4-2016

Hardware accelerated redundancy elimination in network system

Kelu Diao
Purdue University

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Kelu Diao

Entitled
Hardware Accelerated Redundancy Elimination in Network System

For the degree of Master of Science

Is approved by the final examining committee:

Ioannis Papapanagiotou	
Co-chair	
Thomas J. Hacker	
Co-chair	
Baijian Yang	

To the best of my knowledge and as understood by the student in the Thesis/Dissertation Agreement, Publication Delay, and Certification Disclaimer (Graduate School Form 32), this thesis/dissertation adheres to the provisions of Purdue University's "Policy of Integrity in Research" and the use of copyright material.

Approved by Major Professor(s): Thomas J. Hacker

Approved by: Jeffrey L. Whitten 4/20/2016

Head of the Departmental Graduate Program Date

HARDWARE ACCELERATED REDUNDANCY ELIMINATION IN NETWORK SYSTEM

A Thesis

Submitted to the Faculty

of

Purdue University

by

Kelu Diao

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

May 2016

Purdue University

West Lafayette, Indiana

ACKNOWLEDGMENTS

I wish to gratefully acknowledge my committee members Dr. Ioannis Papapanagiotou, Dr. Thomas Hacker, and Dr. Baijian Yang for their help in my research. Dr. Ioannis Papapanagiotou guided me in this research, contributed a lot of ideas, and facilitated me with a powerful machine to do all the experiments. I would not have picked this topic and made it this far without him. Dr. Thomas Hacker taught me to be conscientious and responsible in research, guided me in this research with his insightful comments and suggestions. Dr. Baijian Yang mentored me in research and life, and affected me with his kindness and passion.

I am grateful to my friends Faheem, Jason, Saurav, Zhihao in High Performance Computing (HPC) lab. I had a great time working with them and developed friendship with them beyond the lab. Faheem and Saurav are like older brothers to me. I always asked them for advice when I had problems in research and life. Jason managed the lab in order and facilitated all of us when we needed him. I greatly benefited from his expertise in networking and Linux systems. Zhihao's thought provoking questions and insight into research helped me improve my work.

I would like to thank my teachers and mentors back in China, including Yong Yu, Minyi Guo, Yanmin Zhu, Jian-Guang Lou, Qingwei Lin, and all of the teachers who have taught me. I cannot explain how much they helped me with my study and career. I would especially like to thank Yong Yu, the dean of ACM class in Shanghai Jiao Tong University. He guided me into computer science and gave me the chance to work and study with the most talented people in the world. I would not have made computer science my career without him.

I would like to thank my mom, dad and my girlfriend Zhirui for their love and support. Their unconditional and unbounded love was the driving force behind this work. I would like to thanks my grandparents and great grandparents for their support, even though they have no idea what I am doing in a land thousands miles away from home. I am proud of them, and I know they are also proud of the first man who got a Master's degree in the family.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SYMBOLS

$\oplus$	the mathematical symbol for exclusive or
δ	size of sliding window
σ	constant used to compute the module number of hash value to determine fingerprints
μ	number of mappers
γ	number of reducers

ABBREVIATIONS

CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DDT	Distinct Data Rate
DRE	Data Redundancy Elimination
FSM	Finite State Machine
Gbps	billions of bits per second
GPU	Graphics Processing Unit
I/O	Input and Output
LRU	Least Recently Used
SHA	Secure Hash Algorithm

GLOSSARY

(application) object: Requests from or responds to an Internet application.

block: A group of threads in CUDA architecture, which could be treated like a matrix of threads.

buffer: A space to store data temporarily for the purpose of faster access.

CUDA stream: "A sequence of operations that execute in issue-order on the GPU" (Rennich, 2011, p. 3).

data redundancy: Duplicated chunks in a data stream.

Finite State Machine: A mathematical model that describes finite number of states and the behavior of transferring among these states.

kernel: "CUDA C extends C by allowing the programmer to define C functions, called kernels" (*CUDA Toolkit Documentation*, 2015, para. 2.1).

MapReduce: A programming model that assigns workers two roles, mapper and reducer, and make them run under a certain rule. It ensures the program to distribute the work load to multiple workers and let them work in parallel.

middle-box: "A computer networking device that manipulates traffic for purposes other than packet forwarding" (Morreale & Anderson, 2014, p. 129).

multi-threading: The situation where the program creates multiple threads and handles them at the same time.

(network) packet: Data formated under a rule that can be recognized by network devices.

pipeline: The situation where each worker (thread) handles one task, and passes its output as the next worker's input.

(Rabin) fingerprint: A method that transforms strings into polynomials in finite field, and modulo them by a large prime polynomial in the same field, to compute fingerprints (Rabin et al., 1981).

shared memory: The space allocated for each block, and shared by all the threads within the block (*CUDA Toolkit Documentation*, 2015).

ABSTRACT

Diao, Kelu M.S., Purdue University, May 2016. Hardware Accelerated Redundancy Elimination in Network System. Major Professor: Ioannis Papapanagiotou.

With the tremendous growth in the amount of information stored on remote locations and cloud systems, many service providers are seeking ways to reduce the amount of redundant information sent across networks by using data de-duplication techniques. Data de-duplication can reduce network traffic without the loss of information, and consequently increase available network bandwidth by reducing redundant traffic. However, due to the heavy computation required for detecting and reducing redundant data transmission, de-duplication itself can become a bottleneck in high capacity links. We completed two parts of work in this research study, Hardware Accelerated Redundancy Elimination in Network Systems (HARENS) and Distributed Redundancy Elimination System Simulation (DRESS). HARENS can significantly improve the performance of redundancy elimination algorithm in a network system by leveraging General Purpose Graphic Processing Unit (GPGPU) techniques as well as other big data optimizations such as the use of a hierarchical multi-threaded pipeline, single machine Map-Reduce, and memory efficiency techniques. Our results indicate that throughput can be increased by a factor of 9 times compared to a naive implementation of the data de-duplication algorithm, providing a net transmission increase of up to 3.0 Gigabits per second (Gbps). DRESS provides further acceleration to the redundancy elimination in network system by deploying HARENS as the server's side redundancy elimination module, and four cooperative distributed byte caches on the clients' side. A client's side distributed byte cache broadcast its cached chunks by sending hash values to other byte caches, so that they can keep a record of all the chunks in the cooperative distributed cache system. When duplications are detected, a client's side byte cache can fetch a chunk directly from either its own cache or peer byte caches rather than server's side redundancy elimination module. Our results indicate that bandwidth savings of the redundancy elimination system with cooperative distributed byte cache can be increased by 12% compared to the one without distributed byte cache, when transferring about 48 Gigabits of data.

CHAPTER 1. INTRODUCTION

This research was conducted to accelerate the redundancy elimination algorithm so that it can match up the speed of network. This section covers the background, scope, research question, assumptions, limitations, and delimitations of this study.

1.1 Background

Network performance is always a big issue for service providers and users. A recent study reveals that most of the network traffic on the commodity Internet is due to streaming videos (*Global Internet Phenomena Africa, Middle East & North America*, 2015). Many people have experienced and been annoyed by intermittent video streams, especially during network rush hours. Network performance is affected by factors such as available bandwidth, real-time network traffic, and the distance of data transfer. Numerous attempts have been made to improve network performance. For example, a network provider can replace copper with fiber-optic cable to improve bandwidth, apply better routing techniques to find optimal network paths, use middle-box data de-duplication techniques to reduce the impact of redundant data in the traffic stream, set up multiple data centers around the world to shorten the distance between users and data, and use P2P protocols to allow users to share data while downloading.

Eliminating redundancy within the network can both reduce the network traffic and decrease the response time in network-based applications. Generally, there are two methods of network data de-redundancy, proxy-caches (Glassman, 1994) and Data Redundancy Elimination (DRE) (Sijben, van Willigenburg, de Boer, & van der Gaast, 2002). *Proxy-cache* is a method in which a proxy server detects duplicate Internet requests for data, and returns objects to users directly if the requested data is in its cache. Another method, *DRE*, places middle-boxes on both the servers' and users' end. The user's middle-box simply retains data in its cache with LRU cache replacement. A server's middle-box detects the duplication in the data to be sent to the users and sends metadata indicating the location of the chunk cached in the user's middle-box. The user's middle-box then fetches this chunk directly from its cache. If the chunk is no longer in cache, the client requests that chunk from the server. We implemented a general de-duplication method that could be deployed within proxy caches or DRE middle-boxes. Although our approach would perform better using DRE because DRE middle-boxes are more powerful and flexible than proxy caches.

For this research, the redundancy elimination procedure we deployed in servers' middle-boxes employed the following steps:

1. fetch data, which could be packets or objects, from a network port,

2. divide the received packets or objects into smaller chunks,

3. compute a hash value for each chunk, and

4. use the hash values to match the chunks.

In practice, we found that the data fetching step is fast enough, and does not require acceleration, so this step can be safely skipped. There are two approaches for the object/packet chunking step, the "classical" sliding window approach and SampleBytes, a window sampling approach proposed by Aggarwal et al. (2010). In the sliding window approach, the sliding window is put at the beginning of the data stream and slid to the end of the stream one byte a time. For each step, the sliding window algorithm computes the Rabin hash value of the window, applies a certain sampling function (e.g. MODP, MAXP) on the hash values to choose a subset of the windows. The first bytes of the windows are denoted as the fingerprints, which divide the objects/packets into chunks. Another approach is SampleBytes, which picks up some sample windows and computes their Rabin hash values, and then applies a sampling function to choose fingerprints from the sample windows. We use the sliding window approach in our work because the SampleBytes approach has the potential to lose redundancy detection opportunities. In our work, the length of the sliding window is 32 bytes, and sampling function is MODP. For the chunk hashing step, a general redundancy elimination algorithm computes the hash values of each chunk. For the chunk matching step, the hash values computed in the previous step are used to detect redundant chunks, and replace the redundant chunks with metadata that indicates the location of these chunks in the cache.

An existing redundancy elimination algorithm, the Rabin fingerprint, is a slow technique in terms of network throughput, due to the time required for computing the hash values and the memory management of this algorithm. This drawback has made a Rabin fingerprint based redundancy elimination algorithm impractical for high-speed networks. Many researchers have been studying this problem and have proposed many solutions. Aggarwal et al. (2010) applies sampling techniques which only analyzes a subset of all the windows to avoid the huge amount of computing required to analyze all of the data. Bhatotia, Rodrigues, and Verma (2012) uses GPU acceleration with a multi-threaded pipeline to improve disk de-duplication performance. The fingerprint sampling technique introduced by Aggarwal et al. (2010) may miss many duplicate bytes due to the sampling technique it uses, which may affect the overall redundancy elimination. Shredder has demonstrated good performance improvements for content chunking, but is still not fast enough for fast networks (up to 10 Gbps), hence it is only applied for disk de-duplication.

Furthermore, the resources in network systems are not used optimally due to the lack of information. For example, when a client request for a web content, the client's side middle-ware would request server for this content if it is not cached. It would be faster if the middle-ware could request a device, which is closer to this middle-ware than the server and has the content in cache, for the data. A solution for this problem is to use cooperative caches, which can share information about cache contents to each other. Pitkänen and Ott (2007) and J. M. Wang, Zhang, and Bensaou (2013) have applied cooperative distributed byte caches in the network data de-duplication problem, but both of their works are limited in certain network. Our purpose in this work is to find a general method that is suitable for the common network structures.

1.2 Statement of Problem

With the tremendous growth in the amount of data produced and stored on the Internet, many service providers are seeking ways of data de-duplication. Data de-duplication can remove the replicative patterns from network traffic, and decrease response time for time sensitive applications (Papapanagiotou, Callaway, & Devetsikiotis, 2012). However, the redundancy elimination algorithm is a slow technique compared to network throughput, which makes the approach very limited in terms of bandwidth saving in network traffic. It is an open problem to accelerate the redundancy elimination algorithm so that its throughput can match up to the network throughput.

The redundancy elimination algorithm typically consists of the following steps: fetching data, partitioning packets/objects chunks, computing hash values for chunks, and detecting and removing duplicate chunks using hash values. The problem is how to accelerate the other three steps so that their throughput can match up to the throughput of the first step. Bhatotia et al. (2012) has proposed an optimization of data chunking step. But the large amount of memory allocation, release, and access in chunk hashing and chunk matching steps makes these two step extremely time consuming, and hence slows down the whole process.

Besides, two routers that are located nearby each other in the network would need to fetch full package of data respectively, even if they requested for the same data. Network redundancy elimination is based on the logic that Internet users in a region, like university campus, tend to visit similar web contents. Likewise, Internet users in another university campus nearby would also tend to visit similar web contents to the previously mentioned university campus. It would save a lot of bandwidth and make respond faster if the routers can fetch data from each other's cache.

1.3 Research Question

How can the redundancy elimination algorithm be accelerated so that its throughput speed can match up to the network throughput speed?

- For each step of redundancy elimination algorithm, which optimization technique would be the best match?
- Can we make better use of the devices, such as CPU and GPU, so that each device doesn't need to wait until the other devices have done their work before it begins a new task?
- Can this program be deployed in multiple middle-boxes, which can synchronize frequent patterns with each other?

1.4 Significance

Network redundancy is a problem that has troubled service providers as well as clients for years, as the duplicated data occupies a considerably large proportion of the network traffic and reduces the Distinct Data Throughput (DDT) of the network. With a redundancy elimination module in the network device, one can not only increase the DDT, but also reduce the response time for Internet applications.

Many methods were introduced for network data de-duplication. Anand, Muthukrishnan, Akella, and Ramjee (2009) used an object cache to track the similar contents within the same object, but could not detect duplication chunks across objects. Douglis and Iyengar (2003), and Mogul, Douglis, Feldmann, and Krishnamurthy (1997) introduced delta-encoding, which can compare chunks across objects, but it is application-specific and has problems dealing with dynamic contents. Spring and Wetherall (2000) applied a middle-box approach to IP-layer devices, but it is a slow technique. Aggarwal et al. (2010) introduced a SampleByte technique, which can achieve a throughput of 2.2-5.8 Gbps with sampling period of 32-512 bytes, but it will miss some fingerprints in certain cases.

This research study aimed to find a way to accelerate the redundancy elimination technique in network systems without affecting the duplication detection rate. The researcher divided the redundancy elimination algorithm into a few steps, and apply a few optimization techniques on both the hardware and software levels to accelerate each step. The researcher used a multi-threaded pipeline to overlap the execution time of each step to achieve the best performance. The researcher also deployed distributed byte caches that can cooperate with each other to further reduce the redundancy in network.

1.5 Scope

The work conducted by this research study included the following parts:

1. Apply a multi-threaded pipeline architecture for the redundancy elimination algorithm steps to overlap their execution time.

2. Use Compute Unified Device Architecture (CUDA) to accelerate the packet/object chunking step.

3. Use the optimization techniques of CUDA to take the best advantage of CUDA architecture.

4. Find a way to accelerate the chunk hashing and matching steps. The chunk hashing step is very computationally intensive and the chunk matching step is also time consuming due to intensive memory allocation, release, and access.

5. Compare the performance of algorithms with and without the multi-threaded pipeline, and algorithms with and without CUDA acceleration.

6. Use distributed byte caches, which can synchronize with each other, to allow them fetch redundant chunks from other cache rather than fetch from server.

7. Implement a simulation system to compare the performance of the data redundancy elimination system with and without synchronized distributed byte cache.

Hash functions used in the chunk hashing step should be carefully chosen, because this step is very computationally intensive, and it would severely slow down the whole process with an improper hash function.

1.6 Assumptions

The assumptions for this study included:

- The middle-boxes upon which we deployed our algorithm supports multi-threaded programs, have installed NVIDIA GPU(s) that support CUDA 6.5, and have a 64-bit operating system.

- The middle-boxes have memory no smaller than 16 GB so that we can have enough space for the algorithm to run properly, and store the chunks, meta-data, and intermediate data in it.

- The operating system running on the middle-boxes supports C++ version 11, which is the earliest C++ version which has implemented standard multi-threading library (*Working Draft, Standard for Programming Language C++*, 2012, p. 1114).

- The middle-boxes have or are at least compatible with GCC 4.7 or later, because GCC versions earlier than GCC 4.7 would not support the C++11 features used in the program (*C++0x/C++11 Support in GCC*, 2015).

- The middle-boxes have installed Nvidia GPU(s) which can support CUDA architecture 6.5.
- The server is a video server, and users request for video by string IDs.
- The nodes and links in the network are stable and never failed or corrupted.

1.7 Limitations

The limitations for this study included:

- This research only covers redundancy elimination algorithms that use a content-based chunking method.
- This research uses the Rabin Fingerprint method as the fingerprinting method.
- This research uses CUDA architecture to accelerate the fingerprinting process.
- This research uses C++11 multi-threading libraries to implement multi-threaded programs.
- This research uses ns-3 to simulate network behaviors of the network redundancy elimination system with cooperative distributed byte caches, which consists of one server's side middle-box (HARENS) and four clients' side middle-boxes (distributed byte cache).

1.8 Delimitations

The delimitations for this study included:

- This research will not cover compression algorithms.
- This research will not cover assembly optimization or hardware architecture optimization.

- This research will not cover the implementation of a real-world network traffic redundancy elimination system.

1.9 Summary

This chapter provided the background, statement of problem, research question, significance, scope, definitions, assumptions, limitations, and delimitations for the research project. The next chapter provides a review of the literature relevant to this research.

CHAPTER 2. REVIEW OF RELEVANT LITERATURE

This chapter highlights the evolution of methods used for data de-duplication in network systems.

2.1 Identical Object Detection

Data redundancy in the network is one of the major problems that researchers aim to tackle. A trivial solution is to use proxy caches to detect and eliminate the identical objects/packets. Glassman (1994) is one of the first who studied the network data redundancy problem. After that, researchers kept looking into this problem and found more optimization techniques and applications of Glassman (1994). Abrams, Standridge, Abdulla, Williams, and Fox (1995), Barford, Bestavros, Bradley, and Crovella (1999), Markatos (1996), and Williams, Abrams, Standridge, Abdulla, and Fox (1996) studied the cache replacement policies. Abrams et al. (1995) and Markatos (1996) explored other replacement policies, which are derived from Least Recent Used (LRU) policy, to achieve better performance in general or special cases. Barford et al. (1999) and Williams et al. (1996) explored other replacement policies based on the file size. Bolot and Hoschka (1996) applied an Auto-Regressive Integrated Moving Average (ARIMA) model to track the pattern of occurrence of redundancy. Bolot, Lamblot, and Simonian (1997) applied the proxy caching method to group Internet requests by sites or web servers and send the requests in the same group at the same time. Meira Jr, Fonseca, Murta, and Almeida (1998) presented an approach that can examine the performance of proxy caches. Papapanagiotou, Nahum, and Pappas (2012) summarized the usage and benefits of browser cache and proposed a modification on cache replacement strategy to achieve the optimal bandwidth savings.

The early research studies introduced in this section are highly limited, because they can only detect identical objects/packets and are protocol-specific. However, their work is still valuable because proxy caching is a fast technique compared to the redundancy elimination algorithms of today. For example, Papapanagiotou, Callaway, and Devetsikiotis (2012) proposed a hybrid method, which integrated proxy caching and a redundancy elimination module to take advantage of both techniques.

2.2 Delta Encoding

Delta encoding is a method that compares a series of data to another series of data, and transforms it to the difference of these two series of data. Researchers applied this method to detect and compress similar objects/packets in network systems. In the delta encoding method, the analysis program computes the delta encoding of an object/packet based on the original one, and the compressed delta encoded data would be sent rather than the full data. Jacobson (1990) applied this technique to compute the delta encoding of the TCP/IP header to be sent based on the previous header and compress it, which is based on the assumption that both senders and receivers keep the previous header. It took researchers many years to apply the delta encoding technique to HTTP traffic analysis. Williams et al. (1996) first proposed delta encoding in their work addressing HTTP cache replacement strategies, but they did not work on this idea until later (Williams (1996) according to Mogul et al. (1997)). Housel and Lindquist (1996) is believed to be the first publication that investigated the application of the delta encoding technique to HTTP traffic. Mogul et al. (1997) and Douglis and Iyengar (2003) proposed their refined delta encoding method in a protocol-specific situation.

Delta encoding is a big improvement over the trivial proxy cache technique. But it only has significant benefit when dealing with very similar data, which makes it highly limited in the era of information explosion.

2.3 Duplicate Chunk Detection

A better idea is to reuse the duplicated chunks in the data series, that the data de-duplication algorithm can still detect duplications even if there is only little similarity between two objects/packets.

Previous studies have proposed two approaches both related to the fingerprinting method. Fingerprinting is a way to use a small signature to represent a larger object. The researcher of this thesis first categorized them into Fingerprint Expansion (FPE) and Fingerprint Partition (FPP). The FPE method is applied in the work of Manber (1994), Spring and Wetherall (2000), and Schleimer, Wilkerson, and Aiken (2003). It treats the fingerprints as anchors in the data series, matches the anchors between two objects/packets, and expands the anchors to find the maximum chunk match when two anchors match. The FPP method is applied in the work of Muthitacharoen, Chen, and Mazières (2001), Rhea, Liang, and Brewer (2003), Tolia, Kaminsky, Andersen, and Patil (2006), Pucha, Andersen, and Kaminsky (2007), Anand et al. (2009), Aggarwal et al. (2010), Bhatotia et al. (2012), and Papapanagiotou, Callaway, and Devetsikiotis (2012). It treats the fingerprints as break points within the data series, divides the data series into chunks from these break points, and matches the chunks within and across objects/packets based on the non-collision hash value (normally SHA) of each chunk. The FPE method would find a longer match than FPP once it finds an anchor match, but it is a slow technique because the program needs to compare every byte before and after the two matching anchors until it reaches different bytes. The researcher of this thesis applied the FPP method, because it is much faster than FPE, and speed is the main concern in this work.

There are also different methods for picking fingerprints. A simple implementation would be to choose a fingerprint every several bytes. However, a slight modification would make a significant change in the fingerprint set. For example, if there is a byte inserted in the data series, the fingerprints after that insertion would all be shifted left by one byte. This problem could be solved by content-based chunking, which is categorized to MODP (MOD p) and MAXP (local MAX in p bytes) by Anand et al. (2009). Besides MODP and MAXP, Aggarwal et al. (2010) proposed SampleByte in the work EndRE. MODP is a method that chooses a subset of the fingerprints that every fingerprint in the subset modulo p equals a certain number. p is a pre-determined number. Manber (1994) applied this technique in his work related to finding similar files in a large file system. Spring and Wetherall (2000) first proposed applying this technique in redundancy elimination in network systems. Bhatotia et al. (2012), Muthitacharoen et al. (2001), Papapanagiotou, Callaway, and Devetsikiotis (2012), Pucha et al. (2007), Rhea et al. (2003), and Tolia et al. (2006) also adopted MODP method or used the tools that adopted MODP method in their work. The MAXP method is choosing a local maximum (or minimum) in every continuous block with length p. Anand et al. (2009), and Schleimer et al. (2003) adopted MAXP method in their work. The SampleByte method proposed by Aggarwal et al. (2010) adopted the ideas from both MODP and MAXP. It skips $p/2$ bytes every time when a fingerprint is chosen. Because MODP uses a pre-determined value p to filter fingerprints, and the network data could be clustered, it could miss some redundancy detection opportunity. The SampleByte method would lose even more redundancy detection opportunity than MODP. However, this work adopted the MODP method, because it has approximately the same detected redundancy rate as MAXP, according to the experiment conducted by Aggarwal et al. (2010), and partitions data into fewer chunks, which would make the critical steps faster.

2.4 Content-based Chunking

The content-based chunking algorithm computes fingerprints for the set of selected substrings in the data series, and applies a certain rule to filter these fingerprints. The data series is partitioned by the first bytes of the remaining substrings. The Rabin fingerprint method proposed by Rabin et al. (1981) is the most famous and commonly used fingerprinting technique. The fingerprints of different objects are expected to be different, which means there is only very small probability that two different objects have the same fingerprint. Mathematically, denoting k as the number of bits in a fingerprint, Rabin's hash method can be represented by Equation 2.1

$$\mathcal{F} = \{f : O \to \{0,1\}^k\} \tag{Eqn. 2.1}$$

In equation 2.1, symbol O represents all possible objects inputted for fingerprinting. It is expected that for any real-life set of objects S, the function satisfies Equation 2.2, which means the size of the set of fingerprinting results should be approximately equal to the size of the set of input values.

$$\frac{\big||f(S)| - |S|\big|}{|S|} \approx 0 \tag{Eqn. 2.2}$$

The fundamental idea proposed in the work of Rabin et al. (1981) represent a string as a series of bits, which could be interpreted as a large polynomial over a Galois field of order 2 (denoted as $\mathcal{GF}(2)$), and to compute the signature of the string as this string modulo a polynomial, which is of degree $k-1$ and irreducible over $\mathcal{GF}(2)$. In Rabin's method, a string $\equiv$ a series of bits $\equiv$ a polynomial over field $\mathcal{GF}(2)$. The irreducible polynomial could be easily generated by the algorithm proposed by Gao and Panario (1997). In this research, this irreducible polynomial is dynamically generated to protect information security.

A brute force implementation of Rabin hash is very expensive. The complexity of the modulo operation used for computing the fingerprints that uses a Euclidean algorithm would be $O((8n-k+1)k) = O(nk)$, where n is the length of string in bytes, and k is the term number of irreducible polynomial. Broder (1993) introduced an optimized form of the Rabin hash algorithm. This algorithm pre-computes the results of modulo operation for all possible inputs and stores them in a table, from which the algorithm can simply refer to in the fingerprint computation. Notice that the input could be very large, which would require a huge table to store all results for all possible inputs. The clever part of the design in Broder (1993) is that they only computed the results of all possible eight-bit inputs shifted left by $k + 24$, $k + 16$, $k + 8$, and k bits, and stored them into four tables named TA, TB, TC, and TD, with 256 data entries each. The data structure of the tables is array, in which the index could perform as key of table. Hence, the result of possible 32-bit inputs shifted left by k bits could be computed by concatenating one entry from each of the four small tables. The modulo operation is computed iteratively, as shown in Algorithm 1. The input of this algorithm is a string,

Algorithm 1 Optimized Algorithm by Broder (1993)

```
$result \leftarrow 0$
while $S(x)$ is not empty do
    $A(x) \leftarrow$ first 32 bits in $S(x)$
    $r_0, r_1, r_2, r_3 \leftarrow result_0$
    $result \leftarrow (result << 32) + TA[r_0] + TB[r_1] + TC[r_2] + TD[r_3] + A(x)$
end while
```

denoted as $S(x)$. The variable $result_0$ is the first 32 bits of the value $result$, and $P(x)$ denotes the irreducible polynomial in the following discussion. The algorithm reads 32 bits of $S(x)$ each time, concatenates the variable $result$ with the newly read bits, and compute the result of $result$ modulo $P(x)$ as the new $result$. The modulo operation is done by decomposing the first 32 bits of result into four eight-bit numbers and referring to the tables for the results. Notice that this algorithm actually computed the result of $(S_0(x) << 64) \mod P(x) + S_1(x)$, where $S_1(x)$ is the last 64 bits of $S(x)$, and $S_0(x)$ is the remaining part. However, $S(x) \mod P(x) = \big((S_0(x) << 64) + S_1(x)\big) \mod P(x)$, so that the result of this algorithm is not exactly the same as the result of $S(x) \mod P(x)$. But the distribution of the results of optimized algorithm is as sparse and even as Rabin hash, which means it can also satisfy Equation 2.2. Therefore, the researcher adopted this optimized Rabin fingerprint algorithm in this research.

2.5 Efforts of Acceleration

Several previous studies have contributed to accelerating the redundancy elimination algorithm to allow it to match the speed of the network. Anand et al. (2009) and Schleimer et al. (2003) applied MAXP, which is faster than MODP. Anand et al. (2009) applied a bloom filter to accelerate chunk matching, but the bloom filter has a relatively high false positive probability, does not have the information of chunk location, and has difficulty in the case of deletion due to lacking of information of chunk location. Aggarwal et al. (2010) proposed the SampleByte method, which only pick some sample bytes to compute Rabin fingerprints, so that it loses redundancy detection opportunities. Bhatotia et al. (2012) used a GPU to accelerate the Rabin fingerprint process, but their technique is mainly targeted on redundancy elimination in incremental storage, and their technique was not addressed in accelerating chunk hashing and matching procedures, which are critical steps in network redundancy elimination. Papapanagiotou, Callaway, and Devetsikiotis (2012) adopted some of the work of Aggarwal et al. (2010) and proposed a hybrid method using both a proxy cache and a redundancy elimination module. The work of Papapanagiotou, Callaway, and Devetsikiotis (2012) was later adopted in the work of Callaway and Papapanagiotou (2013a), Callaway and Papapanagiotou (2013b), and Callaway and Papapanagiotou (2014). But their work was not addressed in accelerating the algorithm in redundancy elimination module. In this thesis, we mainly focused on accelerating the whole process of redundancy elimination, which is a complementary of the work made by Papapanagiotou, Callaway, and Devetsikiotis (2012).

Dal Bianco, Galante, and Heuser (2011), and Kolb, Thor, and Rahm (2012) proposed a data de-redundancy approaching using map-reduce. But their work applied brute force approach in objects chunking and duplicate chunk matching, which is inefficient, and used existing map-reduce tools, which may not perfectly suit for the redundancy elimination task. Our work used the Rabin fingerprinting algorithm for objects/packets chunking, which is much faster. We also implemented a single-machine map-reduce. It eliminated the key-value pair process, because the value itself is the key. It also eliminated the merge/sort step, and passes the output of map phase to reducers immediately instead of passing a list after mapper is done. This real-time map-reduce communication approach is the key of acceleration.

2.6 Distributed Cache

Distributed cache is a solution for addressing the limitations posed by the limited computational power and storage of the middle-boxes. It can also distribute network traffic to multiple paths. Gadde, Rabinovich, and Chase (1997) and Povey and Harrison (1997) were among the first to propose using distributed Internet caches to reuse data and reduce the redundant network transfer. Gadde et al. (1997) proposed a Caching and Replication for Internet Service Performance (CRISP) method, which allows multiple caches share a directory of the cache contents to distributed traffic load to the caches. Povey and Harrison (1997) proposed a method that can replicate data among a hierarchical cache, which reduces the network traffic load in successive levels in the Internet. Chuang and Sirbu (2000), and Tay, Feng, and Wijeysundera (2000) also used distributed caches for servers and proposed strategies to ensure data consistency. Borst, Gupt, and Walid (2010) developed a cache management algorithm to maximize cache usage and rate of non-redundant data in the network traffic. Iyer, Rowstron, and Druschel (2002), C. Wang, Xiao, Liu, and Zheng (2004), and Huang, Sun, Chen, Mao, and Zhang (2012) showed the advantage of the distributed Internet cache method over the traditional proxy cache in P2P applications. Among these three works, Iyer et al. (2002) proposed Squirrel, a decentralized web cache that enables the nodes in the peer-to-peer network to share their local caches. We also used a decentralized method for the clients' side middle-boxes in this thesis as it's more efficient and fault tolerant. C. Wang et al. (2004) proposed a Distributed Caching and Adaptive Search (DiCAS) protocol, which uses a multilayer P2P network and restrict traffic within one layer based on the group ID of the node that sends requests. Huang et al. (2012) proposed a BufferBand method with distributed caches and a three-layer hierarchical mapping, which can decrease the delay compared to traditional P2P networks.

Although both redundancy elimination in network system and distributed cache are well researched, there is not much research about using distributed cache in redundancy elimination. Pitkänen and Ott (2007) proposed an opportunistic model in a delay tolerant network, which allows nodes in the network to communicate end-to-end when a link to server cannot be set up in a given time. However, most Internet applications are not delay tolerant, which makes this work very limited when applied to real cases. J. M. Wang et al. (2013) proposed a cache cooperation scheme to distribute traffic load among caches and reduce redundancies. However, this work is also very limited in real cases, because it is only designed for Content-Centric Networking (CCN) architectures. In this thesis, I used synchronized distributed caches in client's side middle-boxes and a redundancy elimination module in a server's side middle-box, which can be deployed in a normal network. Distributed caches on the clients' side rather than on the server's side can also reduce the network latency because the clients' side middle-boxes are geographically close to each other.

2.7 Summary

This chapter provided a review of the literature relevant to redundancy elimination in network systems. The next chapter provides the framework and methodology to be used in the research project.

CHAPTER 3. FRAMEWORK AND METHODOLOGY OF HARENS

This chapter provides the framework and methodology that were used in the research study of Hardware Accelerated Redundancy Elimination in Network Systems (HARENS).

3.1 Research Type

This research study was a mix of qualitative and quantitative research. our goal was finding a way to accelerate the redundancy elimination algorithm, and to estimate the improvement of throughput speed and the effect on detected redundancy rate by each acceleration technique.

3.2 Hypothesis

H_0: HARENS does not improve the performance (throughput speed and redundancy detection) of a redundancy elimination algorithm in network systems.

H_a: HARENS improves the performance (throughput speed or redundancy detection) of a redundancy elimination algorithm in network systems.

3.3 Methodology

In this section, we introduce an overview of the algorithm applied to solve the data duplication problem. Then the optimization techniques applied to accelerate the algorithm are introduced.

3.3.1 Algorithm Overview

The common procedure for a redundancy elimination algorithm consists of four steps: fetching data, partitioning packets/objects chunks, computing hash values for chunks, and matching chunks by comparing hash values. We found that the data fetching step is fast enough, and does not require acceleration, so this step can be safely skipped. The other three steps will be referred as chunk partitioning, chunk hashing, and chunk matching in the rest parts of this chapter.

3.3.1.1. Packet/Object Chunking

In this step, the program reads in the traffic stream either on-line or off-line, applies a sliding window to scan through the whole input stream, and marks the fingerprints based on given rule. The fingerprints divide the stream into chunks.

Algorithm 2 Packet/Object Chunking

```
window ← 0
fingerprints ← empty set
while window + $\delta$ < stream length do
  hashValue = ComputeRabinHash(stream[window], $\delta$)
  if hashValue  mod $\sigma = 0$ then
    fingerprints.Add(window)
  end if
  window ← window + 1
end while
```

The Rabin fingerprint algorithm (Rabin et al., 1981) is used as the fingerprint sampling algorithm in this step. An overview of the procedure of the Rabin fingerprint algorithm is briefly shown in Algorithm 2. During initialization, *window* is put at the beginning of the stream and *fingerprints* is an empty set. Then this *window* is slide to the end of the stream, moving right by one byte each step. In each step, the algorithm also computes the Rabin hash value of the stream that covered by the *window* (σ is size of the *window*). If the hash value module by

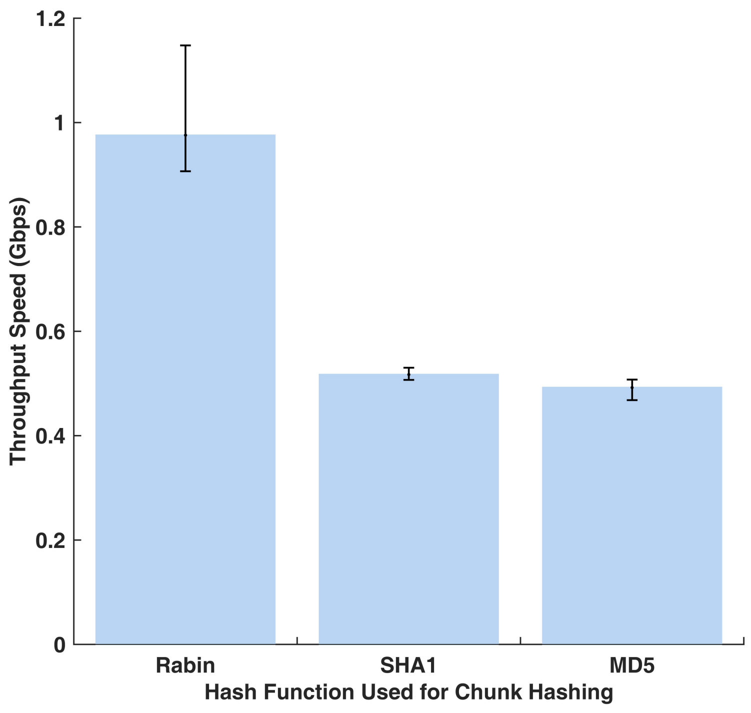

Figure 3.1. Chunk hashing throughput speed using Rabin, SHA1, and MD5 hash (single thread)

a given number δ equals to 0, the algorithm mark the beginning of this *window* as a $fingerprint$ and add it into the set.

3.3.1.2. Chunk Hashing

Having partitioned the stream into chunks, a hash value is computed for each chunk. The Rabin, SHA1, and MD5 hash are all good choices for chunk hashing. The hash function used for chunk hashing is expected to be effective (satisfies Equation 2.2) and efficient. So that we tested the throughput (shown in Figure 3.1) and hash collision rate (Rabin hash has 0.1% – 0.5% collision rate, while no hash collision observed using SHA1 and MD5) of each method affected by these two hash functions against the YouTube video accessing data we generated from the trace file of UMass Trace Repository *UMass Trace Repository* (2014) The experiment was CPU based, conducted under the environment described in Section 3.4. According to these results, Rabin hash has much better performance and SHA1 hash has slightly better performance than a MD5 hash, but Rabin hash could occasionally encounter hash conflicts, while SHA1 and MD5 would not have this problem. In the method proposed in this work, a duplication is claimed when a duplicated hash value is detected, hence hash conflicts are intolerable. Therefore, we choose SHA1 as the chunk hashing method.

3.3.1.3. Chunk Matching

In this step, the algorithm stores the hash values computed on the previous step in a hash table. It reports a duplicate chunk when it finds that the hash value already exists in the hash table. Hash conflicts would not be a problem because the hash conflict rate of SHA1 is small enough to be safely ignored Spring and Wetherall (2000).

A replacement algorithm for the chunks is required, so that only a limited number of chunks are kept in memory. We chose LRU as the chunk replacement algorithm because the repetitive patterns are also grouped in time period (Willinger, Taqqu, Sherman, & Wilson, 1997), which means the redundant chunks are most likely to be redundant to recent data.

3.3.2 CUDA Acceleration

In the packets/objects chunk partitioning step, we need to compute the Rabin hash for each window, which is (stream length - $\sigma + 1$) windows, which makes this step very computationally intensive. But it also has a great feature in that the Rabin hash takes exactly σ bytes as input, and will follow exactly the same mathematical procedure, which is independent to each other and takes about the same amount of time. The features of the MODP Rabin fingerprinting algorithm in this step makes it a perfect candidate of CUDA acceleration. Besides, we also applied CUDA optimization techniques to make the best advantage of CUDA.

Shared memory access is much faster than global memory access in the CUDA architecture, which makes it beneficial to transfer data to shared memory before computing. Besides, there are multiple threads reading the same memory location because of the overlap of windows, which causes access conflicts and hence only one of the threads that access the same memory slot would run while the others are waiting. To solve this problem, we made two copies of some input data in shared memory and aligned the data to avoid half of the access conflict as well as improve the memory bandwidth. It is impossible to make more copies of data in shared memory because of the limited size of shared memory.

The CUDA kernel instructions could be stalled by many reasons. We ran an un-optimized CUDA algorithm in Visual Studio and generated Figure 3.2, which shows the potential reasons that could slow down CUDA kernel execution. As shown in Figure 3.2, the issues that could affect the performance includes: the speed of instruction fetch and constant miss, which requires well arranged code order to make the full advantage of cache; execution dependency and memory dependency, which requires avoidance or reduction of the scenario that the threads in GPU depends on one another; and memory throttle and pipe busy, which requires avoidance or reduction of the scenario that multiple threads access the same slot of data.

Moreover, in CUDA compute capability 3.5, the maximum threads per multiprocessor is 2048; the maximum shared memory per multiprocessor is 49152 bytes; the maximum register file size per multiprocessor is 65536 bytes (*CUDA Toolkit Documentation*, 2015). To achieve the most threads per multiprocessor, one need to balance the size of shared memory and register allocated per multiprocessor. In this method, we transfer data from global memory in GPU to two shared memory slots in each block, which stores two shifted copies of input data, and used CUDA kernel registers for the other memory usage. The GPU threads are synchronized after transferring data into shared memory, so that a large part of stall

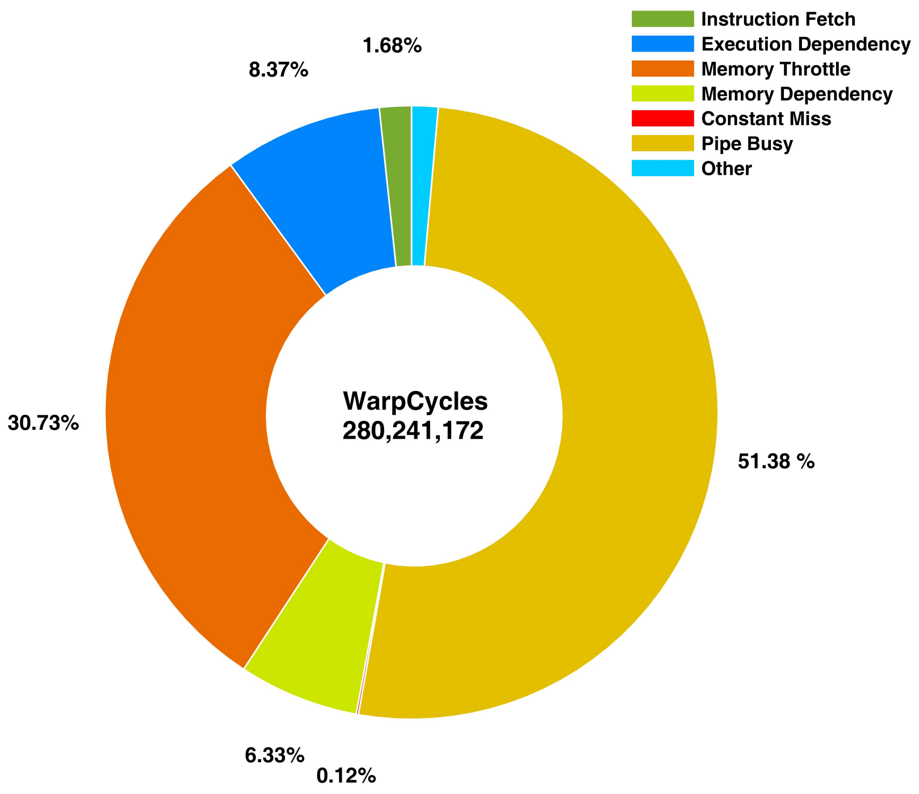

Figure 3.2. Issue stall reasons before optimization

reasons for the threads are switched from memory throttle, memory dependency, and pipe busy to synchronization, as shown in Figure 3.3. Although there are more warp cycles got stalled in the optimized code, the execution time was reduced by around 10% because a) the average stall time was reduced and b) a large part of issue stalls (synchronization) were under control. A GPU with CUDA architecture consists of multiple Streaming Multiprocessors (SM). Each SM has its independent instruction unit, constant cache, texture cache, shared memory, and multiple Streaming Processors (SP), which have independent registers and share the “shared memory”. To fully utilize the GPU, we expect the occupancy of each SM to be as

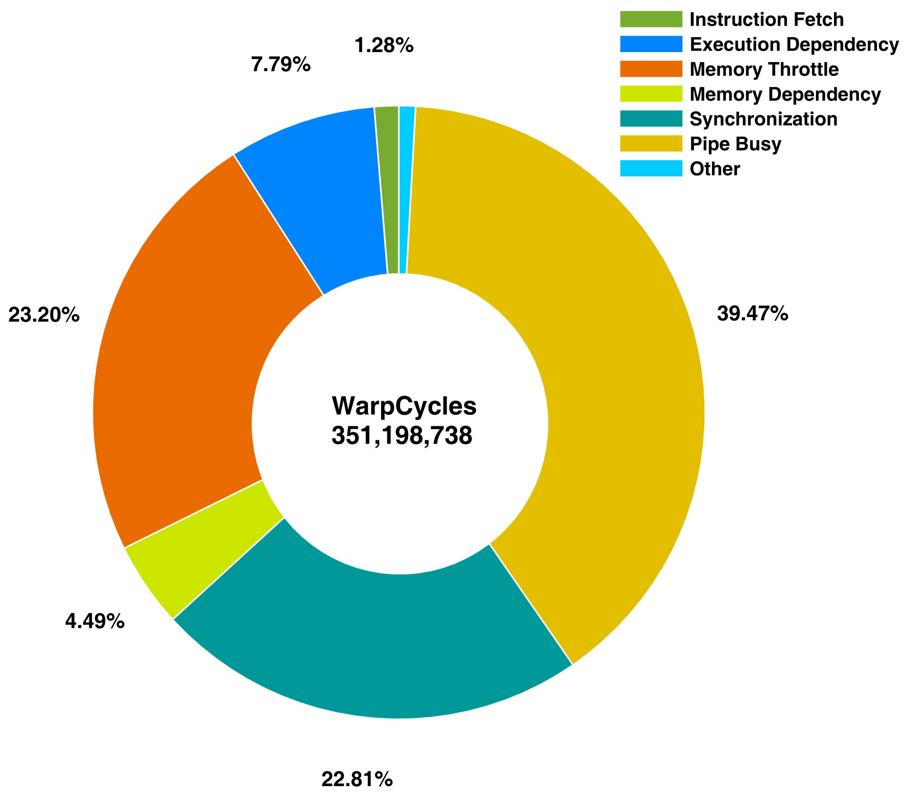

Figure 3.3. Issue stall reasons after optimization

high as possible. Table 3.1 shows the GPU occupancy of our program. This work achieved 100% theoretical occupancy and 93.89% achieved occupancy.

3.3.3 Single Machine Map-Reduce

In the chunk hashing and chunk matching step, a simple approach is to compute the hash value of each chunk, store it in a hash table, and detect duplicate chunks by referring to the hash table. The problem with such an approach is that a large amount of time will be consumed in randomly accessing the memory when the hash table is repeatedly accessed. Another idea is to launch multiple threads to execute chunk matching task, which would result in much better performance.

Table 3.1 *GPU occupancy*

Variable	Achieved	Theoretical	Device Limit
Occupancy Per SM			
Active Blocks	-	4	16
Active Warps	60.09	64	64
Active Threads	-	2048	2048
Occupancy	93.89%	100%	100%
Warps			
Threads/Block	-	512	1024
Warpss/Block	-	16	32
Block Limit	-	4	16
Registers			
Registers/Thread	-	17	255
Registers/Block	-	12288	65536
Registers/SM	-	49152	65536
Block Limit	-	5	16
Shared Memory			
Shared Memory/Block	-	1034	49152
Shared Memory/SM	-	4136	49152
Block Limit	-	38	15

However, there exists an upper boundary of throughput in the approach of simply launching multiple threads to execute a chunk matching task, as there would be massive time spent in waiting for acquiring locks on the a) hash table and b) chunk hashing result queue, both of which would need to be accessed by all the threads to prevent a data race condition. We developed our own data mapper and reducer inspired by Hadoop Map-Reduce with the goal of parallelizing data chunk hashing (using the mapper developed in this research) and data chunk hash matching (using the reducer developed in this research). We call our approach *Map-Reduce*, however, we did not use Hadoop Map-Reduce software - we developed our own implementation. We used our single machine map-reduce architecture, which significantly improved the performance of these two steps.

Figure 3.4 shows our Map-Reduce structure in this research. Assume there are μ threads of chunk hashing as the mappers, and γ threads of chunk matching as

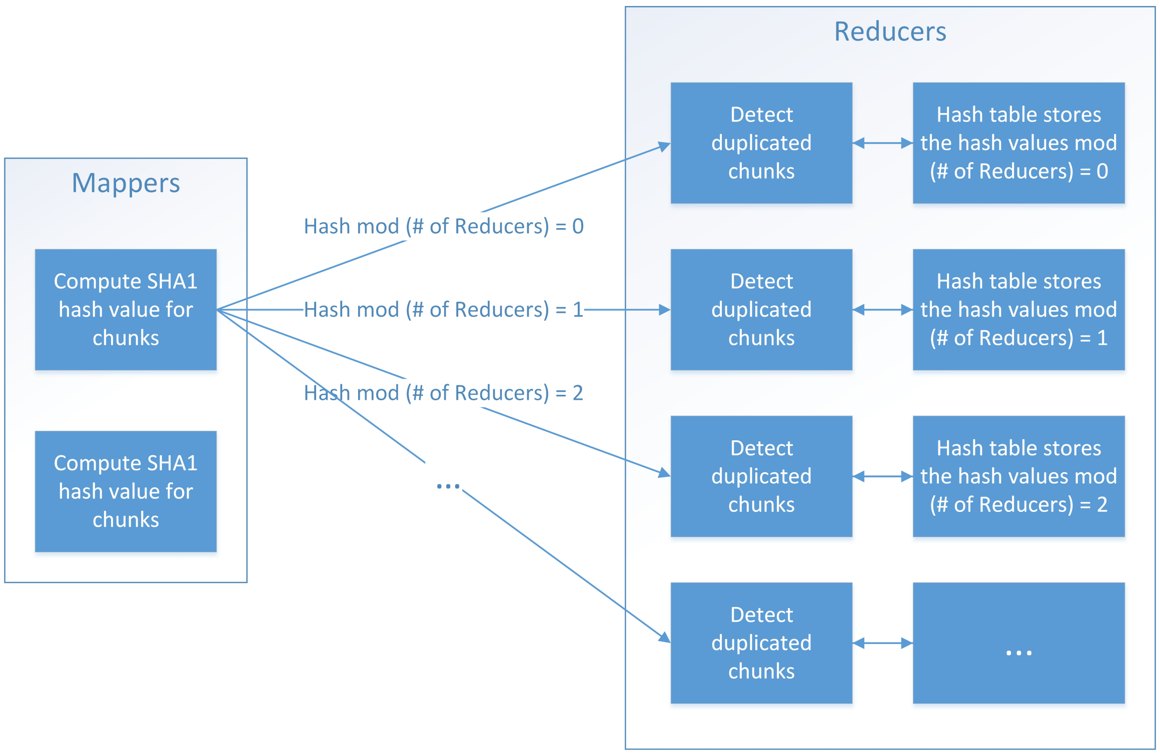
Mappers
Compute SHA1 hash value for chunks
Compute SHA1 hash value for chunks
Hash mod (# of Reducers) = 0
Hash mod (# of Reducers) = 1
Hash mod (# of Reducers) = 2
...
Reducers
Detect duplicated chunks
Hash table stores the hash values mod (# of Reducers) = 0
Detect duplicated chunks
Hash table stores the hash values mod (# of Reducers) = 1
Detect duplicated chunks
Hash table stores the hash values mod (# of Reducers) = 2
Detect duplicated chunks
...

Figure 3.4. Map-Reduce structure

the reducers. The mapper computes a SHA1 hash value for each chunk and sends to hash value to a reducer based on the value of $(hash\_value \mod \gamma)$. Each reducer i maintains a hash table separately, which only stores hash values satisfying Equation 3.1.

$$hash\_value \mod \gamma = i \qquad \text{(Eqn. 3.1)}$$

So that Equation 3.1 performs the functionality of a shuffle between mappers and reducers. Furthermore, γ should be greater than μ because maintaining a hash table is much slower than computing a SHA1 hash value.

We conducted an experiment using a different number of mappers and reducers to analyze its effect on the throughput of our algorithm. As shown in Figure 3.5, the throughput speed increases with the number of mappers. However, the number of mappers does not significantly impact the throughput speed when there are more than 8 mappers. The throughput speed fluctuates when the number

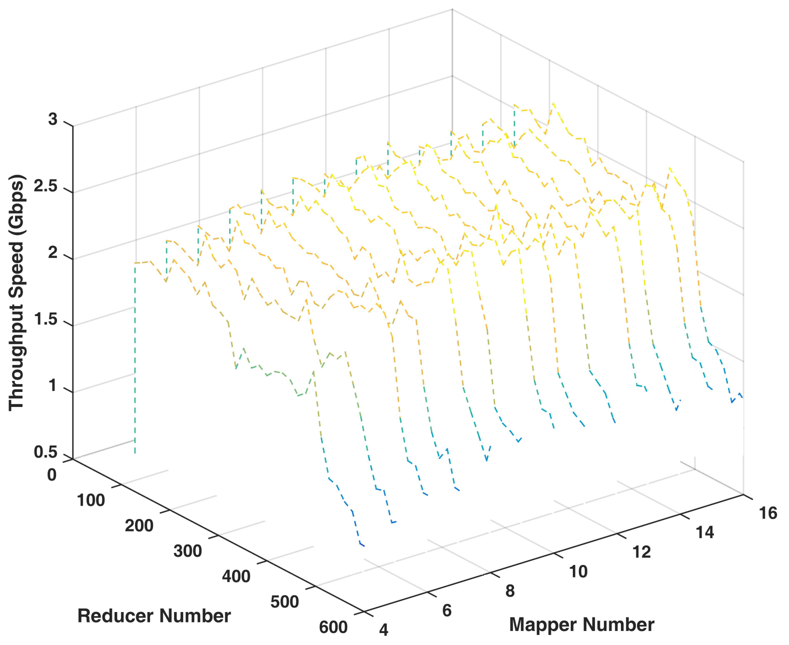

Figure 3.5. Influence of the number of mappers and reducers on performance

of reducers is smaller than 500, and deteriorates when the number of reducers is higher than 500. An explanation for this is that the system we ran experiment on cannot handle that many threads. Experimental results showed that the algorithm performed best with $\mu \in [8, 14]$ and $\gamma \in [92, 218]$. We used $(\mu, \gamma) = (11, 212)$ in the experiments in Section 3.4.

3.3.4 Multi-threaded Pipeline

Pipelining is the task mode in which the workers perform their assigned tasks simultaneously, and pass their output to the next worker as input. To increase the throughput of the redundancy elimination algorithm, we used a multi-threaded pipeline technique to minimize the idle time of each device.

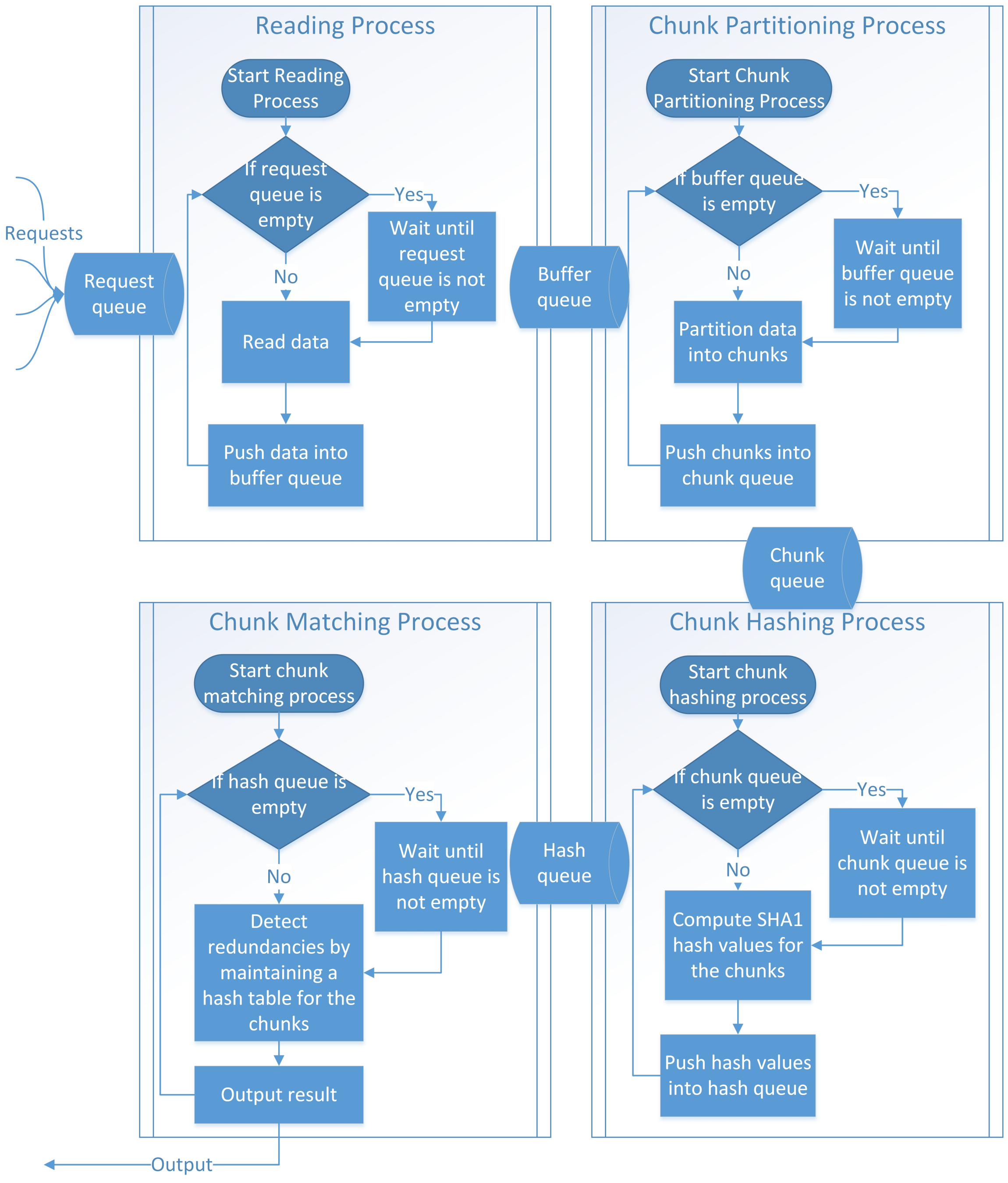
Reading Process
Start Reading Process
If request queue is empty
Yes
Wait until request queue is not empty
No
Read data
Push data into buffer queue
Requests
Request queue
Buffer queue
Chunk Partitioning Process
Start Chunk Partitioning Process
If buffer queue is empty
Yes
Wait until buffer queue is not empty
No
Partition data into chunks
Push chunks into chunk queue
Chunk queue
Chunk Matching Process
Start chunk matching process
If hash queue is empty
Yes
Wait until hash queue is not empty
No
Detect redundancies by maintaining a hash table for the chunks
Output result
Output
Hash queue
Chunk Hashing Process
Start chunk hashing process
If chunk queue is empty
Yes
Wait until chunk queue is not empty
No
Compute SHA1 hash values for the chunks
Push hash values into hash queue

Figure 3.6. The pipeline process

Figure 3.6 shows the work flow of the multi-threaded pipeline redundancy elimination algorithm:

- When the program receives a request, it pushes the request into the request queue.
- The reading process retrieves a request from the request queue, reads the data for the request into memory, and then pushes the memory page into the buffer queue.
- The chunk partitioning process retrieves data from the buffer queue, partitions the data into chunks, and then pushes the chunks into the chunk queue.
- The chunk hashing process retrieves data from the chunk queue, computes the SHA1 hash for each chunk, and pushes the hash values into the hash queue.
- The chunk matching process retrieves data from the hash queue, detects duplicate chunks by maintaining a hash table for the chunks, and outputs the result for the request.

Each process keeps running as long as the input buffer is not empty, which makes it much faster than the serialized method.

Furthermore, we used asynchronized memory transfer in the CUDA accelerated packets/objects chunk partitioning step. The CUDA stream technique allows the kernel to execute the instructions in the order they are sent by the program, and the host-device memory transfer and kernel functions can be simultaneously executed. We used this technique as the second layer pipeline within the pipelining of the whole process.

3.4 Measurements

The methods were evaluated on a machine with an Intel(R) Core(TM) i7-5930K CPU @ 3.50 GHz, with 6 cores and 16.0 GB RAM. The GPU installed in this machine is an NVIDIA Tesla K40c. The hardware accelerated redundancy elimination is evaluated on Windows 8 64 bit operating system.

We generated experimental data for our evaluation using data from Youtube traces that are publicly available in the *UMass Trace Repository* (2014). We wrote a batch downloading script *gotube*[1] inspired by a python YouTube video downloading library written by Ficano (2015).

We generated six test cases as the experiment data, which are 10 GB files. The redundancies are detected in a window about 8 GB, so that the least recently accessed chunks would be removed from the cache. Figure 3.7 and Table 3.2 show the average value and standard deviation of the experimental results for all six test cases. We did not intend to simulate the natural of downloading in real-world, where the videos might be downloaded simultaneously and overlap with each other. As the size of video files (normally 10-20 MB) is relatively small in terms of size of the cache (8-16 GB), the video files would not compete on the cache. Therefore, the order of packets of the videos would not affect the experimental result.

We evaluated our method by comparing the throughput and the detected redundancy rate of the program introduced in this research (HARENS) with the other three benchmark programs[2]. The algorithms we evaluated were:

- Naive C++ implementation
- Multi-threaded pipeline accelerated algorithm
- CUDA accelerated algorithm
- HARENS

In Figure 3.7, we showed the throughput of the four methods. As is shown in the figure, the multi-threaded pipeline accelerated algorithm demonstrated about a 60% throughput improvement over the naive C++ implementation, because the multi-threaded pipeline accelerated algorithm overlapped the execution time of each steps. The improvement in the multi-threaded pipeline accelerated algorithm was not significant compared with the naive C++ implementation because each step was

[1]The script can be found in https://github.com/KeluDiao/gotube
[2]All the programs can be found in https://github.com/keludiao/REinNS

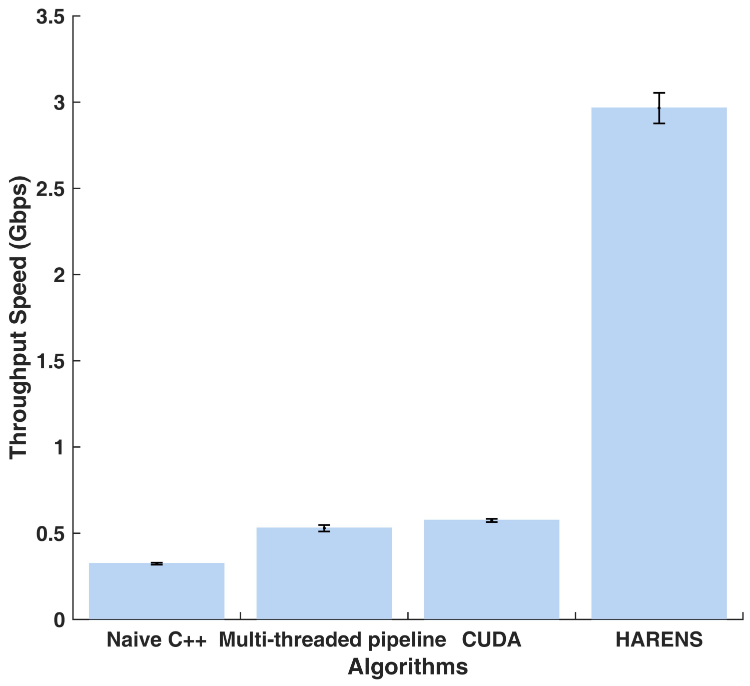

Figure 3.7. Throughput speed of the four methods (sample number = 6)

Table 3.2 *Redundancy Rate Detected (sample number = 6)*

	Mean	Standard Deviation
Naive C++	19.00	2.96
Multi-threaded pipeline	19.00	2.96
CUDA	18.98	2.91
HARENS	19.02	2.98

still time consuming. The CUDA accelerated algorithm had about 70% throughput improvement over the naive C++ implementation, because the CUDA accelerated algorithm shortened the time required for objects/packets chunk partitioning. HARENS, the method introduced in this work had about a factor of 9 times throughput improvement over the naive C++ implementation, because HARENS not only shortened the execution time of objects/packets chunk partitioning, chunk hashing, and chunk matching, but also overlapped the execution time of each step.

In Table 3.2, we show the detected redundancy rate of the trace file using the four methods. The redundancy rates detected by the naive C++ implementation, the multi-threaded pipeline accelerated approach, and the CUDA accelerated approach are similar. Our method detected slightly more redundancy than the other three methods. As the buffer is limited, the older chunks would be removed from the buffer according to the LRU replacement technique. The Map-Reduce chunk hashing/matching mechanism has a better chance of keeping the redundant chunks because the redundancies are generally clustered.

3.5 Summary

In this chapter, the author presented HARENS, an efficient approach developed for redundancy elimination in network systems. The author used a multi-threaded pipeline to overlap the execution time of each step. For the object/packet chunk partitioning step, this research used a GPU to accelerate the Rabin fingerprinting algorithm. In this step, this research made use of shared memory to improve memory bandwidth, applied asynchronized memory transfer to minimize the blocking time of the kernel instructions, and balanced the usage of GPU registers and shared memory to activate the largest possible number of threads in execution. Hence, this research achieved the highest possible theoretical GPU occupation. For the chunk hashing and chunk matching step, this research applied our single machine Map-Reduce architecture which distributed and scheduled the work load of these two steps in thousands of threads which improved the overall performance by about a factor of 9 times.

There are several interesting avenues for future work. The Map-Reduce architecture this research used could be deployed on a distributed system, such as Openstack. We could also adopt the hybrid method introduced by Papapanagiotou, Callaway, and Devetsikiotis (2012), making our redundancy elimination module work with a proxy cache module, to see if it can improve performance. Besides, the author found that there is significant time spent in lock acquisition and release. A lock-free method would be another good topic for further investigation.

CHAPTER 4. FRAMEWORK AND METHODOLOGY OF DRESS

This chapter provided the framework and methodology that were used in the research study of Distributed Redundancy Elimination System Simulation (DRESS).

4.1 Research Type

This research study was a mix of qualitative and quantitative research. Our goal was finding a way to make the clients' side middle-boxes fetch data from a network device that is closer to it in the network than a server. We estimated how cooperative distributed caches impacted the bandwidth savings and affected the detected redundancy rate.

4.2 Hypothesis

H_0: DRESS does not improve the bandwidth savings of a network data redundancy elimination systems.

H_a: DRESS improves the bandwidth savings of a network data redundancy elimination systems.

4.3 Methodology

The use of distributed caches is a widely used method to distribute computational and I/O workload over multiple machines. We deployed the hardware accelerated redundancy elimination module in the server's side middle-box, and distributed caches in the clients' side middle-boxes. The clients'

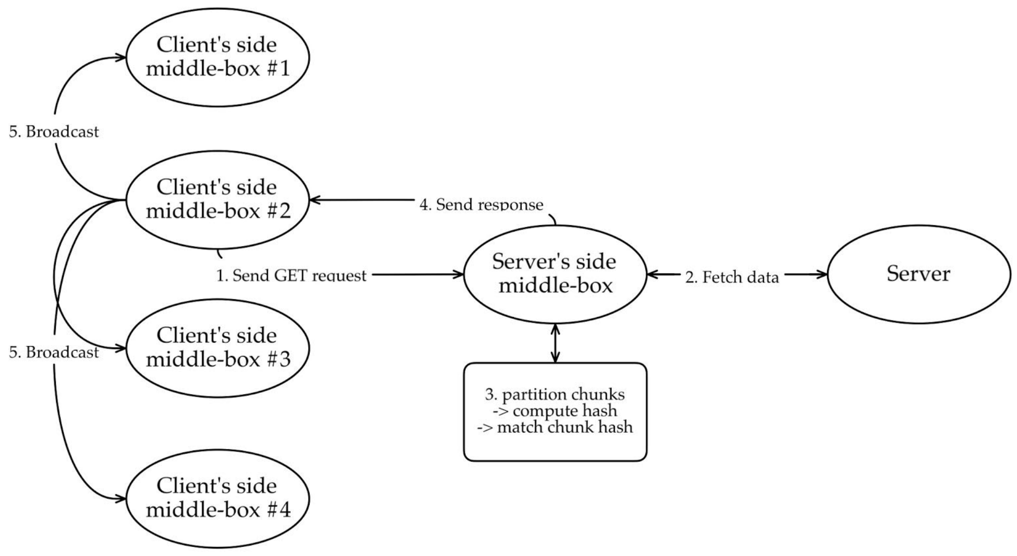
Client's side middle-box #1
5. Broadcast
Client's side middle-box #2
4. Send response
1. Send GET request
Server's side middle-box
2. Fetch data
Server
Client's side middle-box #3
5. Broadcast
3. partition chunks -> compute hash -> match chunk hash
Client's side middle-box #4

Figure 4.1. Work flow if no redundancy detected

side middle-boxes are geographically close to each other, so the latency of data transfer between clients' side middle-boxes are generally shorter than latency of data transfer between a server's side middle-box and a client's side middle-box. It is also easier to improve the bandwidth between clients' side middle-boxes. Based on our assessment of DRESS using a small network configuration, we believe that our method is a good fit for a small cooperative cache system. In terms of scaling, however, managing the distributed caches is likely to pose a challenge in terms of performance and in discovering the client who holds a data block. Additional work would be needed to investigate the challenges posed by scaling.

Figure 4.1 shows the work flow of such a system when no redundant chunks are detected:

1. When a client sends a GET request, the client's side middle-box redirects the request to server side middle-box.

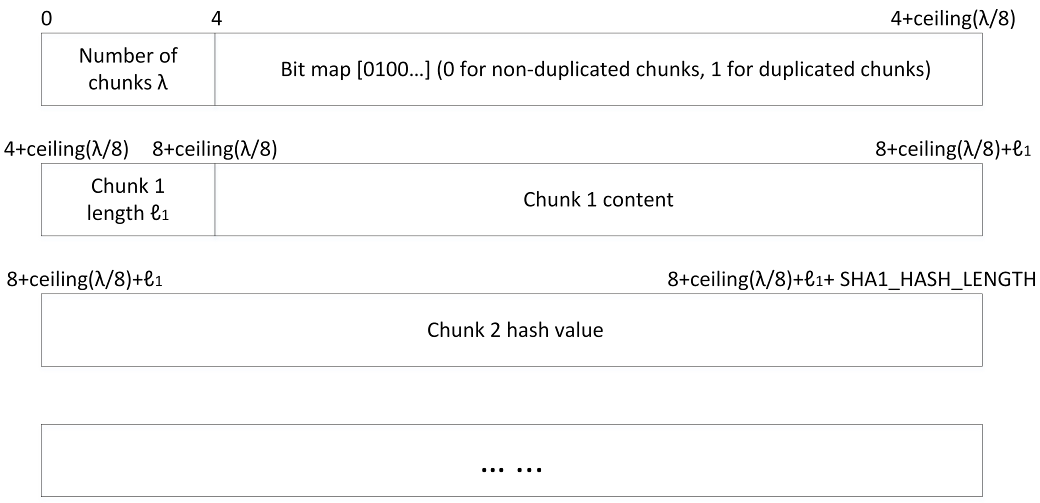

Figure 4.2. Format of response for GET request

2. The server's side middle-box fetches the data from server. One can also apply other redundancy elimination methods here, for example, caching static web pages, to further increase the performance.

3. The server's side middle-box then applies the aforementioned redundancy elimination algorithm to the data: partitions data into chunks, computes hash values for the chunks, and matches the hash values to detect redundancies.

4. The format of the response for a GET request from a client's side middle-box as shown in Figure 4.2 consists of three parts: an integer indicating the number of chunks in the data, a bit map indicating whether each chunk is duplicated or not, followed by an array of chunks if chunk is not duplicated, or hash values if chunk is duplicated. The numbers marked above each bar is the offset from the beginning of the payload of the response.

5. When a client's side middle-box receives a response, it stores the transmitted chunks in its cache and computes hash values for the non-duplicated chunks. Then it broadcasts the hash values of the newly accepted chunks to cooperating middle-boxes. The client's side middle-boxes that receive the

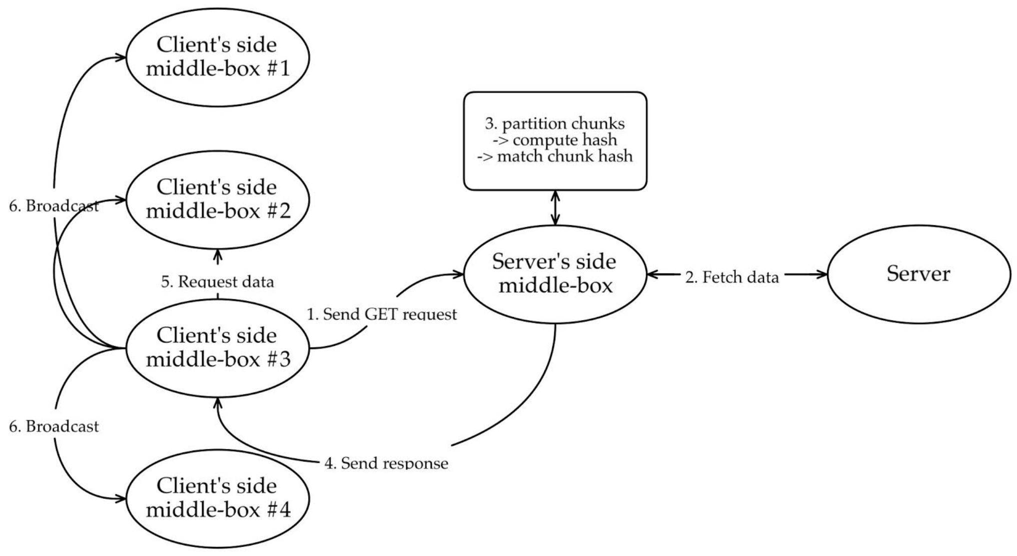
Client's side middle-box #1
Client's side middle-box #2
Client's side middle-box #3
Client's side middle-box #4
6. Broadcast
5. Request data
6. Broadcast
3. partition chunks -> compute hash -> match chunk hash
Server's side middle-box
1. Send GET request
2. Fetch data
Server
4. Send response

Figure 4.3. Work flow with redundancy detected

broadcast maintain a hash table, recording the hash values that the other middle-boxes have.

Figure 4.3 shows the work flow of the system when redundant chunks are detected. The first four steps of this scenario are the same as first four steps in the work flow where no redundancy is detected. When the client's side middle-box that requested for the data gets the response, it would fetch the duplicated chunks from caches. If the duplicated chunk does not exist in its own cache, the client's side middle-box would request the data from other client's side middle-boxes according to the hash map.

4.4 Measurements

The methods were evaluated on a machine with Intel(R) Core(TM) i7-5930K CPU @ 3.50 GHz, with 12 cores and 16.0 GB RAM. The GPU installed in this machine is NVIDIA Tesla K40c. The cooperative distributed cache is evaluated on Debian 8 64 bit operating system, since the experiment is conducted over a

Table 4.1 *Test cases for distributed byte cache*

Test Case	Number of Files	Total Size (Bytes)
1	400	5,196,155,543
2	400	5,810,427,132
3	400	6,566,425,587
4	400	5,919,295,453
5	400	4,762,626,716

simulated network using NS-3 (*NS-3 documentation*, 2011), which only runs in Linux systems.

Experimental data for DRESS are also generated based on Youtube traces that are publicly available in *UMass Trace Repository* (2014) using the batch downloading script *gotube*. Table 4.1 shows the five test cases in this experiment, which were generated as mentioned earlier. The video data were stored in files named by the video IDs assigned by YouTube. The order of requests (for video files) that clients' side middle-boxes send to server's side middle-box was based on the trace file in *UMass Trace Repository* (2014), and the request was generated randomly under a Poisson distribution. Figure 4.4 and Figure 4.5 show the experimental result of all the test cases separately.

We evaluated our method by comparing the detected redundancy and bandwidth savings of the program introduced in this research with the other 2 benchmark programs [1]. The algorithms evaluated are listed below:

- Redundancy elimination system with cooperative distributed byte cache
- Redundancy elimination system without cooperative distributed byte cache
- Network system without redundancy elimination program

In Figure 4.4, we show the average redundancy rate in the distributed caches. One can see that the average detected redundancy rate of the redundancy

[1]All the programs can be found in https://github.com/keludiao/DRESS

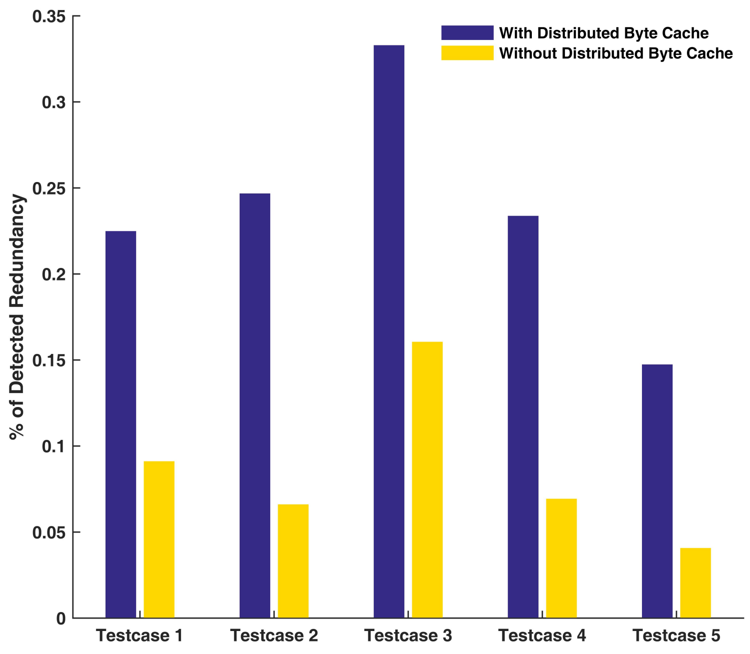

Figure 4.4. The average detected redundancy rate in the distributed caches

elimination system with distributed byte caches was about two to three times greater than the average detected redundancy rate of the redundancy elimination system without distributed byte caches. In the redundancy elimination system without distributed byte cache, the server's side middle-box needed to keep a hash table for each of the client's side middle-box. It only sent a hash value instead of chunks when the chunk exists in the cache of the client's side middle-box that sent the current request. In the redundancy elimination system with distributed byte cache, the server's side middle-box only needed to keep one hash table for the chunks. It would send a hash value for the duplicated chunks even if it does not exist in the cache of the client's side middle-box that sent the current request. Therefore, the distributed cache significantly increased the chance to detect duplicated data and reduces the memory space required in server's side middle-box for the hash tables.

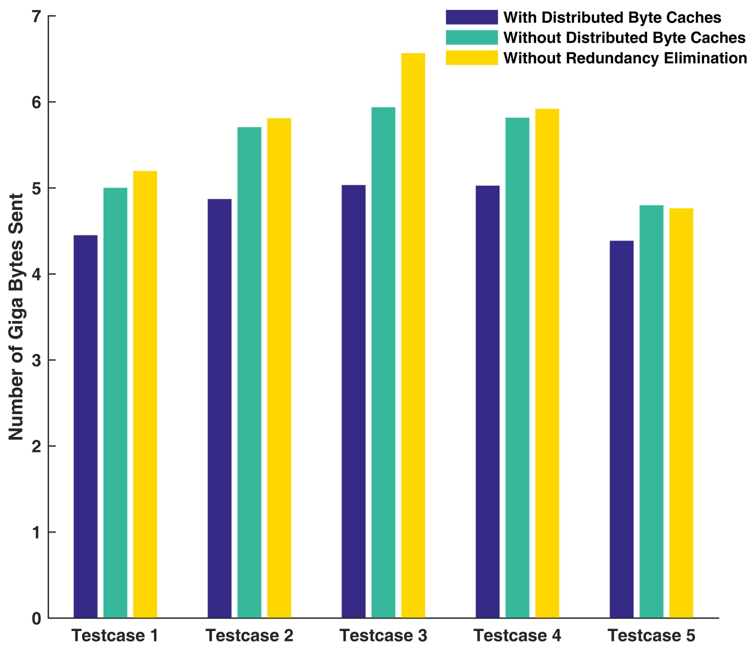

Figure 4.5. Size of data needed to send

In Figure 4.5, we show the amount of data need to send in a redundancy elimination system with distributed byte caches, a redundancy elimination system without distributed byte caches, and a network system without redundancy elimination program. The amount of data needed to send in the redundancy elimination system without redundancy elimination program was the same as the amount of data requested. One can see from the figure that the redundancy elimination system with distributed byte caches can save about 15% of the bandwidth, while the redundancy elimination system without distributed byte caches can only save 3% of the bandwidth. The redundancy elimination system with distributed byte caches saved 12% more bandwidth than the one without distributed byte caches. In test case 5, the redundancy elimination system without distributed byte caches generated even more traffic than the baseline system, since there were only small amount of redundancies detected. This results in smaller amount of bandwidth savings compared to the meta data (including the integer indicating number of chunks and the bit map)

4.5 Summary

In this chapter, the author presented DRESS, a distributed Internet cache that scales to improve the performance of a redundancy elimination system. The author deployed HARENS as the server's side redundancy elimination module, and four cooperative distributed Internet caches in the clients' side middle-boxes. Results showed that the redundancy elimination system with cooperative distributed Internet caches saved 12% more bandwidth than the redundancy elimination system without a cooperative distributed Internet cache.

There are several interesting avenues for future work. The cooperative distributed caches in this research are close to each other in the network. We can find a method for wider Internet and let the distributed caches to cooperate with their neighbors. We can also make the server's side redundancy elimination module a part of the P2P network, so the redundancy elimination program can be applied to both download and upload streams. As mentioned in previous chapter, we can adopt the hybrid method introduced by Papapanagiotou, Callaway, and Devetsikiotis (2012), making HARENS work with a proxy cache redundancy elimination module. It would be interesting to research about how to put this hybrid method in DRESS network system.

CHAPTER 5. SUMMARY

In this thesis, the author presented HARENS, an efficient approach developed for redundancy elimination in network systems, and a simulation of distributed Internet cache to improve the performance of redundancy elimination.

HARENS provided an acceleration to the redundancy elimination method based on Rabin fingerprinting algorithm. We divided the process of redundancy elimination into four steps: fetching data, partitioning packets/objects chunks, computing SHA1 hash values for chunks, and matching chunks by comparing hash values. Each step was treated as a Finite State Machine (FSM), which ran separately but shared data buffers to synchronize with each other. These FSMs performed the role of workers in a pipeline, which can keep the tasks running simultaneously. Due to the acceleration methods, HARENS answered the hypothesis by: a) improving the throughput of network data de-duplication by about a factor of 9 times and b) witnessing no significant improvement on redundancy detection.

DRESS scaled HARENS in a larger network by deploying cooperative distributed Internet caches. HARENS performed the role of server's side redundancy elimination module, and the cooperative distributed Internet caches performed the role of clients' side redundancy elimination modules. There was higher chance to detect duplicated data in this network, since the distributed Internet caches shared the information of their cached chunks with each other. DRESS answered the hypothesis by improving 12% more bandwidth savings than the network redundancy elimination system without cooperative distributed Internet caches.

There are many potential improvements and further works for both HARENS and DRESS. We can use hybrid methods combining HARENS redundancy elimination module with traditional proxy-cache methods. We can make better use of hardwares and reduce the time consumption of synchronizing the

FSMs. We can also further scale our redundancy elimination system to a larger network by making the distributed Internet caches cooperate with their neighbors.

To conclude, redundancy elimination is a good solution for the network traffic jam. It is a much cheaper solution than increasing the network bandwidth by deploying more coppers or replacing them with better network cables. However, current redundancy elimination methods are too old and slow for the rapidly increasing bandwidth. We are glad to be one of the first to pick up this topic again, and hope there will be more research to further dig this gold mine and solve the real problems.

LIST OF REFERENCES

LIST OF REFERENCES

Abrams, M., Standridge, C. R., Abdulla, G., Williams, S., & Fox, E. A. (1995). *Caching proxies: Limitations and potentials* (Tech. Rep.). Blacksburg, VA, USA.

Aggarwal, B., Akella, A., Anand, A., Balachandran, A., Chitnis, P., Muthukrishnan, C., ... Varghese, G. (2010). EndRE: An end-system redundancy elimination service for enterprises. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation* (pp. 28–28). Berkeley, CA, USA: USENIX Association. Retrieved from http://dl.acm.org/citation.cfm?id=1855711.1855739

Anand, A., Muthukrishnan, C., Akella, A., & Ramjee, R. (2009, June). Redundancy in network traffic: Findings and implications. *SIGMETRICS Perform. Eval. Rev.*, *37*(1), 37–48. Retrieved from http://doi.acm.org/10.1145/2492101.1555355 doi: 10.1145/2492101.1555355

Barford, P., Bestavros, A., Bradley, A., & Crovella, M. (1999, January). Changes in web client access patterns: Characteristics and caching implications. *World Wide Web*, *2*(1-2), 15–28. Retrieved from http://dx.doi.org/10.1023/A:1019236319752 doi: 10.1023/A:1019236319752

Bhatotia, P., Rodrigues, R., & Verma, A. (2012). Shredder: GPU-accelerated incremental storage and computation. In *FAST* (p. 14).

Bolot, J.-C., & Hoschka, P. (1996). Performance engineering of the World Wide Web: Application to dimensioning and cache design. In *Proceedings of the Fifth International World Wide Web Conference on Computer Networks and ISDN Systems* (pp. 1397–1405). Amsterdam, The Netherlands, The Netherlands: Elsevier Science Publishers B. V. Retrieved from http://dl.acm.org/citation.cfm?id=232710.232762

Bolot, J.-C., Lamblot, S. M., & Simonian, A. (1997). *Design of efficient caching schemes for the World Wide Web.*

Borst, S., Gupt, V., & Walid, A. (2010). Distributed caching algorithms for content distribution networks. In *Infocom, 2010 proceedings ieee* (pp. 1–9).

Broder, A. Z. (1993). Some applications of Rabins fingerprinting method. In *Sequences II* (pp. 143–152). Springer.

C++0x/C++11 support in GCC. (2015). Retrieved 2015-09-28, from https://gcc.gnu.org/projects/cxx0x.html

Callaway, R. D., & Papapanagiotou, I. (2013a, March 13). *Dynamic caching module selection for optimized data deduplication.* Google Patents. (US Patent App. 13/800,289)

Callaway, R. D., & Papapanagiotou, I. (2013b, October 22). *Dynamic caching module selection for optimized data deduplication.* Google Patents. (US Patent App. 14/059,959)

Callaway, R. D., & Papapanagiotou, I. (2014, September 4). *Scheduler training for multi-module byte caching.* Google Patents. (US Patent App. 14/477,093)

Chuang, J. C.-I., & Sirbu, M. A. (2000). Distributed network storage service with quality-of-service guarantees. *Journal of Network and Computer Applications*, *23*(3), 163–185.

CUDA toolkit documentation. (2015). Retrieved 2015-09-24, from https://docs.nvidia.com/cuda/cuda-c-programming-guide/#kernels

Dal Bianco, G., Galante, R., & Heuser, C. A. (2011). A fast approach for parallel deduplication on multicore processors. In *Proceedings of the 2011 acm symposium on applied computing* (pp. 1027–1032). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/1982185.1982411 doi: 10.1145/1982185.1982411

Douglis, F., & Iyengar, A. (2003). Application-specific Delta-encoding via resemblance detection. In *USENIX Annual Technical Conference* (pp. 113–126).

Ficano, N. (2015). *pytube.* https://github.com/nficano/pytube.git. GitHub.

Gadde, S., Rabinovich, M., & Chase, J. (1997). Reduce, reuse, recycle: An approach to building large internet caches. In *Operating systems, 1997., the sixth workshop on hot topics in* (pp. 93–98).

Gao, S., & Panario, D. (1997). Tests and constructions of irreducible polynomials over finite fields. In *Foundations of computational mathematics* (pp. 346–361). Springer.

Glassman, S. (1994). A caching relay for the World Wide Web. In *Computer Networks and ISDN Systems* (pp. 69–76).

Global Internet phenomena Africa, Middle East & North America. (2015). https://www.sandvine.com/downloads/general/global-internet-phenomena/2015/global- (Accessed:2016-04-12)

Housel, B. C., & Lindquist, D. B. (1996). WebExpress: A system for optimizing web browsing in a wireless environment. In *Proceedings of the 2nd Annual International Conference on Mobile Computing and Networking* (pp. 108–116). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/236387.236416 doi: 10.1145/236387.236416

Huang, B., Sun, Z., Chen, H., Mao, J., & Zhang, Z. (2012). Bufferbank: A distributed cache infrastructure for peer-to-peer application. *Peer-to-Peer Networking and Applications*, *7*(4), 485–496.

Iyer, S., Rowstron, A., & Druschel, P. (2002). Squirrel: A decentralized peer-to-peer web cache. In *Proceedings of the twenty-first annual symposium on principles of distributed computing* (pp. 213–222).

Jacobson, V. (1990). *Compressing TCP/IP headers for low-speed serial links.*

Kolb, L., Thor, A., & Rahm, E. (2012, August). Dedoop: Efficient deduplication with hadoop. *Proc. VLDB Endow.*, *5*(12), 1878–1881. Retrieved from http://dx.doi.org/10.14778/2367502.2367527 doi: 10.14778/2367502.2367527

Manber, U. (1994). Finding similar files in a large file system. In *USENIX Winter 1994 Technical Conference* (pp. 1–10).

Markatos, E. P. (1996). Main memory caching of web documents. In *Proceedings of the Fifth International World Wide Web Conference on Computer Networks and ISDN Systems* (pp. 893–905). Amsterdam, The Netherlands, The Netherlands: Elsevier Science Publishers B. V. Retrieved from http://dl.acm.org/citation.cfm?id=232710.232714

Meira Jr, W., Fonseca, E., Murta, C., & Almeida, V. (1998). Analyzing performance of cache server hierarchies. In *Computer Science, 1998. SCCC'98. XVIII International Conference of the Chilean Society of* (pp. 113–121).

Mogul, J. C., Douglis, F., Feldmann, A., & Krishnamurthy, B. (1997). Potential benefits of Delta encoding and data compression for HTTP. In *ACM SIGCOMM Computer Communication Review* (Vol. 27, pp. 181–194).

Morreale, P. A., & Anderson, J. M. (2014). *Software defined networking: Design and deployment.* CRC Press.

Muthitacharoen, A., Chen, B., & Mazières, D. (2001, October). A low-bandwidth network file system. *SIGOPS Oper. Syst. Rev.*, *35*(5), 174–187. Retrieved from http://doi.acm.org/10.1145/502059.502052 doi: 10.1145/502059.502052

NS-3 documentation. (2011). Retrieved from https://www.nsnam.org/documentation (Accessed:2016-04-19)

Papapanagiotou, I., Callaway, R. D., & Devetsikiotis, M. (2012). Chunk and object level deduplication for web optimization: A hybrid approach. In *Communications (ICC), 2012 IEEE International Conference on* (pp. 1393–1398).

Papapanagiotou, I., Nahum, E. M., & Pappas, V. (2012). Smartphones vs. laptops: Comparing web browsing behavior and the implications for caching. In *Acm sigmetrics performance evaluation review* (Vol. 40, pp. 423–424).

Pitkänen, M. J., & Ott, J. (2007). Redundancy and distributed caching in mobile dtns. In *Proceedings of 2nd acm/ieee international workshop on mobility in the evolving internet architecture* (p. 8).

Povey, D., & Harrison, J. (1997). A distributed Internet cache. *Australian Computer Science Communications*, *19*, 175–184.

Pucha, H., Andersen, D. G., & Kaminsky, M. (2007). Exploiting similarity for multi-source downloads using file handprints. In *Proceedings of the 4th USENIX Conference on Networked Systems Design &#38; Implementation* (pp. 2–2). Berkeley, CA, USA: USENIX Association. Retrieved from http://dl.acm.org/citation.cfm?id=1973430.1973432

Rabin, M. O., et al. (1981). *Fingerprinting by random polynomials.* Center for Research in Computing Techn., Aiken Computation Laboratory, University.

Rennich, S. (2011). *CUDA C/C++ streams and concurrency* [PDF document]. Retrieved 2015-09-24, from http://on-demand.gputechconf.com/gtc-express/2011/presentations/StreamsAndConcurrencyWebinar.pdf

Rhea, S. C., Liang, K., & Brewer, E. (2003). Value-based web caching. In *Proceedings of the 12th International Conference on World Wide Web* (pp. 619–628). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/775152.775239 doi: 10.1145/775152.775239

Schleimer, S., Wilkerson, D. S., & Aiken, A. (2003). Winnowing: Local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data* (pp. 76–85). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/872757.872770 doi: 10.1145/872757.872770

Sijben, P., van Willigenburg, W., de Boer, M., & van der Gaast, S. (2002). Middleboxes: Controllable media firewalls. *Bell Labs technical journal*, *7*(1), 141–157.

Spring, N. T., & Wetherall, D. (2000). A protocol-independent technique for eliminating redundant network traffic. *ACM SIGCOMM Computer Communication Review*, *30*(4), 87–95.

Tay, T., Feng, Y., & Wijeysundera, M. (2000). A distributed Internet caching system. In *Local computer networks, 2000. lcn 2000. proceedings. 25th annual ieee conference on* (pp. 624–633).

Tolia, N., Kaminsky, M., Andersen, D. G., & Patil, S. (2006). An architecture for Internet data transfer. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3* (pp. 19–19). Berkeley, CA, USA: USENIX Association. Retrieved from http://dl.acm.org/citation.cfm?id=1267680.1267699

UMass trace repository. (2014). Retrieved 2015-10-09, from http://traces.cs.umass.edu/index.php/Network/Network

Wang, C., Xiao, L., Liu, Y., & Zheng, P. (2004). Distributed caching and adaptive search in multilayer P2P networks. In *Distributed computing systems, 2004. proceedings. 24th international conference on* (pp. 219–226).

Wang, J. M., Zhang, J., & Bensaou, B. (2013). Intra-AS cooperative caching for content-centric networks. In *Proceedings of the 3rd acm sigcomm workshop on information-centric networking* (pp. 61–66).

Williams, S. (1996). *Personal communication.* Retrieved from http://ei.cs.vt.edu/ williams/DIFF/prelim.html

Williams, S., Abrams, M., Standridge, C. R., Abdulla, G., & Fox, E. A. (1996). Removal policies in network caches for World-Wide Web documents. In *Conference Proceedings on Applications, Technologies, Architectures, and Protocols for Computer Communications* (pp. 293–305). New York, NY, USA: ACM. Retrieved from http://conferences.sigcomm.org/sigcomm/1996/papers/williams_orig.pdf doi: 10.1145/248156.248182

Willinger, W., Taqqu, M. S., Sherman, R., & Wilson, D. V. (1997). Self-similarity through high-variability: Statistical analysis of Ethernet LAN traffic at the source level. *IEEE/ACM Transactions on Networking (ToN)*, *5*(1), 71–86.

Working draft, standard for programming language C++. (2012). Retrieved 2015-09-28, from http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3337.pdf