees idle periods and eliminates the latency penalty of memory power down events. The request scheduling is done by the memory arbiter described in section 4.1.4 and has the following assumptions:

- The total required memory bandwidth is a fraction of the available bandwidth. This might be a low performance scenario.
- The two clients accessing memory, the de-blocking filter and the motion compensator, have built-in latency tolerance without affecting performance.

The result is reduction of power dissipated by external memory devices with noticeable impact on performance.

1.4.2. Reference Data Scheduling

Eliminating pipeline stalls helps reduce power dissipation by increasing the amount of time the logic could be power gated or clock gated. To that end, surveyed research made heavy use of reference data caching to reduce stalls due to memory fetches. The following techniques offer additional reduction of pipeline stalls:

- Extend caching to re-use pixels across macro blocks instead of limiting it to sub-partition inside the macro block.
- When switching to a new sub-partition, use the new motion vector to prefetch reference data ahead of time.

Applying these techniques would reduce the time it takes to interpolate a given frame thus allowing a bigger opportunity for power down and clock gating.
CHAPTER 2: Power Reduction Techniques

2.1. Memory Access Arbitration

For a given configuration, if \( B \) is the total available memory bandwidth and \( X \) is the memory bandwidth required by the motion compensator, then theoretically, we should be able to power down the memory modules \((B-X)/B\) of the time. To get close to this power down ratio, a memory arbiter would take requests from the motion compensator and the deblocking filter and issue them to the memory devices. The memory arbiter would have the following specifications:

1) Batching up memory requests and issuing them in a bursty manner while allowing the memory modules to be put in power down mode in between bursts. The arbiter would have programmable low and high watermarks. When the number of batched requests exceeds the high watermark, the arbiter will start issuing the requests until the number of queued requests reaches the low watermark.

2) Memory power down causes the motion compensator to see larger memory read latency than normal. This would require the compensator to implement latency tolerance by prefetching data in advance and storing it in a latency hiding buffer. When the data in the buffer runs low, the predictor can signal the arbiter that it is in critical need of data which the arbiter will try to service as soon as possible.

3) To ensure that the write stream coming from the deblocking filter is also bursty, a coalescing buffer will be used to accumulate the writes. This buffer will also implement a high and low watermarks policy to decide when to flush the buffer.

4) Since memory devices are not efficient when read and write accesses are finely interleaved, the memory arbiter will bunch these accesses to minimize overhead.

The specification for this idea can be divided into the following three concepts:

A. Sizing the latency buffer for motion compensation reads
B. Sizing the coalescing buffer for blocking filter writes

C. Arbitration Policy and Assumptions

Here we present calculations to determine the requirements for each of these aspects. This is a feasibility analysis to determine that these requirements can be realistically met in an energy-saving CODEC architecture.

A. Sizing the latency buffer for motion compensation access

Latency Buffer (LB) Size = Read Bandwidth (RBW) \* Read Latency (RL)

Ideally we want to design the smallest latency buffer to support the highest performance configuration, so:

\[\text{Min LB size} = \text{Max RBW} \times \text{Min RL}\]

i. Read Latency (RL)

\[RL = \text{Arbiter holdoff} + \text{Memory Pwr up} + \text{Read return latency}\]

Where

- Arbiter holdoff is the time between the arbiter seeing the read request and acting on it.
- Memory power up is the time between the arbiter deciding to power up memory until it can issue the read request.
- Read return latency is the time between issuing the read request and the read data arriving at the input of the latency buffer.

Since we are using the calculation of RL to size the latency buffer, we need to look for the worst case latency. This should include programmable knobs like arbiter holdoff. Furthermore, the time between issuing the read and when the read data actually returns is only a handful of clocks, thus the equation for RL reduces to:

\[RL = \text{Memory Pwr Up}\]

which is memory dependent, so for DDR2-533, mem pwr up from self refresh is 200tck = 750 ns
ii. \textit{Bandwidth (BW)}

BW is dependent on:

- Num. of frames/sec
- Size of frames
- The sampling format (4:2:2, 4:4:4...)
- Number of reference frames needed to predict current frame

The read BW for motion compensation can be summarized as follows:

I.

\[
BW_{MC} = \text{Num of frames/sec} \times \left( \frac{f \times \text{# lines} \times \text{#luma/line}}{\text{where } f = 3 \ (4:4:4) \ 2 \ (4:2:2) \ 3/2 \ (4:2:0)} \right) \times \text{Num of luma and chroma samples/frame} \times \text{Num of bits/sample} \times \text{Num of Reference Frame(s)}
\]

The write BW for the deblocking filter is the same as the read BW for the motion compensation when the number of reference frames is one. This is because the deblocking filter need only to write one frame at a time.

\[
BW_{Filter} = \frac{BW_{MC}}{\# \text{ reference frames}}
\]

The key assumption here is that the motion compensator is able to re-use pixels across the whole frame and no reference pixel is read twice. This is why the Reference Data Scheduling technique is important.

The total BW for the H.264 decoder can be described as follows.

\[
\text{Total BW} = BW_{MC} + BW_{Filter} = (1 + \# \text{ reference frames}) \times \frac{f \times \text{# lines} \times \text{#luma/line}}{\text{where } f = 3 \ (4:4:4) \ 2 \ (4:2:2) \ 3/2 \ (4:2:0)} \times \text{frames/sec} \times \text{frames/sec} \times \text{#bits /sample}
\]
So for 1080p: 

\[
(1+ \# \text{reference frames}) \times 25 \times 2 \times 1920 \times 1080 \times 8 \\
= (1+ \# \text{reference frames}) \times 400 \times 1920 \times 1080
\]

Worst case (B macroblock, bi predicted) => \# reference frames = 2

=> Total BW for 1080p (worst case) = 296 MB/s

Assuming

\[
\begin{align*}
64\text{b memory} &= 8B \\
\text{DDR3 800MHz} & \quad 70\% \text{efficiency} \\
\end{align*}
\]

=> 8B \times 800 \times 0.7 = 4.480MB/s = 4.48GB/s

Total available BW from a fast memory configuration is:

\[
\begin{align*}
32\text{b memory} &= 4B \\
\text{DDR2 667MHz} & \quad 70\% \text{efficiency} \\
\end{align*}
\]

=> 1.86GB/s

Total available BW from a slow memory configuration is:

\[
\begin{align*}
32\text{b memory} &= 4B \\
\text{DDR2 400MHz} & \quad 70\% \text{efficiency} \\
\end{align*}
\]

=> 1.12GB/s

From all the above numbers we can conclude that we have room for memory power down.

\[
\text{Min LB} = \text{Max BW} \times \text{Mem PwrUp} \\
= 296\text{MB/s} \times 200 \text{tck} \\
= 222\text{Bytes}
\]
222Bytes is pretty small, so we have plenty of room in the LB size. However we should size the buffer based on a “typical” memory read latency. This typical memory is somewhere between the “best” and “worst” case latency.

**Best Case Latency**

a. Memory is already powered up  
b. The read is to a page that is already open  
c. The bus is not being used to drain the write buffer (for the deblocking filter accesses)

**Average Case Latency**

a. Memory is already powered up  
b. The read is not to a page that is already open  
c. The bus is being used to drain the write buffer (for the deblocking filter accesses)

**Worst Case Latency**

a. Memory is not powered up  
b. The read is not to a page that is already open  
c. The bus is being used to drain the write buffer (for the deblocking filter accesses)

Note that the worst case latency could be reduced if the decoder pipeline provided an early “Need to Read” signal which will hide some of the power up latency. What this means is that if for some reason the actual latency turned out to be at our typical latency, then the motion compensator will stutter. However, this should be rare due to early power up and some arbiter optimizations. Thus, using typical latency to size the buffer should be a reasonable compromise to save power and will be verified by simulation.

⇒ Average latency is a function of the following:

1. The read is not to a page that is already open: We need to issue a page management command before issuing a read. From the DDR2 we can get a rough estimate of this factor. The worst case
value is when a targeted bank has a page operation such that we need to close an open page and open a new one.

2. The bus is being used to drain the write buffer: Assume we can interrupt the write stream and then add some guard band to account for letting writes go so that we don’t block the filter.

Therefore latency = precharge \rightarrow activate \rightarrow read \rightarrow data

\[ \text{Latency} = t_{rp} + t_{rd} + t_{cl} \]

- DDR2-800E = 15 + 15 + 15 = 45ns
- DDR2-667D = 15 + 15 + (5x300) = 45ns
- DDR2-533C = 15 + 15 + (4x3.78) = 45 ns

So typical latency = 45 ns (this is from pin to pin)

\[ \Rightarrow \text{Latency Buffer Size} = 296 \text{ MB/s} \times 45\text{ns} = 13B \]

The fact the buffer need to cover memory latency is small is good news. This means that we could increase the latency tolerance of the motion compensator at a relatively low cost which would have two advantages:

1- Keep memory powered down for longer stretches of time and exit power down mode less frequently.

2- Allow bigger batches of write requests which would increase the memory bus efficiency by having less turnaround time.
B. Sizing the coalescing buffer for blocking filter access

Requirements for the write buffer:

- Needs to be deep enough to absorb deblocking filter writes at the frame rate * frame size for the duration required to keep the motion compensation from stuttering.
- Coherency. Make sure that frames that are being read by the motion estimator are flushed first if it is in the write buffer.
- Need to provide an urgency indicator to the arbiter when close to full.

The depth of write buffer affects the read latency and the BW efficiency. A write buffer would allow the memory arbiter to bunch writes and reads and prevent costly turn around time on the memory bus. It also reduces the chance of a read burst getting interrupted to serve writes.

In a similar fashion to our calculation of the latency buffer for the motion compensator,

\[ BW_{\text{filter}} = 400 \times 1920 \times 1080 \]

\[ = 98\text{MB/s} \]

Another configurable knob is how long to hold off writes, either due to memory power down or because the arbiter is doing reads, before issuing them to memory. Assume we block for T secs,

\[ \Rightarrow 98\text{MB/s} \times T \text{ sec} \Rightarrow 98T \text{ MB} \]

i.e. DDR2-666 where tck = 3ns, write buffer = 98 * 1.5 B
C. Memory Arbiter

There are two key points:

i. Arbitration policy

The arbitration policy is configurable via the following knobs:

- Request bandwidth: Each client will have a programmable request bandwidth knob which tells the arbiter how many requests this client needs to issue before it can switch to the other client.
- Sticky timer: Each client will have a programmable timer which tells the arbiter how many cycles to park on a client that has not satisfied its request bandwidth but has not request before it can switch to the other client.

We’re assuming just 2 core clients, the motion compensator (C1) and the filter (C2). So basically switch from C1 to C2 if:

- C1 met its BW sharing & (C2 has a request || C1 didn’t have a request for X1 cycles)
- C1 did not meet its BW sharing && (C2 has a request & C1 did not have a request for X1 cycles)

where X1 is the number of cycles to wait for C1 requests before bailing out on C1.

i. Power up/down

- Self refresh
- Active power down (check the spec on semantics & how each mode is entered and exited)
- Entry and Exit Criteria:
  - Entry: Hierarchical programmable counter
  - Exit: Early signals from clients
2.2. Reference Data Scheduling

2.2.1. Reference Pixel Re-use

In motion compensated prediction, based on the interpolation type (see Table 2 below), reference pixels from neighboring sub-partitions might be used when interpolating pixels in the current sub-partition.

<table>
<thead>
<tr>
<th>Interpolation position</th>
<th>Interpolation filters</th>
<th>Minimized reference data size</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>No</td>
<td>MxN</td>
</tr>
<tr>
<td>a,b,c</td>
<td>6-tap horizontal</td>
<td>Mx(N+5)</td>
</tr>
<tr>
<td>d,h,n</td>
<td>6-tap vertical</td>
<td>(M+5)xN</td>
</tr>
<tr>
<td>e,f,g,i,j,k,p,q,r</td>
<td>6x6 tap filter</td>
<td>(M+5)x(N+5)</td>
</tr>
</tbody>
</table>

Figure 5: Full pixel samples (shaded and with upper case letters) and sub-pixel sample positions (unshaded with lower case letters) for quarter sample luma interpolation [12]
Table 2 shows how to determine the interpolation type from the motion vector. Example:

Given reference pixel $G$ with a motion vector $(\frac{1}{2}, \frac{1}{2})$, the predicted pixel according to Figure 5 would be pixel $j$. Now, according to Table 2, predicting pixel $j$ requires a 6x6 tap filter and $(M+5) \times (N+5)$ reference data.

Figure 6: Horizontal and vertical data reuse across 4x4 sub-partitions

- Figure 6 shows how pixels that were read to predict one sub-partition can be re-used when predicting the adjacent sub-partition. In this case, the motion vector for sub-partition A moves the pixels by $\frac{1}{4}$ or $\frac{1}{2}$ pixels. This means that reference pixels from sub-partition B will be used when interpolating sub-partition A pixels. When predicting sub-partition B,
some of the same reference pixels used for A will also be used when interpolating sub-
partition B pixels.

- The idea here is to keep the common pixels around to prevent reading them twice and
  increase memory access efficiency. A key challenge is when the two neighboring sub-
  partition A and B have different sizes and/or motion vectors. In this case, the prediction
  logic needs to be smart enough to know how to re-position the re-used pixels in the input
  buffer to the interpolation logic.

The pixel reuse logic determines whether an entry read from the reference pixel cache
needs to be marked for reuse. At a high level, it takes as input from the motion
compensator:

- Current sub partition bounding box + motion vector
- Next sub partition bounding box + motion vector

2.2.2. Reference Data Prefetching

When the motion compensated prediction moves from one sub-partition to the next, the motion
vector value might change which would change the type of the interpolation filter used, what
reference pixels are pushed through the filter, and the crossbar configuration.

During the sub-partition switch, our goal is for the pixel cache and the crossbar to feed the
interpolator with data every clock cycle with no stuttering. While the current sub-partition is
being interpolated, a prefetcher will look ahead at the next sub-partition and, based on the
motion vector, determines what reference data needs to be read from memory. It will prefetch
the data into the pixel cache so that it is ready for interpolation.

The prefetcher tries to read reference pixels needed to interpolate future sub partitions. It will
exclude reference pixels needed to interpolate current sub-partitions. At a high level, it takes
as input from the motion compensator:
• Current sub partition bounding box + motion vector
• Next N sub partitions bounding box + motion vectors.

For these 2 concepts of pixel reuse and data prefetching, a pixel cache is needed and that is managed by a Cache Manager. The Cache Manager interfaces to the memory arbiter and the reference Pixel Cache and its main function is:

1. Keeps track of what lines are in the cache so that we can dismiss MC reads and prevent them from being issued to memory.
2. Makes sure that data is read from the cache in the same order that the MC issued the read request.
3. Manages the Read Pending Queue and Cache Lookup Table structures.
   a. Read Pending Queue:
      i. Contains entry for all requests issued to memory
      ii. Contains a list of all MC reads (issued + dismissed)
      iii. Indicates whether read data is available (hit in the cache or coming from memory)

   ![](Read Pending Queue.png)
   
   **Figure 7: Read pending queue.**

   b. Cache lookup table. Each table entry has:
      i. Line Address
      ii. State:
1. Empty
2. MC Pending
3. MC Prefetch Pending
4. Prefetch
5. Reuse
   iii. Age counter
4. Invalidates lines on writes
5. Implements the following cache entry replacement policy

   if (MC request)
   
   grab an empty line
   
   else grab a prefetched line (FIFO)
   
   else grab a reused line (FILO)
   
   else hold off the request (bad – means the latency buffer sizing was incorrect)

   else if (Prefetch request)
   
   grab an empty line
   
   else grab a prefetched line (FIFO) or hold off request based on input from prefetcher to prevent trashing.

Whenever a line is read out from the pixel cache it is marked as either a reuse line (based on input from the reuse block) or empty.
CHAPTER 3: System Model

3.1. Model Architecture

The following figure shows the high level block diagram of the system model used to verify the theory behind these techniques. All blocks in this figure are implemented using SystemC classes. The prefetcher, motion compensator and deblocking filter will issue read or write requests to memory. The arbiter will select between these requests, snoop and update the cache, and issue requests to memory. The cache manager implements the tag cache lookup along with the caching policies. The pixel cache handles memory read return data as well as providing the motion compensator with read data.

Figure 8: System model top level diagram.
3.1.1. Motion Compensator

The motion compensator class instantiates a traffic generator class which implements the following functionality:

a. Reads a sequence of blocks from a user file. The file contains a list of different size macro blocks for the motion compensator to read from memory. A batch of macro blocks forms a sub partition. At the end of a sub partition, the user file will have a command to pass the sub-partition to the deblocking filter which will write it to memory.

b. Before passing them to the motion compensator, the generator expands macro blocks to include neighboring reference pixels. The number of neighboring pixels included depends on the motion vector for this macro block.

c. After expanding a macro block, the generator will dice it into 2x2 pixel regions (quads) each with a corresponding memory address.

d. The generator will go over all the macro blocks specified in the user file until it reaches the end of the block sequence.

Here is a snippet of a user file describing a block sequence:

```
IMAGE WIDTH : 256  # Image dimensions used for address generation.
IMAGE HEIGHT : 256
0 : 0,0 7,7  # Upper left and lower right coordinates of macro block to read from estimate.
1 : 0,0 7,7  # Write command for passing to the deblocking filter.
0 : 8,0 11,3
0 : 12,0 15,3  # Macro blocks forming a Sub-partition.
0 : 8,4 11,7
0 : 12,4 15,7
1 : 8,0 15,7
```
The motion compensator will issue one read request to memory for each quad of pixels it receives from the traffic generator. It receives quad data, in order, from the pixel cache which it processes at a configurable rate. The motion compensator can have a certain number of read requests in flight that it cannot exceed. A read request is retired after the corresponding response has been received from the pixel cache and processed.

The motion compensator is responsible for indicating to arbiter, via the mc_arb_urgent signal, when it is running low on read data. The arbiter uses this information to prioritize motion compensator reads over other memory clients and to power up memory if needed. The motion compensator derives its urgency in the following manner:

a. The mc_arb_urgent signal is set high (urgent) when the number quad data in the input buffer goes below a configurable low watermark.
b. The mc_arb_urgent signal is set low (normal) when the number of quad data in the input buffer goes above a configurable high watermark.

Finally, the motion compensator is responsible for conveying a sub-partition to the deblocking filter after issuing read requests for all the macro blocks in that sub partition.

### 3.1.2. Debloking filter

The deblocking filter is responsible for removing blocking artifacts from a sub-partition that has been estimated by the motion compensator. It accepts “blocks” from the motion compensator which are defined by their upper left and lower right pixel coordinated.

```c
typedef enum
{   BLOCK_RD = 0,
    BLOCK_WR = 1
} block_type;

typedef struct
{
```
typedef struct {
    location up_left;
    location lo_right;
    block_type type;
} block;

The filter will process the block and write it to memory one quad at a time at a configurable rate. The filter is responsible for computing the memory address of every quad using the same parameters and computations as the motion compensator’s traffic generator.

Finally, the write data buffer for the filter lives in the arbiter class. As described later, the arbiter, based on the write buffer occupancy, will derive the urgency of the filter’s memory write requests.

3.1.3. Prefetcher

The prefetcher’s role is to look ahead in the block sequence and read reference pixels for macro blocks that the motion compensator has not started processing yet. The advantage of doing that is to reduce the motion compensator’s read latency (i.e. no stuttering) and, more importantly, minimize powering up memory as the motion compensator can be serviced directly from the cache.

The prefetcher will receive future macro blocks from the motion compensator’s traffic generator using the same “block” packet defined above. It will dice the block into quads, generate a memory address for every quad and issue a request to memory.

In the current implementation, the prefetcher does not get acknowledgement that its requests have been issued to memory or whether the response data has been received. As will be seen
later, the arbiter will service prefetch requests opportunistically when the memory is powered up.

Finally, the prefetcher issues it requests at a configurable rate which allows for an additional knob to reduce the prefetch request traffic.

3.1.4. Arbiter

The arbiter class will act as a consumer for the three memory requestors in the system. The arbiter will have an input fifo to buffer a certain number of requests from each requestor before it will start blocking new requests until a slot in the fifo becomes available.

The arbitration policy used by the arbiter is designed to meet to the following criteria:

1. Prioritize clients that are urgent in order to guarantee a certain performance level for the system.
2. Maximize the memory bus efficiency by reducing the read to write and write to read turnaround times.
3. Power down memory as aggressively as possibly while adhering to the above two criteria.

Figure 9 shows the fifos and logic blocks inside the arbiter that implement the above three aspects.
As described above, the motion compensator drives a mc_arb_urgent signal indicating its request urgency. Because the write data buffer for the deblocking filter lives in the arbiter, the filter_arb_urgent signal is generated locally. The filter_mc_urgent is derived in the following manner:
- The filter_arb_urgent signal is set high (urgent) when the number quad data in the write buffer goes above a configurable high watermark.
  
a. The filter_arb_urgent signal is set low (normal) when the number of quad data in the write buffer goes below a configurable low watermark.

Here is the pseudo code used to implement the arbiter’s policy:

```c
arb_pick_mc = FALSE;
arb_pick_filter = FALSE;
arb_pick_prefetcher = FALSE;
if (mc_arb_urgent)
    arb_pick_mc = TRUE;
else if (cur_filter_urgent)
    arb_pick_filter = TRUE;
else if (doing_reads && MC has requests)
    arb_pick_mc = TRUE;
else if (doing_reads && Prefetcher has requests)
    arb_pick_prefetcher = TRUE;
else if (doing_writes && filter has requests)
    arb_pick_filter = TRUE;
else if (MC has requests)
    arb_pick_mc = TRUE;
else if (filter has requests)
    arb_pick_filter = TRUE;
else if (Prefetcher has requests)
    arb_pick_prefetcher = TRUE;
```

Here is the pseudo code for the logic that is trying to minimize read to write turn around.

```c
if (arb_pick_mc || arb_pick_prefetcher)
```
num_back2back_reads++;  
num_back2back_writes=0;  
if (num_back2back_reads == ARB_MAX_NUM_B2B_READS)  
     num_back2back_reads=0;  
     doing_reads = FALSE;  
     doing_writes = TRUE;  
else  
     doing_reads = TRUE;  
     doing_writes = FALSE;  
else if (arb_pick_filter)  
     num_back2back_writes++;  
     num_back2back_reads=0;  
if (num_back2back_writes == ARB_MAX_NUM_B2B_WRITES)  
     num_back2back_writes=0;  
     doing_writes = FALSE;  
     doing_reads = TRUE;  
else  
     doing_writes = TRUE;  
     doing_reads = FALSE;  

Once the arbiter picks a winner from the three potential requestors, it will:
   a. Pop the request from the winner’s request fifo.
   b. Send the request to the cache manager to snoop the cache.
   c. Push the request into the Pending Address FiFo to wait for the hit/miss response from
      the cache manager.

The arbiter will generate a request to the cache manager indicating that it is about to issue a
read or a write request to memory. For read requests, the arbiter will wait for the cache manager
response before issuing the read. The response could be one of the following:
- Miss: The data corresponding to the request is not present in the pixel cache. The arbiter can go ahead and issue the read request to memory since the cache manager already reserved an entry for it.

- Hit: The data corresponding to the request is present in the pixel cache. The arbiter should dismiss the read request to memory since the cache manager already locked the corresponding entry in the pixel cache to prevent it from getting evicted.

Note that, for reads, the arbiter needs to tell the cache manager whether the request source is the motion compensator or the prefetcher. This is because the Pixel Cache will need to know whether to forward the response data to the motion compensator or keep it in the cache if the source was the prefetcher.

Also, to be able to stream reads to memory, the interface between the arbiter and the cache manager will be pipelined where the arbiter could have up to N requests outstanding before it could get the response for the oldest outstanding request. N is computed as the latency from the arbiter request to the cache manager’s response (assuming no blocking due to cache being full) divided by the rate of the arbiter requests.

For writes requests, the arbiter will send a request to the cache manager to invalidate the line if it is present in the cache. This is needed to make sure no stale data in the pixel cache are sent to the MC.

The cache manager response packets to the arbiter will go into the Cache Hit FiFo. Here’s the pseudo code of the logic used by the arbiter to drain the Pending Address and Cache Hit FiFos:

```python
if (Head of Pending Address FiFo is a write request &&
    Write data buffer has quad data)
    Pop an entry from the Pending Address Fifo;
    Take quad data from the write buffer.
    Issue write request to memory.
```
else if (Head of Pending Address Fifo is a read &&
    Cache Hit Fifo has a valid entry)
    if (cache response is a miss)
        Issue read request to memory
        Pop an entry from the Pending Address Fifo
        Pop an entry from the cache Hit Fifo

Note how read requests which cache response is a hit will be dropped inside the arbiter and not
issued to memory.

Finally, the following describes the logic implemented to put the memory in power down mode.

    if (mc_arb_urgent || filter_arb_urgent)
        if (memory_in_powerdown)
            Wake memory from powerdown mode.
            memory_in_powerdown = FALSE;
            memory_powerdown_counter = ARB_CYCLES_TO_PWRDN;
        else if (memory_powerdown_counter > 0)
            memory_powerdown_counter--;
        else if (memory_powerdown_counter == 0)
            Put memory in powerdown mode.
            memory_in_powerdown = TRUE;

It is important to note the following arbiter’s behavior when memory is powered down:

a. The arbiter policy would still pick a winner from the three clients’ request fifos.
b. If the Pending Address FiFo is not full, the arbiter will issue a cache manager snoop
   request.
c. If the entry at the head of the Pending Address Fifo is read and the corresponding cache
   manager’s response indicate that the read hit in the cache, then the read will be dropped.
3.1.5. Cache Manager

The cache manager class is responsible for managing the pixel cache, allocating cacheline(s) for read requests issued by the arbiter and responding to the arbiter with Miss/Hit messages. The pixel cache is a slave to the cache manager which will have a tag table to keep track of the state of all the cachelines.

For this purpose, the cache manager instantiates a cache model with the following characteristics:

1. 64 Byte cacheline size.
2. Tags are produced by a hash function based on the cacheline aligned address.
3. Replacement policy is Least Recently Used.

The LRU replacement policy is implemented using a doubly linked list of nodes representing cachelines in the cache.

- When an existing cacheline is referenced (hit in the cache), the exiting node is moved to the front of the list.
- When a new cacheline is allocated (miss in the cache), the newly created node is placed at the front of a linked list.
- If attempting to add a new node and all cachelines are busy, a node is removed from the rear of the list causing a cacheline to be freed up.

After performing the cache lookup on behalf of the arbiter, the cache manager can send one of the following commands to the pixel cache based on the request type (read vs. write), request source (MC vs. Prefetcher), and whether the request hit or miss in the cache:
a. **ALLOC**: This will instruct the pixel cache to await read return from memory and store the data in the cacheline index specified in the packet.

b. **EMIT**: This will instruct the pixel cache to send the data stored in the cacheline specified by the index in the message to the motion compensator.

c. **ALLOC_EMIT**: This will instruct the pixel cache to await read return from memory and store the data in the location indicated in the message. The data will also be sent to the pixel cache output.

d. **INVALIDATE**: Invalidate the cacheline specified by the index in the message. This instruction is not strictly needed since the cache manager, through the cache model, maintains the available tags in the cache and invalidating a cacheline is just a matter of removing the corresponding node from the cache model’s linked list.

Here is the pseudo code for the logic in the cache manager that is processing the arbiter’s requests:

```plaintext
If (Arbiter request fifo has a valid entry)
    Pop the arbiter request.
    Align to cacheline size and perform a tag lookup
    If (request is READ)
        Send hit response packet to arbiter
    If (request hits in cache)
        If (request is WRITE)
            Invalidate the corresponding cacheline.
            Send INVALIDATE to the Pixel Cache
        else if (source != Prefetcher)
            Send EMIT to the Pixel Cache
    Else
        If (request is READ)
            If (source == MC)
```
Send ALLOC_EMIT to Pixel Cache
Else
    // This is a Prefetcher request
    Send ALLOC to Pixel Cache

Note that upon a tag lookup, the cache model along with the hit information will return an index in the cache that represents:

- The location of an already occupied cacheline in the case of a hit.
- The location of an empty/reused cacheline in the case of a miss.

The cache manager will send the cacheline index along with the command to the Pixel Cache.

3.1.6. Pixel Cache

The pixel cache class implements the physical cache storage. It acts as a slave to the cache manager by receiving “commands” and executing them. It also receives response data from memory for read requests made by the arbiter.

The following pseudo code describes the logic in the Pixel cache that processes cache manager commands:

```python
if (cache manager command fifo has a valid request)
    if (command == EMIT)
        Send the cacheline specified by the command to the motion compensator.
    else if ((command==ALLOC) || (command==ALLOC_EMIT))
        Wait for memory response data to be available
        Store data in the cacheline specified
```
if (command==EMIT)
    Forward data to motion compensator
else if (command==INVALIDATE)
    // Do nothing

Pop the cache manager’s command.

3.1.7. Memory

The memory model uses is a SystemC class that performs two main functions:

1. When it receives read request from the arbiter, it forms the response data and forwards it to the pixel cache.
2. When it receives a write request from the arbiter, it drains the corresponding write data and drop both request and data on the floor.

3.2. Model Interfaces

Three approaches for communication between components were evaluated. The approach of using ports derived from sc_port and a fifo as a channel seemed a bit heavy weight and slow. The approach of using sockets and a generic TLM payload and transport interfaces offers much more than is needed for this project. The approach of using sc_port on requestor/producer and having the responder/consumer use sc_export seems to offer the best balance between functionality and speed. For example, in the case of the arbiter, it implements three sc_export input interfaces which will be bound to the three arbiter requestors. Also, I defined data structures to carry messages/requests for each interface. So one data structure will be defined for the messages between the requestors and the arbiter and one data structure will be defined
for each of the other interfaces in the system (arbiter->cache manager, cache manager <-> pixel cache, and arbiter -> memory).

### 3.2.1. Arbiter

The following are the arbiter’s interfaces to other blocks in the system.

<table>
<thead>
<tr>
<th>Table 3: Arbiter Interfaces</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
</tr>
<tr>
<td>CM implements</td>
</tr>
<tr>
<td>sc_export&lt;sc_fifo_in_if&lt;arb_req&gt; &gt; req</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Arbiter implements</td>
</tr>
<tr>
<td>sc_export&lt;sc_fifo_in_if&lt;cm_rsp&gt;</td>
</tr>
<tr>
<td>r</td>
</tr>
<tr>
<td>Memory implements</td>
</tr>
<tr>
<td>sc_export&lt;sc_fifo_out_if&lt;arb_req &gt;&gt; arb_mem_req</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Arbiter implements</td>
</tr>
<tr>
<td>sc_export&lt;sc_fifo_in_if&lt;client_req&gt;&gt; mc_arb_req</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Arbiter implements</td>
</tr>
<tr>
<td>sc_export&lt;sc_fifo_in_if&lt;client_req&gt;&gt; filter_arb_req</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Arbiter implements</td>
</tr>
<tr>
<td>sc_export&lt;sc_fifo_in_if&lt;client_req&gt;&gt; pref_arb_req</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Arbiter implements</td>
</tr>
</tbody>
</table>

### 3.2.2. Cache Manager

The following are the Cache Manager’s interfaces to other blocks in the system.
Table 4: Cache Manager Interfaces

<table>
<thead>
<tr>
<th>Type</th>
<th>Src</th>
<th>Dst</th>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC implements sc Export&lt;sc fifo in_if&lt;cm req&gt; &gt; cm pc req</td>
<td>Cache Manager</td>
<td>Pixel Cache</td>
<td>cm pc req.type</td>
<td>00 = ALLOC, 01 = EMIT, 10 = ALLOC_EMIT, 11 = INVALIDATE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cm pc req.index</td>
<td>Index in the cache to store and/or read the CM data.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cm pc req.subid</td>
<td>Subid used to tag read requests and responses.</td>
</tr>
</tbody>
</table>

3.2.3. Pixel Cache

The following are the Pixel Cache’s interfaces to other blocks in the system.

Table 5: Pixel Cache Interfaces

<table>
<thead>
<tr>
<th>Type</th>
<th>Src</th>
<th>Dst</th>
<th>Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC implements sc Export&lt;sc fifo out_if&lt;mem data&gt; &gt; mem pc data</td>
<td>Memory</td>
<td>Pixel cache</td>
<td>mem pc data.data</td>
<td>Read response data</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>mem pc data.cnt</td>
<td>Number of data transfers</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>mem pc data.src</td>
<td>Source of read request 1=MC, 2=Prefetcher</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>mem pc data.subid</td>
<td>Subid used to tag read requests and responses.</td>
</tr>
<tr>
<td>MC implements sc Export&lt;sc fifo out_if&lt;mem data&gt; &gt; pe mc data</td>
<td>Pixel Cache</td>
<td>Motion Comp.</td>
<td>pe mc data.data</td>
<td>Read response data</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>pe mc data.cnt</td>
<td>Number of data transfers</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>pe mc data.src</td>
<td>Source of read request 1=MC, 2=Prefetcher</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>pe mc data.subid</td>
<td>Subid used to tag read requests and responses.</td>
</tr>
</tbody>
</table>
CHAPTER 4: Experimental Results

4.1. Model Assumptions

To limit the implementation scope of the model and focus on producing data for analysis rather than building a full-fledged H.264 decoder, the following design assumptions were made.

1. Instead of capturing a read request stream from a real motion compensator, a traffic generator was built that allows the user to specify a custom macro block sequence. The user would describe the sequence one block at a time using:
   a. The block upper left and lower right coordinates.
   b. Indicating whether a block is a macro block for read or a sub-partition to be written by the deblocking filter.
   c. The height and width of the frame.

The traffic generator will take care of generating a sequence of read and write requests as well as computing the memory addresses.

The following figure shows the frame that is used to analyze the model behavior. It is taken from a frame of the popular “Foreman” QCIF clip [3]. For this work, we constrained the analysis to the red highlighted region.
This example frame is coded as a P slice frame with the choice of partitions overlaid on the frame. P slice frames use partitions with a single motion vector and a single reference frame. Note that the area chosen for the study has both macro blocks coded using a single 16x16 partition as well as smaller partitions around complicated area of movements like the mouth.

2. The motion compensator and the deblocking filter in the model are naïve with respect to their actual functionality in a read H.264 decoder. For example, rather than implementing the motion compensator pipeline in the model, a configurable parameter was used to
specify how many cycles it takes the motion compensator to consume a chunk of reference pixel data. Also, in reality, the number of neighboring pixels read when estimating a macro block depends on the motion vector received with the macro block specification. For modeling purposes, however, the number of neighboring pixels read for a macro block is dictated by a configurable parameter. Finally, the deblocking filter uses a similar mechanism by producing a write request at a configurable frequency to try to mimic the behavior of a real filter.

3. In order to map pixel coordinates to memory address the following assumptions were made:

   a. Each pixel is 16bit wide.
   b. Macro blocks are traversed in 2x2 pixel regions (quads) which are stored in a contiguous 64bit block in memory.
   c. Memory width is 64bit which matches the request size so request, read or write, map to a single data transfer.

   Based on the above, the equation to generate the memory address of a pixel quad is:

   \[
   \text{Frame Offset} + \left(\left(\text{quad anchor } Y \times \text{Frame Width}\right) + \text{quad anchor } X\right) \times 4 \times \text{Bytes per Pixel}.
   \]

   Where:
   Frame offset is the start of the frame in memory.
   Quad anchor is the upper left pixel of the quad.
   Frame width comes from the user’s block sequence file.
   Bytes per pixel is 2 as described above.
4. It was originally expected that this work would make use of a transaction-level model that accurately represented synchronous DRAM command latencies. Unfortunately, this model was never completed, and was considered out of the scope of this work. Therefore, the decision was made to write a simple memory model that does not implement the full DRAM functionality. This decision allowed the arbiter to be much simpler as it does not have to implement a full-fledged memory protocol. Instead the arbiter’s implementation can focus on the ideas being presented here.

5. In order to quantify the power savings from putting the memory in self refresh mode, we used a Micron x16, 512Mb, 133 MHz, Mobile DDR device running at 1.8V. This memory has the following characteristics[18]:

\[
\begin{align*}
P(\text{Sref}) &= IDD6a \times VDD = 0.3mA \times 1.8V = 0.54mW \\
P(\text{Std\_modes}) &= P(\text{PRE\_PDN}) + P(\text{PRE\_STDBY}) + P(\text{ACT\_PDN}) + P(\text{ACT\_STBY}) + P(\text{REF}) + P(\text{ACT}) + P(\text{WR}) + P(\text{RD}) + P(\text{DQ}) = 89.79mW \\
P(\text{Average}) &= P(\text{Std\_modes}) \times \%\text{Std} + P(\text{SRef}) \times \%\text{SRef}
\end{align*}
\]

Where \%Std and \%SRef are the percentage of time the memory is in standard mode and self refresh mode respectively. Note that the standard mode power considers a typical usage scenario as defined in [18] and uses a weighted average of the power of the various memory operations (page management, reads, writes, refresh, etc.).

6. To accurately model \%SRef, we have to take into account the self refresh entry and exit latency.
   a. The entry latency, defined from the arbiter’s last read/write request to memory going into self refresh, is the memory read latency plus burst length. For Micron LPSDR-166 this is 7 cycles [19].
   b. The exit latency, defined from memory existing self refresh to arbiter issuing the first read/write request, is tXSR (self refresh to read command). Micron recommends 20 cycles for LPSDR-166 [19].
4.2. Memory Access Arbitration

The first chart below shows when the memory was powered down (value=1) and when it was powered up (value=0) over the time of the simulation. At the beginning of the simulation, the motion compensator asserts the mc_arb_urgent signals to reflect that its data pool is below the low watermark. This prevents the arbiter from powering down the memory until it services enough read requests from the motion compensator to get its data pool above the high watermark. At that point, the motion compensator deasserts the urgent signal allowing the arbiter to power down the memory.

While the memory is powered down, the motion compensator will be processing reference pixel data and reducing the data pool amount as can be seen in the second chart. At some point, the data pool goes below the low watermark causing the mc_arb_urgent signal to assert at which point the arbiter will power up memory and service enough read requests to reduce the urgency of the motion compensator. This cycle will be repeated throughout the simulation.

Note the asymmetry between power up and power down periods which is due to the difference between the motion compensator processing rate and the memory rate. This characteristic is typical of H.264 decoders and is the premise of the Memory Access Arbitration scheme. Power up memory for short period of times and run it at maximum bandwidth to allow for longer power down periods.

The third chart shows the number of outstanding motion compensator read requests over the simulation time. This number is incremented when the motion compensator issues a read request and is decremented when the corresponding response data is processed. This number fluctuates between 31 and 32 where the latter is the maximum number of outstanding reads configured in this simulation. The conclusion here is that by increasing the number of outstanding reads along with the data pool capacity of the motion compensator, we can have longer power down periods at the expense of more area and little longer power up time to
service the additional requests. This would result in a net power reduction due to higher power
down to power up ratio as well as less power transitions.

Finally, the fourth chart shows the filter write data pool size over the simulation time. Note
that the filter works out of phase with the motion compensator as it waits for a whole block to
be estimated before using frame n-1 reference pixel before it generates frame n reference
pixels. The filter write data pool will start empty and builds up because memory is powered
down and the arbiter is not draining. At some point, the pool exceeds the high watermark
causing the filter_mc_urgent signal to be asserted at which point the arbiter will power up
memory and drain the pool. The key thing here is that with the right pool sizes, we can have
the motion compensator and the filter go urgent and power up memory around the same time.
That way when memory is power down, the motion compensator pool is close to full and filter
pool is close to empty. The opposite would be true when we power up where the motion
compensator pool would be almost empty and the filter pool almost full.

For the block sequence in question, the memory is put in self refresh 1,269,376 cycles out of
1,415,320 cycles the simulation was run. This sets %Sref to 89.69% and %Std to 10.31%. The
average power is:

$$89.79\text{mW} \times 10.31\% + 0.54\text{mW} \times 89.69\% = 9.74\text{mW}$$

This results in 89.15% power savings compared to an implementation that does not implement
self refresh.
Figure 11: Memory Access Arbitration Results

Memory Self Refresh

MC Data Pool

Number of Outstanding MC Read Requests

Filter Data Pool
4.3. Reference Data Scheduling

The chart below shows the fraction of the motion compensator memory reads that hit in the cache (blue) as opposed to the ones going to memory (red) for various cache sizes. As expected, increasing the number of entries in the direct mapped cache reduces the number of evictions and allows the motion compensator to hit in the cache when it goes back in raster order and revisits a partition that is neighboring an already processed partition.

![Figure 12: Motion Compensator Hits/Misses in the Cache vs. Cache size](image)

The graph below shows the motion compensator hit rate as a percentage of total memory reads for various cache sizes. As expected, the benefit of increasing the cache size will flatten out as the number of misses approach the size of the frame. As stated earlier, the best we can do here is read every reference pixel once.
The graph below shows the percentage of system power saved as a function of cache size. The graph was derived from data presented in section 1.2 where we can see that 75% of memory accesses come from the motion compensator and 20% of the system power is spent in off chip memory. Assuming that a reduction in memory accesses corresponds to a proportional reduction in memory power, then we can translate the motion compensator hit rate of $H$ into power savings as follows.

\[
\text{Off chip memory savings} = H \times 75\%
\]

\[
\text{System savings} = \text{Off chip memory savings} \times 20\% = H \times 75\% \times 20\% = H \times 15\%
\]

Figure 13: Motion Compensator hit rate in the Cache vs. Cache Size.
The graph below shows the cache area for the various sizes chosen for the study. This data was derived using TSMC 28nm process. According to [26], the area per bit for a 28nm SRAM is 0.127um². With every cache entry requiring 512 bit of data storage and 26 bit of address tag storage, we can derive the total cache size. Note that we are assuming 20% overhead for SRAM read and write decoding logic.
The graph below shows the increase in the area of a typical H.264 decoder when a pixel cache is integrated into the design. The reference decoder being considered is described in [27] and takes up $5.9\text{mm}^2$ in 28nm process. Note that the design in [27] includes a pixel cache, but the size is not reported. The graph shows that the pixel cache is small compared to the total size of the decoder, and that adding a pixel cache is an economical way to reduce the off-chip memory portion of the H.264 decoder overall power.
Figure 16: System Cost vs. Cache Size

The bar chart below shows how the motion compensator hit to miss ratio changes when the prefetcher is turned on with a 512 entries cache.

Figure 17: Motion Compensator Hits in the Cache
When the prefetcher is turned off, the motion compensator reads hit 96.13% in the pixel cache. These hits are due to data reuse where reference pixels are kept around to be reused by the interpolator filter in the motion compensator. The fact that we are reusing reference pixels across macro blocks boundary helps quite a bit.

When the prefetcher is turned on and is not being throttled, the motion compensator hit rate goes up to 99.08% at the expense of 2.92% extra reads issued to memory. We also looked at an intermediate usage case where the prefetcher is turned on but is throttled to only issue one read request every 16 cycles. The hit rate was pretty much unchanged but the number of extra prefetcher memory reads went down to 2.47%. This suggests that throttling the prefetcher might be a good compromise between motion compensator hit rate and memory power.

The prefetcher also increases the likelihood that motion compensator reads hit in the cache but it does not reduce the actual number of memory accesses. In fact, if the prefetcher is not carefully constrained it could increase the number of memory accesses. The main benefit of the prefetcher is to bring in the reference data early in the cache which would help by servicing the motion compensator from the cache and not powering up the memory.
CHAPTER 6: Conclusion

In this work we proved that Memory Access Arbitration and Reference Data Scheduling can considerably reduce off-chip memory power. This was done by simulating these two techniques using a SystemC cycle accurate model and input stimulus that mimics real H.264 traffic.

Memory Access Arbitration implemented by the memory arbiter allows memory to be accessed in bursts with idle periods in between where memory is put in self refresh mode. It also reduces memory bus inefficiency due to read to write and write to read turnaround.

Reference Data Scheduling reduces the motion compensator memory read requests by keeping reference pixels in an on chip pixel cache to be reused by the interpolation logic. An added prefetcher helps in reducing the motion compensation read latency by increasing its hit rate in the pixel cache. That in turn reduces the motion compensator power as it eliminate pipeline stalls and allows it to be more power efficient (race to sleep).

An extension to this work might entail looking more closely at the pixel cache size to balance chip area with memory power. The prefetching policy can also be tweaked and analyzed. Finally, the arbitration policy could be extended to include more clients like display.
REFERENCES


[23] Vcodex, [www.vcodex.com](http://www.vcodex.com)


[26] Mark LaPedus, “TSMC devises SRAM cell at 28-nm” *EE Times, June 17, 2009*

APPENDICES
Appendix A
**H.264 Block Overview**

*Entropy Decoder*

The H.264 entropy encoder is responsible for converting syntax elements such as quantization coefficients, motion vectors, prediction modes etc. into a bit stream that is entropy decoded to recover the syntax elements.

There are two main entropy coding schemes, variable length coding and arithmetic coding. The first scheme uses a Huffman table where a symbol can be represented with one or more integer number of bits. In arithmetic coding, a symbol is encoded based on the probability of its appearance, so it can represent a symbol with a fractional number of bits, thus giving this scheme the edge in compression efficiency over variable length coding.

Table 6 describes the different parameters that need to be encoded. The default entropy coding method uses a single infinite codeword set (i.e. 1 VLC table) for all syntax elements except the quantized transform coefficients. The single codeword table used is an exp-Golomb code with very simple and regular decoding properties. For transmitting the quantized transform coefficients a more sophisticated method called CAVLC (Context-adaptive Variable Length Coding) is used. In this method, numerous VLC tables are used that match corresponding conditioned statistics thus improving performance compared to a single VLC table.
Table 6: Coding methods for syntax elements [3]

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Desc.</th>
<th>Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence, picture and slice layer syntax elements</td>
<td>Headers and parameters</td>
<td>Exp-Golomb</td>
</tr>
<tr>
<td>Macroblock type mb_type</td>
<td>Prediction method for each coded macroblock</td>
<td></td>
</tr>
<tr>
<td>Coded Block Pattern</td>
<td>Indicates which blocks within a MB contain coded coefficients</td>
<td></td>
</tr>
<tr>
<td>Quantization Parameter</td>
<td>Transmitted as a delta value from the previous value of QP</td>
<td></td>
</tr>
<tr>
<td>Reference frame index</td>
<td>Identify reference frame(s) for inter prediction</td>
<td></td>
</tr>
<tr>
<td>Motion Vector (MV)</td>
<td>Transmitted as a difference (mvd) from predicted MV</td>
<td></td>
</tr>
<tr>
<td>Residual Data</td>
<td>Coefficient data for each 4x4 or 2x2 block</td>
<td>CAVLC</td>
</tr>
</tbody>
</table>

To further improve entropy coding efficiency Context-Adaptive Binary Arithmetic Coding (CABAC) can be used. Since CABAC is an arithmetic coding scheme it allows the assignment of non-integer number of bits to each symbol which is beneficial for symbol probabilities much greater than 0.5. Furthermore, the context modeling of CABAC allows the statistics of already-coded syntax elements to be used to estimate the conditional probabilities. Of course this efficiency improvement comes at the expense of being more computationally extensive than CAVLC. The choice of CABAC or CAVLC depends on the value of the entropy coding mode.

**Inverse Quantization and Transformation**

Similar to previous video coding standards, H.264 also utilizes transform coding of the prediction residual data that is sent from the encoder. However, in H.264 the transformation is applied to 4x4 blocks, and instead of a 4x4 discrete cosine transform (DCT), a separable integer transform with basically the same properties as a 4x4 DCT is used. Since the inverse transform is defined by exact integer operations, inverse-transform mismatches are avoided.

Three different types of transforms are used. The first type is applied to all samples of all prediction error blocks of the luma and chroma components regardless whether inter or intra prediction is used. The transform is a 4x4 matrix H1. The second type of
transform is a Hadamard transform $H_2$ that is used in addition to $H_1$ if the type of prediction is INTRA16x16. The third transform is also a Hadamard transform but of size 2x2. It is used for the transform of the 4 DC coefficients of each chrominance component.

\[
H_1 = \begin{bmatrix}
1 & 1 & 1 & 1 \\
2 & 1 & -1 & -2 \\
1 & -1 & -1 & 1 \\
1 & -2 & 2 & -1
\end{bmatrix} \quad H_2 = \begin{bmatrix}
1 & 1 & 1 & 1 \\
1 & 1 & -1 & -1 \\
1 & -1 & -1 & 1 \\
1 & -1 & 1 & -1
\end{bmatrix} \quad H_3 = \begin{bmatrix}
1 & 1 \\
1 & -1
\end{bmatrix}
\]

*Figure 18: Matrices of the three different transforms applied in H.264*

For the quantization of transform coefficients, H.264 uses scalar quantization. The main function of quantization is to scale down the transformed coefficients to reduce the coding information. The quantization step is chosen by the Quantization Parameter (QP) which supports 52 different quantization parameters. These values are arranged so that an increase of 1 in quantization parameter means an increase in quantization step by approximately 12%.

*Inter/Intra Prediction*

In H.264, prediction is trying to find a reference MB that is similar to the current MB under processing so that only the difference needs to be coded instead of the whole current MB. Depending on where the reference MB comes from, prediction can be classified as either intra prediction and inter prediction. Intra prediction exploits the spatial redundancy among neighboring blocks in a video frame whereas inter prediction exploits the temporal redundancy of a video sequence.
Intra prediction means that samples of a macroblock are predicted by using only information of already transmitted macroblocks of the same image. For the luminance component there are two different types of intra prediction. The first type is called INTRA_4x4 and the second is INTRA_16x16. In an I-type 16x16 MB the luminance component (one 16x16) and the chrominance components (two 8x8 blocks) are predicted separately. Using INTRA_4x4, each 4x4 block utilizes one of nine prediction modes (one DC prediction mode and eight directional prediction modes). In the INTRA 16x16 case, which is well suited for a smooth image area, a uniform prediction is performed for the whole luminance component of a macroblock. Four prediction modes are defined (vertical prediction, horizontal prediction, DC-prediction and plane-prediction).

Intra prediction for the chrominance component of a macroblock is similar to the INTRA 16x16 case because chrominance signals are very smooth in most cases. It is carried out upon 8x8 blocks using one of the four modes mentioned above.

Inter prediction, also known as motion estimation, uses temporal redundancy to reduce data to be encoded. Macroblocks are predicted from the image signal of already transmitted reference images. For this reason, each macroblock can be divided into smaller portions of various sizes as shown in the figure below.

Figure 19: Partitioning of macroblock and sub-macroblock for motion compensated prediction.
In the case of an 8x8 sub macroblock in a P-slice, an additional syntax element specifies if the 8x8 can be further divided into smaller partitions. A motion vector is estimated and transmitted for each block; this vector refers to the corresponding position of the image signal in an already transmitted reference image. The accuracy of motion vectors in H.264 is a quarter pixel; an improvement on previous standards which had at most half of a pixel accuracy. Furthermore, in previous standards only blocks of size 16x16 and 8x8 were supported, so in this fixed block-size motion estimation the same effort is spent when estimating the motion of moving objects and background objects (no motion). In H.264, variable block-size motion estimation (VBSME) is used thus allowing for smaller block sizes to be used for moving objects and larger block size for background. VBSME increases video quality and coding efficiency.

Multiple reference frames is another improvement in motion estimation in H.264. Instead of just using one reference frame as in previous standards, a P frame in H.264 can use up to 5 frames whereas a B frame can use 10 frames. More frames result in smaller residual data thus lower bit rate, however this comes at the expense of computational complexity and higher memory traffic.

Filter

Block-video coding produces undesirable artifacts known as blocking artifacts that degrade the quality of the decoded frame. Two components of the H.264/AVC decoder that can be a source of blocking artifacts are the block-based integer discrete cosine transforms and the motion compensation prediction, i.e. the prediction and the residual difference coding stages of the decoding process. The deblocking filter is applied to reduce the effects of these undesirable artifacts.
There are two approaches to integrate the de-blocking filter into video codecs, post filtering or loop filtering. Post filters only operate on the outside of the coding loop whereas loop filters operate within the coding loop. There are several advantages to using a loop de-blocking filter instead of a post filter. First, the requirement to use the filter in the coding loop guarantees a certain quality of service which is especially important in today’s modern communication systems. Second, there is no need for an extra frame buffer in the decoder since loop filtering is carried out on macroblocks during the decoding process. Third, empirical tests have shown that loop de-blocking filters improve both the objective and subjective quality of video with significant reduction in decoder complexity [4]. These advantages come at the price of one critical factor which is the computational complexity. The loop filter accounts for one-third of the computational complexity of a decoder and this is mainly because of its high adaptivity since it requires conditional processing on block edge and sample levels. Conditional branches are known to be very time consuming and pose quite a challenge for parallel processing whether using DSP hardware or general-purpose processors.

Filtering is performed on 1 16x16 MB at a time. It filters every boundary defined by 4x4 blocks within the MB. The filtering order consists of a horizontal filtering across all vertical edges followed by a vertical filtering across all horizontal edges.

Inputs to the filter include boundary strength (bS) and threshold values. As the boundary strength is increased, more blocking artifacts are eliminated. Threshold values are used to distinguish a true edge from a false one.