Real-time Distributed Video Transcoding on a Heterogeneous Computing Platform

Zhi Hao Chang

Faculty of Engineering, Computing, and Science (FECS)
Swinburne University of Technology
Sarawak Campus, Malaysia

Submitted for the degree of Master of Engineering

2017
Dedicated to ...

my beloved family
Abstract

The requirements for real-time video transcoding systems have been significantly increased due to the easy and widely available access to high resolution video streams and large-scale applications in recent years. These requirements basically comprise processing video in-messages, resilient to missing & out-of-order frames, capability of storing, accessing & modifying state information grouped along live streaming videos, capability to distribute transcoding instances across multiple machines, deliver low latency response times for high volume video transcoding applications and etc. In this thesis, a real-time distributed video transcoding system working in a heterogeneous environment has been proposed to tackle the high requirements of such applications. It allows multiple computers on the same network to execute the same transcoding task together. As such, the system can then process more video streams in real time.

The proposed approach emphasizes throughput of video data aggregation which involves the continuous input video stream and outcomes of transcoded video streams that are accessible on-the-fly in contrast to the batch-oriented approach such as the MapReduce framework, where output latencies can be significant. Data transmission in terms of micro batches of GOP is selected as the granularity level of data in the proposed system. This is to ensure maximum system throughput obtained by incorporating the trade-off between computation and communication overheads. Besides that, an offline resource-aware task allocation scheme is proposed and has exhibited a fair amount of tasks allocated among heterogeneous computers.

The performance of the proposed system can be further optimized by using a more intelligent scheduler and dynamic task allocation strategies for an optimal fair workload distribution among heterogeneous computing systems.
Acknowledgments

This master research study is made possible through the financial support from Sarawak Information System Sdn. Bhd. as part of a bigger project: "A High Performance Network Video Recorder and Video Management Systems". Swinburne University of Technology has also kindly provided research studentship funding for my postgraduate study, for which I am deeply indebted.

First, my sincere gratitude goes to my main supervisor, Professor M. L. Dennis Wong for the continuous support, his individual insights, patience, immense knowledge, kindness and the motivation he provided for the accomplishment of this dissertation. His guidance helped me always at the time of doing research, in keeping my research focused and on track throughout my Master degree. It was an honour for me to work under his supervision.

In addition, I also want to thank my co-supervisor Dr. Ming Ming Wong, who reminded me to look for research topics when the project first kick started, not just only focus on project milestones and deliverables. My sincere thanks also go to Associate Professor Wallace Shung Hui Wong, David Lai Soon Yeo, Mark Tze Liang Sung, Lih Lin Ng, who provided me an opportunity to participate in a collaborated project by Sarawak Information Systems (SAINS) Sdn. Bhd. and Swinburne Sarawak (SUTS).

My thanks to my fellow friends, Dr. Nguan Soon Chong, Dr. Wei Jing Wong, Dr. Lin Shen Liew, Bih Fei Jong, Wen Long Jing, Jia Jun Tay, Ivan Ling and Tien Boh Luk, for being the most stimulating discussion group that I have had in the past two years. My gratitude goes to Bih Fei Jong, Dr. Nguan Soon Chong, Dr. Wei Jing Wong, Ivan Ling, and Derick Tan who were my project partners and who contributed their time and effort throughout the project milestones.

Last but not least, I would like to thank my parents, my brother and sister for encouraging me to pursue my Master degree and in supporting me spiritually throughout all these years.
Declaration

I declare that this thesis contains no material that has been accepted for the award of any other degree or diploma and to the best of my knowledge contains no material previously published or written by another person except where due reference is made in the text of this thesis.

CHANG ZHI HAO
Publications Arising from this Thesis


# Contents

1 Introduction

1.1 Background to the Media Content Delivery System ............................................. 3
  1.1.1 Video Acquisition, Pre-processing and Encoding ........................................ 3
  1.1.2 Video Transmission, Decoding and Post-processing .................................... 3
  1.1.3 Fundamentals of Video Coding ................................................................. 4
  1.1.4 Video Coding Standards .............................................................................. 5
  1.2 Research Problem and Motivation ..................................................................... 8
  1.3 Research Objective .......................................................................................... 10
  1.4 Significant of Research .................................................................................. 10
  1.5 Thesis Contributions ....................................................................................... 11
  1.6 Thesis Organization ........................................................................................ 12

2 Literature Review

  2.1 Specialized High-Performance Hardware Accelerators ...................................... 13
  2.2 Graphics Processing Units (GPUs) ................................................................. 14
  2.3 The Nvidia Hardware Video Encoder .............................................................. 14
  2.4 Scalable Video Coding ................................................................................... 16
  2.5 Latest Video Coding Standards ....................................................................... 17
  2.6 Adaptive Bit-rate Video Streaming .................................................................. 17
  2.7 Peer-Assisted Transcoding ............................................................................. 18
  2.8 Cloud Video Transcoding ............................................................................... 18
  2.9 Video Transcoding using a Computer Cluster ................................................ 19
  2.10 Resource-Aware Scheduling in a Distributed Heterogeneous Platform .......... 20
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Apache Hadoop: A Batch Processing Infrastructure</td>
<td>22</td>
</tr>
<tr>
<td>3.1.1</td>
<td>A Distributed Storage System in the Batch Processing Infrastructure</td>
<td>22</td>
</tr>
<tr>
<td>3.1.2</td>
<td>The Hadoop Distributed File System</td>
<td>22</td>
</tr>
<tr>
<td>3.1.3</td>
<td>MapReduce</td>
<td>23</td>
</tr>
<tr>
<td>3.1.4</td>
<td>Data Locality in the Hadoop Cluster</td>
<td>24</td>
</tr>
<tr>
<td>3.1.5</td>
<td>Fault-Tolerance</td>
<td>25</td>
</tr>
<tr>
<td>3.1.6</td>
<td>Task Granularity</td>
<td>25</td>
</tr>
<tr>
<td>3.1.7</td>
<td>Discussion</td>
<td>26</td>
</tr>
<tr>
<td>3.2</td>
<td>Apache Storm: A Stream Processing Infrastructure</td>
<td>27</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Big Data Processing in the Real-time Stream Processing Infrastructure</td>
<td>27</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Domains of Real-time Processing</td>
<td>28</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Data Flow Model and Execution Architecture in Storm</td>
<td>29</td>
</tr>
<tr>
<td>3.2.4</td>
<td>Stream Groupings in Storm</td>
<td>30</td>
</tr>
<tr>
<td>3.2.5</td>
<td>Storm Internals</td>
<td>30</td>
</tr>
<tr>
<td>3.2.6</td>
<td>Levels of Parallelism in Storm</td>
<td>31</td>
</tr>
<tr>
<td>3.2.7</td>
<td>Processing Semantics in Storm</td>
<td>32</td>
</tr>
<tr>
<td>3.2.8</td>
<td>Fault Tolerance Daemons in Storm</td>
<td>33</td>
</tr>
<tr>
<td>3.2.9</td>
<td>Daemon Metrics and Monitoring in Storm</td>
<td>34</td>
</tr>
<tr>
<td>3.2.10</td>
<td>Auto Scaling in Storm</td>
<td>34</td>
</tr>
<tr>
<td>3.2.11</td>
<td>The Isolation Scheduler in Storm</td>
<td>35</td>
</tr>
<tr>
<td>3.2.12</td>
<td>Ease of Development in Storm</td>
<td>35</td>
</tr>
<tr>
<td>3.2.13</td>
<td>Discussion</td>
<td>35</td>
</tr>
<tr>
<td>4</td>
<td>Distributed Video Transcoding on a Selected Big Data Processing Platform</td>
<td>37</td>
</tr>
<tr>
<td>4.1</td>
<td>The Batch-oriented Distributed Video Transcoding Method</td>
<td>37</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Adopting a Distributed File System in Storing Video Data</td>
<td>37</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Mapper Task in the Batch Video Transcoding Approach</td>
<td>38</td>
</tr>
<tr>
<td>4.1.3</td>
<td>Reducer Task in the Batch Video Transcoding Approach</td>
<td>39</td>
</tr>
<tr>
<td>4.2</td>
<td>The Proposed Real-time Distributed Transcoding Method</td>
<td>40</td>
</tr>
</tbody>
</table>
CONTENTS

4.2.1 System Architecture ........................................... 40
4.2.2 Data Granularity .................................................. 41
4.2.3 Transcoding Videos at A Larger Scale ......................... 42
4.2.4 The Video Data Spout ........................................... 43
4.2.5 The Transcoding Bolt ........................................... 43
4.2.6 The Sorting Bolt .................................................. 44
4.2.7 The Reporting Bolt .............................................. 44
4.2.8 The Topology Builder ........................................... 44
4.2.9 The Storm Pluggable Scheduler ................................. 49
4.2.10 Storm Custom Stream Grouping ............................... 49
4.2.11 The Resource-Aware Task Allocation Scheme ................. 51

5 Experimental Results and Discussions .......................... 55
5.1 Performance Evaluation ........................................... 55
5.1.1 The Nvidia Hardware Encoder ................................. 55
5.1.2 The Batch-Oriented Distributed Video Transcoding System .... 57
5.1.3 Our Proposed System .......................................... 60
5.1.4 Load Distribution in the Proposed System .................... 61
5.2 Comparative Evaluation of the Fine- and Coarse-Grain Approach for the Proposed System ...................... 65
5.3 Performance Evaluation of the Proposed Resource-Aware Task Allocation Scheme .............................. 70
5.4 Comparative Evaluation of Various Groupings to the Proposed System ........................................ 74
5.5 Quality of Service .................................................. 81

6 Conclusion and Future Works ..................................... 85
6.1 Conclusion .......................................................... 85
6.2 Future Work ....................................................... 86
6.2.1 Online Resource-Aware GOP Allocation ....................... 86
6.2.2 A Resource-Aware Scheduler .................................. 86
6.2.3 Incorporation of GPU’s Parallel Processing to The Proposed System ........................................ 87
# List of Figures

1.1 A typical setup example of DVRs versus the modern NVR .......................... 2  
1.2 A typical example specifying the arrangement order of intra and inter frames in a GOP structure .................................................. 5  
1.3 Frame-level bit size analysis over a time-window of 162 frames .......... 6  
1.4 An example of a sequence of three MPEG coded images, reproduced from [1] ................................................................. 7  
1.5 Format conversion using a video transcoder ........................................... 8  
2.1 The NVENC block diagram, taken from [2] ........................................... 15  
2.2 Adaptive scalable video coding for simulcast applications, taken from [3] 16  
3.1 Data Redundancy in HDFS, reproduced from [4] ................................. 22  
3.2 Daemons of a MapReduce job, reproduced from [4] ................................. 24  
3.3 Domains of real-time processing .......................................................... 28  
3.4 The high level architecture of Storm, reproduced from [5] ...................... 31  
3.5 Different levels of parallelism in Storm, reproduced from [6] ............... 32  
4.1 The proposed Storm-based video transcoding topology .......................... 40  
4.2 Processing data at a larger scale ........................................................... 42  
4.3 FFprobe sample output example ........................................................... 45  
4.4 Selection of computational resources and task allocation ..................... 54  
5.1 The replication factor and block size used in HDFS ............................... 57  
5.2 Percentage increase in system throughput of the batch approach and our proposed system compared to the baseline benchmarked .............. 60
LIST OF FIGURES

5.3 Computer throughput of different data granularities set at resolution 640×480. ........................................... 66
5.4 Computer throughput of different data granularities set at resolution 640×480. ........................................... 67
5.5 Computer throughput of different data granularities set at resolution 640×480. ........................................... 68
5.6 Cluster throughput of different data granularities set at resolution 1920×1080. 69
5.7 Cluster throughput of different data granularities set at resolution 1280×720. 69
5.8 Cluster throughput of different data granularities set at resolution 640×480. 69
5.9 Load distribution of video frames among heterogeneous computers using storm default fields grouping at resolution 1920×1080. .................. 75
5.10 Load distribution of video frames among heterogeneous computers using key fields grouping at resolution 1920×1080. .................. 75
5.11 Load distribution of video frames among heterogeneous computers using the proposed resource-aware scheme on fields grouping at resolution 1920×1080. .................. 75
5.12 Load distribution of video frames among heterogeneous computers using the proposed resource-aware scheme on key fields grouping at resolution 1920×1080. .................. 75
5.13 Load distribution of video frames among heterogeneous computers using default fields grouping at resolution 1280×720. .................. 76
5.14 Load distribution of video frames among heterogeneous computers using key fields grouping at resolution 1280×720. .................. 76
5.15 Load distribution of video frames among heterogeneous computers using the proposed resource-aware scheme on fields grouping at resolution 1280×720. .................. 76
5.16 Load distribution of video frames among heterogeneous computers using the proposed resource-aware scheme on key fields grouping at resolution 1280×720. .................. 76
5.17 Load distribution of video frames among heterogeneous computers using storm default fields grouping at resolution 640×480. .................. 77
5.18 Load distribution of video frames among heterogeneous computers using key fields grouping at resolution 640×480. .................. 77
5.19 Load distribution of video frames among heterogeneous computers using the proposed resource-aware scheme on fields grouping at resolution $640 \times 480$. .................................................. 77

5.20 Load distribution of video frames among heterogeneous computers using the proposed resource-aware scheme on key fields grouping at resolution $640 \times 480$. .................................................. 77

5.21 The time required to transcode the video in a heterogeneous environment using fields grouping at resolution $1920 \times 1080$. .................................................. 78

5.22 The time required to transcode the video in a heterogeneous environment using partial key grouping at resolution $1920 \times 1080$. .................................................. 78

5.23 The time required to transcode the video in a heterogeneous environment using the proposed resource-aware scheme on fields grouping at resolution $1920 \times 1080$. .................................................. 78

5.24 The time required to transcode the video in a heterogeneous environment using the proposed resource-aware scheme on partial key grouping at resolution $1920 \times 1080$. .................................................. 78

5.25 The time required to transcode the video in a heterogeneous environment using fields grouping at resolution $1280 \times 720$. .................................................. 79

5.26 The time required to transcode the video in a heterogeneous environment using partial key grouping at resolution $1280 \times 720$ .................................................. 79

5.27 The time required to transcode the video in a heterogeneous environment using the proposed resource-aware scheme on fields grouping at resolution $1280 \times 720$. .................................................. 79

5.28 The time required to transcode the video in a heterogeneous environment using the proposed resource-aware scheme on partial key grouping at resolution $1280 \times 720$. .................................................. 79

5.29 The time required to transcode the video in a heterogeneous environment using fields grouping at resolution $640 \times 480$. .................................................. 80

5.30 The time required to transcode the video in a heterogeneous environment using partial key grouping at resolution $640 \times 480$. .................................................. 80

5.31 The time required to transcode the video in a heterogeneous environment using the proposed resource-aware scheme on fields grouping at resolution $640 \times 480$. .................................................. 80
LIST OF FIGURES

5.32 The time required to transcode the video in a heterogeneous environment using the proposed resource-aware scheme on partial key grouping at resolution 640\(\times\)480. ................................................................. 80

5.33 The response time of the batch transcoding system compared to the proposed system. ................................................................. 82

5.34 The signal to noise ratio of the proposed system when output resolution is set to 1920\(\times\)1080, 1280\(\times\)720 and 640\(\times\)480 respectively .............................................. 83

5.35 The PSNR of a transcoded video sample produced from the proposed system, generated using the MSU Video Quality Measurement Tool ........................................... 84
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>CPU and GPU specification, the average transcoding rate of four computers and two GPUs used in the experiment, each with a unique label.</td>
<td>56</td>
</tr>
<tr>
<td>5.2</td>
<td>Transcoding speed of NVENC when output resolution is set to 1920×1080, 1280×720 and 640×480 respectively.</td>
<td>56</td>
</tr>
<tr>
<td>5.3</td>
<td>Transcoding speed of the batch-oriented distributed video transcoding approach when output resolution is set to 1920×1080, 1280×720 and 640×480 respectively.</td>
<td>59</td>
</tr>
<tr>
<td>5.4</td>
<td>The transcoding speed of the proposed system when output resolution is set to 1920×1080.</td>
<td>62</td>
</tr>
<tr>
<td>5.5</td>
<td>The transcoding speed of the proposed system when output resolution is set to 1280×720.</td>
<td>63</td>
</tr>
<tr>
<td>5.6</td>
<td>The transcoding speed of the proposed system when output resolution is set to 640×480.</td>
<td>64</td>
</tr>
<tr>
<td>5.7</td>
<td>Transcoding speed of the proposed system when data granularity is set to a frame interval of 3, 5, 7, 9, 11 and 15 at resolution 1920×1080.</td>
<td>66</td>
</tr>
<tr>
<td>5.8</td>
<td>Transcoding speed of the proposed system when data granularity is set to a frame interval of 3, 5, 7, 9, 11 and 15 at resolution 1280×720.</td>
<td>67</td>
</tr>
<tr>
<td>5.9</td>
<td>Transcoding speed of the proposed system when data granularity is set to a frame interval of 3, 5, 7, 9, 11 and 15 at resolution 640×480.</td>
<td>68</td>
</tr>
<tr>
<td>5.10</td>
<td>The frame distribution of the proposed system with various groupings when output resolution is set to 1920×1080.</td>
<td>71</td>
</tr>
<tr>
<td>5.11</td>
<td>The frame distribution of the proposed system with various groupings when output resolution is set to 1280×720.</td>
<td>72</td>
</tr>
<tr>
<td>5.12</td>
<td>The frame distribution of the proposed system with various groupings when output resolution is set to 640×480.</td>
<td>73</td>
</tr>
<tr>
<td>5.13</td>
<td>Mapping PSNR to MOS, taken from [106].</td>
<td>82</td>
</tr>
</tbody>
</table>
# Commonly Used Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABR</td>
<td>Adaptive Bit Rate</td>
</tr>
<tr>
<td>AVC</td>
<td>Advance Video Coding</td>
</tr>
<tr>
<td>AWS</td>
<td>Amazon Web Services</td>
</tr>
<tr>
<td>CCTV</td>
<td>Closed-Circuit Television</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DSMS</td>
<td>Data Stream Management System</td>
</tr>
<tr>
<td>DVR</td>
<td>Digital Video Recorder</td>
</tr>
<tr>
<td>EC2</td>
<td>Elastic Compute Cloud</td>
</tr>
<tr>
<td>GOP</td>
<td>Group Of Pictures</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphical Processing Unit</td>
</tr>
<tr>
<td>HDFS</td>
<td>Hadoop Distributed File System</td>
</tr>
<tr>
<td>HEVC</td>
<td>High-Efficient Video Coding</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hyper Text Transfer Protocol</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>IPB</td>
<td>Intra, Predictive and Bi-directional Predictive Frames</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>MOS</td>
<td>Mean Opinion Score</td>
</tr>
<tr>
<td>MPSOC</td>
<td>Multiprocessor System-on-Chip</td>
</tr>
<tr>
<td>NVENC</td>
<td>Nvidia Hardware Video Encoder</td>
</tr>
<tr>
<td>NVR</td>
<td>Network Video Recorder</td>
</tr>
<tr>
<td>OLAP</td>
<td>Online Analytical Processing</td>
</tr>
<tr>
<td>OLTP</td>
<td>Online Transaction Processing</td>
</tr>
<tr>
<td>ONVIF</td>
<td>Open Network Video Interface Forum</td>
</tr>
<tr>
<td>P2P</td>
<td>Peer-to-Peer</td>
</tr>
<tr>
<td>PKG</td>
<td>Partial Key Grouping</td>
</tr>
<tr>
<td>PSNR</td>
<td>Peak Signal to Noise Ratio</td>
</tr>
<tr>
<td>RTP</td>
<td>Real-time Transport Protocol</td>
</tr>
<tr>
<td>RTSP</td>
<td>Real-time Streaming Protocol</td>
</tr>
<tr>
<td>SVC</td>
<td>Scalable Video Coding</td>
</tr>
</tbody>
</table>
CHAPTER 1

Introduction

Over the past decades, we have seen the rapid development and tremendous growth of video surveillance solutions in today’s security market demands [7]. Video surveillance systems have changed from a conventional analogue closed-circuit television (CCTV) and video tape archiving environment to a self-contained digital video recorder (DVR) environment; and it is now further evolving into a centralized network video recorder (NVR) consisting of centrally managed digital Internet Protocol (IP) cameras, which is used to record, playback and view video streams [8]. Fig. 1.1 depicts the differences between a traditional DVRs and the modern NVR based setup. Many standards were leveraged to enable such transformation. These include the Real-Time Streaming Protocol (RTSP [9]) for video streaming via network transmission, HTML-5 [10] for multimedia web browsing without additional plug-ins and the Open Network Video Interface Forum (ONVIF [11]) which overcomes interoperability limitation among IP devices. Enterprises around the world have already spent a lot of money and resources on video surveillance systems including network systems, storages and computing infrastructures, supporting all systems as a whole [8].

As cloud computing and distributed technologies are becoming more mature for use in large-scale frameworks such as big data mining [12], machine learning [13] and behavioural patterns analytics on social network platforms [14], it becomes logical for one to look at using distributed technologies in video surveillance systems to process large scale video data. The most computationally expensive process found in an IP network video surveillance system is video transcoding [15]. A video transcoder performs several key operations which include video resolution, bit rate and format conversion from one compressed video stream into another [16]. An increased number of multiple transcoding instances in a large-scale surveillance system would lead to a
huge amount of CPU resources consumption, leaving limited available CPU resources for the media server and web applications. Consequently, it slows down the overall transcoding process. Therefore, a more cost-effective method for accessing computing resources is required to realize multiple instances transcoding and speed up the overall performance of the video surveillance system. The remaining parts of this chapter outlines the basic components of video processing which include video compression, video data structures, video container formats and video streaming, from the acquisition of video sequences and transmission to display.

Figure 1.1: A typical setup example of DVRs versus the modern NVR
1.1 Background to the Media Content Delivery System

1.1.1 Video Acquisition, Pre-processing and Encoding

In a digital video coding system [17], a pool of video sequences is first captured by a digital capture device such as a high resolution digital video camera. Pre-processing operations are then performed on the raw uncompressed video frames to: 1) reduce video resolution; and 2) correct and convert the color pixel format conforming to a chosen standard. Next, an encoding process is carried out to transform the pre-processed video sequences into coded bit-streams. The purpose of video encoding is to provide a compact and bandwidth saving representation for the ease of network transmission later.

1.1.2 Video Transmission, Decoding and Post-processing

Prior to transmitting video streams over a network, the coded bit-streams are packetized into an appropriate transport format as prescribed by the chosen transport protocol, e.g. the Real-time Transport Protocol (RTP [18]). The transmission of video streams involves client-server initiation, the receipt of video streams at the client side, as well as, supported protocols, recovery of lost packets. At the client end, reconstructed video sequences of the received bit-stream are achieved through a decoding process. On the other hand, video encoding often adopts lossy compression in order to meet the targeted transmission bandwidth constraint where the decoded video typically suffers from quality degradation from the original video source as some reconstructed pixels will be at best an approximation of the originals. If a lost packet fails to be recovered during transmission, the decoder will apply an error concealment technique, focusing on the recovery of the corrupted video frame. After decoding, post-processing operations are performed on reconstructed video sequences, which include color correction, trimming and re-sampling. Finally, the video sequence is ready to be displayed. The latency from decoding to display of the video sequences is one of the critical factors in determining the viewing experiences of the viewers.
1.1.3 Fundamentals of Video Coding

The Structure of a Video Sequence

Video sequences basically are a series of time varying pictures, which are presented with successive pictures in a constant time interval of milliseconds [19]. In order to provide real-time video motions, a picture rate of at least 25 pictures per second is required. The unit of picture rate is often referred to as frame rate (frames per second). High definition captured videos found today can easily apply picture rates of 50 to 60 frames per second.

The Structure of a Coded Bit-stream

A video encoder basically transforms an input video sequence into coded bit-streams which contain the essential information that is required to decode the bit-streams correctly and re-construct an approximation of the original video sequence for displaying at the client side. A video encoding process involves applying compression to the video source by eliminating redundancies found in the video sequence. A video sequence basically consists of two different types of redundancy: namely, the spatial and temporal redundancies. The correlation presents between pixels within a video frame is known as spatial redundancy. A video frame with the removal of such spatial redundancy that is present within the frame is often named as an intra coded frame. On the contrary, temporal redundancies are present between successive frames. Successive frames of a sufficiently high frame rate video sequence are likely to be highly identical to one another. Hence, the removal of such temporal redundancies involves between identical successive frames in a video sequence is referred as Inter Coding. The spatial redundancy of a frame is removed through the implementation of transform coding techniques while temporal redundancies present between successive frames are discarded through techniques of motion estimation and compensation [20].

IPB Frames in the GOP Structure

There are three common types of frames are adopted in video compression, namely: the intra frame, predictive frame and bi-directional predictive frame [21]. In short, I frames are also referred to as intra frames, while B and P are known as inter frames. The repeated arrangement of such a collection of successive types of frames forms a
Group of Pictures (GOP [22]) structure, as shown in Fig. 1.2. The GOP’s size is generally described by the distance between two successive I frames. For example, when the GOP length is set constant at 11, the coding order of the GOP structure is set as IPBPBPBPBPBPBPBPBPBPBPBPBPBPBPBPBPBPBP... and the intra-period adopt value is usually between 3 to 12 frames. Furthermore, Intra frames can be differentiated from inter frames by their byte size differences [1]. For example, a test sequence of encoded H.264 video with a GOP length of 15 frames is used to plot Fig. 1.3. The figure illustrates that larger byte sizes correspond to I frames, while smaller byte sizes are B frames and intermediate byte sizes are P frames. Hence, the size of GOP used can be indicated by the interval that appeared at the peaks of I frames which corresponds with the distance of two successive I frames. It is also worth noting that the byte sizes of GOP frames follow the spatial and temporal activities of the test sequence, where complex frames require more bits for frames representation, while the static and simple frames are presented in fewer bits.

1.1.4 Video Coding Standards

Intra-frame-only Compression

Motion JPEG (M-JPEG) is an example of intra-frame-only compression codecs where all frames are individually compressed into JPEG images. Hence, the resulting quality of the compressed image is independent from the motion of successive images. This type of video compression usually has a low latency in image processing and each compressed image has a consistent bits size. In a poor bandwidth network, image resolution is given a higher priority so that the transmitted images can retain their qualities and lost packets of images would be discarded. Key advantages of M-JPEG are less delay issues when is used with audio, ideal for courtroom evidences, less error propagation since every frame is coded independently, technology is simpler which
Figure 1.3: Frame-level bit size analysis over a time-window of 162 frames

Can lower the cost of the media server and easier to work with video analytics and parallel processing. It had become the most widespread image compression format used and found in every network camera until today [23].

Group of Pictures based Compression

Modern inter-frame video codecs such as MPEG1 [24], MPEG2 [25], MPEG-4 [26], H.264 [27] and H.265 [28] achieve better compression ratios than the intra-only video codecs. For example, the MPEG-4 video standard which was introduced in the late 1998, is primarily used for web media streaming, CD distribution and video recording. MPEG-4 uses both key frames and inter-frames for high efficient compression. Simply described, the first frame of a MPEG stream must be a key frame (I frame), followed by inter-frames (B and P frames) that are different from one or more reference images. The adoption of MPEG video compression can greatly reduce the amount of data to be transmitted over the network compared to M-JPEG compression. This is illustrated in Fig. 1.4 where information containing the differences between the first and successive frames is transmitted only. Key advantages of MPEG video compression are to reduce the amount of storage and network bandwidth used by a surveillance system. However, since inter frames only contain the differences between the previous and subsequent frames, they may raise issues when used as court evidence. Furthermore, other disadvantages include high error propagation where multiple coded frames rely heavily on one another, harder to implement for parallel processing and proprietary
licensing is required for commercial use.

The Video Container Format

Video container [29] is an encapsulation method to guarantee that the packaged audio/video, metadata, caption text, subtitles and index points stay synchronized. Video containers are also widely known as video wrapper formats which carry compressed data but this does not make them a codec format. The container also plays an important role in defining how metadata should be formatted for the interchange and wide use guarantees [30]. Most popular multimedia containers include 3GP [31], WMV [32], AVI [33], Matroska [34], MP4 [35], FLV [36] and Ogg [37].

Video Transcoding

The process of converting a given video from one encoding format into another format is known as video transcoding [38]. As shown in Fig. 1.5, a video encoding format can be characterised into coding standard, spatial resolution, frame rate, bit rate and video content. A typical application of video transcoding is to adapt to the lower network bandwidth by lowering the bit rate of previously compressed video bit-streams. For instance, a high bit rate TV program is initially compressed for studio applications, and also for channel broadcasting at a much lower bit rate [39]. A video transcoding operation basically comprises a decoding stage, content-adapting operations, followed by an encoding stage. This re-encoding process is inherently a highly intensive task as there is no correlation exploited between the input and output of the transcoder [40].
**1.2 Research Problem and Motivation**

Over the last few years, many attempts have been made to improve the performance of video transcoding in video surveillance applications. H.264/AVC video dedicated hardware encoders have been accommodated in commodity hardware including multi-core Central Processing Units (CPU) and many-core Graphics Processing Units (GPU) [41]. The adoption of Nvidia Video Encoder (NVENC) has demonstrated a high FHD video encoding speed at 200 frames per second in a single transcoding session [41]. However, real-time multiple transcoding sessions on a single GPU are often limited by supported GPUs. In [42], affordable Nvidia Kepler and Maxwell class GeForce GPUs are allowed to transcode maximum of two concurrent streams per system while high performance computing GPUs such as Tesla, Grid and Quadro are fully supported up to the transcoder’s throughput capacity. Issue arises as industrial developers often prefer to add new workstations or servers to a separate network of IP cameras rather than an expensive investment on high performance computing GPUs specifically dedicated for video transcoding only. Furthermore, adding more GPUs to the system can further scale up but until limited by CPU utilization [43]. For instance, if each transcoding process utilizes CPU at 22 percent, the number of simultaneous full HD encodes in a system will be bound by 5. There are yet other aspects of CPU utilization such as OS scheduling, video ingest overhead, initial processing and etc that are not taken into consideration.

Meanwhile, cloud computing has emerged as a long-held promising technology for storage and computing services all over the internet around the globe [44]. Therefore, video transcoding and streaming services in the cloud offer a highly reliable, scalable and efficient solution for online video services due to their high speed computation capabilities that are highly parallel. Cloud computing basically consists of a cluster of loosely or tightly connected computers working together as a single system. On a single computer, each instance operates independently from any other instances and

![Figure 1.5: Format conversion using a video transcoder](image)
therefore all computers working together can provide the advantages of powerful parallel computing capabilities. Since computing nodes can be heterogeneous, the cloud computing platform becomes relatively inexpensive and therefore has an easily expandable cluster size. In order to improve such high efficiency of distributed computing and minimize the overall response time of tasks, authors in [45] had proposed the most popular distributed computing paradigm, named MapReduce. In this paradigm, computing efficiency of the system has been improved via load balancing on all computing nodes which is, splitting the input data into key/value pairs and handle them in a distributed manner. Hadoop [46], or more specifically the MapReduce framework, is originally designed as a batch-oriented processing algorithm which focuses on optimizing the overall completion time for batches of parallel transcoding tasks [47]. Video transcoding in such architectures is not designed to work on-the-fly which means users cannot start viewing the video as soon as the first video segment is transcoded and received.

Another issue arises when the batch processing approach is attempted on hadoop heterogeneous platforms. Taking data locality into account when handling multiple input files can greatly reduce the amount of network traffic generated between compute instances. The current Hadoop block placement policy assumes that the homogeneity nature of computing nodes always holds in a Hadoop cluster. In launching speculative map tasks, data locality has not been taken into consideration as it is assumed that the data is mostly available locally to mappers. The initial data placement algorithm proposed in [48] begins by first splitting a large input file into even-sized chunks. The data placement algorithm then schedules the sharded chunks to the cluster according to the node’s data processing speed. Regardless of heterogeneity in node’s processing capacity, the initial data placement scheme distributes input chunks evenly so that all nodes can have their local data to be processed within almost the same completion time. In the case of an over-utilized node, file chunks are repeatedly moved by the data distribution server from an over-utilized node to an under-utilized node until the workload is fairly distributed. Thus, their proposed data placement scheme adaptively distributes the amount of data stored evenly in each node in order to achieve an improved data-processing speed while maintaining the initial data locality. However, this method is not suitable for dynamic resource-aware task allocation via video content due to corrupted video frames acquired when video chunks are stored as HDFS blocks. Also, video chunks re-distribution approaches do not distribute fairly enough
as the reconstructed batches of video chunks are much bigger blocks than a few dozen of video frames found in a GOP structure.

Apart from difficulty in taking advantage of data locality during task re-allocation for heterogeneous systems, Hadoop distributes worker slots by the amount set in static configuration and will never be changed when the batch job is running. To address this issue, a data pre-fetching mechanism proposed by [49] can be used to pre-fetch the data to the corresponding under-utilized compute nodes in advance. The basic idea of the data pre-fetching mechanism is to overlap data transmission with data processing. In this way, when the input data of a map task is not available locally, the overhead of data transmissions is hidden. However, a highly coordinated approach is needed between the scheduling algorithm and the data pre-fetching mechanism for task re-allocation. For this reason, we prefer stream processing over the batch distributed approach when furthering our future study on dynamic resource allocation for distributing video transcoding in a heterogeneous computing environment.

1.3 Research Objective

The main research objective of this thesis is to investigate the existing parallel computing infrastructures, how they are being implemented from the point of view of stream processing and batch processing, followed by proposing a method or architecture on how to implement video streams transcoding on such infrastructures in an efficient manner. The investigation will begin with comparative studies and a search for the required system components. In addition to this, a baseline version of such systems needs to be implemented on a related work, selecting a big data processing platform for the purpose of experimentation, comparative studies, performance analysis of system components and the proposed system as a whole.

1.4 Significant of Research

Many public cloud video transcoding platforms found today do not offer the real-time requirement in producing transcoded video file or streams that are accessible on-the-fly while transcoding. The real-time capability of the proposed system has enabled distributed live transcoding to be realized on a larger scale, without having to wait for the entire transcoding process to complete. Meanwhile, dedicated standalone compute
instances found in the public cloud market are scaled on an on-demand basis. Such a single-tenancy approach would become costly for small and medium-sized enterprises to transcode videos at a larger scale. Thus, a shared or multi-tenancy architecture in providing such cloud video transcoding services is desired and can be facilitated by using the large scale and flexible data aggregation capabilities in Apache Storm. The main idea is to allow clients to contribute an affordable number of cloud compute instances to the cluster managed by our proposed system. An expanding cluster of such systems enables a cost effective way to share a collection massive transcoding resources among all clients. This is achieved by having the cluster’s resources to be shared among all clients where compute resources utilization can be maximized during idle or off-peak hours. In other words, online customer communities can leverage the unused resource availability of the remaining shared systems onto their transcoding processes. In private cloud applications, a resource-aware allocation scheme would play an important role in distributing workloads fairly among heterogeneous cloud computing instances in a company’s private data center.

1.5 Thesis Contributions

The thesis has contributed to the design of a method for transcoding streaming videos based on a big data processing platform. More specifically, a large-scale video transcoding platform that is capable of distributedly transcode incoming live video streams in real time has been realized in the course of the research. This has enabled the outcomes of transcoded video streams to be accessible on-the-fly, and thereby facilitates users to preview the content of the video output streams while transcoding, making sure the desired quality is delivered. Secondly, impacts of data granularity on the proposed real-time distributed video transcoding platform were studied and the best trade-off between the load and communication overheads has allowed the parallel computers to perform identically to their local baseline benchmarks which are to their maximum computing capacities. Apart from that, a resource-aware task allocation scheme has also been proposed for workload balancing by taking into consideration of the heterogeneous capabilities of computing nodes in a Storm cluster. This has enabled a fair amount of sharded video streams to be allocated for each computer in accordance with their computing ratios in real time.
1.6 Thesis Organization

This section gives a brief overview of chapter organization in this thesis. Chapter 2 will investigate the related literature and studies that are relevant to the understanding of development in parallel video transcoding. In Chapter 3, overviews of Apache Hadoop and Apache Storm for data processing are introduced. After that, implementation of the batch and real-time distributed video transcoding architectures are presented in Chapter 4. Chapter 5 presents experiments and results obtained for related studies and the proposed system. Last but not least, Chapter 6 summarizes the proposed methods and results obtained in previous chapters along with some limitations, putting forward some possible directions for future research.
Chapter 2

Literature Review

This chapter presents a wide range of relevant literatures on cost-efficient video transcoding solutions. It includes ten sections. The first section describes the concept of hardware efficiency for a multi-core processor in parallelizing the tasks of a H.264 encoder. The second section presents another category of a many-core GPU in offloading portions of the video transcoding tasks. The third section reviews the first commodity hardware H.264 encoder released by Nvidia. In the fourth section, we explain the reasons why the scalable video coding strategy is so well-known for simulcast applications by efficiently having the video transcoded once only. The fifth section discusses the latest video coding standards available today. The sixth section covers the related study of how video transcoding can be traded-off from the number of viewers in adaptive bit-rate streaming. The remaining sections investigate the state-of-the-art video transcoding strategies in peer-to-peer systems, cloud services and distributed computing systems.

2.1 Specialized High-Performance Hardware Accelerators

Efficiency issues of general-purpose processors on multimedia applications, including video decoding and encoding, have attracted a lot of research efforts. Various approaches have been studied to improve hardware efficiency. Olli Lehtoranta et al. [50] presented a scalable MPEG-4 hardware encoder for FPGA based Multiprocessor System-on-Chip (MPSOC). Their results showed the scalability of their encoder framework in MPSOC using horizontal spatial parallelism and the measured speed-up of two and three encoding slaves are 1.7 and 2.3 times respectively. Xiao et al. [51] proposed an on-chip distributed processing approach in parallelizing a baseline H.264 en-
coder at the macroblock (16x16) and sub-macroblock (4x4) levels on a fine-grained of a many-core processor. However, their works focus mainly on mapping the H.264 codec to a many-core processor and energy efficiency for a single H.264 video encoding session in real time. Although many approaches have been proposed to boost the speed of a single video transcoding instance, a large-scale video transcoding system usually experiences bottlenecks that are caused by the limited processing power of CPU when multiple transcoding tasks are executed simultaneously.

### 2.2 Graphics Processing Units (GPUs)

More recent approach towards accelerating H.264/AVC is by parallel processing using a many-core GPU. For example, the Nvidia CUDA Video Encoder (NVCUVENC), a hybrid CPU/GPU accelerated H.264/AVC video encoder that leverages on hundreds of CUDA cores in accelerating encoding tasks of NVCUVENC [52]. In another work, Momcilovic et al. [53] proposed an efficient approach for collaborative H.264/AVC inter-loop encoding in a heterogeneous CPU + GPU system. The integrated scheduling and load balancing routines allow efficient distribution of workload among modules across all processing units. In [54], Ko et al. proposed a novel motion estimation (ME) algorithm implemented in a Nvidia GPU using sub-frame ME processing at frame-level parallelization. However, these approaches have limited scalability issues: i.e only a few GPUs can be employed and could not fully exploit capabilities of a multiple GPUs system due to limitation bounded by CPU utilization.

### 2.3 The Nvidia Hardware Video Encoder

The emergence of dedicated hardware H.264/AVC encoders has recently accommodated to commodity computing hardware such as CPUs and GPUs. In the past, hardware video encoders were only commonly found in System-on-Chip (SoC) inside mobile phones. These hardware encoders are often limited only to the built-in cameras on mobile devices for video compression. On the other hand, hardware video encoders found in the video broadcasting industry are often expensive and do not provide enough flexibility in encoding parameters [41].

The first commodity hardware H.264 encoder was introduced and released along with the Kepler GPU architecture by Nvidia in 2012. The first generation of the Nvidia HW
Figure 2.1: The NVENC block diagram, taken from [2]

encoder is named as NVENC [2, 55]. The second generation hardware encoder was later released in 2014 along with the Maxwell GPU architecture. NVENC basically provides a power-efficient fixed-function encoder that is able to receive coded input video, decode, pre-process, and encode H.264-based content. Besides that, NVENC is a fully dedicated H.264 hardware encoder which does not use CPU resources or CUDA cores for computation, leaving available resources for user space or to perform other computes. NVENC supports video resolutions up to 4096 × 4096 (4K resolution) at 30 frames per second, and is capable of encoding multiple streams in parallel. The hardware encoder also supports advanced features of H.264 such as B-frames, MVC & CABAC entropy coding and external motion estimation. Under different encoding scenarios, presets can be defined in the hardware encoder API. Fig. 2.1 shows the hardware block diagram of NVENC. NVENC is built to perform all tasks that are critical to parts of the end-to-end H.264 encoding apart from the rate control and picture type determination. The rate control algorithm is implemented in the GPU's firmware and is controlled via the Nvidia driver software. From the application's point of view, the rate control feature is a hardware function controlled by parameters defined in NVENC APIs. The hardware encoder also provides ability to use a motion estimation engine externally and customizable quantization parameter maps for video encoding at a region of interest (ROI). Stensland et al. [56] evaluated the basic functionality of the streaming pipeline using the NVENC hardware encoder and the measured interaction latency is less than 90ms. However, the maximum number of allowable concurrent transcoding streams in one system is limited to two regardless of the number of Nvidia GeForce GPUs the system has [43]. High performance GPUs such as Nvidia Tesla and Quadro
do not have this limitation but they are too expensive for deployment in most cases.

2.4 Scalable Video Coding

Scalable video coding (SVC [57]) has been an active area of research, development and standardization for more than 20 years. Scalable video coding is also highly desirable for future video surveillance applications. Efficient scalable video coding can provide a number of advantages for streaming applications. For example, heterogeneous clients in a video transmission service would usually request multiple bit streams of the same video source and contents differing in bit rate, coded picture size and frame rate should be delivered simultaneously. In a properly configured scalable video coding scheme, the video source only has to be encoded once, with the lowest desired resolution and bit rate as the base layer, resulting in multi-layer bit streams from which representation with higher resolution, frame rate and quality can be obtained by decoding the base layer and required subsequent enhancement layers shown in Fig. 2.2. This design eliminates the need of transcoding process as the video stream produced from a SVC enabled IP camera can be decoded directly based on the desired resolution, frame rate and quality through discarding enhancement layers. However, scalable profiles of this standard never gained popularity in commercial use for a number of reasons. One of the reasons includes the characteristics of long-established traditional video transmission systems that are difficult to be changed. Additionally, the spatial, temporal and quality scalabilities coding create significant losses in compression efficiency, larger output file, lower encoding speed and higher decoding complexity compared to the non-scalable profiles [58].
2.5 Latest Video Coding Standards

Video coding standards have been evolved from the well-known H.262/MPEG-2 standard [59] to H.264 Advanced Video Coding (AVC) [27]; and now to the emergence of H.265 High-Efficiency Video Coding (HEVC) [28] and VP9 [60]. The H.264/MPEG-AVC standard has been successfully satisfied the growing need for higher coding efficiency in standard-definition videos where an increase of about 50% in coding efficiency has been achieved compared to its predecessor H.262. However, both these video coding standards were not designed for High Definition and Ultra High Definition video content in the first place, the demand which is expected to be dramatically increased in the near future. Consequently, the H.265/MPEG-HEVC standard has emerged and to be applied to almost all existing H.264 applications which emphasize high-resolution video coding. Substantial bit-rate saving is achieved in H.265 encoding for the same visual quality when compared to its predecessor H.264. At the same time, a few companies have also developed their own video codecs, which often were kept secretly and partly on the variants of the state-of-the-art technologies used in their standardized counterparts. One of these kinds of proprietary video codecs is the VP8 codec [61], which was developed privately by On2 Technologies Inc and was later acquired by Google. Based on VP8, Google started the development of its successor VP9 [60] in 2011, which was recently announced to be finalized. Grois et al. [62] presented a performance comparison between H.265, H.264 and VP9. The HEVC encoder provides significant gains in terms of coding efficiency compared to both VP9 and H.264 encoders. On the other hand, the average encoding time of VP9 is more than 100 times slower than the H.264 encoders and 7.35 times slower than H.265 encoders.

2.6 Adaptive Bit-rate Video Streaming

Over past few years, adaptive bit rate (ABR [63]) techniques have been deployed by content providers to cope with the increased heterogeneity of network devices and connections, for instance mobile phones with 3G/4G networks, tablets on Wi-Fi, laptops and network TV channels. This technique consists of delivering multiple video representations and is being applied to live streams in video surveillance systems. However, only a limited number of video channels are dedicated to the employment of ABR
streaming in the streaming service system. In a large-scale live video streaming platform such as Twitch, a huge amount of commodity servers used in the data center has brought new issues in serving a large number of concurrent video channels. For each video channel, a raw live video source is transcoded into multiple streams at different resolutions and bit rates. These transcoding processes consume many computing resources which induce significant costs, especially those dealing with a large number of concurrent video channels. On the other hand, clients can be benefited from the improvement of QoE by the reduction of delivery bandwidth costs. Aparicio-Pardo et al. [64] proposed a better understanding on how viewers could benefit from ABR streaming solutions by trading-offs video transcoding in ABR streaming solutions. This is done by filtering out broadcasters and only those who can provide their own raw live videos at all resolutions (480p, 720p and 1080p) are considered as candidates for ABR streaming. Apart from that, candidate broadcasters with a minimum of \( N \) viewers are only be selected for transcoding. This approach can satisfy a time varying demand by efficiently exploiting an almost constant amount of computing resources without wasting resources for streaming channels without many viewers.

### 2.7 Peer-Assisted Transcoding

Since the emergence of peer-to-peer (P2P) networking in recent years, there are some study of literatures on transcoding techniques utilized in a P2P streaming system. In [65, 66], Ravindra et.al and Yang et.al have proposed a P2P video transcoding architecture on a media streaming platform in which transcoding services are coordinated to convert the streaming content into different formats in a P2P system. Liu et.al. [67] proposed a peer-assisted transcoding system working on effective online transcoding and ways to reduce the total computing overhead and bandwidth consumptions in P2P networks. However, large-scale video transcoding application poses high requirement of CPU and network I/O that incorporated with the unreliability low availability of P2P clients makes it difficult to envision a practical implementation of such systems.

### 2.8 Cloud Video Transcoding

A cloud computing system basically comprises a computing resources infrastructure that is made available to the end consumer. Cloud computing has been known for its
powerful and cost-effective resources compared to those provided by their own single machine system. Such high resource availability can be provided on a service-based web interface where consumers can access the available resources in an on-demand manner, reducing the hardware and maintenance costs to one usage-based billing. By applying this utility-computing concept where computing resources are consumed in the same way as electricity would, huge commercial endeavours like Google Cloud Platform [68], Amazon Web Services [69], Microsoft Cloud [70], Linode [71], AliCloud [72] and Telestream Cloud [73] were set up [74]. Li et al. [75] introduced a system using cloud transcoding to optimize video streaming services for mobile devices. Video transcoding on a cloud platform is a good solution to transcode a large volume of video data because of its high throughput capacity. Netflix has [76] also deployed their video transcoding platform on Amazon’s cloud services.

2.9 Video Transcoding using a Computer Cluster

Another widely researched area in speeding up the video transcoding process is the use of computer clusters or also well known as the distributed computing platform. Myoungjin Kim et al., Haoyu et al., Schmidt et al. and etc [77–87] have implemented a distributed video transcoding system using the popular MapReduce framework running on top of the Hadoop Distributed File System (HDFS). Myoungjin et al.’s proposed distributed video transcoding system provides an excellent performance of 33 minutes in completing the transcoding process in 50 gigabytes of video data set conducted on a 28 nodes HDFS cluster. Besides that, Rafael et al. [88] has demonstrated that by scaling up the split and merging architectures with in-between key-frames of fixed 250 frames chunk per worker, a fixed encoding time is guaranteed among workers independently of the size of the input video. This allows total elasticity where it is possible to add and remove workers on-the-fly according to demand. Hence, the cost of operation of the public cloud service is thus minimal. On the other hand, Horacio et al. [89] has showed that under overloaded condition, an increased number of split tasks produced from a naive scheduler would result in a longer completion time than those non-split cases. This indicates that further study on the distributed scheduling strategy is required to fully accommodate available worker resources to task allocation.

However, these approaches are all batch-oriented which means they are not suitable for continuous and real-time processing whenever a new data is produced continuously.
especially video data.

2.10 Resource-Aware Scheduling in a Distributed Heterogeneous Platform

The heterogeneity of data processing clusters has become relatively common in computational facilities. Polo, Jorda, et al. [90] presented a novel resource-aware adaptive scheduler working on resource management and job scheduling for MapReduce. The author’s technique leverages job profiling information in order to adjust the number of slots dynamically on every machine, as well as workload scheduling across heterogeneous cluster to maximize resource utilization of the whole cluster. Boutaba et al. [91] have analyzed various types of heterogeneity challenges in cloud computing systems in terms of distribution of job lengths and sizes, the burstiness of job arrival rates and heterogeneous resource requirements of the job. Ahmad et al. [92] proposed Tarazu which achieves an overall speed up of 40% over Hadoop, incorporated with communication-aware load balancing in scheduling of map jobs and predictive load-balancing of reduce jobs. Rychly and Marek [93] proposed a Storm scheduler which utilizes both the offline decision derived from results of the performance test and of the performance captured online to re-schedule processors deployment based on application needs. However, many of these approaches focus mainly on deployment and scheduling of resources for job execution (spouts and bolts in Storm). They do not consider the amount of tasks (task/tuple allocation in Storm) processed per resource based on computing ratios.
Chapter 3

Apache Hadoop and Storm: The Batch and Real-time Big Data Processing Infrastructure

Over the past decade, the world has seen a revolution in large-scale data processing. MapReduce over Hadoop [94], HDFS [95] and Spark related technologies have made the possibility of storing big data economically and processing data distributed at volume that was previously unthinkable scale. This chapter explains the various essential components of Hadoop: the architecture of the hadoop distributed file system, MapReduce job, data locality and the handling of data in computation and machine failures. Hadoop is one of the state-of-the-art batch processing framework used by many for large-scale video transcoding, that is closely related to our work. In order to meet the baseline requirement of our proposed method, Hadoop is used as a comparative study to our work. Unfortunately, these batch oriented data processing technologies do not possess real-time processing capabilities. There is no other way that Hadoop can turn into a real-time system; real-time data processing requires fundamentally new requirement sets compared to batch processing [96]. There is a number of popular stream processing platforms currently available such as Storm, Samza, Spark Streaming and Flink. All of them offer exciting capabilities and appear promising. However, based on the time factor, we chose to narrow down the detailed evaluation candidate to Storm only. The remaining of this chapter explains the significant components of Storm, its processing semantics, how Storm provides mechanisms to guarantee fault tolerance, followed by metrics visualization in Storm, isolation scheduler and its benefits in video transcoding.
CHAPTER 3: APACHE HADOOP AND STORM: THE BATCH AND REAL-TIME BIG DATA PROCESSING INFRASTRUCTURE

3.1 Apache Hadoop: A Batch Processing Infrastructure

3.1.1 A Distributed Storage System in the Batch Processing Infrastructure

The core component of Apache Hadoop project comprises a file storage system that is used to store data reliably and provide high throughput access to application data, known as the Hadoop Distributed File System or HDFS [4]. The main idea is to split the data into blocks and store them across multiple computers known as the HDFS cluster. When it comes to processing the data, it is done exactly where its blocks are stored. Rather than retrieving the data over the network from a central server, it is already in the cluster and ready to be processed on site. When the volume of data that has to be stored in HDFS grows, users can always increase the cluster size by adding more machines needed to the cluster. Ever since Hadoop has started gaining its popularity, many other software applications have been built upon Hadoop. These software applications are also collectively grouped within the Hadoop Ecosystem which was intended to facilitate the data uploading process to the Hadoop cluster. Presently, there are many ecosystems and projects that can communicate with each other to simplify the installation process, thus maintaining the cluster. Cloudera, the company that has put together the distribution of Hadoop called cloudera distribution or CDH which integrates all key ecosystems along with Hadoop itself and packages them together so that installation can be done easily.

3.1.2 The Hadoop Distributed File System

Figure 3.1: Data Redundancy in HDFS, reproduced from [4]
Fig. 3.1 illustrates the architecture of the reliable scalable data storage (HDFS) in Hadoop. When a file is uploaded into HDFS, it is sharded into small chunks known as blocks. Each block has a default size of 64MB and is given a unique label. As the file is uploaded to HDFS, each block will be stored on one of the DataNode in the hadoop cluster. In order to identify which necessary blocks are required to re-construct the original file, a NameNode is used to store this information as metadata. Apart from that, data can be lost during network failures among the nodes and disk failure on an individual DataNode. To address these issues, Hadoop replicates each block three times and stores them in HDFS. If a single node fails, there are two other copies of the block on other nodes. In the case of under-replicated blocks, the NameNode will replicate those blocks onto the cluster. In the past, NameNode was the single point of failure in Hadoop. If the NameNode dies, the entire cluster is inaccessible as the metadata on the NameNode is lost completely, so too the entire cluster data. Even the cluster has the replicas of all blocks in DataNodes, there is no way of knowing which block belongs to which file without the metadata. To avoid this problem, NameNode is configured to store the metadata not only on the local drive but also in the network. These days, NameNode is no longer a single point of failure in most production clusters as two NameNodes are configured. The active NameNode works like before and the standby NameNode can be configured to take over when the active NameNode has failed.

### 3.1.3 MapReduce

Apart from storing the data reliably and effectively, the data also have to be read and processed efficiently. Considering a large data file, processing serially from top to bottom could take a long time. Instead, MapReduce is designed to process data in parallel where the data file is broken into chunks and processed in parallel. When running a MapReduce job, the job is submitted to JobTracker which divides the work into mappers and reducers. The actual map/reduce tasks are handled by slave node daemons called TaskTrackers. Each TaskTracker will run on every node and have map tasks work directly onto the pieces of data that are stored locally on a machine. This would save a large amount of network traffic. By default, Hadoop use HDFS blocks as the input split for each Mapper. This ensures that a mapper works on data on the same machine. If one of the blocks needs processing, TaskTracker on the machine with the block will likely be chosen to process that block. However, TaskTracker on the machine that has the block replica could be busy, in which case a different node will be chosen.
to process the block and the block will be stream over the network. This is illustrated in Fig. 3.2. In brief, mappers read data from inputs, produce an intermediate data and pass to the reducers.

### 3.1.4 Data Locality in the Hadoop Cluster

Data locality is defined as the ability to keep compute and storage close together in different locality levels such as process-local, node-local, rack-local and etc [97]. Data locality is one of the most critical factors considered at task scheduling in data parallel systems. Intermediate data generated from map tasks are stored locally (not upload to HDFS) so that data locality of map tasks is benefited. As such, MapReduce conserves network bandwidth by taking the shortest distance between computes and storages, that is data blocks to be processed are selected nearest to the storage drives and the compute instances of the machines that create the Hadoop cluster. For instance, the MapReduce master node locates the input data and a machine that contains a replica of the corresponding data block is to be scheduled to the map task. If such a task can be found, node-level data locality benefits and no network bandwidth is consumed. Otherwise, Hadoop will attempt to schedule a map task that is nearest to a replica of that input data block to achieve rack-level data locality (the input data block and task are randomly picked and dispatched). However, such data locality in Hadoop does not consider other factors such as system load and fairness.
3.1.5 Fault-Tolerance

Since MapReduce processes terabytes or even petabytes of data on hundreds or thousands of machines, it must be a fail-safe system [45]. In the following paragraph, we discuss how Hadoop keeps data safe and resilient in case of worker and master failures. Periodically, the master node checks for active connection with all workers. If a worker fails to respond within a certain amount of time, the worker is marked failed by the master node. Workers with their map tasks completed are restoring back to their idle state, and these workers consecutively become available for re-scheduling. In a similar way, when a worker fails, all map or reduce tasks in progress under the failed worker node are terminated and therefore become eligible for re-scheduling onto other worker nodes. Completed map tasks of a failed machine are inaccessible upon machine failures and therefore have to be re-executed as their outputs are locally stored on the failed machine. On the other hand, the output of reduce tasks is stored globally so that the completed reduce task does not need to be re-executed in case of machine failures. When a worker fails, the map task is re-executed on another worker. Re-execution of the map task will be notified to every reducer. Any reducers that do not receive the notification message from the failed mapper will read from the re-allocated mapper. Furthermore, MapReduce is resilient towards failures of large-scale workers. For instance, network maintenance and updates of a running MapReduce cluster can cause a cluster of 100 machines to become inaccessible for several minutes. In order to make forward progress and complete MapReduce operations, the master node simply has to re-execute the work completed by the inaccessible worker machine.

If the master daemon dies, a new copy of the master daemon is started over from the last check-pointed state. However, even the master failure is unlikely where there is only one master; the current implementation terminates MapReduce computation if the master fails. Users would have to monitor for this failure and debug the MapReduce operation if they needed to [45].

3.1.6 Task Granularity

In terms of task granularity, Hadoop’s map phase is subdivided into $M$ chunks (typically 64MB) and the reduce phase into $R$ chunks. The number for $M$ and $R$ is preferably greater than the number of worker nodes. As such, all worker nodes can perform many different tasks which then improve the dynamic of load balancing and speed up the re-
covery process in the case of workers failures. This is because the remaining map tasks, given the fine granularity, can be more easily reallocated across all other worker machines. Since the master node must schedule the map and reduce tasks and store them in memory in the first place, the implementation of MapReduce has practical bounds on how numerous $M$ and $R$ can be. Additionally, $R$ is often constrained by user’s preferences because the output of multiple reduce tasks ends up with multiple output files. Thus, $M$ is tended to be chosen as task granularity so that every task is roughly 16MB to 64MB of the input data (where throughput optimization is the most effective), and $R$ is a small multiple of the number of worker machines [45].

3.1.7 Discussion

There is a number of benefits Hadoop can provide for video transcoding. First, video data can be stored reliably in HDFS. Due to its data locality design, Hadoop can provide high throughput video transcoding, maximizing the utilization of all computing resources. Besides that, the Hadoop cluster can be easily scale based on resource demand for video transcoding, eliminating the delay which is needed to re-run the whole job. Apart from that, the Hadoop cluster can be easily deployed in available cloud services such as the Amazon Elastic Compute Cloud (AWS EC2 [98]). However, cloud compute instances are expensive to be deployed in a larger scale. The pricing rate multiplied corresponding to the number of instances used in the Hadoop-based video transcoding system.
3.2 Apache Storm: A Stream Processing Infrastructure

3.2.1 Big Data Processing in the Real-time Stream Processing Infrastructure

Big data is basically characterized into four dimensions, namely the volume, velocity, variety and veracity of big data, and Hadoop-like frameworks handle the volume and variety parts of big data [99]. Along with the volume and variety, a real-time system requires handling the high velocity of processing, that is the velocity of data. And maintaining the velocity of data in big data computation is a complicated task. First, the system has to be able to gather data generated by real-time stream events which are coming in the rate of millions of message streams per second. Even as the data is being gathered, the system has to be robust enough to process the data in parallel. Then comes the complex task of performing event correlation and extracting meaningful information from these moving streams. All of these need to happen in a fault tolerant and distributed manner. All employed real-time systems must be low latency systems, enabling faster computation and facilitating near real-time response capabilities. Data streams can be gathered from various sources, processed in the stream processing engine, the result of which is written to a destination system. In between, queues are used to store/buffer messages. Before Storm, one would need to build a network of queues and workers manually to do data processing in real time. However, issues such as tedious coding, poor fault-tolerance, data lost and painful to scale are limitations to such naive approaches. This is where Storm comes into the picture.

*Apache Storm* is a big data framework fine-tuned to handle unbounded streams of data. As mentioned, modern big data processing environments have evolved from batch computing against large volumes of static data and now into real-time computing on data streams that are produced in real time. This is especially important in social media platforms such as Twitter which requires making complex decisions based on the data created by user interactions [5]. Storm includes multiple features such as horizontal scalability, resilient, guaranteed data processing, extensibility and efficient as listed below:

1) **Scalable:** Nodes can be added and removed non-disruptively without affecting the uniform distribution of works.

2) **Resilient:** Fault-tolerance is essential for hardware failures and the continuation of job execution if a slave node goes down.
CHAPTER 3: APACHE HADOOP AND STORM: THE BATCH AND REAL-TIME BIG DATA PROCESSING INFRASTRUCTURE

3) **Efficient**: Storm must have in-memory data warehouses for fast data transfer since it is designed for real-time processing application.

4) **Extensibility**: A Storm topology can invoke any external processes from external modules (e.g. persistent storage using SQL services). Thus, Storm is an extensible framework.

Storm was originally developed by Nathan Marz and his team at BackType, which was later acquired by Twitter and became open-source in 2012 [100]. Twitter has improved Storm in several ways, by scaling out to a larger amount of nodes, and reduce Zookeeper dependencies in Storm [101]. Storm was then adopted by various other corporations. Storm has been used or experimenting by more than 60 companies and it is currently considered the most popular scalable data stream management system (DSMS). For instance, Groupon [102], Yahoo! [103], The Weather Channel [104], Taobao [105], iQIYI [106], Alibaba [107], Baidu [108], and Rocket Fuel [109] are organizations who are officially implemented Storm into their systems.

Even today, stream processing systems are constantly expanding and are a rich body of literature from which stream processing researchers can examine; for example, declarative query languages are not supported in many of these "modern" systems. Thus, stream processing has been an active research which evolves rapidly for advanced development [5].

### 3.2.2 Domains of Real-time Processing

Before going into Storm’s architecture in details, we first discuss what we mean by real-time processing. The definition of real-time may vary from one application to another.

![Figure 3.3: Domains of real-time processing](image-url)
CHAPTER 3: APACHE HADOOP AND STORM: THE BATCH AND REAL-TIME BIG DATA PROCESSING INFRASTRUCTURE

Fig. 3.3 illustrates three different categories of data processing where the respective processing time ranges from 500 ms to a few hours. For instance, both online transaction processing (OLTP) and online analytical processing (OLAP) are characterized by a fine millisecond latency budget and is one of the fastest analyses available nowadays. On the other end of the spectrum, batch processing is characterized by high throughput. One of the reasons why batch processing takes hours and days, is that it is munching through massive amounts of data. If there is too much data, the storage drive has to transfer the data at a rate of 100 Mbps or more. A simple method of solution to such limitation is to employ multiple storage devices so that data transfers can be performed across each physical device simultaneously. One can have more physical devices working to retrieve the data but if the data is much higher than the disks can actually transfer, processing will eventually take hours and days. Therefore, throughput is important for this kind of batch processing and is mostly used for ad-hoc queries.

In between the two ends of the spectrum, there is another category of near real-time (within seconds) processing where number crunching gives an approximation to the real values. It gives insights into one’s business or any activities requiring monitoring in "real-time". The caveat of this approximation is how much tolerance one can accept for this approximation. As long as the approximation is 5 to 10% off the actual value, then in most cases, it may still be acceptable. In this thesis, we focus on near real-time approaches as transcoded video frames have to be available as soon as possible for user interaction. This is where Storm comes into action.

3.2.3 Data Flow Model and Execution Architecture in Storm

Storm data processing architecture basically consists of tuple streams flowing through different stages of components (spouts and bolts) in a Storm topology. A Storm topology is expressed as a directed graph where nodes represent components of computation and edges identify computation components which data can flow in-between. Nodes are further divided into two disjoint sets, namely spouts and bolts. Spouts are sources of tuples in a storm topology. Typically, spouts consume messages by pulling data from message queues, such as Kafka or Kestrel. On the other hand, bolts process incoming tuples and emit them downstream to the next stages of bolts. Additionally, it is possible for having cycles in a Storm topology. From database system’s point of view, we can think of a Storm topology as a directed graph of operators.
A list of terms shown below is used in the Storm data stream model:

1. **tuple** - the basic unit data between inter-connected data operators (referred to *spouts* and *bolts*)
2. **spout** - a source of data streams received or read at live
3. **bolt** - a unit of computation which emits new stream of data
4. **task** - a Storm task that runs per thread normally

### 3.2.4 Stream Groupings in Storm

Storm also provides flexible stream groupings in combining data. Shuffle grouping allows *tuples* to be sent in random sequences. Secondly, field grouping combines one or more fields in *tuples* where *tuples* with the same field will always go to the same job execution. An all-grouping strategy replicates the entire *tuple* to the next stage of bolt instances. Direct grouping decides which task of the consuming bolt will receive the *tuple*. Global grouping sends all *tuples* to a single next stage *bolt* instance. Local grouping allows sending the *tuple* to task in the same machine. There are more other groupings that are not listed in this thesis. All components of Storm topology are executed in parallel. This allows users to specify the number of parallelism associated with every component (*spouts* and *bolts*) and Storm spawns the necessary number of threads across the cluster in order to perform the execution of all components.

### 3.2.5 Storm Internals

Storm runs on a distributed message system, based on a Master/Slave pattern similar to Apache Hadoop. The master and slave processes are coordinated by a third-party dependency so-called ZooKeeper. Users submit topologies to the master node, which is handled by a daemon called *Nimbus*. *Nimbus* is mainly accountable for code distribution around the cluster and allocating works among *Worker* nodes. *Nimbus* assigns tasks to slave nodes, monitors the task progress and re-schedules the task to another *Worker* node if a *Worker* node fails. In contrast to JobTracker in Hadoop, *Nimbus* is stateless where all data are stored in the *Zookeeper* cluster instead. This avoids a single point of failure. If *Nimbus* fails, it can be restarted without influencing any running tasks. Actual tasks are done on the *Worker* nodes. *Worker* nodes are also referred as *Supervisors* in Storm. The *Supervisor* node starts a new Java Virtual Machine (JVM) instance for
each Worker process, in which each worker process runs one or more Executors. Each Executor may run one or more tasks. Tasks are the actual work performed for a spout or a bolt. Similar to Nimbus, Supervisor stores its data in the ZooKeeper cluster and therefore is a fail-fast process. Each Worker node in Storm normally runs multiple Worker processes. Each machine may have more than one Worker process with every Worker process mapped to the submitted topology. Hence, different components (spouts or bolts) of the topology can be executed by the Worker processes of the same machine.

Fig. 3.4 illustrates the high-level architecture of Storm.

3.2.6 Levels of Parallelism in Storm

Storm exploits four different levels of parallelism in its partitioning strategies. The following levels are used to run a topology in Storm:

1). Supervisor (Slave node)
2). Worker (JVM)
3). Executor (Thread)
Inter-topology, intra-topology and intra-bolt/intra-spout parallelisms are three different levels of parallelism provided by Workers, Executors and Tasks respectively. These different levels of parallelism in Storm are illustrated in Fig. 3.5. The figure shows one Supervisor daemon, which includes Worker processes. Each worker consists of five Executor threads. The maximum number of Workers is specified within Supervisor configuration. By default, Storm runs one Task per Executor thread but an Executor thread might also execute multiple Tasks in series. The feature of having multiple Tasks that are executed in a serial fashion within an Executor thread, allows users to test the parallelism of the system.

3.2.7 Processing Semantics in Storm

One of the key features of Storm is its ability to provide delivery and a guarantee of the data it processes. Three types of semantic guarantees are presented, namely "at-least-once", "at-most-once" and "exactly-once" semantic. The "at-least-once" semantic basically guarantees that every tuple produced from a Storm component (spout or bolt) will be processed at least once in the downstream bolt. Messages are replayed when there
are failures. For at-most-once semantic, every tuple is processed once and dropped in case of failures. Exactly-once semantic guarantees that no duplicated messages are delivered to the processing bolts.

To provide "at-least-once" semantic, Storm topologies are augmented with "acker" bolts that track tuples-tree for every tuple that is emitted by the spouts. Storm tagged every processed tuple with a randomly generated unique 64-bit "message/tuple id". Trace ids are then grouped together with tuples in spouts once the data streams are first pull from the input sources. Whenever a tuple is processed, new tuple or multiple tuples can be produced. For example, tuple containing the entire sentence is sharded into a set of words by a bolt, producing new set of tuples: tuple of words and tuple of sequence numbers from the input tuple of the sentence. Every produced tuple is then assigned to a new random 64-bit id, and the list of new tuple ids that is associated with the output tuples are retained in a provenance tree. Backflow mechanism is used to acknowledge the tasks that have contributed to the output tuples when these tuples about to leave the topology. In order to retire these tuples, the backflow mechanism reaches out to the spouts that have already started processing tuples in the first place.

Exactly-once semantic in Storm can be achieved by transactional topology where strong ordering of tuples can provide such capabilities. Each tuple is associated with a transaction id and is incremental with every tuple. If a tuple fails and needs to be replayed, then it is re-emitted with the exact same transaction id.

### 3.2.8 Fault Tolerance Daemons in Storm

A Storm cluster has several different daemon processes: Nimbus that schedules jobs to worker processes, Supervisors that start and kill Worker processes, the log viewer that gives access to log files, and Storm UI that shows the status of a cluster. When a Worker process dies, it is restarted by the Supervisor daemon. If failures are on the start-up and heartbeats, Nimbus will re-schedule the Worker process to another machine. In the mean time, tasks assigned on the failed machine will be timed-out and Nimbus will re-assign those tasks to the new scheduled Worker. The Nimbus and Supervisor daemons are supervised by daemon tools or monit so that if the Nimbus or Supervisor daemons die, they will be backed-up as if nothing has happened. In addition, since all states are stored in the Zookeeper cluster, the Nimbus and Supervisor daemons become fail-fast and stateless.
CHAPTER 3: APACHE HADOOP AND STORM: THE BATCH AND REAL-TIME BIG DATA PROCESSING INFRASTRUCTURE

Most notably, no running worker processes are affected by the death of Nimbus or their Supervisors as cluster communication and heartbeats are all managed by the Zookeeper cluster. This is in contrast to Hadoop where if a JobTracker dies, all running jobs are completely lost. Even if the Nimbus node is down, workers will still continue to function and supervisors will continue to restart workers if the supervisor daemon dies. However, if Nimbus goes down, workers cannot be reassigned to other machines.

3.2.9 Daemon Metrics and Monitoring in Storm

Storm User Interface (UI) provides a web interface for visualization of running topology; with a complete break-up of internal spouts and bolts. Storm UI also contributes information regarding any errors coming in tasks and fine-grained stats on the throughput and latencies of every component of the running topology. Storm UI helps in debugging issues at high-level, for example as soon as an error happens in a topology, a stack trace is displayed. The Storm’s built-in metrics feature provides framework-level support for collecting and reporting metrics to external systems. Worker logs are another source of valuable information used for troubleshooting any topology errors.

3.2.10 Auto Scaling in Storm

Storm supports configuring initial parallelism at various levels per topology; number of worker processes, number of executors, number of tasks. Storm supports dynamic re-balancing which allows increasing or decreasing the number of worker processes and/or executors without restarting the cluster or the topology. Although the number of executors for a component may change over time, the number of initial tasks configured is always the same throughout the lifetime of a topology. Once all supervisor nodes are fully saturated with worker processes and there is a need to scale out, one simply has to begin a new supervisor node and pointing it to the Zookeeper cluster. All resources on the newly added supervisor will then be available for scheduling. It is possible to automate the logic of monitoring current resource consumption on each node in the Storm cluster, and dynamically add more resources.
CHAPTER 3: APACHE HADOOP AND STORM: THE BATCH AND REAL-TIME BIG DATA PROCESSING INFRASTRUCTURE

3.2.11 The Isolation Scheduler in Storm

Each worker process runs executors for a particular topology only where mixing of different topology tasks is not allowed at worker process level. This provides topology level run-time isolation. Furthermore, each executor thread runs one or more tasks of the same component (spout or bolt) without mixing tasks across components. In a shared cluster setup, a topology that needs a large number of tasks, and resources (CPU cores, memory) can be submitted along with a smaller topology. To handle this, the Storm's isolation scheduler provides multi-tenancy across topologies and avoids shared-resource contention among topologies. It allows specifying which topology should be isolated, that is run on a dedicated set of machines within a cluster. All isolated topologies are given higher priority than non-isolated ones, taking resources from the non-isolated.

3.2.12 Ease of Development in Storm

Storm provides a really simple, rich and intuitive API that simply describes the DAG nature of processing flows (topology). Storm tuples, which provide the abstraction of data flowing between nodes in the DAG, are dynamically typed. Developers can start with writing topologies and run them in the local cluster mode or the pseudo-distributed mode. In the local mode, threads are used to simulate worker nodes, allowing developers to set breakpoints, halt execution, inspect variables, and profile their topology before deploying it to the real cluster where all of this is much more difficult.

3.2.13 Discussion

There are a number of benefits Storm can provide for video transcoding. First, video data can be gathered in terms of live video streams from IP cameras into the Storm based data ingestion component called Spout. Due to its stream processing architecture, the output from processing can be visualized on-the-fly, providing real-time transcoded video streams ready to begin streaming to the client end. Since the input video stream is distributed all over to the Storm cluster and all computers are utilized under the proposed system, the CPU and memory usage of each computer can be reduced in real time when the input data rate is lower than the total data processing rate. In the context of a large-scale video transcoding system, more input video streams can be processed
in such system.

In cloud computing application, Storm can provide a shared video transcoding service using the available public cloud services such as the Amazon Elastic Compute Cloud (AWS EC2 [98]). Resource utilization of such shared cluster can be maximized for the fact that not all clients are transcoding videos in a 24/7 basis. This allows idle clients to share their available allocated compute resources to the cluster. In this way, online clients can leverage resource availability of the remaining unused resources in the shared cluster onto their transcoding processes.

An expanding cluster of such shared systems enables a cost effective way to share a collection of massive transcoding resources among all customers by having customers pay only for their contributed number of instances and charges for maintaining the entire cluster. On the contrary, dedicated standalone instances in an on-demand basis can cost a lot for a customer to transcode video at a larger scale.
Transcoding a large volume of daily recorded video files with a volume in terabytes or petabytes gives rise to potential challenges in reducing the processing time. High cost of reliable storages is needed to store the video reliably in traditional storage systems and to stream the distributed data from traditional storage devices across the network to the central processor, which can take up a long time and processing can be extremely slow. When there are more cameras deployed along the way, more video streams are needed to be stored. In order to store such a huge amount of videos, a way to easily scale the video storage capacity up to a massive volume is required. The chapter first presents the architecture of a Hadoop-based video transcoding system and its data placement issues in heterogeneous systems. The remaining of this chapter presents the architecture of the proposed Storm-based video transcoding system and its components reflected in a Storm topology.

4.1 The Batch-oriented Distributed Video Transcoding Method

4.1.1 Adopting a Distributed File System in Storing Video Data

To store such a huge volume of video data reliably, the Hadoop Distributed File System (HDFS) is adopted. HDFS is a fault-tolerant storage system, not only does it store data in a distributed way across multiple machines and scales up incrementally, it also
survives the failure of significant parts in the storage infrastructure without losing data in case of machine failures. Video transcoding in Hadoop requires special treatment as media content files are very different from file formats that usually handle in Hadoop. First, HDFS manages the cluster storage by breaking down recorded video files into blocks and storing each video block redundantly across a pool of machines. As Hadoop uses HDFS blocks as input splits for each mapper, the mapper can work on data blocks on the same machine. Transcoding video blocks stored in HDFS is not a simple task. Naive HDFS splits performed on a video file would lead to corrupted transcoded video content when transcoding on the HDFS blocks directly since not every video block has the latest header information or the key frame after HDFS splits. This made video files become non-splittable files for transcoding on top of HDFS. Hence, input video files may be required to split into multiple limited size of chunks whereby each chunk has a size not more than the default block size of 64MB before uploading to HDFS. However, if the sharded file size is less than the default block size, HDFS splits are forced to be split into a smaller size than the default block size [110, p. 234] and eventually corrupt the content of the HDFS video blocks. Thus, video blocks for a particular video chunk are required to be processed as a whole by re-construct back the video chunk that is stored as HDFS blocks. For this particular related work, bash scripts are used to execute the batches of mapper/reducer task. This simplifies our code development phase and enables it to stimulate in a pseudo distributed mode before realizing into the cluster.

4.1.2 Mapper Task in the Batch Video Transcoding Approach

Algorithm 1 Mapping transcoding tasks to sharded video chunks

**INPUT:** A mapper set of indexed video chunks stored in HDFS.

**OUTPUT:** A set of unsorted transcoded video chunks with a coded GOP size of 15 frames.

1. while Get the input directory of a set of video chunks to be processed do
2. Read the corresponding HDFS’s sharded video blocks as a whole video chunk & stored temporary
3. Transcode each video chunk using FFmpeg
4. Store the transcoded video chunk back to HDFS
5. end while

In MapReduce processing, the Hadoop cluster is split into tasks of mappers and reducers. Instead of reading the entire video file by a single mapper, the video file is sharded into multiple video chunks with each chunk to be processed by one of the mappers.
The HDFS's sharded video blocks are required to be reconstructed back to their original video chunks without corrupting the output video content during the transcoding process. However, the reconstruction process has involved data gathered over network and thereby loses its data locality advantage over HDFS, the reason why HDFS work efficiently with MapReduce in the first place. Nevertheless, all mappers work simultaneously, having each mapper processes over a video chunk rather than a small fraction of the video chunk. For multiple video files transcoding, the mapper would process a set of indexed video blocks, and pile them up so that the video chunks from the same video source go in the same pile. By the end of the map phase, each mapper would have a pile of transcoded video chunks per source. In a mapper task shown in Algorithm 1, FFmpeg processes are used to transcode a set of indexed video chunks. After that, a collective intermediate set of unsorted transcoded video chunks produced from the mappers are then passed to the reducers.

### 4.1.3 Reducer Task in the Batch Video Transcoding Approach

**Algorithm 2** Sort and merge all indexed transcoded video chunks into a set of single video file

**INPUT:** A set of unsorted transcoded video chunks.  
**OUTPUT:** A set of transcoded output video files consisting all sorted video chunks.

1. **while** Get the directory of a set of transcoded video chunks to be merged **do**
2. **end while**

```plaintext
1: while Get the directory of a set of transcoded video chunks to be merged do  
2:    Read the corresponding HDFS sharded transcoded video blocks into a whole chunk  
3:    Sort & merge a set of sorted video chunks to their corresponding video source using mkvmerge  
4:    Store back the set of merged video files to HDFS  
5: end while
```

Once the mappers have finished processing, the reducers would collect an intermediate set of unsorted video chunks and aggregate the transcoded video chunks to their corresponding initial video source. This is fast for the reducers to perform data aggregation as the mappers have already separated the indexed video chunks into a pile per source. Once the reducers have retrieved all their data, they collect all the small piles and create a larger pile by going through the piles one at a time. To keep the video chunks in the sorted order, a phase of shuffle and sort takes place. The shuffling phase is the movement of intermediate transcoded video chunks from the mappers to the reducers. Sorting is the fact that the reducers will organize these set of transcoded video chunks into a sorted order. Each reducer goes through a set of piles in a sorted order. Hence, the video chunks must have been indexed into keys and value where the key
corresponds to the sequence number for sorting and the value corresponds to the input video chunks (the value supposed to be each particular piece of the input video chunks stored in HDFS). In Algorithm 2, a reducer performs merging tasks by concatenating a set of sorted transcoded video chunks into one final set of outputs and writes back to HDFS.

4.2 The Proposed Real-time Distributed Transcoding Method

4.2.1 System Architecture

![Figure 4.1: The proposed Storm-based video transcoding topology](image)

We shall now present the framework for real-time distributed video transcoding. It is based on Apache Storm to provide such real-time capabilities. One compute node among several compute nodes is used as a master node and the rest of the nodes are used as worker nodes. In a real life case, the master node could be a media server which gets several requests on multiple live streams at different resolutions and bitrates. Fig. 4.1 depicts the proposed distributed video transcoding system based on Storm. The video transcoding rate is often limited by the video data acquiring speed. In order to maximize transcoding throughput, one video file can be sharded into multiple video chunks so that the multiple video streams can be acquired and transcoded simultaneously. For this, an input video stream (single video file source) is divided into multiple
sub-streams (three sharded video chunks are used in this thesis) which are ingested to
the three spouts and are further partitioned or sharded into tuples/tasks before emitted
to the next stages of transcoding bolt instances. From the figure, a single transcoder bolt
instance is replicated four times for each sub-stream generated. In the first stage, three
spouts are created to read from the video chunks. A video spout is capable of replay-
ing a video tuple if it fails to be processed by the transcoding bolt. This ensures that no
video frame will be missed in the transcoding process. Each of the output tuple emitted
from the video spouts consists of a "video data" field and the "emit seq" field grouped
in a hash partitioning scheme called fields grouping. This facilitates segregation of the
stream consisting both the "video data" and "emit seq" fields. For example, if the stream
is grouped by the "video-data" field, tuples with the same "video data" field will always
go to the same task (e.g., transcoding task), but tuple with different fields (e.g., "emit
seq" field) may go to different tasks (e.g., indexing task). The second level parallelism
is set at the transcoding stage to evenly distribute every video tuple emitted by the cor-
responding spout. Every new video tuple produced from the same task id is collected
back in the sorting stage. The sorting bolt then performs real-time sorting based on the
spout's emitting sequence number to ensure every transcoded video tuple received is
in the correct order. At last, sorted output video streams are collected using streaming
joins, joining all streams into the single reporter bolt before writing to the storage drive.

4.2.2 Data Granularity

A closed group of pictures (GOP) with a size of 15 IBP frames is selected as data gran-
ularity in Storm's thread-level parallelism in the proposed system. This is to ensure
maximum compression ratio achieved in validating our proposed system and to main-
tain a low communication-to-computation ratio for maximum compute throughput.
On the other hand, coarse granularity of parallelism does cause load imbalance in ho-
mogeneous platforms and is recently addressed by a partial key grouping technique.
A single intra-frame transcoding (GOP size of 1) basically yields a higher transcoding
rate but its lower compression ratio also results in a larger output storage size. Further-
more, the thread-level parallelism in Storm is not restricted to a local machine itself but
throughout the entire cluster.
4.2.3 Transcoding Videos at A Larger Scale

By referring to the experiment results of the proposed system in Table 5.10, the input frame rate for Spout 1, 2 and 3 is 815 fps, 1568 fps and 1003 fps respectively. Transcoder backends used in the experiment are capable of transcoding at 13 fps, 23 fps and 36 fps when output resolution is set to 1080p. Therefore, the cluster takes 47 secs to transcode 1 sec of video frames. Meaning that by the end of 47 secs, 155756 frames are going to be held in the data buffer and waiting to be processed. Since the cluster transcoding rate is much lower than the total input frame rate, frames are keep on holding all way in the queues. Thus, the system will never catch up with the input data rate and eventually suffered from insufficient memory.

Fig. 4.2 shows how the stream processing system can handle the case where the input frame rate is greater than the processing rate. In the previous example, our proposed system acquires input videos at a total input video frame rate of 3386 frames per second. To fully utilize such high input frame rate, Storm divides them into 47 small buckets with each bucket transcoding a total of 72 frames per second. Basically, the optimized cluster size is equivalent to a cluster that is duplicated 47 times under the same computers that the existing cluster is doing for one whole input stream acquiring at 3386 frames per second. This process is often refers to data sharding, where the entire input stream is divided into multiple streams (e.g., 47 streams in our case) and running the same transcoding program backend.
4.2.4 The Video Data Spout

The video data spout is the source of a storm-based video transcoder topology. At start-up, it reads the GOP size, loads the video in exact GOP bytes sizes into main memory and encode into base64 string characters before emitting them as Storm tuples. For the input source of a camera stream, FFmpeg is used to read from the RTSP stream and the video stream is converted to raw H.264 bit-streams. Similarly, the H.264 bit-streams are then later encoded into base64 string characters before emitting to the transcoding bolt.

Algorithm 3 illustrates a reliable video data spout component. At initialization, the video data is loaded into memory. Every time the nextTuple function is called, it will emit a tuple in a list of GOPs in base64-encoded text, its emitting sequence and its identification. Spout assigns every tuple with a global message identifier, which is later used by a transcoding bolt to acknowledge (ack) the tuple. If the transcoding bolt fails or the tuple experiences a timeout, the guaranteed message processing of Storm will replay the tuple.

```
Algorithm 3 The GOP level of granularity for the input video spout method

INPUT: Input video frames.
OUTPUT: A tuple grouped in a micro-batch list of 15 frames GOP, seqid and msgID.

1: function NEXTTUPLE() ▷ The tuple to be emitted to the next consuming bolt
2:     msgID ← randomUUID
3:     while (buffer.size() < current computed GOP bytes size) do
4:         if nRead ← input video stream read up to GOP byte size ! = empty then
5:             buffer ← nRead video bytes are concatenated to the previously stored buffer
6:         end if
7:     end while
8:     s ← encode the buffer into base64 string characters
9:     seqid + +; ▷ incrementing the emit sequence id for every tuple
10:    The output collector emits tuple grouped in ((s,seqid), msgID)
11: end function
```

4.2.5 The Transcoding Bolt

The transcoding bolt is the first stage of bolt instances found in the storm-based video transcoder topology. In a bolt instance, the received tuple is decoded back to the video bytes format and piped to the FFmpeg and FFprobe process respectively. The transcoded output from the FFmpeg process is temporarily stored in key-value TreeMap. Fig. 4.3 illustrates the sample output data generated from the FFprobe process. FFprobe is used to compute the GOP byte size for each video frame stored in TreeMap. Once the FFm-
Chapter 4: Distributed Video Transcoding on a Selected Big Data Processing Platform

A peg process has completed the very last frame of a coded GOP, a single GOP packet has been emitted to the reporter bolt in the final stage.

Algorithm 4 illustrates the algorithm of a transcoding bolt component. Similarly at initialization, received tuples are loaded into memory. The tuple is then partitioned into fields of "coded video" and "emit seq". The video source tuple received is converted back to byte streams. Every time the next tuple is executed, a new output tuple in a list of transcoded GOP in base64-coded text and its received sequence order will be emitted. The bolt acknowledges (ack) the input tuple after it has successfully emitted the transcoded GOP tuple. If the transcoding bolt fails or the input tuple experiences a timeout, the guaranteed message processing of Storm will replay the tuple.

4.2.6 The Sorting Bolt

The sorting bolt is basically responsible for sorting the small piles of video tuples which facilitates the final sort in the reporting bolt to create a larger pile from those small piles stored in keys and value pair in terms of a sequence number and a set of video tuples in accordance to their corresponding video source. This is done by using Treemap where a set of video tuples are sorted in real time in a balanced binary search tree. This sorting bolt can be neglected and needed only when there is way too many number of transcoded video streams to be stored in memory for sorting purposes and which could not handled by a reporting bolt alone.

4.2.7 The Reporting Bolt

Algorithm 5 shows a reporting bolt component. A reporting bolt is responsible for gathering all transcoded GOP segments emitted from all worker nodes using streaming joins. Every time the next tuple is received; the transcoded GOP is stored and sorted in the correct order before writing them to the storage drive.

4.2.8 The Topology Builder

Algorithm 6 shows the spouts and bolts in building the topology and their parallelism hint for thread-level parallelism. A spout emitted tuple consists of two fields, the GOP and emitting sequence fields. The spout tuples are then distributed to the transcoding bolts by fields grouping at parallelism of three. In the final stage, a reporter bolt gathers
[FRAME]
key_frame=1
pkt_pos=48049315
pkt_size=96856
width=1920
height=1080
pict_type=I
coded_picture_number=1035
[/FRAME]

[FRAME]
key_frame=0
pkt_pos=48146171
pkt_size=57757
width=1920
height=1080
pict_type=P
coded_picture_number=1036
[/FRAME]

[FRAME]
key_frame=0
pkt_pos=48203928
pkt_size=13152
width=1920
height=1080
pict_type=B
coded_picture_number=1037
[/FRAME]

[FRAME]
key_frame=0
pkt_pos=48217080
pkt_size=11636
width=1920
height=1080
pict_type=B
coded_picture_number=1038
[/FRAME]

Figure 4.3: FFprobe sample output example
Algorithm 4 The Transcoding Bolt Algorithm

**INPUT:** Tuple grouped with a micro-batch of 15 frames, seqid and msgID

**OUTPUT:** Tuple grouped with a micro-batch of transcoded 15 frames and seqid.

1. `function EXECUTE(tuple)` ▶ Transcode video tuple
2. `gopstring ← get the tuple string by the field "group of picture"`  
3. `seq ← get the tuple integer by the field "seq"`  
4. `gop ← decode the gop coded text to byte format`  
5. `write the gop to the output stream and redirect to the transcoding process`  
6. `function THREAD` ▶ Store video streams in memory & pipe to FFprobe
7. `while true do`  
8. `if nRead ffmpeg ← transoded video stream read ! = empty then`  
9. `buffer ← transoded video stream of nRead ffmpeg bytes`  
10. `s ← encode the buffer to base64 string characters`  
11. `TreeMap ← store (counting / s) as a key/value pair`  
12. `write the buffer to the output stream and redirect to the FFprobe process`  
13. `counting ++`  
14. `end if`  
15. `end while`  
16. `end function`  
17. `function THREAD` ▶ Get the GOP size from FFprobe and store in memory
18. `while true do`  
19. `if nRead ffmpeg probe ← computed packet size ! = empty then`  
20. `pktsizetreeMap ← store (count ffmpeg probe, nRead ffmpeg probe) as a key/value pair`  
21. `count ffmpeg probe ++`  
22. `end if`  
23. `end while`  
24. `end function`  
25. `function THREAD` ▶ Read GOP up to its exact byte size and emit to the reporting bolt
26. `while true do`  
27. `if FFprobe have computed the size of video packet up to the GOP size then`  
28. `for each video frame in the GOP read`  
29. `gopsize + = get the corresponding video packet size stored in the pktsizetreeMap`  
30. `end for`  
31. `if overflowdata > 0 then`  
32. `bufferemit ← decode the pktdatatreeMap[pktdataindex] to bytes format`  
33. `if (overflowdata2 ← overflowdata - gopsize) > 0 then`  
34. `bufferTotal ← concatenate the bufferemit without overflow`  
35. `pktdataindex ++`  
36. `overflowdata ← overflowdata2`  
37. `else if overflowdata < gopsize then`  
38. `bufferTotal ← concatenate bufferemit`  
39. `pktdataindex ++`  
40. `overflowdata ← 0`  
41. `end if`  
42. `bufferTotal ← concatenate bufferemit`  
43. `pktdataindex ++`  
44. `overflowdata ← 0`  
45. `while bufferemitTotal < gopsize`
bufferemit ← decode pktdatatreeMap[pktdatindex] to bytes format

if gopsize > bufferemitTotal + bufferemit.length then
  if bufferemitTotal == 0 then
    bufferTotal ← concatenate bufferemit
  else
    bufferTotal ← concatenate bufferemit
    bufferemitTotal = bufferemit.length
    pktdatindex = +
    overflowdata ← 0
  end if
else if overflowdata ← bufferemitTotal + bufferemit.length − gopsize > 0 then
  bufferTotal ← concatenate bufferemit without overflow
  bufferemitTotal + = (gopsize − bufferemitTotal)
  break
else if gopsize == bufferemitTotal + bufferemit.length
  bufferTotal ← concatenate bufferemit
  bufferemitTotal + = bufferemit.length
  pktdatindex = +
  overflowdata ← 0
  break
end if
end while

The collector emits tuple grouped in (encode(bufferTotal), seqtreeMap[emitindex])

The collector acks the emitted tuple

counttemp + = groupofpicturesize
bufferemitTotal ← 0
gopsize ← 0
emitindex + +
end if
end if
end while
end function
Algorithm 5 The Reporting Bolt Algorithm

**INPUT:** Tuple grouped with a micro-batch of transcoded 15 frames and seqid

**OUTPUT:** Sorted micro-batches of GOP ready to view

1. **procedure** EXECUTE(tuple)  
   ▶ Joins streams from upstream transcoding bolt instances and store to memory

2. **componentId** ← get the source component of the received tuple

3. **if** componentId equals to “transcoder1”

4. mainstream ← get the string tuple by the field “MainStream”

5. indexseq ← get the integer tuple by the field “seq”

6. sortingtreeMap ← store (indexseq, mainstream) as a key/value pair

7. **else if** componentId equals to “transcoder2”

8. substream ← get the string tuple by the field “MainStream2”

9. indexseq2 ← get the integer tuple by the field “seq2”

10. sortingtreeMap2 ← store (indexseq2, substream) as a key/value pair

11. **else if** componentId equals to “transcoder3”

12. substream3 ← get the string tuple by the field “MainStream3”

13. indexseq3 ← get the integer tuple by the field “seq3”

14. sortingtreeMap3 ← store (indexseq3, substream3) as a key/value pair

15. **end if**

16. **function** THREAD ▶ Sort GOPs of video output 1 into order and write them to drive

17. **while** true do

18. **if** sortingtreeMap contains the key nextindex then

19. bufferReporter decode sortingtreeMap[nextindex] to bytes format

20. write bufferReporter to outputStream

21. nextindex ++

22. **end if**

23. **end while**

24. **end function**

25. **function** THREAD ▶ Sort GOPs of video output 2 into order and write them to drive

26. **while** true do

27. **if** sortingtreeMap2 contains the key nextindex2 then

28. bufferReporter2 decode sortingtreeMap2[nextindex2] to bytes format

29. write the bufferReporter2 to the outputStream2

30. nextindex2 ++

31. **end if**

32. **end while**

33. **end function**

34. **function** THREAD ▶ Sort GOPs of video output 3 into order and write them to drive

35. **while** true do

36. **if** sortingtreeMap3 contains the key nextindex3 then

37. bufferReporter3 decode sortingtreeMap3[nextindex3] to bytes format

38. write the bufferReporter3 to the outputStream3

39. nextindex3 ++

40. **end if**

41. **end while**

42. **end function**

43. **end procedure**
all transcoded GOPs by joining *tuples* to one point. Again, each *tuple* is grouped by fields grouping.

### 4.2.9 The Storm Pluggable Scheduler

In Storm, Nimbus uses a scheduler to assign *executors* to *supervisors*. The default scheduler of Storm aims at scheduling computing resources evenly to the topology components (*spouts* and *bolts*). In a homogenous Storm cluster, the default scheduler works well in terms of fairness distribution among topology components but it is unpredictable for users to locate the placement of every topology component in the Storm cluster. In a heterogeneous platform such as one used in our proposed system, we have faced unfair scheduling in the storm default scheduler where more *executors* are assigned to a *supervisor* results in executing more *tasks* and underloading on the other side. Therefore, we have implemented a custom scheduler to address this issue. On the other hand, users can reserve the name of *supervisors* under storm.scheduler.meta in a configuration file, in which users can specify any supervisors in a "key-value" pairs to facilitate task scheduling. For instance, the scheduler ensures that a *spout* named "Main-Spout" in the topology "video topology" runs on a *supervisor* named "supervisor1" can be configured at supervisor.scheduler.meta. If a Storm’s component consists of three *executors*, the scheduler will allocate three *workers* from three different selected *supervisors* to the *executors* accordingly. The number of *executors* for a Storm’s component is basically corresponding to the number of parallelism hint set in stream groupings. Algorithm 7 shows the portion of the scheduler code for video *spouts*. First, we find out all components and *executors* of the topology that are needed for scheduling. Next, component’s *executors* are scheduled to specified *supervisors* for *task* execution. Similar method is applied to the transcoding *bolt* instances and the reporter *bolt* instance in the proposed system.

### 4.2.10 Storm Custom Stream Grouping

There are many stream groupings provided in Storm. These groupings are responsible for the task assignment in which the emitted tuples are sent to the consuming *bolt* instances. A key or fields grouping is a strategy used in the stream processing framework to simplify the development of parallel stateful operators. Inside the fields grouping approach, a stream of *tuples* is usually partitioned based on the key of the
Algorithm 6 The Topology Builder Algorithm

1: procedure MAIN() ▷ create the proposed system topology with parallelism hints
2:   Declare a TopologyBuilder called builder
3:   set MainSpout at parallelism of 1
4:   set SubStreamSpout at parallelism of 1
5:   set SubStreamSpout3 at parallelism of 1
6:   set TranscoderBolt1 at parallelism of 3 with ResourceAwarePKG from MainSpout
7:   set TranscoderBolt2 at parallelism of 3 with ResourceAwarePKG from SubStreamSpout
8:   set TranscoderBolt3 at parallelism of 3 with ResourceAwarePKG from SubStreamSpout3
9:   set ReportBolt at parallelism of 1 with ResourceAwarePKG from TranscoderBolt1, TranscoderBolt2 and TranscoderBolt3
10: end procedure

Algorithm 7 The Storm Custom Scheduler Algorithm

1: procedure SCHEDULE(Topologies topologies, Cluster cluster) ▷ create our own scheduler.
2:   TopologyDetails ← get by the name "transcodertopology"
3:   ...
4:   componentToExecutors ← map listings of component to executors details that need scheduling
5:   for int i ← 0; i < Spoutname.length; i ++ do
6:     if componentToExecutors contains the key Spoutname[i] then
7:       ExecutorDetails ← get componentToExecutors by the name in Spoutname[i]
8:       specialSupervisor ← get supervisors details
9:       SpecialSupervisorDetails ← null
10:      for each supervisor do
11:         Map meta ← get supervisor metadata
12:         if meta name equals to Supervisorname for Spout[i] then
13:            specialSupervisor ← supervisor
14:            break
15:       end if
16:      end for
17:     if specialSupervisor! = null then
18:       List availableWorkerSlots ← get available slots from the specialSupervisor
19:       if availableWorkerSlots is empty && executors is not empty then
20:          for Integer port ← get used ports from the specialSupervisor do
21:             free workerslot ports under the specialSupervisor
22:          end for
23:       end if
24:       availableSlots ← get available slots for the specialSupervisor
25:       assign availableSlots to executors
26:     end if
27:   end if
28: end procedure

50
stream into several disjoint sub-streams, ensuring *tuples* with the same key are processed by the workers of the same *bolt*. Typically, a hash function is used to map the tuple key to its corresponding sub-stream. Such hash-based routing allows sub-stream messages to be routed from their sources solely via their key, without needing to keep any state or to coordinate among *bolts*. Fields grouping is also known as a "single-choice" paradigm which causes imbalance in task distribution across the Storm cluster since it disregards the frequency of messages with the same key appearing during stream grouping. Hence, fields grouping basically groups data based on hash assignment which is scalable, has low memory but is stateful and load imbalance. In order to achieve excellent load balancing, another partitioning scheme called shuffle grouping is used. This is done in a round-robin routing approach which sends messages to a bolt in cyclic order, irrespective of their keys. However, this scheme is mostly used for stateless computation. An additional aggregation phase and memory are required for computation, and grouped by shuffle grouping to become stateful. Nasir, Muhammad Anis Uddin, et al. [111] introduced Partial Key Grouping (PKG), a new stream partitioning scheme that achieves better load balancing than fields grouping while incurring less overhead memory compared to shuffle grouping. Authors in [111] focus on solving issues of load balancing on stateful applications when key distribution is skewed (limited scalability). In PKG, the upstream *spout* creates a balanced partition of messages for the downstream *bolts*, attaining load balancing for each edge of DAG. They claimed that by comparing it to fields grouping, PKG reduces load imbalance to multiple orders of magnitude. Thus, the throughput and latency of an example application can be improved by up to 45%.

### 4.2.11 The Resource-Aware Task Allocation Scheme

In a homogeneous computing environment, every node has identical performance and capacity. Effective task scheduling is measured by the quantity of nodes instead of their individual quality since the resources are allocated evenly across all available nodes. For most cases, task scheduling and resource allocation in the homogeneous computing environment divide workload across all nodes evenly, which results in identical workload distribution across all nodes. Contrary to the homogeneous computing environment, computing nodes have different computing performances and capacities in heterogeneous systems. Under identical workload distribution, high-performance nodes can process data faster than the low-performance nodes. This session proposes
a resource-aware task allocation scheme for leveraging heterogeneous resources to the proposed system. The proposed scheme assigns computational tasks based on computer processing performance. In comparison with the resource-aware task allocation scheme in Hadoop or the MapReduce framework, which splits the data file based on computer processing performance, our proposed scheme on-the-fly decides which supervisor will receive the video tuple based on real-time monitoring of the number of GOP tuples received and computing capabilities of supervisors per bolt. Fig. 4.4 illustrates the concept of the proposed scheme. A task manager decides how the tasks are allocated to the computational resources in the proposed scheme. Algorithm 8 depicts the processing capability $P_{Ci}$ of each computer and the incremental count $F_{Ci}$ is triggered every time a GOP tuple is emitted where allocating task to a computer has occurred. The objective of the proposed scheme is to reduce the processing time of the worst-performing computers among the heterogeneous slave computers $C_i$.

$$\text{Selected supervisor} = \frac{F_{C_1} \text{ corresponding } P_{C_1}}{F_{C_2} \text{ corresponding } P_{C_2}}$$  \hspace{1cm} (4.2.1)

where $F_{C_1}$ corresponds to FirstChoice in the partial key grouping scheme

$F_{C_1}$ corresponds to SecondChoice in the partial key grouping scheme

The two-choice paradigm is where the Partial Key Grouping scheme comes into action. The final selected supervisor is computed using equation (4.2.1) shown above. The Partial Key Grouping algorithm will select the two best supervisor candidates and our proposed scheme computes which supervisor candidate is currently underloaded or overloaded based on their computer processing capabilities $P_{Ci}$ and the present number of GOPs allocated $F_{Ci}$ for the two best supervisor candidates.
Algorithm 8: The Resource-Aware Stream Grouping Algorithm

1: procedure PREPARE(WorkerTopologyContext, GlobalStreamId & a list of targetTasks)
2:   this.targetTasks ← targetTasks;
3:   if GlobalStreamId.fields! = null then
4:     GlobalStreamId.outFields ← get component output fields for the stream
5:   end if
6:   declare an integer array named assignedtuples with the size of targetTasks.size()
7: end procedure

8: procedure CHOOSETASKS(int taskId, List of values)
9:   List of integer boltIds ← ArrayList[1]
10:   if values.size() > 0 then
11:     \( \text{i.e. Partial Key Grouping Algorithms} \quad \triangleright \text{Any Grouping Algorithm} \)
12:     \( \ldots \)
13:     firstChoice ← Math.abs(h1.hashBytes(raw).asLong())%targetTasks.size()
14:     secondChoice ← Math.abs(h2.hashBytes(raw).asLong())%targetTasks.size()
15:   \}
16:   switch firstChoice do
17:     case 0
18:       assert(b1 ← \( P_{C_0} \))
19:       case 1
20:         assert(b1 ← \( P_{C_1} \))
21:       case 2
22:         assert(b1 ← \( P_{C_2} \))
23:       switch secondChoice do
24:         case 0
25:           assert(b2 ← \( P_{C_0} \))
26:         case 1
27:           assert(b2 ← \( P_{C_1} \))
28:         case 2
29:           assert(b2 ← \( P_{C_2} \))
30:           selected ← if \((\text{assignedtuples[firstChoice]} \ast 15/b1) > (\text{assignedtuples[secondChoice]} \ast 15/b2))\ then choose secondChoice else choose firstChoice\quad \triangleright \text{resource-aware task allocation, refer to equation (4.2.1)}\)
31:           boltIds add targetTasks.get(selected)
32:           assignedtuples[selected] + +;
33:       end if
34:     return boltIds;
35: end procedure
Figure 4.4: Selection of computational resources and task allocation
CHAPTER 5

Experimental Results and Discussions

The proposed framework in Section 4.2 is implemented in this chapter. The first few sections investigate the performance evaluation on some existing researches that have been previously done on this area, comparing with the proposed system. The remaining of this chapter validates our proposed system in terms of data granularity and fair resource-aware task allocation to heterogeneous systems. Finally, the produced output videos are rated to ensure meeting the expected quality standard.

5.1 Performance Evaluation

Four computers with their CPU and GPU specifications listed in Table 5.1 were used in the experiment. GPUs were used as a related work comparison to the proposed system. A test sequence (Big Buck Bunny, 1920×1080, 24 fps, 9 minutes 57 seconds) was used to simulate a real-time video stream and was transcoded into the H.264 format with multiple output resolutions (1920×1080 or 1080p, 1280×720 or 720p and 640×480 or 480p) using the proposed real-time distributed system to obtain the maximum capability of the system.

5.1.1 The Nvidia Hardware Encoder

Table 5.2 presents the average transcoding rate of the related work, NVENC when different output resolutions are selected. The input video stream is divided into multiple sub-streams, allowing multiple transcoding instances to be executed simultaneously.
### Table 5.1: CPU and GPU specification, the average transcoding rate of four computers and two GPUs used in the experiment, each with a unique label.

<table>
<thead>
<tr>
<th>Computer and GPU label</th>
<th>CPU and GPU specification</th>
<th>Baseline transcoding rate (fps)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1080p</td>
</tr>
<tr>
<td>C1</td>
<td>Intel® Core™ i5-3210M, 2.5GHz</td>
<td>13</td>
</tr>
<tr>
<td>C2</td>
<td>Intel® Core™ i5-3470, 3.2GHz</td>
<td>23</td>
</tr>
<tr>
<td>G1</td>
<td>GTX 660, Kepler-based, 1st Gen NVENC</td>
<td>170</td>
</tr>
<tr>
<td>C3 and C4</td>
<td>Intel® Xeon® E5-2630 v2, 2.6GHz</td>
<td>36</td>
</tr>
<tr>
<td>G2</td>
<td>Tesla K40c, Kepler-based, 1st Gen NVENC</td>
<td>170</td>
</tr>
</tbody>
</table>

### Table 5.2: Transcoding speed of NVENC when output resolution is set to 1920×1080, 1280×720 and 640×480 respectively.

<table>
<thead>
<tr>
<th>GPU label</th>
<th>Video input 1</th>
<th>Video input 2</th>
<th>Video input 3</th>
<th>Total transcoding rate (fps)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Transcoded frames</td>
<td>Transcoding rate (fps)</td>
<td>Transcoded frames</td>
<td>Transcoding rate (fps)</td>
</tr>
<tr>
<td>G1</td>
<td>4949</td>
<td>86</td>
<td>4949</td>
<td>85</td>
</tr>
<tr>
<td>G2</td>
<td>4949</td>
<td>57</td>
<td>4949</td>
<td>57</td>
</tr>
</tbody>
</table>
However, even with running multiple transcode instances to obtain the maximum utilization of NVENC capacity, the total transcoding rate remains the same. This has indicated that the maximum processing rate of NVENC at a full HD resolution (1080p) video is 170 fps, regardless of the increased number of transcoding sessions requested. Furthermore, Nvidia Kepler-based Geforce GPUs are limited to a maximum of two encoding sessions that can be executed simultaneously. Scaling up the system with multiple NVENC GPUs is also limited to a maximum of two encoding sessions per system when there is a low-end Nvidia GPU such as a Geforce GPU or lower is present in the system. This limits the implementation of NVENC transcoders onto the Storm cluster which requires more transcoding instances at Storm’s thread-level parallelism.

### 5.1.2 The Batch-Oriented Distributed Video Transcoding System

In the related study on the batch-oriented video transcoding approach, the default block size of 64MB is selected for simplicity. The parallelism of map phases in MapReduce computation is done by splitting the input video file according to the number of computers deployed in the hadoop cluster. The reason is the difference in properties of data sets can affect the performance of the Hadoop cluster [112] as processing fine grains of input data could result in trivial processing in the Map task. The result of having the setup time becomes the dominating time factor of the execution time of the Map phase by not fully utilizing compute resources. This effect is further amplified as the amount of small data files to the process increases.

Table 5.3 depicts the transcoding rate of the related work on batch processing using the MapReduce framework or the Hadoop cluster. The transcoding rate per computer of the batch approach yields similar throughput compared to the baseline benchmarked, that fully utilizes processing resources to their maximum capacities. However, there is a major latency issue in uploading video files onto HDFS with the specification shown in Fig. 5.1, which took 13 minutes just to upload a total size of 431MB 1080p videos.
CHAPTER 5: EXPERIMENTAL RESULTS AND DISCUSSIONS

Once the video transcoding batch job begins, the batch job continues until it is done. There is no interaction with the user while the transcoding process is being executed. The user has to wait until the job finishes since the transcoded video frames are unreachable during video transcoding as the output video frames are stored locally on other compute nodes. Uploading output video files back to HDFS only begins when the transcoded video chunks have been completely written to local temporary storages. Furthermore, the input video file is not sharded into size according to computing ratios in this related study experiment, leaving poor load distribution in the hadoop heterogeneous computing platform.
<table>
<thead>
<tr>
<th>Output Video Resolution</th>
<th>Computer label</th>
<th>Input video file 1</th>
<th>Input video file 2</th>
<th>Input video file 3</th>
<th>Transcoding time per computer (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>video acquisition rate = 400 fps</td>
<td>video acquisition rate = 360 fps</td>
<td>video acquisition rate = 214 fps</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>4949 frames</td>
<td>4949 frames</td>
<td>4430 frames</td>
<td></td>
</tr>
<tr>
<td>Transcoding time (secs)</td>
<td></td>
<td>Transcoding time (secs)</td>
<td>Transcoding time (secs)</td>
<td>Transcoding time (secs)</td>
<td></td>
</tr>
<tr>
<td>1080p</td>
<td>C1</td>
<td>380.69 (13fps)</td>
<td>-</td>
<td>-</td>
<td>380.69 (13fps)</td>
</tr>
<tr>
<td></td>
<td>C2</td>
<td>-</td>
<td>215.17 (23fps)</td>
<td>-</td>
<td>215.17 (23fps)</td>
</tr>
<tr>
<td></td>
<td>C3</td>
<td>-</td>
<td>-</td>
<td>110.75 (40fps)</td>
<td>110.75 (40fps)</td>
</tr>
<tr>
<td>Total Cluster transcoding time</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>380.69 (76fps)</td>
</tr>
<tr>
<td>720p</td>
<td>C1</td>
<td>176.75 (28fps)</td>
<td>-</td>
<td>-</td>
<td>176.75 (28fps)</td>
</tr>
<tr>
<td></td>
<td>C2</td>
<td>-</td>
<td>98.98 (50fps)</td>
<td>-</td>
<td>98.98 (50fps)</td>
</tr>
<tr>
<td></td>
<td>C3</td>
<td>-</td>
<td>-</td>
<td>56.08 (79fps)</td>
<td>56.08 (79fps)</td>
</tr>
<tr>
<td>Total Cluster transcoding time</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>176.75 (157fps)</td>
</tr>
<tr>
<td>480p</td>
<td>C1</td>
<td>79.98 (66fps)</td>
<td>-</td>
<td>-</td>
<td>79.98 (66fps)</td>
</tr>
<tr>
<td></td>
<td>C2</td>
<td>-</td>
<td>41.59 (119fps)</td>
<td>-</td>
<td>41.59 (119fps)</td>
</tr>
<tr>
<td></td>
<td>C3</td>
<td>-</td>
<td>-</td>
<td>25.91 (171fps)</td>
<td>25.91 (171fps)</td>
</tr>
<tr>
<td>Total Cluster transcoding time</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>79.98 (356fps)</td>
</tr>
</tbody>
</table>

Table 5.3: Transcoding speed of the batch-oriented distributed video transcoding approach when output resolution is set to 1920×1080, 1280×720 and 640×480 respectively.
5.1.3 Our Proposed System

![Percentage increase in processing throughput](image)

**Figure 5.2:** Percentage increase in system throughput of the batch approach and our proposed system compared to the baseline benchmarked

Tables 5.4 to 5.6 show the average transcoding rate of each computer in the proposed system when different output resolutions are selected. The number of pending tuples in the experiment is restricted to 200 tuples. When the output video is set to full HD resolution (1080p), the proposed system yields a total cluster transcoding rate of 115.2 fps which is 3 times faster than the highest local CPU baseline transcoding rate benchmarked in Table 5.1, 1.5 times slower than the NVENC hardware transcoder but outperforms the performance of the NVENC hardware transcoder when the output video resolution of 720p or below is selected. Fig. 5.2 depicts the transcoding rate of the MapReduce over Hadoop approach that has yielded similar throughput compared to the proposed system in the same cluster size. This indicates that the proposed system’s throughput does not bounded by network latencies or data localities where computation to communication ratio is fully utilized to system maximum processing capacity. Scaling out the cluster size can further increase the performance and resource availability of the proposed system to handle more transcoding requests. In a real-time video surveillance system, the proposed method is able to transcode more live full HD streams produced from cameras compared to the NVENC approach where only two encoding sessions per system are permitted. On the other hand, batch distributed ap-
proaches cannot to be implemented on real-time transcoding of live video streams since they only consume data via files. Additionally, a full HD resolution video stream request is normally delivered using the original video source without transcoding. Thus, the high transcoding rate at lower resolutions in the proposed system is more demanding than the NVENC approach.

5.1.4 Load Distribution in the Proposed System

From Tables 5.4 to 5.6, one can see that the number of frames transcoded for all computers is similar to each other. The Storm’s task allocation scheme distributes video frames across the cluster evenly. This is due to the fact that stream groupings used in Storm apply a simple round-robin algorithm that does not consider heterogeneous processing rates, leaving poor load distribution in the heterogeneous video transcoding platform. Therefore, the performance of the entire system is bounded by the worst-performing computer in the system. This issue has been addressed in sections 5.3 and 5.4 by applying the resource-aware task allocation scheme proposed in section 4.2.11.
### Table 5.4: The transcoding speed of the proposed system when output resolution is set to 1920×1080.

<table>
<thead>
<tr>
<th>Computer label</th>
<th>Video spout 1</th>
<th>Video spout 2</th>
<th>Video spout 3</th>
<th>Total transcoding time per computer (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C1</td>
<td>1267 324 (3.9fps)</td>
<td>1261 315.25 (4.0fps)</td>
<td>1095 232.97 (4.7fps)</td>
<td>324 (12.6fps)</td>
</tr>
<tr>
<td>C2</td>
<td>1227 92.95 (13.2fps)</td>
<td>1211 163.65 (7.4fps)</td>
<td>1142 178.44 (6.4fps)</td>
<td>178.44 (27fps)</td>
</tr>
<tr>
<td>C3</td>
<td>1198 106.96 (11.2fps)</td>
<td>1216 90.07 (13.5fps)</td>
<td>1074 85.92 (12.5fps)</td>
<td>106.96 (37.2fps)</td>
</tr>
<tr>
<td>C4</td>
<td>1257 87.90 (14.3fps)</td>
<td>1261 112.59 (11.2fps)</td>
<td>1119 86.74 (12.9fps)</td>
<td>112.59 (38.4fps)</td>
</tr>
<tr>
<td><strong>Total Cluster transcoding time</strong></td>
<td><strong>324 (115.2fps)</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 5.5: The transcoding speed of the proposed system when output resolution is set to 1280×720.

<table>
<thead>
<tr>
<th>Computer label</th>
<th>Video spout 1</th>
<th>Video spout 2</th>
<th>Video spout 3</th>
<th>Total transcoding time per computer (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4949 frames emitted video acquisition rate = 273 fps</td>
<td>4949 frames emitted video acquisition rate = 271 fps</td>
<td>4430 frames emitted video acquisition rate = 318 fps</td>
<td></td>
</tr>
<tr>
<td>Transcoded frames</td>
<td>Transcoding time (secs)</td>
<td>Transcoded frames</td>
<td>Transcoding time (secs)</td>
<td>Transcoded frames</td>
</tr>
<tr>
<td>C1</td>
<td>1039</td>
<td>152.79 (6.8fps)</td>
<td>1345</td>
<td>164.02 (8.2fps)</td>
</tr>
<tr>
<td>C2</td>
<td>1341</td>
<td>55.86 (24fps)</td>
<td>1273</td>
<td>90.93 (14fps)</td>
</tr>
<tr>
<td>C3</td>
<td>1287</td>
<td>49.5 (26fps)</td>
<td>1075</td>
<td>39.81 (27fps)</td>
</tr>
<tr>
<td>C4</td>
<td>1282</td>
<td>47.48 (27fps)</td>
<td>1256</td>
<td>44.86 (28fps)</td>
</tr>
<tr>
<td>Total Cluster transcoding time</td>
<td></td>
<td></td>
<td></td>
<td>164.02 (233.8fps)</td>
</tr>
</tbody>
</table>

*Video spout 1, 2, and 3 are output frames from three different computer labels (C1, C2, C3, C4) respectively.*
### Table 5.6: The transcoding speed of the proposed system when output resolution is set to 640×480.

<table>
<thead>
<tr>
<th>Computer label</th>
<th>Video spout 1</th>
<th>Video spout 2</th>
<th>Video spout 3</th>
<th>Total transcoding time per computer (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Transcoded frames</td>
<td>Transcoding time (secs)</td>
<td>Transcoded frames</td>
<td>Transcoding time (secs)</td>
</tr>
<tr>
<td>C1</td>
<td>1254</td>
<td>66 (19fps)</td>
<td>1259</td>
<td>74.06 (17fps)</td>
</tr>
<tr>
<td>C2</td>
<td>1203</td>
<td>35.38 (34fps)</td>
<td>1204</td>
<td>29.37 (41fps)</td>
</tr>
<tr>
<td>C3</td>
<td>1268</td>
<td>22.64 (56fps)</td>
<td>1207</td>
<td>13.12 (92fps)</td>
</tr>
<tr>
<td>C4</td>
<td>1224</td>
<td>18.83 (65fps)</td>
<td>1279</td>
<td>24.60 (52fps)</td>
</tr>
<tr>
<td><strong>Total Cluster transcoding time</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
5.2 Comparative Evaluation of the Fine- and Coarse-Grain Approach for the Proposed System

One of the important aspects in designing the architecture of our proposed system is the issue of data granularity. This experiment is run on different granularity sizes with the same number of parallelism, testing the behavior on how large or fine grained tasks can affect the throughput and consistency of the proposed system. The best trade-off between the load and communication overheads has to be found in order to attain the greatest parallel performance. When the data granularity level is set too fine, the greater potential for parallelism can cause the system performance to suffer from increased communication overheads. On the other hand, the system performance can suffer from load imbalance if data granularity is too coarse. In the following experiments, the number of pending tuples is not restricted. Thus, the video acquisition rate has increased drastically. Tables 5.7 to 5.9 and Figures. 5.6 to 5.8 show that the finer the data granularity, the greater is the loss in the overall speedup from 12.7% (Transcoding at 1080p) up to 30.5% (Transcoding at 480p). Fig. 5.3 depicts finer granularity of data has negligible impact on the 1080p video transcoding throughput as the emitting rate of tuples in Storm manages to catch up with the maximum transcoding capacity of all computers. On the other hand, throughput degradation of computer C3 which implements grain sizes of 3 to 11 frames in Fig. 5.4 has illustrated that the tuple emitting spout no longer catches up with the maximum rate of transcoding capacity in computer C3. On the contrary, computers C1 & C2 experience less throughput degradation in their input frame acquisition rate. This is the reason why sufficiently high emitting throughput in Storm has managed to hide communication overheads caused by fine granularity. Besides that, Fig. 5.5 has noticeably showed significant throughput degradation in computers C2 & C3 where the emitting rate of fine granularity (3 to 9 frames) has failed to keep up with computers C2 & C3 to their maximum video transcoding capacities. In order to catch up with such higher transcoding capacity of C2 & C3, video data is delivered in batches of higher number of frames to hide communication overhead. This is done by overlapping communication over computation which results in a drastic increase of cluster throughput at the grain size of 11 shown in Fig. 5.8. Finally, a coarser grained of 15 frames has allowed the parallel computers to perform almost identical to their local baseline benchmarks which are to their maximum computing capacities.
Chapter 5: Experimental Results and Discussions

<table>
<thead>
<tr>
<th>Data Granularity (frames)</th>
<th>Computer label</th>
<th>Video spout 1</th>
<th>Video spout 2</th>
<th>Video spout 3</th>
<th>Total transcoding rate per computer (fps)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>4949 frames emitted</td>
<td>4949 frames emitted</td>
<td>4430 frames emitted</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>video acquisition rate = 815 fps</td>
<td>video acquisition rate = 1568 fps</td>
<td>video acquisition rate = 1003 fps</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>C1</td>
<td>2.9</td>
<td>2.8</td>
<td>2.7</td>
<td>8.4</td>
</tr>
<tr>
<td></td>
<td>C2</td>
<td>7.0</td>
<td>7.4</td>
<td>7.5</td>
<td>21.9</td>
</tr>
<tr>
<td></td>
<td>C3</td>
<td>11.0</td>
<td>12.0</td>
<td>12.0</td>
<td>35.0</td>
</tr>
<tr>
<td>Total cluster transcoding rate</td>
<td>65.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>C1</td>
<td>3.8</td>
<td>3.7</td>
<td>4.0</td>
<td>11.5</td>
</tr>
<tr>
<td></td>
<td>C2</td>
<td>7.4</td>
<td>7.0</td>
<td>7.6</td>
<td>22.0</td>
</tr>
<tr>
<td></td>
<td>C3</td>
<td>11.0</td>
<td>11.0</td>
<td>10.0</td>
<td>32.0</td>
</tr>
<tr>
<td>Total cluster transcoding rate</td>
<td>65.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>C1</td>
<td>3.8</td>
<td>3.7</td>
<td>3.9</td>
<td>11.4</td>
</tr>
<tr>
<td></td>
<td>C2</td>
<td>7.9</td>
<td>7.6</td>
<td>6.9</td>
<td>22.4</td>
</tr>
<tr>
<td></td>
<td>C3</td>
<td>9.1</td>
<td>13.0</td>
<td>12.0</td>
<td>34.1</td>
</tr>
<tr>
<td>Total cluster transcoding rate</td>
<td>67.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>C1</td>
<td>3.9</td>
<td>3.8</td>
<td>4.0</td>
<td>11.7</td>
</tr>
<tr>
<td></td>
<td>C2</td>
<td>7.5</td>
<td>8.7</td>
<td>7.4</td>
<td>23.6</td>
</tr>
<tr>
<td></td>
<td>C3</td>
<td>9.6</td>
<td>13.0</td>
<td>12.0</td>
<td>34.6</td>
</tr>
<tr>
<td>Total cluster transcoding rate</td>
<td>69.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>C1</td>
<td>4.2</td>
<td>4.0</td>
<td>3.8</td>
<td>12.0</td>
</tr>
<tr>
<td></td>
<td>C2</td>
<td>7.6</td>
<td>8.4</td>
<td>6.9</td>
<td>22.9</td>
</tr>
<tr>
<td></td>
<td>C3</td>
<td>14.0</td>
<td>9.7</td>
<td>12.0</td>
<td>35.7</td>
</tr>
<tr>
<td>Total cluster transcoding rate</td>
<td>70.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>C1</td>
<td>3.7</td>
<td>4.1</td>
<td>4.3</td>
<td>12.1</td>
</tr>
<tr>
<td></td>
<td>C2</td>
<td>7.3</td>
<td>9.1</td>
<td>9.1</td>
<td>25.5</td>
</tr>
<tr>
<td></td>
<td>C3</td>
<td>11.0</td>
<td>12.0</td>
<td>13.0</td>
<td>36.0</td>
</tr>
<tr>
<td>Total cluster transcoding rate</td>
<td>73.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.7: Transcoding speed of the proposed system when data granularity is set to a frame interval of 3, 5, 7, 9, 11 and 15 at resolution 1920×1080.

![Figure 5.3](image-url): Computer throughput of different data granularities set at resolution 640×480.
### Table 5.8: Transcoding speed of the proposed system when data granularity is set to a frame interval of 3, 5, 7, 9, 11 and 15 at resolution 1280×720.

<table>
<thead>
<tr>
<th>Data Granularity (frames)</th>
<th>Computer label</th>
<th>Video spout 1</th>
<th>Video spout 2</th>
<th>Video spout 3</th>
<th>Total transcoding rate per computer (fps)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C1</td>
<td>4949 frames emitted</td>
<td>4949 frames emitted</td>
<td>4430 frames emitted</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C2</td>
<td>video acquisition rate = 815 fps</td>
<td>video acquisition rate = 1568 fps</td>
<td>video acquisition rate = 1003 fps</td>
<td></td>
</tr>
<tr>
<td></td>
<td>C3</td>
<td>Transcoding rate (fps)</td>
<td>Transcoding rate (fps)</td>
<td>Transcoding rate (fps)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>C1</td>
<td>8.5</td>
<td>8.6</td>
<td>8.5</td>
<td>25.6</td>
</tr>
<tr>
<td></td>
<td>C2</td>
<td>14.0</td>
<td>14.0</td>
<td>14.0</td>
<td>42.0</td>
</tr>
<tr>
<td></td>
<td>C3</td>
<td>21.0</td>
<td>20.0</td>
<td>19.0</td>
<td>60.0</td>
</tr>
<tr>
<td></td>
<td>Total cluster transcoding rate</td>
<td>127.6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>C1</td>
<td>10.0</td>
<td>8.3</td>
<td>7.8</td>
<td>26.1</td>
</tr>
<tr>
<td></td>
<td>C2</td>
<td>14.0</td>
<td>14.0</td>
<td>14.0</td>
<td>42.0</td>
</tr>
<tr>
<td></td>
<td>C3</td>
<td>21.0</td>
<td>29.0</td>
<td>15.0</td>
<td>65.0</td>
</tr>
<tr>
<td></td>
<td>Total cluster transcoding rate</td>
<td>133.1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>C1</td>
<td>9.8</td>
<td>7.6</td>
<td>9.5</td>
<td>26.9</td>
</tr>
<tr>
<td></td>
<td>C2</td>
<td>15.0</td>
<td>15.0</td>
<td>17.0</td>
<td>47.0</td>
</tr>
<tr>
<td></td>
<td>C3</td>
<td>29.0</td>
<td>20.0</td>
<td>16.0</td>
<td>65.0</td>
</tr>
<tr>
<td></td>
<td>Total cluster transcoding rate</td>
<td>138.9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>C1</td>
<td>8.3</td>
<td>8.1</td>
<td>8.7</td>
<td>25.1</td>
</tr>
<tr>
<td></td>
<td>C2</td>
<td>15.0</td>
<td>14.0</td>
<td>15.0</td>
<td>44.0</td>
</tr>
<tr>
<td></td>
<td>C3</td>
<td>29.0</td>
<td>20.0</td>
<td>22.0</td>
<td>71.0</td>
</tr>
<tr>
<td></td>
<td>Total cluster transcoding rate</td>
<td>140.1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>C1</td>
<td>10.0</td>
<td>8.0</td>
<td>8.8</td>
<td>26.8</td>
</tr>
<tr>
<td></td>
<td>C2</td>
<td>16.0</td>
<td>16.0</td>
<td>14.0</td>
<td>46.0</td>
</tr>
<tr>
<td></td>
<td>C3</td>
<td>22.0</td>
<td>18.0</td>
<td>28.0</td>
<td>68.0</td>
</tr>
<tr>
<td></td>
<td>Total cluster transcoding rate</td>
<td>140.8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>C1</td>
<td>9.3</td>
<td>9.6</td>
<td>9.3</td>
<td>28.2</td>
</tr>
<tr>
<td></td>
<td>C2</td>
<td>15.0</td>
<td>15.0</td>
<td>16.0</td>
<td>46.0</td>
</tr>
<tr>
<td></td>
<td>C3</td>
<td>24.0</td>
<td>25.0</td>
<td>25.0</td>
<td>74.0</td>
</tr>
<tr>
<td></td>
<td>Total cluster transcoding rate</td>
<td>148.2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.4: Computer throughput of different data granularities set at resolution 640×480.
### Table 5.9: Transcoding speed of the proposed system when data granularity is set to a frame interval of 3, 5, 7, 9, 11 and 15 at resolution 640×480.

<table>
<thead>
<tr>
<th>Data Granularity (frames)</th>
<th>Computer label</th>
<th>Video spout 1</th>
<th>Video spout 2</th>
<th>Video spout 3</th>
<th>Total transcoding rate per computer (fps)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Computer label</td>
<td>video acquisition rate = 815 fps</td>
<td>video acquisition rate = 1568 fps</td>
<td>video acquisition rate = 1003 fps</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>C1</td>
<td>25.0 fps</td>
<td>20.0 fps</td>
<td>25.0 fps</td>
<td>70.0 fps</td>
</tr>
<tr>
<td></td>
<td>C2</td>
<td>37.0 fps</td>
<td>29.0 fps</td>
<td>42.0 fps</td>
<td>108.0 fps</td>
</tr>
<tr>
<td></td>
<td>C3</td>
<td>39.0 fps</td>
<td>29.0 fps</td>
<td>46.0 fps</td>
<td>114.0 fps</td>
</tr>
<tr>
<td></td>
<td><strong>Total cluster transcoding rate</strong></td>
<td><strong>292.0 fps</strong></td>
<td><strong>292.0 fps</strong></td>
<td><strong>292.0 fps</strong></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>C1</td>
<td>24.0 fps</td>
<td>18.0 fps</td>
<td>27.0 fps</td>
<td>69.0 fps</td>
</tr>
<tr>
<td></td>
<td>C2</td>
<td>42.0 fps</td>
<td>38.0 fps</td>
<td>27.0 fps</td>
<td>107.0 fps</td>
</tr>
<tr>
<td></td>
<td>C3</td>
<td>48.0 fps</td>
<td>51.0 fps</td>
<td>46.0 fps</td>
<td>145.0 fps</td>
</tr>
<tr>
<td></td>
<td><strong>Total cluster transcoding rate</strong></td>
<td><strong>321.0 fps</strong></td>
<td><strong>321.0 fps</strong></td>
<td><strong>321.0 fps</strong></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>C1</td>
<td>20.0 fps</td>
<td>27.0 fps</td>
<td>27.0 fps</td>
<td>74.0 fps</td>
</tr>
<tr>
<td></td>
<td>C2</td>
<td>44.0 fps</td>
<td>32.0 fps</td>
<td>31.0 fps</td>
<td>107.0 fps</td>
</tr>
<tr>
<td></td>
<td>C3</td>
<td>53.0 fps</td>
<td>51.0 fps</td>
<td>51.0 fps</td>
<td>146.0 fps</td>
</tr>
<tr>
<td></td>
<td><strong>Total cluster transcoding rate</strong></td>
<td><strong>327.0 fps</strong></td>
<td><strong>327.0 fps</strong></td>
<td><strong>327.0 fps</strong></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>C1</td>
<td>27.0 fps</td>
<td>23.0 fps</td>
<td>25.0 fps</td>
<td>75.0 fps</td>
</tr>
<tr>
<td></td>
<td>C2</td>
<td>39.0 fps</td>
<td>36.0 fps</td>
<td>35.0 fps</td>
<td>110.0 fps</td>
</tr>
<tr>
<td></td>
<td>C3</td>
<td>46.0 fps</td>
<td>55.0 fps</td>
<td>45.0 fps</td>
<td>146.0 fps</td>
</tr>
<tr>
<td></td>
<td><strong>Total cluster transcoding rate</strong></td>
<td><strong>331.0 fps</strong></td>
<td><strong>331.0 fps</strong></td>
<td><strong>331.0 fps</strong></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>C1</td>
<td>25.0 fps</td>
<td>27.0 fps</td>
<td>20.0 fps</td>
<td>72.0 fps</td>
</tr>
<tr>
<td></td>
<td>C2</td>
<td>46.0 fps</td>
<td>37.0 fps</td>
<td>43.0 fps</td>
<td>126.0 fps</td>
</tr>
<tr>
<td></td>
<td>C3</td>
<td>67.0 fps</td>
<td>48.0 fps</td>
<td>61.0 fps</td>
<td>176.0 fps</td>
</tr>
<tr>
<td></td>
<td><strong>Total cluster transcoding rate</strong></td>
<td><strong>374.0 fps</strong></td>
<td><strong>374.0 fps</strong></td>
<td><strong>374.0 fps</strong></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>C1</td>
<td>21.0 fps</td>
<td>28.0 fps</td>
<td>28.0 fps</td>
<td>77.0 fps</td>
</tr>
<tr>
<td></td>
<td>C2</td>
<td>45.0 fps</td>
<td>40.0 fps</td>
<td>45.0 fps</td>
<td>130.0 fps</td>
</tr>
<tr>
<td></td>
<td>C3</td>
<td>37.0 fps</td>
<td>67.0 fps</td>
<td>70.0 fps</td>
<td>174.0 fps</td>
</tr>
<tr>
<td></td>
<td><strong>Total cluster transcoding rate</strong></td>
<td><strong>381.0 fps</strong></td>
<td><strong>381.0 fps</strong></td>
<td><strong>381.0 fps</strong></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 5.5:** Computer throughput of different data granularities set at resolution 640×480.
Figure 5.6: Cluster throughput of different data granularities set at resolution 1920×1080.

Figure 5.7: Cluster throughput of different data granularities set at resolution 1280×720.

Figure 5.8: Cluster throughput of different data granularities set at resolution 640×480.
5.3 Performance Evaluation of the Proposed Resource-Aware Task Allocation Scheme

A resource-aware grouping strategy is implemented to reduce the impacts of system heterogeneity to the proposed system. Three computers with their CPU specifications listed in Table 5.1 were used in the experiment. Tables 5.10 to 5.12 demonstrate the distribution of workload in Storm’s various stream grouping techniques. One can see that by using the proposed resource-aware task allocation scheme onto either fields grouping or the partial key grouping approach, the number of transcoded frames for each computer is assigned in different amounts. Computers with higher transcoding speed will receive more frames compared to the lower performance computers. This is done by assigning the number of frames emitted for each supervisor based on their computing ratios.
<table>
<thead>
<tr>
<th>Type of Groupings</th>
<th>Computer label</th>
<th>Video spout 1</th>
<th>Video spout 2</th>
<th>Video spout 3</th>
<th>Total transcoding time per computer (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Video frames emitted</td>
<td>Video frames emitted</td>
<td>Video frames emitted</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>video acquisition rate = 815 fps</td>
<td>video acquisition rate = 1568 fps</td>
<td>video acquisition rate = 1003 fps</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Transcoded frames</td>
<td>Transcoding time (secs)</td>
<td>Transcoded frames</td>
<td>Transcoding time (secs)</td>
</tr>
<tr>
<td>Fields Grouping</td>
<td>C1</td>
<td>1630</td>
<td>388.01 (4.2fps)</td>
<td>1559</td>
<td>362.56 (4.3fps)</td>
</tr>
<tr>
<td></td>
<td>C2</td>
<td>1763</td>
<td>220.38 (8fps)</td>
<td>1794</td>
<td>227.09 (7.9fps)</td>
</tr>
<tr>
<td></td>
<td>C3</td>
<td>1556</td>
<td>129.67 (12fps)</td>
<td>1595</td>
<td>132.92 (12fps)</td>
</tr>
<tr>
<td>Total Cluster</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>388.01 (71fps)</td>
</tr>
<tr>
<td>Partial Key Grouping</td>
<td>C1</td>
<td>1660</td>
<td>386.05 (4.3fps)</td>
<td>1655</td>
<td>384.88 (4.3fps)</td>
</tr>
<tr>
<td></td>
<td>C2</td>
<td>1640</td>
<td>179.59 (8.3fps)</td>
<td>1679</td>
<td>212.53 (7.9fps)</td>
</tr>
<tr>
<td></td>
<td>C3</td>
<td>1648</td>
<td>137.33 (12fps)</td>
<td>1615</td>
<td>134.58 (12fps)</td>
</tr>
<tr>
<td>Total Cluster</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>386.05 (71fps)</td>
</tr>
<tr>
<td>Resource-Aware Fields Grouping</td>
<td>C1</td>
<td>917</td>
<td>199.35 (4.6fps)</td>
<td>918</td>
<td>204.00 (4.5fps)</td>
</tr>
<tr>
<td></td>
<td>C2</td>
<td>1729</td>
<td>218.86 (7.9fps)</td>
<td>1739</td>
<td>207.02 (8.4fps)</td>
</tr>
<tr>
<td></td>
<td>C3</td>
<td>2303</td>
<td>191.92 (12fps)</td>
<td>2292</td>
<td>191.00 (12fps)</td>
</tr>
<tr>
<td>Total Cluster</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Resource-Aware Partial Key Grouping</td>
<td>C1</td>
<td>759</td>
<td>194.62 (3.9fps)</td>
<td>737</td>
<td>193.95 (3.8fps)</td>
</tr>
<tr>
<td></td>
<td>C2</td>
<td>1540</td>
<td>205.33 (7.5fps)</td>
<td>1534</td>
<td>196.67 (7.8fps)</td>
</tr>
<tr>
<td></td>
<td>C3</td>
<td>2649</td>
<td>203.77 (13fps)</td>
<td>2678</td>
<td>206.00 (13fps)</td>
</tr>
<tr>
<td>Total Cluster</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.10: The frame distribution of the proposed system with various groupings when output resolution is set to 1920×1080.
<table>
<thead>
<tr>
<th>Type of Groupings</th>
<th>Computer label</th>
<th>Video spout 1</th>
<th>Video spout 2</th>
<th>Video spout 3</th>
<th>Total transcoding time per computer (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fields Grouping</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C1</td>
<td>4949 frames emitted</td>
<td>4949 frames emitted</td>
<td>4430 frames emitted</td>
<td>4949 frames emitted</td>
<td>185.70 (26fps)</td>
</tr>
<tr>
<td>C2</td>
<td>1597 185.70 (8.6fps)</td>
<td>1589 174.62 (9.1fps)</td>
<td>1463 178.41 (8.2fps)</td>
<td>185.70 (26fps)</td>
<td></td>
</tr>
<tr>
<td>C3</td>
<td>1741 96.72 (18fps)</td>
<td>1797 99.83 (18fps)</td>
<td>1305 93.21 (14fps)</td>
<td>99.83 (50fps)</td>
<td></td>
</tr>
<tr>
<td>C3</td>
<td>1610 67.08 (24fps)</td>
<td>1585 72.05 (22fps)</td>
<td>1662 69.25 (24fps)</td>
<td>72.05 (70fps)</td>
<td></td>
</tr>
<tr>
<td>Partial Key Grouping</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C1</td>
<td>1664 184.89 (9fps)</td>
<td>1639 176.24 (9.3fps)</td>
<td>1458 182.25 (8fps)</td>
<td>184.89 (26.3fps)</td>
<td></td>
</tr>
<tr>
<td>C2</td>
<td>1656 97.41 (17fps)</td>
<td>1660 97.65 (17fps)</td>
<td>1483 98.87 (15fps)</td>
<td>98.87 (49fps)</td>
<td></td>
</tr>
<tr>
<td>C3</td>
<td>1628 65.12 (25fps)</td>
<td>1650 71.74 (23fps)</td>
<td>1488 67.64 (22fps)</td>
<td>71.74 (70fps)</td>
<td></td>
</tr>
<tr>
<td>Resource-Aware Fields</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Grouping</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C1</td>
<td>896 104.19 (8.6fps)</td>
<td>945 113.86 (8.3fps)</td>
<td>812 105.45 (7.7fps)</td>
<td>113.86 (24fps)</td>
<td></td>
</tr>
<tr>
<td>C2</td>
<td>1735 108.44 (16fps)</td>
<td>1743 108.94 (16fps)</td>
<td>1576 105.07 (15fps)</td>
<td>108.94 (47fps)</td>
<td></td>
</tr>
<tr>
<td>C3</td>
<td>2318 96.58 (24fps)</td>
<td>2310 96.25 (24fps)</td>
<td>2042 92.82 (22fps)</td>
<td>96.58 (70fps)</td>
<td></td>
</tr>
<tr>
<td>Resource-Aware Partial Key</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Grouping</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C1</td>
<td>894 113.16 (7.9fps)</td>
<td>890 114.10 (7.8fps)</td>
<td>825 115.48 (7.2fps)</td>
<td>114.58 (22.9fps)</td>
<td></td>
</tr>
<tr>
<td>C2</td>
<td>1724 101.41 (17fps)</td>
<td>1728 96.00 (18fps)</td>
<td>1544 102.93 (15fps)</td>
<td>102.93 (50fps)</td>
<td></td>
</tr>
<tr>
<td>C3</td>
<td>2329 97.04 (24fps)</td>
<td>2331 97.13 (24fps)</td>
<td>2060 93.64 (22fps)</td>
<td>97.13 (70fps)</td>
<td></td>
</tr>
<tr>
<td>Total Cluster transcoding time</td>
<td>184.89 (145.3fps)</td>
<td>184.89 (145.3fps)</td>
<td>184.89 (145.3fps)</td>
<td>184.89 (145.3fps)</td>
<td></td>
</tr>
<tr>
<td>Total Cluster transcoding time</td>
<td>113.86 (141.6fps)</td>
<td>113.86 (141.6fps)</td>
<td>113.86 (141.6fps)</td>
<td>113.86 (141.6fps)</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.11: The frame distribution of the proposed system with various groupings when output resolution is set to 1280 x 720.
<table>
<thead>
<tr>
<th>Type of Groupings</th>
<th>Computer label</th>
<th>Video spout 1</th>
<th>Video spout 2</th>
<th>Video spout 3</th>
<th>Total transcoding time per computer (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Transforming frames emitted</td>
<td>Transforming rate = 815 fps</td>
<td>Transforming frames emitted</td>
<td>Transforming rate = 1568 fps</td>
<td>Transforming frames emitted</td>
</tr>
<tr>
<td>Fields Grouping</td>
<td>C1</td>
<td>1608 (84.63 fps)</td>
<td>1577 (83.00 fps)</td>
<td>1466 (81.44 fps)</td>
<td>84.63 (56fps)</td>
</tr>
<tr>
<td></td>
<td>C2</td>
<td>1735 (43.38 fps)</td>
<td>1792 (48.43 fps)</td>
<td>1305 (42.10 fps)</td>
<td>48.43 (108fps)</td>
</tr>
<tr>
<td></td>
<td>C3</td>
<td>1606 (30.88 fps)</td>
<td>1580 (30.38 fps)</td>
<td>1658 (33.16 fps)</td>
<td>33.16 (154fps)</td>
</tr>
<tr>
<td></td>
<td>Total Cluster transcoding time</td>
<td>84.63 (318fps)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Partial Key Grouping</td>
<td>C1</td>
<td>1669 (79.48 fps)</td>
<td>1646 (82.30 fps)</td>
<td>1470 (81.67 fps)</td>
<td>82.13 (322fps)</td>
</tr>
<tr>
<td></td>
<td>C2</td>
<td>1635 (39.88 fps)</td>
<td>1679 (44.18 fps)</td>
<td>1492 (43.88 fps)</td>
<td>44.18 (113fps)</td>
</tr>
<tr>
<td></td>
<td>C3</td>
<td>1646 (32.27 fps)</td>
<td>1624 (32.48 fps)</td>
<td>1468 (29.96 fps)</td>
<td>32.48 (150fps)</td>
</tr>
<tr>
<td></td>
<td>Total Cluster transcoding time</td>
<td>82.13 (322fps)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Resource-Aware Fields Grouping</td>
<td>C1</td>
<td>881 (38.30 fps)</td>
<td>884 (34.00 fps)</td>
<td>867 (45.63 fps)</td>
<td>45.63 (68fps)</td>
</tr>
<tr>
<td></td>
<td>C2</td>
<td>1720 (40.95 fps)</td>
<td>1541 (37.59 fps)</td>
<td>1541 (40.92 fps)</td>
<td>40.95 (120fps)</td>
</tr>
<tr>
<td></td>
<td>C3</td>
<td>2348 (37.27 fps)</td>
<td>2524 (37.12 fps)</td>
<td>2049 (37.25 fps)</td>
<td>37.27 (186fps)</td>
</tr>
<tr>
<td></td>
<td>Total Cluster transcoding time</td>
<td>45.63 (374fps)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Resource-Aware Partial Key Grouping</td>
<td>C1</td>
<td>898 (42.76 fps)</td>
<td>901 (39.17 fps)</td>
<td>834 (43.89 fps)</td>
<td>43.89 (63fps)</td>
</tr>
<tr>
<td></td>
<td>C2</td>
<td>1710 (43.85 fps)</td>
<td>1746 (40.60 fps)</td>
<td>1521 (46.09 fps)</td>
<td>46.09 (115fps)</td>
</tr>
<tr>
<td></td>
<td>C3</td>
<td>2341 (45.02 fps)</td>
<td>2302 (45.14 fps)</td>
<td>2075 (42.35 fps)</td>
<td>45.14 (152fps)</td>
</tr>
<tr>
<td></td>
<td>Total Cluster transcoding time</td>
<td>46.09 (330fps)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.12: The frame distribution of the proposed system with various groupings when output resolution is set to 640 × 480.
5.4 Comparative Evaluation of Various Groupings to the Proposed System

The load distribution of each grouping is plotted in Figures 5.9 to 5.20 and Figures 5.21 to 5.32 are plotted to illustrate the impacts of various groupings on the transcoding time of the proposed system. In Storm’s default fields grouping and partial key grouping, the transcoding time of the entire system is bounded by the worst-performing computer $C1$ in the system when all GOP fragments are evenly distributed across heterogeneous computers in the Storm cluster. On the other hand, our proposed resource-aware task allocation scheme implemented onto the Storm’s stream grouping strategies has shown up to 44% performance improvement of the entire proposed system where input GOP fragments are distributed strictly corresponding to the nodes’s computing ratios in order to overcome performance degradation caused by system heterogeneity. On the contrary, it is worth noting that partial key grouping is still the best approach for distributing workload among computers with equal capabilities in homogeneous systems.
CHAPTER 5: EXPERIMENTAL RESULTS AND DISCUSSIONS

Figure 5.9: Load distribution of video frames among heterogeneous computers using storm default fields grouping at resolution 1920×1080.

Figure 5.10: Load distribution of video frames among heterogeneous computers using key fields grouping at resolution 1920×1080.

Figure 5.11: Load distribution of video frames among heterogeneous computers using the proposed resource-aware scheme on fields grouping at resolution 1920×1080.

Figure 5.12: Load distribution of video frames among heterogeneous computers using the proposed resource-aware scheme on key fields grouping at resolution 1920×1080.
CHAPTER 5: EXPERIMENTAL RESULTS AND DISCUSSIONS

**Figure 5.13:** Load distribution of video frames among heterogeneous computers using default fields grouping at resolution 1280×720.

![Graph 1](image1.png)

**Figure 5.14:** Load distribution of video frames among heterogeneous computers using key fields grouping at resolution 1280×720.

![Graph 2](image2.png)

**Figure 5.15:** Load distribution of video frames among heterogeneous computers using the proposed resource-aware scheme on fields grouping at resolution 1280×720.

![Graph 3](image3.png)

**Figure 5.16:** Load distribution of video frames among heterogeneous computers using the proposed resource-aware scheme on key fields grouping at resolution 1280×720.

![Graph 4](image4.png)
CHAPTER 5: EXPERIMENTAL RESULTS AND DISCUSSIONS

Figure 5.17: Load distribution of video frames among heterogeneous computers using storm default fields grouping at resolution $640 \times 480$.

Figure 5.18: Load distribution of video frames among heterogeneous computers using key fields grouping at resolution $640 \times 480$.

Figure 5.19: Load distribution of video frames among heterogeneous computers using the proposed resource-aware scheme on fields grouping at resolution $640 \times 480$.

Figure 5.20: Load distribution of video frames among heterogeneous computers using the proposed resource-aware scheme on key fields grouping at resolution $640 \times 480$. 
CHAPTER 5: EXPERIMENTAL RESULTS AND DISCUSSIONS

Figure 5.21: The time required to transcode the video in a heterogeneous environment using fields grouping at resolution 1920×1080.

Figure 5.22: The time required to transcode the video in a heterogeneous environment using partial key grouping at resolution 1920×1080.

Figure 5.23: The time required to transcode the video in a heterogeneous environment using the proposed resource-aware scheme on fields grouping at resolution 1920×1080.

Figure 5.24: The time required to transcode the video in a heterogeneous environment using the proposed resource-aware scheme on partial key grouping at resolution 1920×1080.
CHAPTER 5: EXPERIMENTAL RESULTS AND DISCUSSIONS

Figure 5.25: The time required to transcode the video in a heterogeneous environment using fields grouping at resolution $1280 \times 720$.

Figure 5.26: The time required to transcode the video in a heterogeneous environment using partial key grouping at resolution $1280 \times 720$.

Figure 5.27: The time required to transcode the video in a heterogeneous environment using the proposed resource-aware scheme on fields grouping at resolution $1280 \times 720$.

Figure 5.28: The time required to transcode the video in a heterogeneous environment using the proposed resource-aware scheme on partial key grouping at resolution $1280 \times 720$. 
Chapter 5: Experimental Results and Discussions

Figure 5.29: The time required to transcode the video in a heterogeneous environment using fields grouping at resolution $640 \times 480$.

Figure 5.30: The time required to transcode the video in a heterogeneous environment using partial key grouping at resolution $640 \times 480$.

Figure 5.31: The time required to transcode the video in a heterogeneous environment using the proposed resource-aware scheme on fields grouping at resolution $640 \times 480$.

Figure 5.32: The time required to transcode the video in a heterogeneous environment using the proposed resource-aware scheme on partial key grouping at resolution $640 \times 480$. 

80
5.5 Quality of Service

In video on demand services, the concept of quality of service is important. Quality of service is usually used to measure how well a user is satisfied with a given service through different ways and might have different meanings to different users of a given service. In this section, we try to explain and measure some of them. Users often demand the shortest possible response time. Fig. 5.33 illustrates the availability of the system measured in terms of response time over the output video frames. It shows how fast video frames are becoming available to the user in two different distributed transcoding approaches. One way of decreasing the latency of the batch distributed approach is to use smaller chunks. As the size of each chunk decreases, the amount of time required to process these chunks on each processor also decreases. However, too many small chunks can affect data locality, that is more data to be moved and more intermediate files are generated. Another way of measuring the QoS of compressed videos is the peak signal to noise ratio (PNSR) of the output video for various bit rate transcoding jobs. Note that all experiments reported in previous sections were conducted with the resolution transcode at variable bitrates (VBR) or constant quantization. On a scale of 1 to 5, the quality level of a transcoded video sequence can be rated based on Mean Opinion Score (MOS) model, where 5 represents the highest quality [113]. This mapping from PSNR to MOS is shown in Table 5.13. Fig. 5.34 shows the results obtained for bit rate reduction of the sample video used in previous experiments when output resolution is set to 1920×1080, 1280×720, and 640×480 respectively. Fig. 5.35 demonstrates the PSNR measurement per frame of the transcoded video in the proposed system in reference to the original input video frames. The graph has indicated that there is no missing or corrupted frame or even quality degradation during transcoding. The Average Peak Signal to Noise Ratio is most commonly used to measure the quality of a video and is calculated using the following equations:

\[
\text{PSNR} = 10 \times \log \frac{\text{MaxErr}^2 \times W \times H}{\sum_{i=1}^{W} \sum_{j=1}^{H} (x_{ij} - y_{ij})^2}
\]

(5.5.1)

The value of \(x_{ij}\) and \(y_{ij}\) shows pixel values of a input video frame and a compressed video frame. The \(W\) and \(H\) indicate the width and the height of the video frame. The unit for the PSNR is decibels (dB). A higher PSNR value simply indicates less distortion on the video frame.
Figure 5.33: The response time of the batch transcoding system compared to the proposed system.

<table>
<thead>
<tr>
<th>PSNR (dB)</th>
<th>MOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;37</td>
<td>5 (Excellent)</td>
</tr>
<tr>
<td>31-37</td>
<td>4 (Good)</td>
</tr>
<tr>
<td>25-31</td>
<td>3 (Fair)</td>
</tr>
<tr>
<td>20-25</td>
<td>2 (Poor)</td>
</tr>
<tr>
<td>&lt;20</td>
<td>1 (Bad)</td>
</tr>
</tbody>
</table>

Table 5.13: Mapping PSNR to MOS, taken from [106].
Figure 5.34: The signal to noise ratio of the proposed system when output resolution is set to 1920×1080, 1280×720 and 640×480 respectively.
Figure 5.35: The PSNR of a transcoded video sample produced from the proposed system, generated using the MSU Video Quality Measurement Tool.
Conclusion and Future Works

6.1 Conclusion

In this work, we have proposed a distributed real-time video transcoding system, operated on a heterogeneous computing platform. The proposed system is devised primarily with real-time purposes that would be potentially useful for transcoding of live streams at greater scale and speed which is demanding in today’s large-scale video surveillance system. Since input video streams are distributed to multiple computers and all computers are utilized under the proposed system, the CPU load of all computers can be reduced in real time. In the context of a large-scale video transcoding system, more input video streams can be processed in such system.

Experiments have shown the practicality of the proposed system implemented on Apache Storm. However, we found that the task allocation scheme used by Storm does not effectively distribute video frames among computers in heterogeneous systems according to their processing powers, resulting in bottleneck in the overall transcoding time. Subsequently, a resource-aware task allocation scheme is proposed and has overall reduced the amount of tasks allocated for the poor processing computer among heterogeneous computers. The proposed scheme has exhibited a good load balance of tasks based on computing ratios, reducing the overall transcoding completion time of the proposed system significantly.

Another key aspect when designing the proposed system is data granularity. A coarser grain approach with micro batches of 15 frames GOP length has shown maximum throughput and consistency achieved, having the same throughput as each computer benchmarked under their local data. This method suggests that the data locality is not
a concern in affecting the acquiring throughput of the proposed system when a coarser grain approach is used in data transmission. However, the performance of the proposed system can suffer from load imbalance in a homogeneous environment if the data is too coarse. This issue has been addressed by Nasir, Muhammad Anis Uddin, et al. [111] using the partial key grouping strategy.

6.2 Future Work

Large-scale video transcoding systems in the field of distributed computing have attracted increasing research interests in the last decade. As data intensive computing problems evolve, there are many research activities and a series of issues remain to be unsolved. Due to time and resource constraints, there are other interesting research challenges such as an online resource-aware task allocation scheme, an intelligent resource-aware scheduler and incorporation of hardware accelerators could be subjected to further studies.

6.2.1 Online Resource-Aware GOP Allocation

Since the transcoding rate may vary slightly during the transcoding process, an offline resource allocation scheme is not intelligent enough to dynamically allocate tasks on-the-fly based on changed computing ratios. Therefore, the proposed future work is to allow video frames to be dynamically assigned to computers according to their processing capabilities to further optimize the system performance.

6.2.2 A Resource-Aware Scheduler

Similar to many other stream processing systems, Storm has been in the absence of an intelligent scheduling mechanism to deal with heterogeneous systems. The current default round-robin scheduler deployed in Storm neglects resource demands and availability, and therefore can be inefficient in continuous run applications. R-Storm is then proposed in [114] to maximize resource utilization while minimizing the network latency and therefore increases the overall throughput of the system. In order to maximize the utilization of every resource while minimizing the network distance when scheduling tasks, the R-Storm scheduler efficiently schedules tasks closely together with their heterogeneous resource demands. This is true as scheduling every
spout task onto low performance but high memory machines will eventually increase the processing throughput for the rest of the computation systems since more resource availabilities are preserved. On the other hand, data locality is one of the widely adopted scheduler decision in parallel systems. For example, the MapReduce framework performs computation on a node where the required data is stored in order to avoid intensive data loading and removal within the cluster. However, data locality decisions from stream processing have different perspectives from batch processing as stream processing often computes on streams coming from remote sources such as IP cameras instead of static data that is stored locally or in the cluster. Hence, R-Storm has considered data locality to be an effective strategy to minimize communication costs by scheduling processor instances and data to be processed together on the same node or nodes in the same rack.

6.2.3 Incorporation of GPU’s Parallel Processing to The Proposed System

In addition, GPU implementation of video transcoding can be incorporated for better performance. We have experienced throughput degradation of the Nvidia Hardware Encoder (NVENC) when using fine-grain or intra frame transcoding in Storm. There is a number of challenges in making efficient GPU utilization in Storm. To achieve high throughput, stream applications have to make use of batching strategies. Transferring larger bulks of data (GOP size in our case) in between the cluster, GPUs and CPUs can improve the GOP acquiring throughput. The distribution of frames in batches also affects load balancing on GPUs if the data is too coarse. To achieve maximum utilization of the whole system, distribution of workload between CPUs and GPUs has to be considered.


[37] D Singer. 3gpp ts 26.244; transparent end-to-end packet switched streaming service (pss); 3gpp file format (3gp), etsi 3gpp std., rev. 12.3. 0, march 2014, 2014.


[42] NVIDIA. Nvidia video encoder application note.


[55] NVIDIA. nvenc, nvidia kepler hardware video encoder.


[85] Akhmedov Khumoyun, Myoungjin Kim, Yun Cui, Seungho Han, Seunghyun Seo, Seungbum Seo, and Hanku Lee. A study on multimedia transcoding approach by applying multiple mapreduce jobs. 2014.


[88] Rafael Pereira, Marcello Azambuja, Karin Breitman, and Markus Endler. An architecture for distributed high performance video processing in the cloud. In


