Validation of Graphics System Features Using Media Workloads

Major Project Report

Submitted in partial fulfillment of the requirements

For the degree of

Master of Technology

in

Electronics and Communication Engineering

(Communication Engineering)

By

Ishani Pathak

(12MECC09)

Electronics and Communication Engineering Branch

Electrical Engineering Department

Institute of Technology

Nirma University

Ahmedabad-382481

May 2014

Validation of Graphics System Features Using Media Workloads

Major Project Report

Submitted in partial fulfillment of the requirements

For the degree of

Master of Technology

in

Electronics and Communication Engineering

(Communication Engineering)

By

Ishani Pathak

(12MECC09)

Under the Guidance of

Dr. D.K. Kothari

Professor and Section Head, EC

Electronics and Communication Engineering Branch

Electrical Engineering Department

Institute of Technology

Nirma University

Ahmedabad-382481

May 2014

Declaration

This is to certify that

- The thesis comprises my original work towards the degree of Master of Technology in Communication Engineering at Nirma University and has not been submitted elsewhere for a degree.
- Due acknowledgement has been made in the text to all other material used.

- Ishani Pathak
12MECC09

Certificate

This is to certify that the Major Project entitled **"Validation Of Graphics System Features using Media Workloads"** submitted by **Ishani Pathak (12MECC09)**, towards the partial fulfillment of the requirements for the degree of Master of Technology in Communication Engineering of Nirma University, Ahmedabad is the record of work carried out by her under our supervision and guidance. In our opinion, the submitted work has reached a level required for being accepted for examination. The results embodied in this major project, to the best of our knowledge, haven't been submitted to any other university or institution for award of any degree or diploma.

Date:	Place: Ahmedabad
Guide	**Program Coordinator**
Dr. D.K. Kothari (Professor, EC)	Dr. D. K. Kothari (Professor, EC)
Dr. P. N. Tekwani (Head of EE Department)	Dr. K. Kotecha (Director, IT, NU)

Intel Technology India Pvt. Ltd.

Certificate

This to certify that **Ms Ishani Pathak (12MECC09)**, a student of M.Tech EC (Communication), Institute of Technology, Nirma University was working in this organization since 12/06/2013 and carried out her thesis work titled **"Validation Of Graphics System Features using Media Workloads"**. She was working in name of **Graphics Media System Validation intern** under supervision of Mr Mahesh K. Reddy (Mentor), and Sreedhara Murthy (Manager). She has successfully completed the assigned work and is allowed to submit her dissertation report. We wish her all the success in future.

Sreedhara Murthy	Mahesh K. Reddy
Manager, Graphics Media System Validation	Mentor

May 2014
Bangalore

Acknowledgements

It has been an enriching experience for me so far at Intel. The dynamics are different and I learn a lot every day. The work environment is conducive to growth.
I am thankful to my college - Nirma University - for giving me an opportunity to work in such an esteemed organization as Intel. I would like to thank Dr. K. Kotecha, Director, IT-NU and Prof. P.N. Tekwani, Head of Electrical Engineering Department, for providing this to me. I would like to express gratitude to my faculty guide Prof. Dr. D.K. Kothari, Coordinator M.Tech Communication Engineering Program for his support and for his guidelines during the review process.
I am deeply indebted to my project mentor Mahesh Kumar Reddy for his constant guidance and motivation. He has devoted significant amount of his valuable time to plan and discuss the project work. Without his experience and insights, it would have been very difficult to do quality work. I am grateful to the Graphics System Validation Team Manager Sreedhara Murthy for showing confidence in me and supporting me immensely.
I would like to thank everybody at Intel who helped me and made me feel as a part of the team right from the very first day. This includes my Manager Bhavesh Shroff, my mentor J.M. Malavika, all my teammates, previous interns and a few people from other teams as well.

- Ishani Pathak
12MECC09

Abstract

Graphics System Features are global functionalities that are applicable across each and every media unit. Every media test uses an outer cache - either L3 or external LLC (Last Level Cache). So, the outer cache is a global system feature. Yet, there is no universal way of accessing those outer caches and controlling their cacheability. My proposed solution is to use Memory Object Control State (MOCS) as a way of accessing the outer caches. A script needs to be developed which could resolve events that would eventually cause cache failures; like memory issues, invalid translations and cache misses. This script needs to be added and executed with a certain number of media tests to see if the intent is met.

The second scope of work is to enable preemption among media workloads. Traditionally, the graphics driver is used to enable preemption. This is done in the User mode, i.e. the calls (or preemption requests) made by the application have to go through several layers of hardware abstraction (OS, Kernel) to reach the hardware and get the task done. This means a time delay between when the request for preemption is issued and when it is serviced/executed. There are many overhead challenges. My proposed way is to use the graphics microcontroller (GuC) itself for scheduling and preemption of media workloads. This microcontroller is present on the Graphics Processing Engine (GPE) and any task run by it would talk directly to the hardware, and finish sooner. There is also a huge plus of using the microcontroller when it came to power savings - it could run independently in low power mode; even when the main core was in OFF state. Additionally, it has the authority to schedule workload on graphics sub-systems directly, without Host intervention. GuC is also a global system feature.

Contents

List of Figures

Chapter 1

Introduction

The term System features refers to the global features that impact most of the graphics system units. Typically, these features are related to memory accesses, work load scheduling (to match usage models), etc.
Traditionally, these system features are validated using simple (and representative) workloads.
In this work, the validation coverage of the system features shall be enhanced by validating them with actual media workloads, rather than representative workloads.

1.1 Scope

For complex systems like graphics which occupies nearly 50 % of the die in the regular client Central Processing Units (CPU), there is a pressing need to find bugs as early as possible in the design and validation cycle. Emulation platforms enable functional validation teams like System Validation (SV) to execute more cycles much faster. The basic reason for using emulation is to identify the bugs as soon as possible in pre-silicon phase and have minimal bugs post silicon. Traditionally, media system validation involved validation of each individual unit in isolation. However, as the SOCs become more advanced (and complex), additional system level improvements are needed in each generation of the SoC. This mandates that the individual media

units too exercise these system level advancements early in the validation cycle.
The scope of this work is to define and implement a process to exercise media tests (workloads) focused on system level changes in the Graphics sub system.

1.2 Goals

- Defining and implementing methodology for system features validation using media workloads

- Documenting the methodology for re-use in future programs

1.3 Project Flow Of Activities

It is wise to have a sequence of steps that you intend to follow in order to complete and deliver your project on time.
For me, the first requirement was to ramp up on Intel Graphics architecture. Understanding Emulation infrastructure used in the graphics system validation team came next.
I analyzed validation data from Intel's previous platform in order to define a problem statement. The main error buckets were defined. Clearly, there was a pressing need to define a way of outer cache access and controlling the cacheability of the outer caches.Also, graphics microcontroller based preemption in media tests was an unvalidated scenario. By this time, I had a fair idea of what my Ruby script should do. So, we defined a problem statement and also the areas that we would target.
Once the scripts were ready, the next phase was to enable and characterize Media tests under both the scenarios. This would be followed by execution and debug of these tests in emulation. The end goal was to achieve atleast 75 % pass rate in system features tests in each validation cycle.

1.4 Organisation of thesis

The rest of the thesis is organised as follows:

Chapter II - Literature Survey, gives a brief overview of media basics, the encode and decode pipelines used; the hardware and software blocks.

Chapter III - Overview of System Features, describes what essentially are system features. It describes MOCS (Memory Object Control State) and graphics microcontroller; as well as the need for preemption.

Chapter IV - Implementing MOCS (outer cache access mechanism), describes the methodology and implementation I laid out for enabling MOCS; the media units I covered, the scripting involved and the results across two vallidation cycles.

Chapter V - Enabling Preemption, descibes why graphics microcontroller was used for preemption among two media workloads. It also has pseudo-codes of the scripting involved and the units I covered.

Chapter VI describes Conclusion and Future scope of work.

1.5 Summary

System features are those functionalities which are applicable at the system level in the graphics sub-system. The problem statement is to define a mechanism of outer cache access and microcontroller based preemption for media tests. For this, focussed scripts were developed in Ruby.

Chapter 2

Literature Survey

System Features are those functionalities that are applicable across every media unit. The script that I developed was applied onto every media base test for effective enabling of that feature. There are multiple media units - decoder blocks (one for each codec - vp8, mpeg4, avc, vc1, jpeg and a few more), encoder blocks, a memory interface unit, sampler, quantizer, video motion estimation unit and a few more. Each test uses one or more of these units. Hence, it was necessary for me to know what each media unit was meant to do; and to understand the basics of various codecs from a industry specification point-of-view.

2.1 Codec overview

Codec means compression and decompression of digital video. A lot of media tests use the H.264 codec. To understand it, I studied the following article.
The article by Iain E. Richardson, The H.264 Advanced Video Compression Standard, published in *John Wiley & Sons* (2010) described the methods of predicting intra-coded macroblocks in an H.264 video compression codec [1]. If a block or macroblock is encoded in intra mode, a prediction block is formed based on previously encoded and reconstructed (but un-filtered) blocks. This prediction block P is subtracted from the current block prior to encoding.

The paper by Thomas Wiegand, Gary J. Sullivan, Gisle Bjontegaard, and Ajay Luthra, Overview of the H.264/AVC Video Coding Standard, published in *IEEE Transactions On Circuits And Systems For Video Technology* (Vol. 13, No. 7, July 2003) provided an overview of the technical features of H.264/AVC [3]. It described profiles and applications for the standard and outlined the history of the standardization process.
H.264 is different from existing standards in the following ways:

- Enhanced motion-prediction capability
- Use of a small block-size exact-match transform
- Adaptive in-loop deblocking filter
- Enhanced entropy coding

These helped me to gain a fair understanding of the H.264 standard. Other codecs like avc, vc1 etc. were studied from Intel internal presentations. Some are described in brief below.

I. MPEG-2

- Most used compression technology
- From DVD, terrestrial broadcasting (ATSC/DVB), to cable and satellite broadcasting
- Simple and elegant, with limited compression
- Artifacts noticeable for SD under 4mbps, or HD under 12mbps

II. AVC - Advanced Video Coding

- Also referred as MPEG-2 part 10, and H.264
- Aggressively pursuing better compression with heavy toll on complexity

- 4X complexity for 2x compression

III. VC1

- A SMPTE Video standard (Society of Motion Picture and Television Engineers)
- Used to be a proprietary codec of Microsoft
- Complexity and gain are close to that of AVC
- It is today a supported standard found in Blu-ray Discs, Windows Media, Microsoft's Silverlight framework, Slingbox and the now-discontinued HD DVD.

2.2 Media compression

For a few MOCS (Memory Object Control State) tests, there was a need to enable a compression-decompression mode. I needed to know why compression was needed and how to enable it. For this, I went through some internal trainings and the following paper.
The paper by Ian H. Witten, Radford M. Neal, And John G. Cleary, Arithmetic Coding For Data Compression, published in *IEEE Transactions on Communications* (2007) dealt with arithmetic coding [2]. In arithmetic coding, a message is represented by an interval of real numbers between 0 and 1.
Successive symbols of the message reduce the size of the interval in accordance with the symbol probabilities generated by the model. Hence, fewer bits are added to the message.
Arithmetic coding gives greater compression.

2.2.1 The need for compression

Raw data requires a large number of storage. Consider the following:

- SD 720 x 480 x 30fps x 60sec/min x 60 min/hr x 2hr x 3 byte per pixel = 223 GB per disk
- HD(p) 1920 x 1080 x 60fps x 60sec/min x 60 min/hr x 2hr x 3 byte per pixel = 2687 GB per disk

Uncompressed video (and audio) data are huge. In HDTV, the bit rate easily exceeds 1 Gbps; which is a big problem for storage and transport.

Consider the HDTV broadcasting format - 1920 pixels horizontally by 1080 lines vertically, at 30 frames per second. If these numbers are all multiplied together, along with 8 bits for each of the three primary colors, the total data rate required would be approximately 1.5 Gb/sec. Because of the 6 MHz channel bandwidth allocated, each channel will only support a data rate of 19.2 Mb/sec, which is further reduced to 18 Mb/sec by the fact that the channel must also support audio, transport, and ancillary data information.
As can be seen, this restriction in data rate means that the original signal must be compressed by a figure of approximately 83:1. This number seems all the more impressive when it is realized that the intent is to deliver very high quality video to the end user, with as few visible artifacts as possible.
Video Compression can achieve 100 to 1 compression. It means getting rid of redundant data and throwing away details which are not easily noticeable.

Lossy methods have to be employed since the compression ratio of lossless methods (e.g., Huffman, Arithmetic) is not high enough for image and video compression, especially when distribution of pixel values is relatively flat. The following compression types are commonly used in Video compression:

- Spatial Redundancy Removal - Intraframe coding (JPEG)
- Spatial and Temporal Redundancy Removal - Intraframe and Interframe coding (H.261, MPEG)

2.3 Documentations obtained from the extranet

As my project is pertaining to media validation, a pre-requisite was to have a thorough understanding of Intel Graphics Architecture. To achieve the same, I studied the below listed topics:

- Understanding basics of video [8]
- Understanding video encoding and decoding basics [8]
- Overview of Intel Media pipelines[4, 5]
- Understanding Graphics Media System Features [5]
- Understanding System Validation execution environment
- Learning Ruby (developing test content) and Perl (scripting for automation) [7]
- Motion estimation and motion compensation concepts [8]
- Learning Video codecs - AVC,VC1,MVC etc. [3]
- Video compression concepts [6]

2.4 Overview of Media

This section gives an overview of Media basics, the need for video compression, the type of macroblocks and basic Video Encoder and Video Decoder diagrams.

2.5 Video - Overview

Video is an electronic medium for the recording, copying and broadcasting of moving visual images. It basically involves playing a series of still images at a certain speed (frames per second); and the mind perceives that as a continuous video.

Each video file contains several GOPs (or Group Of Pictures). These in turn, are composed of several frames, namely:

- I - Intracoded frame, does not need reference picture to decode, decompress or encode

- B - Bipredictive frame, uses difference between current and successive or previous frame

- P - Predicted frame, uses data from previous frame to decode or encode

Next, each frame is composed of multiple slices. A slice is a spatially distinct region of frame that is encoded separately from any other region in the same frame.
Each slice is composed of Macroblocks (size of 16x16), which are subdivided into blocks (usually 8x8 pixel matrix).
Macroblocks (MBs) have the following modes:

- Intra - encoded without past reference frame
- Inter - forward or backward predicted macroblock
- Skipped - when background content remains the same in several frames, you can skip encoding those MBs altogether
- IPCM - macroblocks are sent as they are, without any encoding. This mode consumes high bit rate, but there is the advantage of zero distortion.

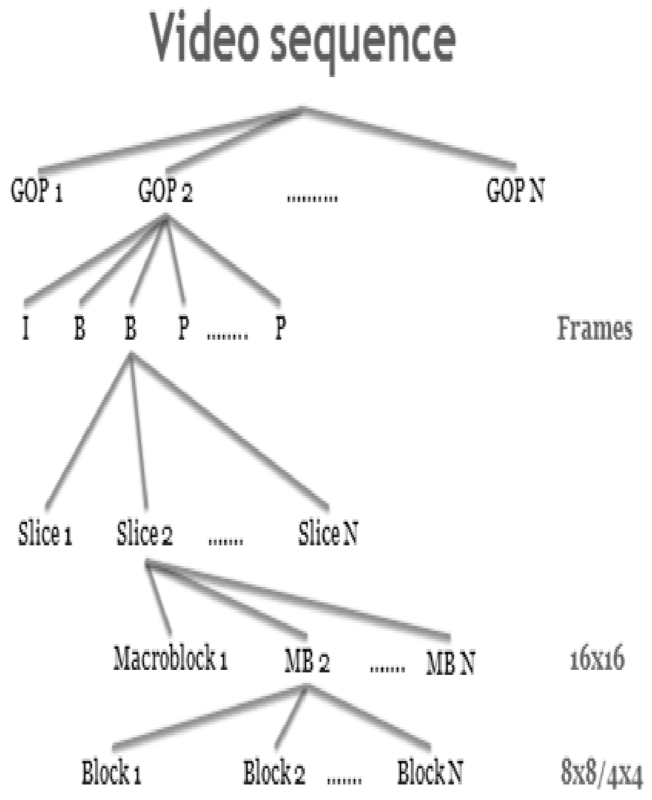


Figure 2.1: A typical video sequence

2.6 Video Decoding - Hardware solution

Multi-Format Codec (MFX) engine contains a full-hardware decoder. It allows Multi-format encode and decode and has a shared data path optimized for area & power.

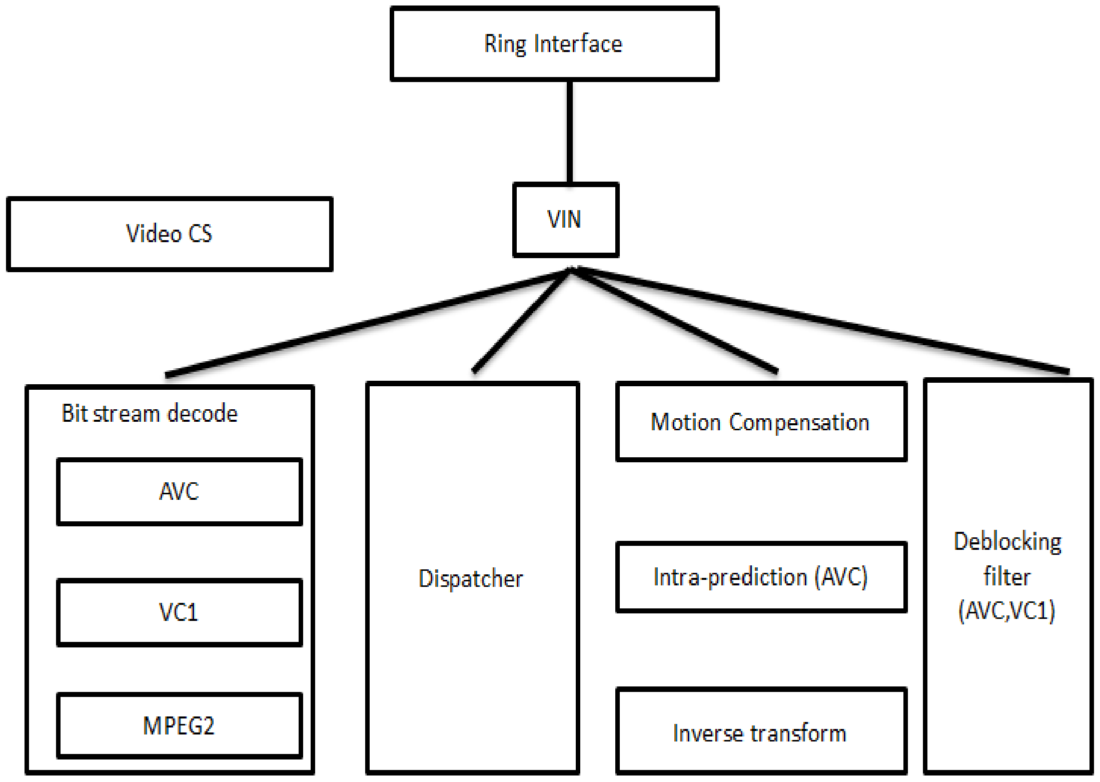


Figure 2.2: Decoder engine

- Ring interface maps to memory.
- VCS (Video Command Streamer) looks at opcodes, MB state and object type and passes it on to VIN.
- VIN (Video input) parses this data, forwards it to every block.

Now, MB modes, as parsed by VIN can be of several types:

I. Intra - encoded without past reference frame

II. Inter - forward or backward predicted macroblock

III. Skipped - when background content remains the same in several frames, you can skip encoding those MBs altogether

IV. IPCM - macroblocks are sent as they are, without any encoding. This mode consumes high bit rate, but there is the advantage of zero distortion.

Bit stream decode unit can decode MPEG2, AVC and VC1 data by employing either Variable Length Decoder (VLD) or Arithmetic Decoding. Video dispatcher unit forwards data streams to the following:

- Motion Compensation block - to predict a MB or block using a weighed combination of 1 or 2 prediction blocks of the same pixel size and block dimension
- Intra-prediction block - to predict a MB without any reference block
- Inverse transform block - employs IDCT and does dequantization

Previously decoded frame is used as a reference to encode the current frame. Deblocking filter smoothens the output.

2.7 Video Encoding - software/hardware solution

Encoding occurs in two phases:

I. ENC maps to Programmable EU Array, runs in GPU (Graphics Processing Unit) in the CPU. It enables algorithm tuning & feature additions

II. PAK maps to MFX hardware pipeline, i.e. it reuses decoders pixel reconstruction circuitry and some units in the MFX pipeline.

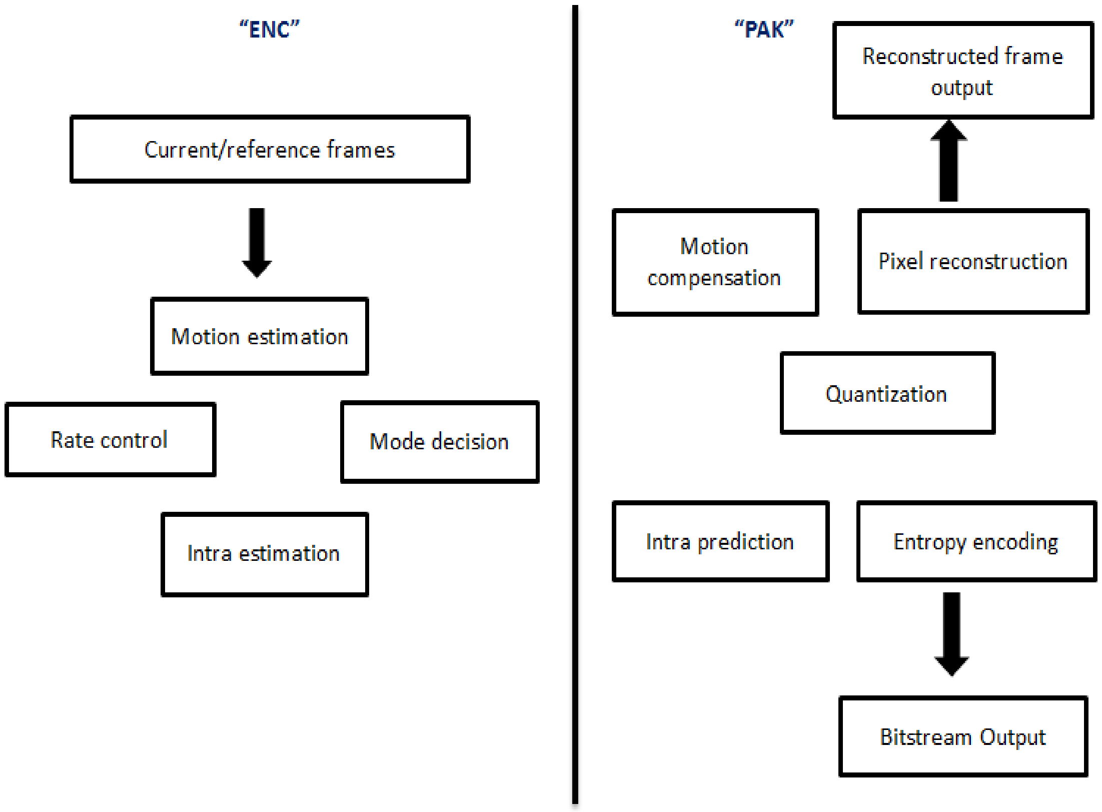


Figure 2.3: Video encoding

2.8 Video Encode - PAK in MFX

Multi-Format Codec (MFX) Engine is the hardware fixed function pipeline for decode and encoding. It includes multi-format decoding (MFD) and multi-format encoding (MFC).

Many decoding function blocks in MFD such as VIP (Video Intra-Prediction), VMC (Video Motion Compensation), IQT (Inverse Quantisation and Inverse Transform), etc. are also used in encoding mode. Two blocks FTQ (Forward Transform and Quantisation) and BSE (Bitstream Encode) are encoding only.

The encoding process is partitioned across host software, GPE (Graphics Processing Engine) engine and the MFX engine. The generation of transport layer, sequence layer, picture layer and slice header layer is required to be done in the host software. GP hardware is responsible for compressing from Slice Data Layer down to all macroblock and block layers. Specifically, GPE w/ VME acceleration is for motion vector estimation, motion estimation, and code decision.

The VME (Video Motion Estimation) is located next to all image processing units, such as DN (denoise) and DI (deinterlace) in sampler in GPE.
MFX is for final bit packing and reconstructed picture generation.
MFC is operated concurrently with and independently from the GPE (3D/Media) pipeline with separate command streamer. The two parallel engines have similar command protocol. They can be executed in parallel with different context. For encoding, motion search, MB mode decision and rate control are performed using GPE pipeline resources.

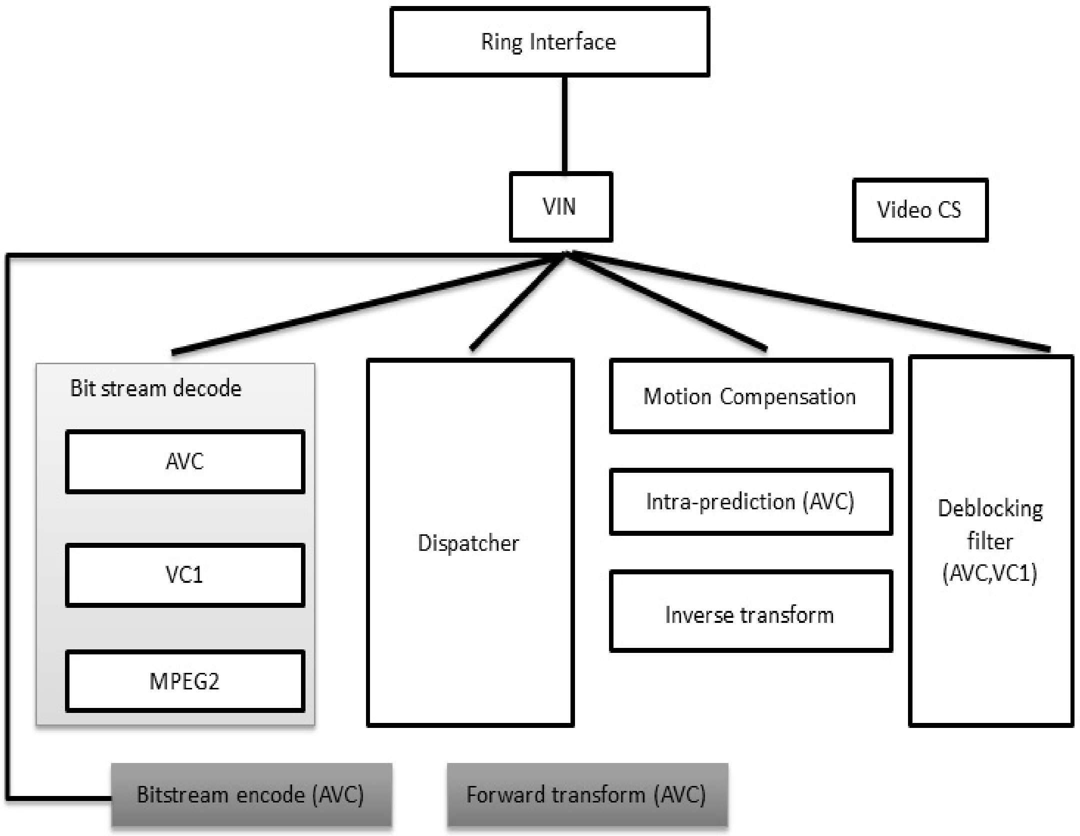


Figure 2.4: MFX - full stack

Bitstream encode block and Forward Transform block is employed for encoding, the Bitstream decode block is used for decoding; the rest of the blocks are shared between MFX and PAK.

As the result of this hardware partitioning, VPP(Video Pre-Processing) and ENC are always running in GPE, and PAK is what runs exactly in MFC.
Functions of PAK:

- Residue packing and entropy coding, including block transformation, quantization, data prediction, bitrate tuning and reference decoding.

- It delivers final packed bitstream and decoded key-frame reference.

2.9 Processor Graphics - Unified 3D and Media architecture

The Main Render Engine is unified for 3D Graphics and Media.

Fixed Function (FF) Pipelines have been implemented which do the following functions

- Thread generation and control
- 3D Graphics or Media FF controls EU array at a given time

The Execution Unit (EU) Array is a programmable entity which is shared between 3D and Media.

Shared Functions

- Accelerators for filtered load, scatter & gather, filtered/ blended store operations

MFX: A parallel codec engine runs on a separate context

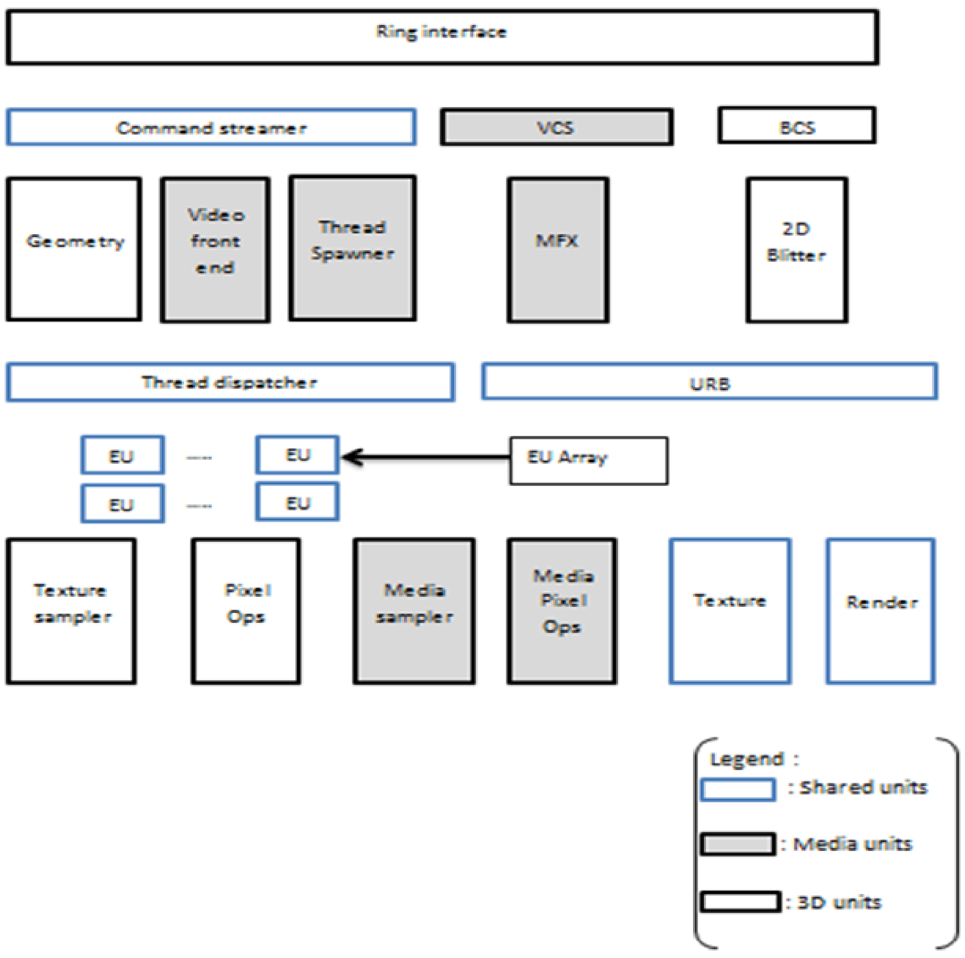


Figure 2.5: Unified architecture

2.10 Scripting language used - Ruby

Test content development as well as scripting to enable System Features is in Ruby. There are several reasons for choosing Ruby.

- Object oriented

  Every value in Ruby is an object, even the most primitive things: strings, numbers and even true and false. Even a class itself is an object that is an instance of the Class class.

  Consider the following example where we apply an action to a number

For example :

```
3.times  print ”Ishani *at* Intel - from Nirma!”
```

- Flexibility

  Ruby is seen as a flexible language, since it allows its users to freely alter its parts. Essential parts of Ruby can be removed or redefined, at will. Existing parts can be added upon.

  For example :

  ```
  class Numeric
  def plus(x)
  self.+(x)
  end
  end
  y = 5.plus 6
  (y is now equal to 11)
  ```

- Single Inheritance only

  Unlike many object-oriented languages, Ruby features single inheritance only.

  ```
  class Wood
  def foo_a
  puts”Here”
  end
  end
  class Teak <Wood
  def foo_b
  puts ”There”
  end
  end
  user=Teak.new
  user.foo_a
  ```

user.foo_b
<implies inheritance in Ruby. Teak becomes a derived class of parent class Wood. The object 'user' will inherit the function foo_a from the parent class.

Ruby has a wealth of other features, among which are the following:

I. Ruby has exception handling features, like Java or Python, to make it easy to handle errors.

II. Ruby features a true mark-and-sweep garbage collector for all Ruby objects. No need to maintain reference counts in extension libraries.

III. Writing C extensions in Ruby is easier than in Perl or Python, with a very elegant API for calling Ruby from C. This includes calls for embedding Ruby in software, for use as a scripting language.

IV. Ruby can load extension libraries dynamically if an OS allows.

V. Ruby features OS independent threading. Thus, for all platforms on which Ruby runs, you also have multithreading, regardless of if the OS supports it or not, even on MS-DOS.

VI. Ruby is highly portable: it is developed mostly on GNU/Linux, but works on many types of UNIX, Mac OS X, Windows 95/98/Me/NT/2000/XP, DOS, BeOS, OS/2, etc.

2.11 Summary

This chapter gave an overview of several media codecs and why Ruby was chosen as the scripting language for the project.
It also discussed the need for video compression and why bitstream packing is needed.
Next, a few Encode/decode block diagrams were described. In Intel's media engine, MFX (Multi-format codec) Engine is the hardware fixed function pipeline for decode

and encoding. It includes multi-format decoding (MFD) and multi-format encoding (MFC). PAK runs in MFC and is responsible primarily for bitsream packing.

Chapter 3

Overview of System Features

System features are global functionalities that are applicable across each and every media unit. My scope of work in the System Features unit in Media is subdivided into two portions (or functionalities) - implementation and validation of MOCS (Memory Object Control State) across two validation cycles; and creating focussed tests to enable preemption of media decoder workloads using graphics microcontroller.

3.1 MOCS

MOCS, or Memory Object Control State, is a per-surface state definition of memory accesses.
The memory object control state defines behavior of memory accesses beyond the graphics core, including graphics data type that allows selective flushing of data from outer caches, and ability to control cacheability in the outer caches.

This control uses several mechanisms. Control state for all memory accesses can be defined page by page in the GTT entries. Memory objects that are defined by state per surface generally have additional memory object control state in the state structure that defines the other surface attributes.
The driver can program a lookup table of memory attribute combinations. Each sur-

face created by driver can have an index to this lookup table rather than defining all the attributes repeatedly.

3.1.1 End user perspective

From the perspective of an end user, MOCS not getting validated could result in a variety of issues:

- Graphics unresponsiveness/Application hangs
- Exceptions
- Page faults

I developed a testing framework from scratch for Intel's upcoming processor platform. MOCS was not validated for Intel's previous processor platform. There was, hence, a gap in validation which I aimed to fill.

3.2 Global MicroController (GuC) or General purpose Microcontroller

A microcontroller (sometimes abbreviated μC or uC) is a small computer on a single integrated circuit containing a processor core, memory, and programmable input/output peripherals. Program memory (Flash or ROM) is also often included on chip, as well as a typically small amount of RAM. Microcontrollers are designed for dedicated tasks, in contrast to the microprocessors; which are used for general purpose applications.
The target segment for a microcontroller is diverse - Digital Signal Processing, Communication and networking, Embedded devices etc.

Tasks:

The microcontroller employed by Intel in media is responsible for graphics workload scheduling on graphics engines.

It does the following tasks:

- Deciding which workload to run next
- Submitting work to Command Streamer (CS)
- Pre-empting existing workloads as per need
- Notifying host software when work is done

Code that runs on GuC is provided by graphics driver during boot-up and graphics initialization page.
There was a need to validate preemption scenarios using graphics microcontroller. I created ten test cases that would involve preemption among codecs on the MFX (Multi Format Encode and Decode) engine. Characterization (simply put, characterization means verifying if the intent of the test has been met) was done by taking waveform captures.

3.3 Need for Emulation

Emulation is the process of imitating the behavior of one or more pieces of hardware (typically a system under design) with another piece of hardware (called an emulator).

To understand why emulation is needed, consider the scenario when validation is not done at the pre-silicon stage. A huge number of bugs would be found at the RTL stage. That would result in the company making another stepping of the chip; which would increase the Time-To-Market as well as increase production and transportation costs. Thus, functional validation is of utmost importance. Emulation can be used to minimise the number of bugs at the RTL stage of the design process.

3.3.1 Emulation vs. Simulation vs. FPGA prototyping

There are pros and cons to each one of them.
Simulation is often not fast enough for large designs and almost always too slow to run complex or stressful tests or application software against the hardware design.
FPGA-based prototypes are fast and inexpensive, but the time required to implement a large design into several FPGAs can be very long and is error-prone. Changes to fix design flaws also take a long time to implement and may require board wiring changes. Traditional FPGA prototypes may have little debugging capability, probing signals inside the FPGAs in real time is very difficult, and recompiling FPGAs to move probes takes too long.

The usual compromise is to use simulation early in the verification process when bugs and fixes are frequent, and prototyping at the end of the development cycle when the design is basically complete and speed is needed to get sufficient testing to uncover any remaining system-level bugs.
Emulation improves somewhat on FPGA prototypings implementation times, and provides a comprehensive, efficient debugging capability.However, it is a scarce resource due to the heavy cost.

3.3.2 Emulation

Verification through emulation reduces project schedules and cost by enabling validation engineers to maximise bugs at the pre-silicon stage.
Steps followed:

- Model is loaded on emulator.
- The test is run through a hardware functional simulator first. Errors are cleaned up. Passes are queued onto the hardware emulator.
- Errors on the emulator require extensive debugging.

  - Debugging techniques include setting breakpoints and using conditional triggers to explore the system.
  - A trigger is used to stop execution when an error condition occurs and allows us to take a waveform capture around the trigger point. This is due to the limitation that we can take a capture only over few cycles.
  - Signal waveforms can be used to analyze relevant hardware signals and events.

3.4 Summary

This chapter aimed to give an overview of system features - MOCS and GuC based preemption of media workloads. The need for emulation was explained and the emulators used at Intel were desribed in brief. My scope of work in the System Features unit in Media is subdivided into two portions (or functionalities).

- Obtaining L3/LLC Coverage by MOCS
- Creating focused GuC tests

Chapter 4

Implementing Memory Object Control State (MOCS)

Often identifying the problem and implementing its solution is the toughest task. This chapter deals with the methodology I proposed for MOCS (way of outer-cache access), work flow, execution, results across two validation cycles and a basic idea of the scripting involved.

4.1 Identifying and implementing Methodology

Traditionally the system level features were validated using focused test development and execution. This involved a lot of manual effort in test development and had limited coverage with respect to media functionality within a specific system feature scenario.

To overcome these issues, I developed a methodology to make a comprehensive script which could be appended onto a few media tests from each unit (for example - across encoders, decoders, motion compensation engine etc.) to enable the specific feature. This "helper" file was validated and characterized to ensure it created the required system feature scenarios in each of the media pipelines. Once the file was validated with each pipeline, we identified more media tests that need to be validated in system features scenarios.

In this manner, a set of 183 tests were included in the test plan for execution with the aim of enabling MOCS. These tests were executed in emulation and the failures were debugged.

My contributions

I shall go through my contributions for implementing MOCS. First, I developed a Ruby script that defined MOCS and additional randomisations like L3 cache random size configurations and memory buffer size configurations. Next, I added that script onto 183 tests and began execution. Execution is at two levels - first being on a hardware simulator; and then on a hardware emulator. Passes on the simulator were queued onto the emulator and failures were debugged. This was done across two validation cycles.

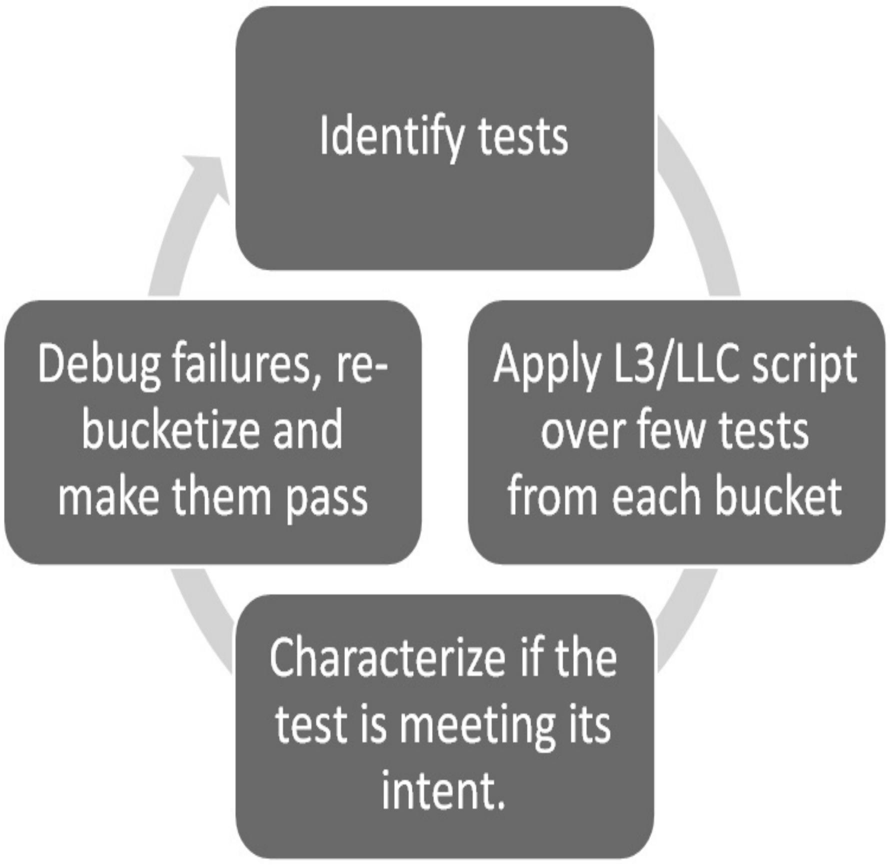

Identify tests
Apply L3/LLC script over few tests from each bucket
Characterize if the test is meeting its intent.
Debug failures, re-bucketize and make them pass

Figure 4.1: Work flow

4.2 Execution

Implementing MOCS functionality across:

- Video decoders - The MFX (Multi Format Codec) engine has a variety of decoders - vc1, vp8, mpeg2, avc, hevc, jpeg to name a few. I took a certain number of tests using source clips of each codec, enabled that particular decoder block and then ran the MOCS script alongwith the decoder base test.

- Video encoders - The PAK (Bitstream Packing and Encoding) engine has an equal number of encoder blocks. A procedure similar to that used in decoders was followed, the only difference being that I enabled the corresponding codec encoder.

- Video enhancement engine - To improve the end user's experience, video enhancement techniques are employed. This includes smoothing the image, applying certain filters to reconstruct damaged pixels, motion compensation, deinterlacing etc. To enable MOCS across this, I first made a generic template (script) that selectively enabled different functions of the video enhancement engine, and appended the MOCS helper file onto this. The resulting tests were executed.

- VME (Video Motion Estimation) unit - VME performs a sequence of operations to find the best mode (i.e. one out of Backward/Forward/Intrapredicted modes) for a given macroblock. It includes motion estimation for various block sizes. It runs only for AVC, MPEG2 and VC1.
  To enable MOCS across this unit, I took a few tests that were developed to run on this unit; and executed them with my MOCS script.

- Audio Video Standard (AVS) - This is a compression standard for digital audio and video. I used a few tests that used the same macroblock and scan specifications as set by the industry standard; and then executed those tests with the MOCS file.

All tests were characterized on the hardware simulator first (checking if they are meeting the intent) and then executed in emulation. The tests that failed in in compiling or in emulation were debugged and fixed.

4.3 Scripting for MOCS

For implementing MOCS across media tests, a comprehensive script was developed. The script defined the MOCS way of cache access, enabled bypassing of L3 cache randomizations as well as some additional randomisations (like memory buffer randomizations).
In the script, a look-up table of memory attribute combinations is defined. Subsequently, the script fixed up a random index in this look-up table.
The script was appended as a helper file above media tests to enable MOCS.

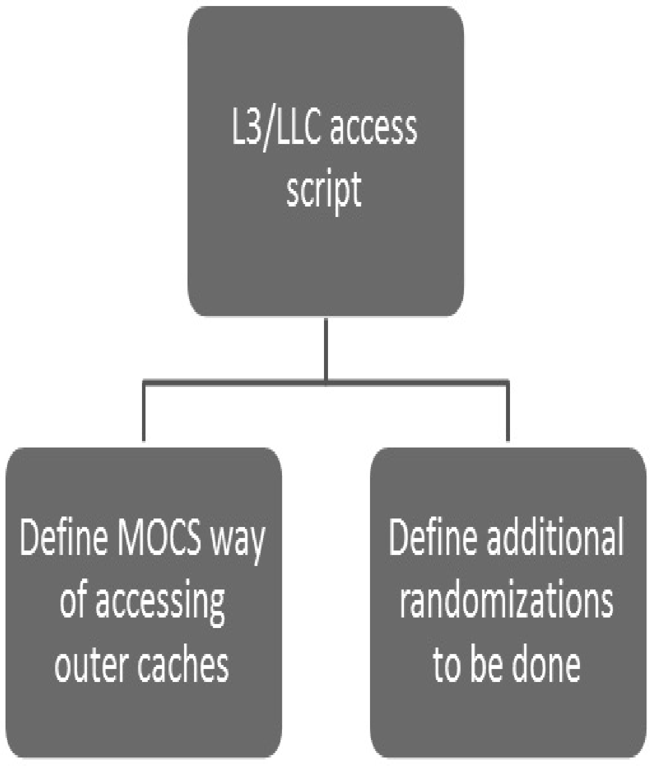
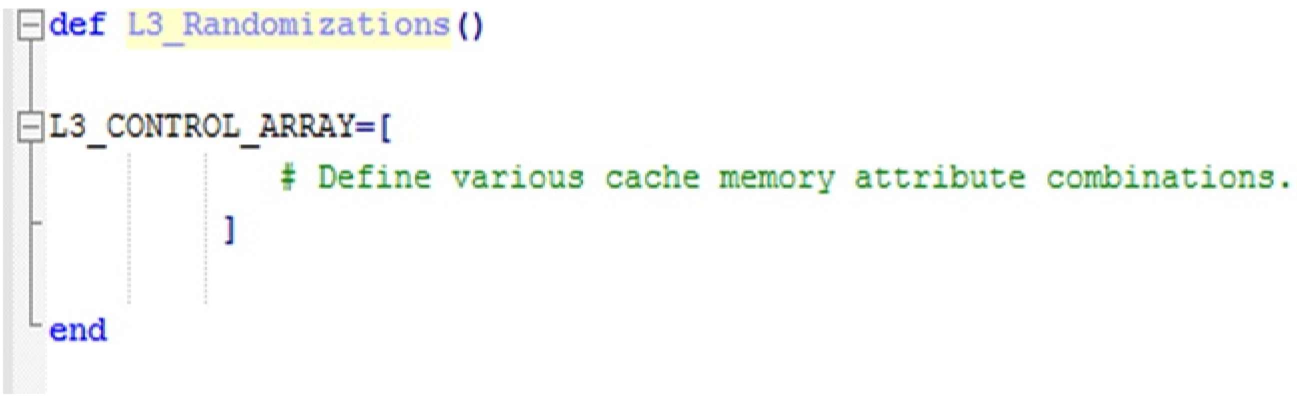

L3/LLC access script
Define MOCS way of accessing outer caches
Define additional randomizations to be done

Figure 4.2: Outer cache access script

4.4 Code snippets

The following code snippets are given below for different scenarios. The next few lines enable media tests to bypass L3 cache randomization settings and go only for LLC randomization. Some media tests do not require the use of L3 cache, and hence the validation engineer can directly specify the same via the command line.



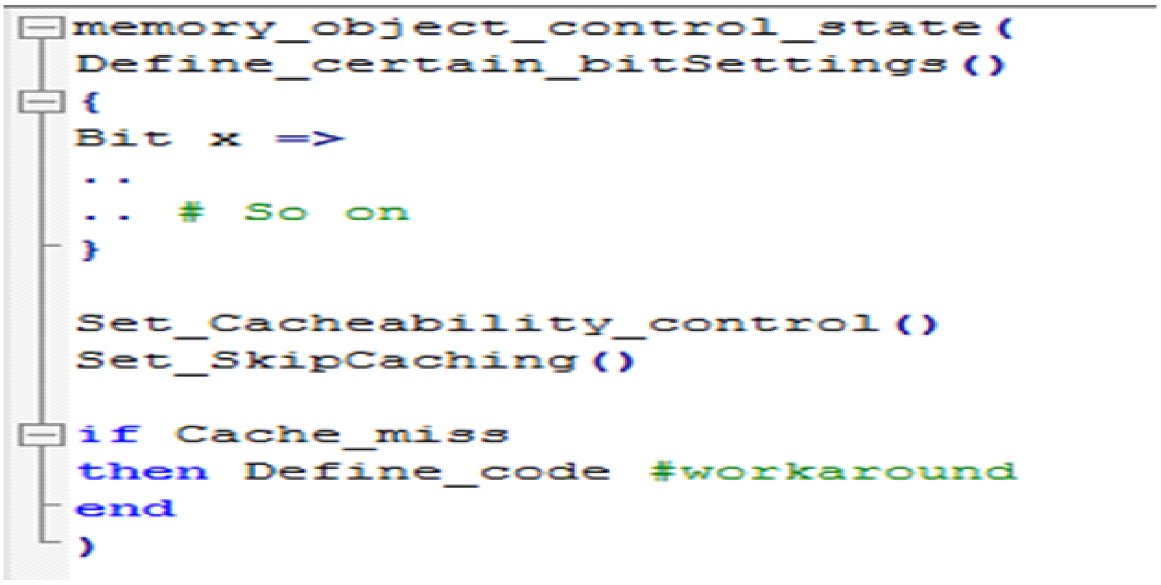
```
def L3_Randomizations()

L3_CONTROL_ARRAY=[
                  # Define various cache memory attribute combinations.
              ]

end
```

Figure 4.3: L3 cache randomizations

The next snippet shows some of the parameters that were hardwired for outer cache access.



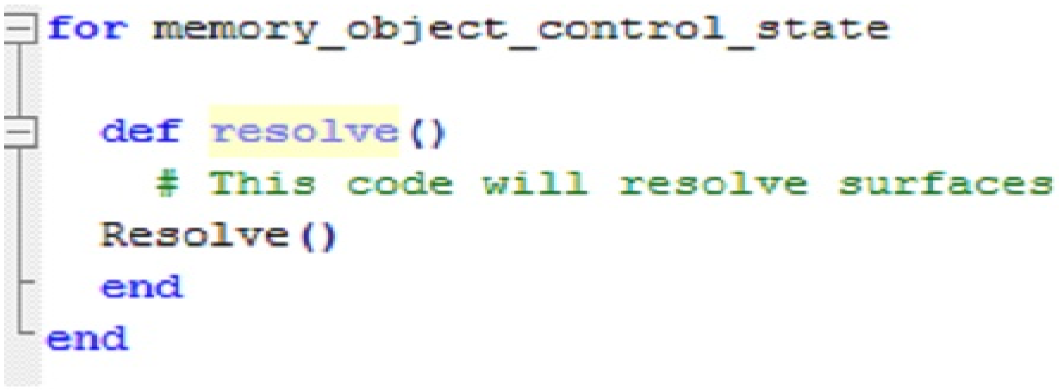
```
memory_object_control_state(
Define_certain_bitSettings()
{
Bit x =>
..
..  #  So  on
}

Set_Cacheability_control()
Set_SkipCaching()

if Cache_miss
then Define_code #workaround
end
)
```

Figure 4.4: Programming MOCS parameters for L3 cache

This snippet shows additional randomizations that were done - for example, memory buffer randomizations.

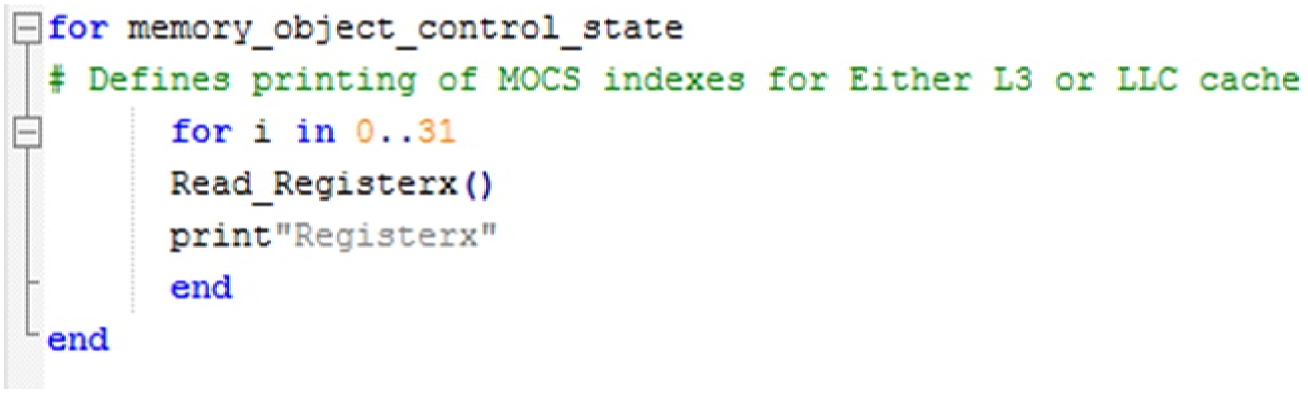


```
for memory_object_control_state

  def resolve()
    # This code will resolve surfaces
  Resolve()
  end
end
```

Figure 4.5: Additional randomizations

At the end, we have a snippet showing the code to print MOCS values for the cache used by the media test.



```
for memory_object_control_state
# Defines printing of MOCS indexes for Either L3 or LLC cache
      for i in 0..31
      Read_Registerx()
      print"Registerx"
      end
end
```

Figure 4.6: Printing MOCS values

4.5 Cache failure buckets: Start of Validation Cycle

The Media pipeline heavily relies upon using L3 cache and LLC (external DRAM Last Level Cache). Yet, there was no universal way of accessing those outer caches and controlling their cacheability. It was a big gap in validation and I aimed to resolve it.

To begin with, I took validation data from the previous processor platform to analyze why cache failures were occuring. This gave me a fair idea of what my script should focus on resolving.

From the previous procesor platform, I took 183 tests to identify and categorize major cache failure buckets. Analysis of errors helped me to narrow down on the possible reasons for it.

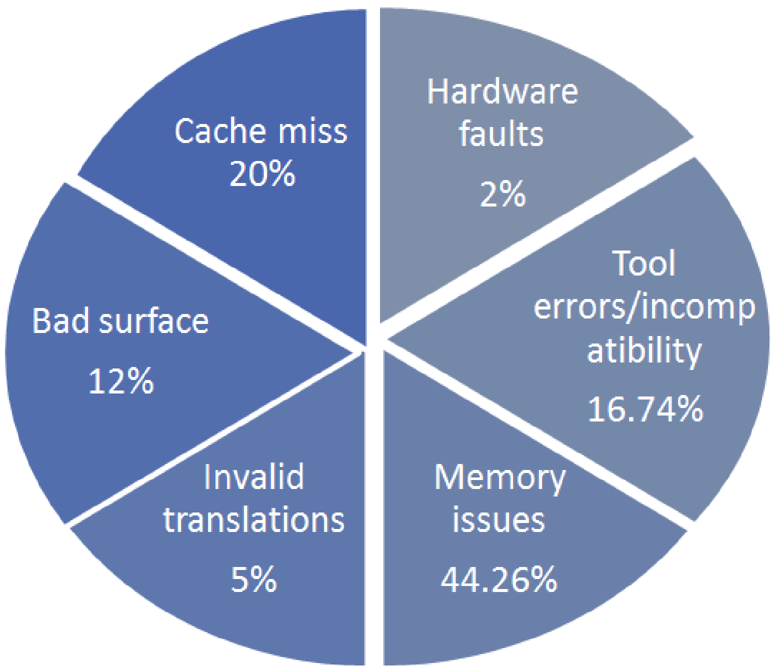


Figure 4.7: Error buckets at the start of first validation cycle

One of the main reasons for cache failure is Memory issues. This could be due to several reasons. Consider the scenario where the test is trying to access the outer cache (say, L3) but there is no space in the cache; or the waiting queue to access the cache is full.

The next major reason is cache miss. Due to large size of cache, there might be a failed attempt to read or write a piece of data in the cache, which results in a main memory access with much longer latency.

Ideally, every media test runs with a specific surface definition. If the processing surface does not match with the one defined in the test, then it causes screen garbage or corruption. It causes a "bad surface".
Another reason is an invalid address translation. Simply put, it means a wrong virtual to physical memory mapping; mainly due to a test content fault or driver fault.
Another reason is Tool errors. We often face software hangs due to a wrong tool version used, or inappropriate command line used to run the test. This is a manual fault and can be avoided if due attention is put.
Lastly, hardware faults also exist. They are due to overheating of the emulator. The only solution to this is to replace the broken fuse

Once I became aware of these errors, I started developing a script that could resolve these errors - mainly memory issues, invalid translations and cache misses. Once the script was ready, I started executing a few tests with my script and checking if MOCS was getting enabled.

4.6 Results

MOCS was enabled and validated across two validation cycles. The process is the same - run the tests first on a hardware simulator, then on a hardware emulator, debug and fix the errors.

4.6.1 Results for first validation cycle

At the end of the first validation cycle, I had a pass rate of 86.96 %; i.e. roughly 157 tests were passing with my MOCS script.
The number of tests failing due to memory issues was reduced to 6 % (10 in number) as opposed to 44 % (80 in number) before enabling MOCS. The number of tests failing due to invalid translations was also reduced to roughly 1 % from 5 % before validation.

Cache misses were also significantly reduced to roughly 1 %; as opposed to 20 % earlier. Bad surface errors were reduced to 1.09 % from 12 % before MOCS enabling.

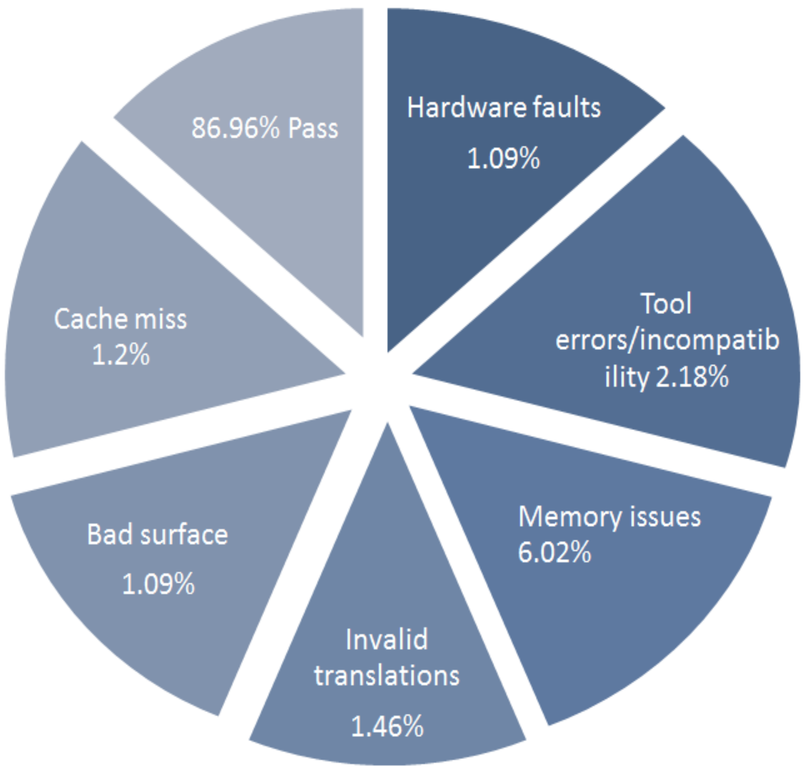


Figure 4.8: Error buckets at the end of first validation cycle

4.6.2 Results for second validation cycle

At the end of the second validation cycle, the results were even more promising. The pass rate was 96.22 %. The bad surface error was completely removed; invalid translations were at a minimum 0.01 % and memory issues were down to 1.85 % (they were at 5 % and 44 % before enabling and validating MOCS).

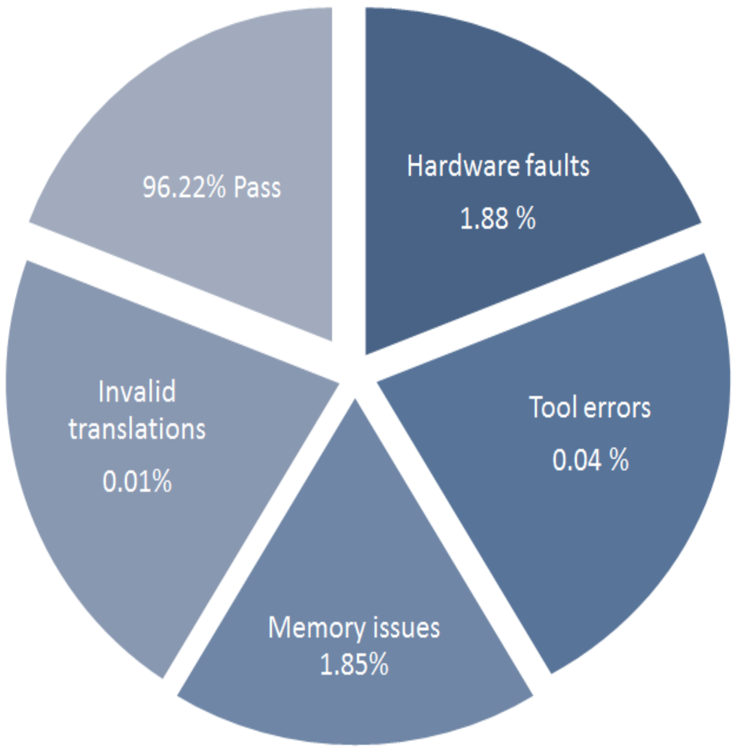


Figure 4.9: Error buckets - current status, second validation cycle

4.7 Summary

I have enabled the MOCS component of system features. Scripting was done in Ruby and Intel tools were used for execution and debug.

For the first validation cycle, the unit had a pass rate of 86.96%. Currently,for the second validation cycle, the unit has a 96.22% pass rate. This has been achieved by continous debug and re-execution of tests on software and later,emulation.

The second area of work was to enable scheduling and preemption of media workloads.

Chapter 5

Enabling Preemption

Preemption of media workloads using graphics microcontroller was a scenario that was not validated in the previous processor platform. This chapter deals with the methodology I proposed for graphics microcontroller based scheduling and preemption, execution, results and a basic idea of the scripting involved (through pseudo-codes).

5.1 Identifying and implementing Methodology

Traditionally, the graphics driver used to enable preemption. This used to be done in the User mode, i.e. the calls (or preemption requests) made by the application had to go through several layers of hardware abstraction (OS, Kernel) to reach the hardware and get the task done. This only meant a time delay between when the request for preemption was issued and when it was serviced/executed. There would be many overhead challenges.

My proposed way is to use the graphics microcontroller (GuC) itself for scheduling and preemption of media workloads. This microcontroller is present on the Graphics Processing Engine (GPE) and any task run by it would talk directly to the hardware, and finish sooner. There is also a huge plus of using the microcontroller when it came to power savings - it could run independently in low power mode; even when the main core was in OFF state. Additionally, it has the authority to schedule workload on graphics sub-systems directly, without host intervention.

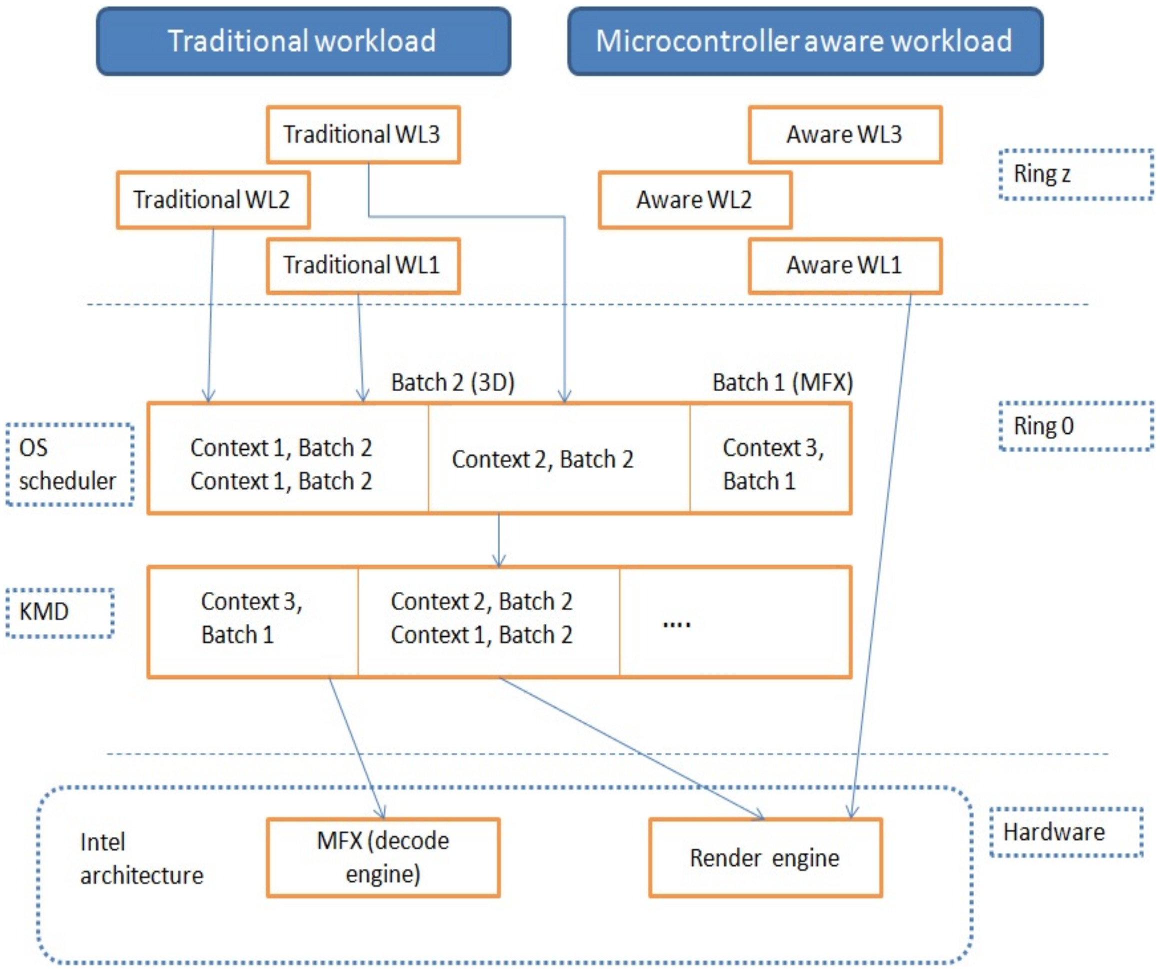


Figure 5.1: Traditional workload scheduling vs. microcontroller based scheduling

As seen in the above figure, there are two kinds of workloads. One that are executed traditionally, by the User mode driver (UMD) and one that are aware of the graphics microcontroller (GuC). The latter are also initiated by the UMD; however they contain instructions to write to dedicated registers that would signal the GuC to wake up, schedule and execute these workloads.

Consider a traditional workload. It is at the highest layer of hardware abstraction (Ring Z); it cannot talk to the hardware directly. Every workload had multiple threads, or contexts. Each thread may require to be run on a different batch, or tar-

get engine. In media terminology, batch may refer to MFX engine, or PAK engine, or Render (display) engine; and so on.
It is the job of the OS schedulerto maintain a queue of incoming contexts and their target batches. The Kernel Mode Driver (KMD) comes in next. It picks up similar contexts and schedules them on the target hardware. Scheduling is done on a first-come-first-serve basis. If a critical context that requires access to MFX engine arrived late, it must wait in queue to get its run on MFX. The time delay is not much, only some nanoseconds, but it is a delay nevertheless.

Now consider a workload that is scheduled by the graphics microcontroller (GuC). It will bypass the OS scheduler, KMD and will directly talk to the hardware. GuC will make sure that there are minimal page faults, it will also ensure optimum usage of target hardware using a priority algorithm. Low priority threads can be preempted by high priority ones.
As is evident, the advantages of using GuC are multifold. It results in the workload getting executed faster and gives better CPU utilisation.

5.2 Work flow

My contributions to the project are:

- Creating focused GuC tests - enabling preemption of media workloads using graphics microcontroller
- Characterisation of the same using pulse triggers and waverform captures

5.3 Execution

I created 10 comprehensive tests which would use the graphics microcontroller to schedule two workloads, and preempt each other once a frame was completed for ei-

ther codec.
For powering on the graphics microcontroller, my script would need to write to dedicated registers in Intel's memory configuration. Once those registers were set, a GuC Interrupt would be raised and GuC would take control from there on.
This was successfully characterised for the following decoder blocks - avc, vc1, vp8, mpeg2 and jpeg. Some additional methods were employed which would preserve the integrity of the data - in simple terms, save the data before the context switch is made and return to the same instruction after preemption is serviced.

5.4 Scripting for Microcontroller based preemption

For enabling scheduling and preemption among media workloads, a comprehensive script was developed.
The basic idea here was to create multiple workloads for MFX (decode engine). The workloads could be any of the following codecs - mpeg2, jpeg, avc, vc1 or vp8. A new workload would be dispatched every time a timer expires. Graphics microcontroller would be used to schedule those workloads on the hardware. Also, the timer would be reset to a pre-defined value after dispatching the workload.

Care was taken to keep the timer value to a moderate value. If it was set too low, then the number of preemptions would increase and no workload would complete its execution. The testing environment would crash. On the other hand, too high a value of the timer would decrease the chances of one codec getting preempted by the other. Hence, i set the timer to a moderate value.
To verify that preemption is indeed happening, captures were taken on emulation.

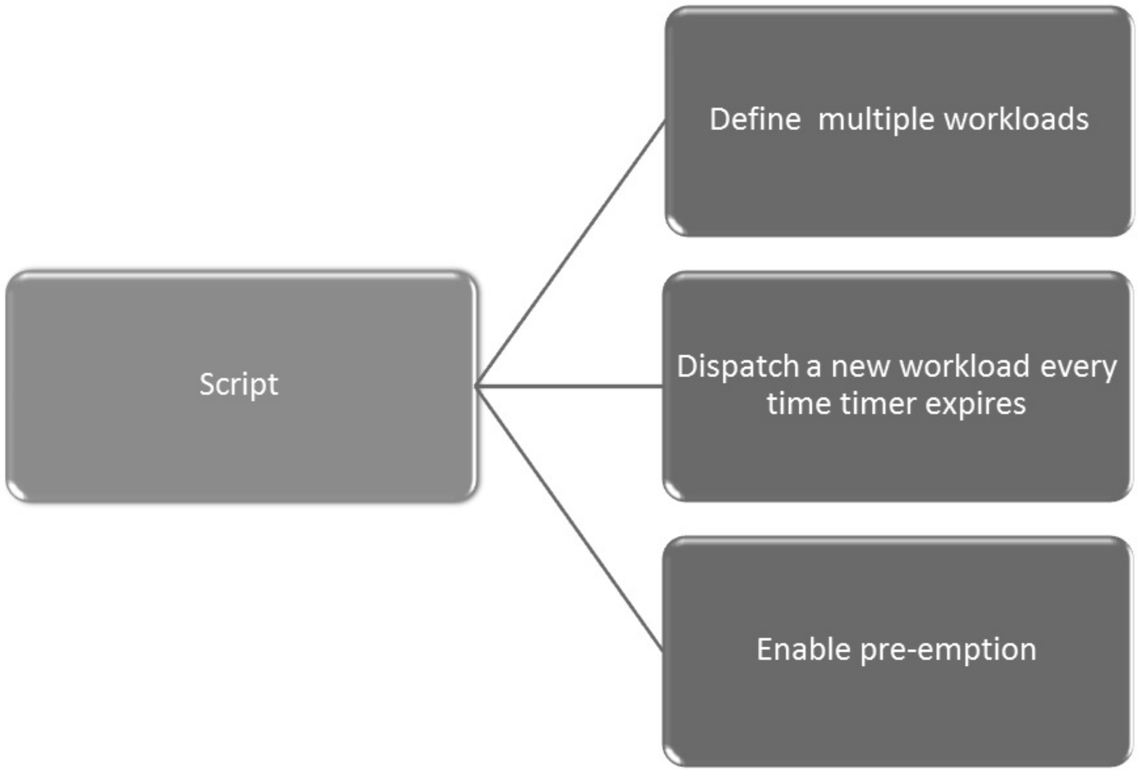
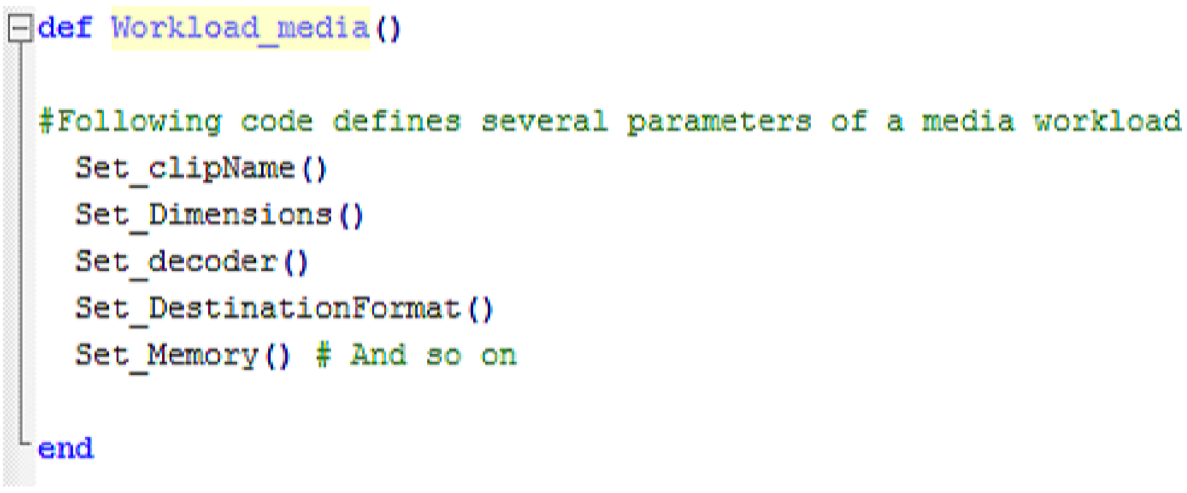

Script
Define multiple workloads
Dispatch a new workload every time timer expires
Enable pre-emption

Figure 5.2: Microcontroller based preemption script

5.5 Code snippets

The below snippet shows basic media parameters that were hardwired (coded) to define a media clip - i.e. to define a workload. Workload could be any one out of jpeg, mpeg2, avc, vc1 or vp8.



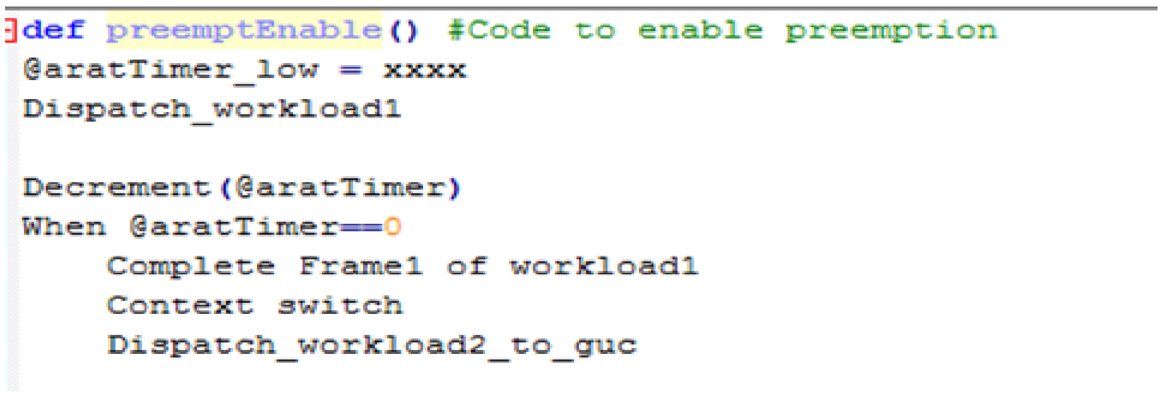
```
def Workload_media()

#Following code defines several parameters of a media workload
  Set_clipName()
  Set_Dimensions()
  Set_decoder()
  Set_DestinationFormat()
  Set_Memory() # And so on

end
```

Figure 5.3: Defining a workload

Next, a timer was defined with a small value, say X. It was decremented to zero. Upon reaching zero, the preemption request was sent alongwith the new workload. After ensuring that the first frame was completed, the graphics microcontroller switched from one workload to another. The timer would then be reset to the same X value.

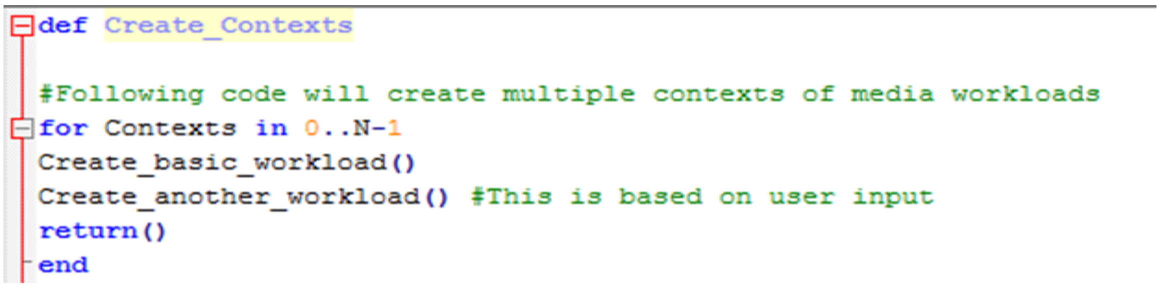


```
def preemptEnable() #Code to enable preemption
@aratTimer_low = xxxx
Dispatch_workload1

Decrement(@aratTimer)
When @aratTimer==0
    Complete Frame1 of workload1
    Context switch
    Dispatch_workload2_to_guc
```

Figure 5.4: Enabling preemption

This snippet shows how multiple instances of different workloads were created.

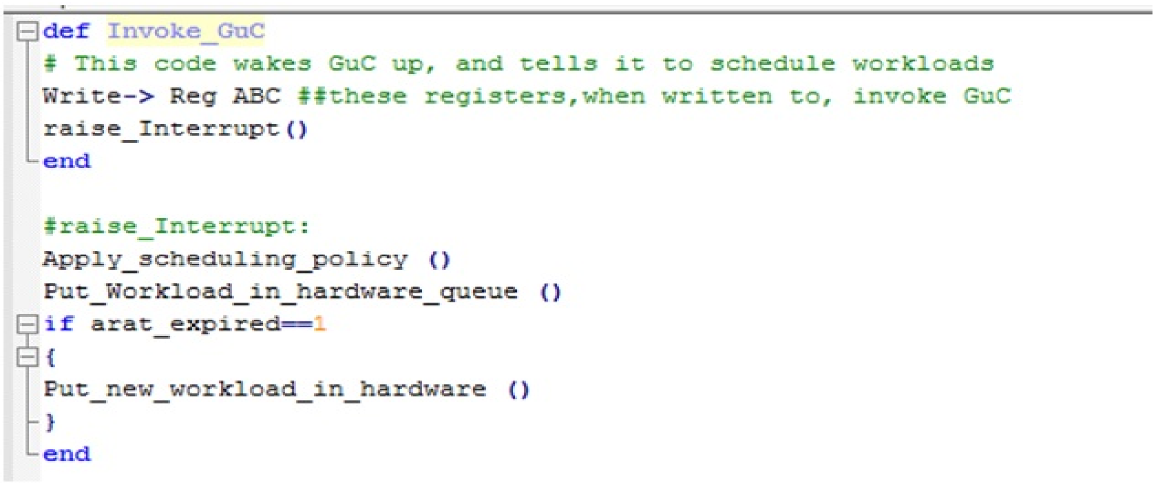


```
def Create_Contexts

#Following code will create multiple contexts of media workloads
for Contexts in 0..N-1
Create_basic_workload()
Create_another_workload() #This is based on user input
return()
end
```

Figure 5.5: Creating a context per workload

This snippet shows how GuC is invoked by setting dedicated registers. Once invoked, Guc schedules the workload on the hardware and also checks for preemption scenarios. It schedules workload based on a priority algorithm.



```
def Invoke_GuC
# This code wakes GuC up, and tells it to schedule workloads
Write-> Reg ABC ##these registers,when written to, invoke GuC
raise_Interrupt()
end

#raise_Interrupt:
Apply_scheduling_policy ()
Put_Workload_in_hardware_queue ()
if arat_expired==1
{
Put_new_workload_in_hardware ()
}
end
```

Figure 5.6: Invoking GuC

5.6 Performance Comparison

I used a generic Intel microcontroller and compared it with GuC on the basis of the time taken for operations like context switch, context save, overheads involved etc. The data was collected by reading the value of dedicated context switch registers when I ran the same media test on both the microcontrollers. The resulting graph is shown below.

As mentioned earlier, the overheads reduce significantly when we use GuC to schedule a workload on the hardware. It can be also seen that in the case of GuC, context switch and execution of the new context takes much less time as compared to that using a generic microcontroller. So, a process scheduled by GuC would finish much sooner than a process scheduled by the generic microcontroller.

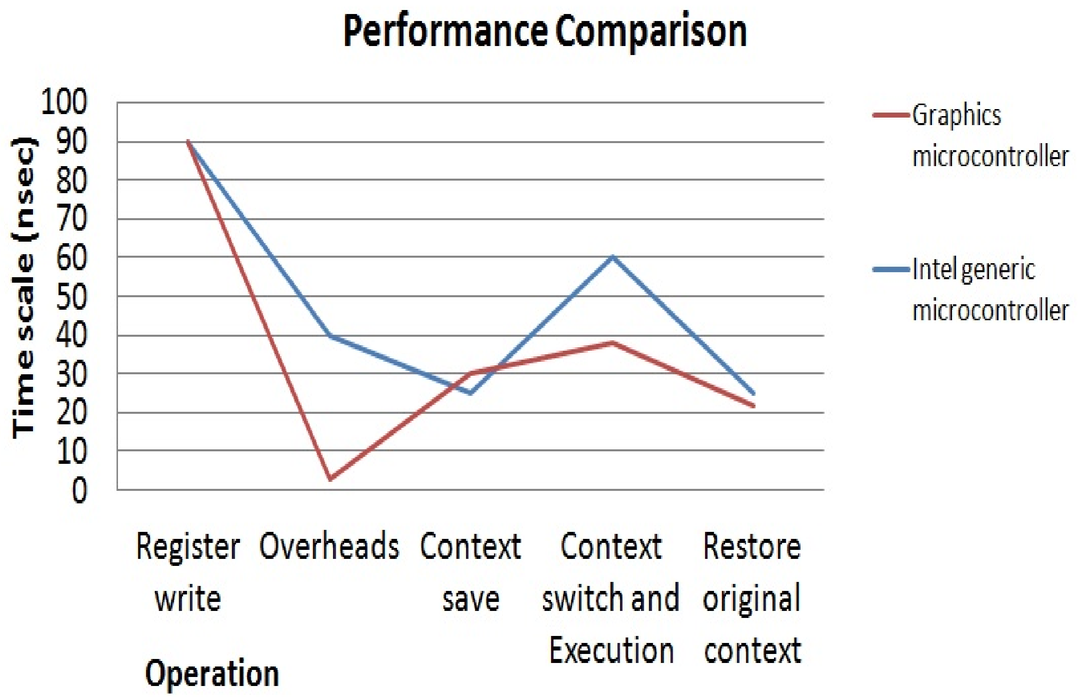


Figure 5.7: Comparison graph

5.7 Intel specific work

Apart from project work, I have gained exposure to what my team does in the Intel framework.

The Graphics System Validation (SV) team is a functional validation team. For complex systems like graphics which occupies nearly 50% of the die in the regular client Central Processing Units (CPU), there is a pressing need to find bugs as early as possible in the design and validation cycle. Emulation platforms enable the team to execute more cycles much faster. The basic reason for using emulation is to identify the bugs as soon as possible in pre-silicon phase and have minimal bugs post silicon. Traditionally, media system validation involved validation of each individual unit in isolation. However, as the SOCs become more advanced (and complex), additional system level improvements are needed in each generation of the SoC. This mandates that the individual media units too exercise these system level advancements early in the validation cycle.

5.8 Summary

Graphics microcontroller based preemption was effectively enabled for Intel's sixth generation processor platform. A set of 10 tests were developed in Ruby, which aimed at preempting two decoder workloads every time a dedicated timer expired. These tests were added in the Media SV validation queue.

Chapter 6

Conclusion

This chapter deals with conclusion and future scope.

6.1 Conclusion

Graphics System Features are global functionalities that are applicable across each and every media unit. Every media test uses an outer cache - either L3 or external LLC (Last Level Cache). So, the outer cache is a global system feature. Yet, there was no universal way of accessing those outer caches and controlling their cacheability. My proposed solution was to use Memory Object Control State (MOCS) as a way of accessing the outer caches. I developed a script that could resolve the main errors that caused cache failures - mainly memory issues, invalid translations, cache misses and bad surfaces. I executed media tests with my script, validated MOCS across two validation cycles. By doing effective execution and debugging of tests; the MOCS unit is now having a pass rate of 96.22% for the second validation cycle. For the first validation cycle, the unit had a pass rate of 86.96%.

The second scope of work was to enable preemption. Traditionally, the graphics driver was used to enable preemption. This was done in the User mode, i.e. the calls (or preemption requests) made by the application had to go through several layers of hardware abstraction (OS, Kernel) to reach the hardware and get the task done. This meant a time delay between when the request for preemption was issued and when it was serviced/executed.

My proposed way was to use the graphics microcontroller (GuC) itself for scheduling and preemption of media workloads. This microcontroller is present on the Graphics Processing Engine (GPE) and any task run by it would talk directly to the hardware, and finish sooner. There was also a huge plus of using the microcontroller when it came to power savings - it could run independently in low power mode; even when the main core was in OFF state. Additionally, it has the authority to schedule workload on graphics sub-systems directly, without Host intervention. GuC is also a global system feature.

A set of 10 tests were developed in Ruby, which aimed at preempting two decoder workloads every time a dedicated timer expired. It was verified that the overheads reduce significantly when we used GuC to schedule a workload on the hardware. Also, context switch and execution of the new context took much less time as compared to when a generic microcontroller was used. So, a process scheduled by GuC finished much sooner than a process scheduled by the generic microcontroller.

6.2 Future scope

- Coverage analysis of the tests developed.

  Media tests are executed on emulation to maximise the number of bugs found at the pre-silicon stage. However, when silicon arrives, it arrives in a very small number and multiple teams at Intel want to validate their features. Its usage, per team and per individual, is limited.

  It is, hence, a good practise to run the tests for each unit through a Coverage Analysis tool and find out what microarchitectural points are being hit (which registers etc.). That way, if it is found that some tests are hitting the same registers, we can go ahead and run only one or two of those tests on the silicon.

- Create and execute tests to plug the gaps identified in coverage analysis.

  If it is found out by the Coverage Analysis tool that some intended registers are not getting hit at all, then additional tests have to be developed that will hit those registers and complete the system coverage.

References

[1] Richardson I. E., "The H.264 Advanced Video Compression Standard", pg: 57-63, *John Wiley & Sons*, Vol. 2, 2010

[2] Ian H. Witten, Radford M. Neal, And John G. Cleary,"Arithmetic Coding For Data Compression", *IEEE Transactions on Communications*, pg:93-98, Vol. 8, September 2007

[3] Wiegand T., Sullivan G. J., Bjontegaard G, and Luthra A., "Overview of the H.264/AVC Video Coding Standard" , *IEEE Transactions On Circuits And Systems For Video Technology*, pg: 560-576, Vol. 13, No. 7, July 2003

[4] Hong Jiang, Sr. Principal Engineer, Chief Media Architect, Intel Corporation, *"The Intel® Quick Sync Video Technology in the 2nd-Generation Intel Core^{TM} Processor Family"*, August 19, 2011

[5] https://01.org/linuxgraphics/documentation/driver-documentation-prms

[6] http://www.larryjordan.biz/what-is-color-sampling-graeme-nattress/

[7] http://www.techotopia.com/index.php/Ruby_Essentials

[8] Prof. S. Sengupta, Department of Electronics and Electrical Communication, I.I.T. Kharagpur, *"Video: basic building blocks*, *"Video encoding basics*, NPTEL lectures

[9] http://www.mentor.com/products/fv/emulation-systems/

[10] http://blogs.mentor.com/embedded/blog/2013/10/11/hardwaresoftware-co-debug-with-sourcery-codebench-virtual-edition-and-veloce-emulation/c