CHALMERS

Design Study of a Computer System Employing Memory Compression

Master of Science Thesis in Embedded Electronic System Design

JONAS ANDERSSON
NIKLAS DOVERBO

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Gothenburg, Sweden, June 2015

Design Study of a Computer System Employing Memory Compression

JONAS ANDERSSON
NIKLAS DOVERBO

© JONAS ANDERSSON, June 2015
© NIKLAS DOVERBO, June 2015
Supervisors: Per Stenström and Angelos Arelakis
Examiner: Per Larsson-Edefors

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Gothenburg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Gothenburg, Sweden June 2015

Abstract

Memory compression is a promising technique for computer systems to increase cache and memory capacity, leading to a decrease of the number of required lower-level accesses without adding significant cost or energy consumption to a system. This thesis answers some of the questions arising when implementing a Huffman algorithm as compression algorithm on main memory content, such as compression factor compared to other compression algorithms and feasibility for compression in a system.

This thesis contains a compression analysis of two scale-out benchmarks from CloudSuite. The two benchmarks, Data analytics and Graph analytics, were set up running on virtual machines. While the benchmarks were running, the virtual machines had their memory extracted at multiple occasions. By using the virtual-to-physical address translation for the benchmarks' processes, in combination with the extracted memory sample, it was possible to create images of the memory used by the benchmarks' processes. These images were analysed with several different compression algorithms including Huffman, BΔI, and FPC.

The Huffman algorithm uses a value frequency table (VFT) which contains values and their frequency. The VFT is used to create the encoding for the Huffman tree where values with high frequency are given a short code word. A larger VFT will give a better compression factor but reduce speed, increase area and energy consumption. This thesis shows that the size of the VFT can be kept small and that the gains in compression factor of having a large VFT significantly decreased with VFT sizes over 1024 entries. The results in the compression analysis show that the memory compression factor for the two CloudSuite benchmarks is around 2.7X for a VFT with 1024 entries.

In a traditional computer system blocks and pages are fixed in size. A computer system using memory compression must allow for blocks and pages to vary in size, however, it would be infeasible to allow for all different block and page sizes. The thesis shows the impact on the Huffman compression factor when only particular compression factors or compressed sizes are allowed. Four additional block addressing bits result in a compression loss of between 4%-7%. A similar analysis for pages show that adding four or six additional addressing bits result in compression losses of between 6%-9% and 1%-2%, respectively.

Following the compression analysis, a few compression schemes were implemented and evaluated in hardware. One of these schemes, the Huffman coding scheme, was also implemented as part of a proof-of-concept. The proof-of-concept was implemented on a ZedBoard, an evaluation board that incorporates both a dual ARM core as well as programmable logic. The ordinary datapath between the cores and the memory was rerouted to pass through the programmable logic of the ZedBoard. In this programmable logic, the Huffman compressing and decompressing logic was implemented.

Acknowledgements

We would like to express our sincere gratitude towards our supervisors Per Stenström and Angelos Arelakis. Per Stenström gave us the opportunity to write our thesis in the very interesting field of memory compression. He has been an excellent supervisor and has provided us with valuable feedback on both our work and this report. He also included us as members in Chalmers' Euroserver team, an experience that has both been fun and valuable. Angelos Arelakis has been very helpful and we would like to thank him for always taking the time for us. His enthusiasm, knowledge, and willingness to answer question and discuss issues and possible solutions have been very valuable.

Finally, we would like to give our appreciation to Per Larsson-Edefors for being our examiner as well as providing helpful feedback on our writing throughout the thesis.

Jonas Andersson & Niklas Doverbo
Chalmers University of Technology
Gothenburg, June, 2015.

Contents

1 Introduction **1**
1.1 Euroserver . . . 2
1.2 Aim of the project . . . 2
1.3 Limitations . . . 3
1.4 Contribution . . . 3
1.5 Thesis outline . . . 4

2 Theory **6**
2.1 Scale-out . . . 6
2.2 CloudSuite . . . 7
2.2.1 Data analytics . . . 9
2.2.2 Graph analytics . . . 9
2.3 Virtualization . . . 10
2.3.1 QEMU and KVM . . . 10
2.4 Compression algorithms . . . 11
2.4.1 Huffman algorithm . . . 11
2.4.2 ZCA . . . 14
2.4.3 BΔI . . . 14
2.4.4 FPC . . . 15
2.4.5 DupDict . . . 16
2.5 Development platform . . . 17
2.5.1 ZedBoard . . . 17
2.5.2 AMBA AXI4 interface protocol . . . 19
2.6 Design flow . . . 21

3 Compression analysis **23**
3.1 Datasets . . . 24
3.1.1 Dataset analysis results . . . 24
3.2 Cloudsuite benchmarks . . . 25
3.2.1 Data analytics . . . 28

3.2.2 Graph analytics 31
3.2.3 VFT size improvement analysis 38
3.2.4 VFT sensitivity analysis 38
3.2.5 Alignment analysis 39
3.2.6 Alternative LO 46

4 System implementation 48
4.1 BΔI 48
4.2 FPC 51
4.3 Proof-of-concept 52
4.3.1 Write component 54
4.3.2 Read component 56
4.3.3 Software 58
4.4 Evaluation 59

5 Discussion 65

6 Conclusion 67
6.1 Future work 68
6.1.1 Compressor improvements 69
6.1.2 Decompressor improvements 69

Bibliography 73

A Appendix A 74
A.1 Data analytics, DA_z0 75
A.2 Data analytics, DA_z1 76
A.3 Data analytics, DA_z2 77
A.4 Data analytics, DA_z3 78
A.5 Graph analytics, GA_z0 79
A.6 Graph analytics, GA_z1 80
A.7 Graph analytics, GA_z2 81
A.8 Graph analytics, GA_z3 82
A.9 Graph analytics, GA_g_0 83
A.10 Graph analytics, GA_g_100 84
A.11 Graph analytics, GA_g_130 85
A.12 Dataset Google Plus 86
A.13 Dataset Twitter, Grammy 87
A.14 Dataset Wikipedia_NB 88
A.15 Dataset Twitter, Superbowl 89
A.16 Dataset Memetracker 90
A.17 Dataset Wikipedia_L 91

B Appendix B 92

B.1 Data analytics, Wikipedia DA_z0 93

C Appendix C **95**

C.1 Block alignment analysis 96

C.2 Page alignment analysis 101

List of abbreviations

B∆I	Base Delta Immediate
CF	Compression Factor
DA	Data Analytics benchmark
FPC	Frequent Pattern Compression
GA	Graph Analytics benchmark
HDL	Hardware Description Language
Huffman	A compression scheme created by D. A. Huffman
KVM	Kernel-based Virtual Machine
LLC	Last-Level Cache
LO	Length Overhead
MMU	Memory Management Unit
PFN	Page Frame Number
PID	Process ID
PL	Programmable Logic
PS	Processing System
RTL	Register Transfer Level
VFT	Value Frequency Table used by the Huffman algorithm
VM	Virtual Machine
VMM	Virtual Machine Manager
ZCA	Zero Content Augmented

1

Introduction

Memories play a significant role in modern computers. They contribute to a large percentage of both the total cost as well as the energy consumption of a computer system. Today, memories stand for roughly 10%-20% of the cost of a computer system, ranging all the way from smart phones to data centers. In addition, they contribute largely to the energy consumption of computers totalling up to 1% of the world's total energy usage [1, 2].

Caches and main memories are key resources in computer systems. The memory hierarchy created by these memories is instrumental for increasing performance by reducing the access time when fetching data and instructions in a computer system. Memories with high hit rate will increase a system's performance while, at the same time, decrease its energy consumption. A miss at one tier in the memory hierarchy results in an access to a lower tier which usually has much longer access time, and it is therefore essential to keep these costly accesses to a minimum. Over the years, several techniques and polices have been developed to increase memory hit rates and new technologies are still emerging.

One of the more promising categories of improvement in memory systems is data compression. Data compression techniques aim to lessen the space needed to store data by utilizing the available memory space more effectively. The penalty paid when using data compression schemes is an increased memory access time and the trade-off between high compression rates versus a short access time has, in the past, lead to small actual performance gains. To have a feasible data compression scheme one has to manage compressed data with a very short added access time. A task that is challenging at best and is probably the reason why data compression has not been used historically.

A recent innovation within the field of data compression addresses the problem with the added additional access time. This innovation includes the use of new specialised hardware in combination with new software. Through the use of this new hardware, it is possible to run Huffman coding, a previously known, aggressive, statistical compression scheme to generate a data compression of up to 4X while suffering almost no performance penalties [2].

A new version of Huffman coding has also been suggested, where a few bits of length information is added to every compressed word. This encoding allows for even faster decompression at the expense of lower compression factors.

1.1 Euroserver

Data centers are crucial to the modern day information era and one of the driving forces is cloud applications. The demand for processing power and data storage is continuously growing and to meet this growing demand, data centers need to improve performance and reduce their energy consumption [3].

The Euroserver project aims to increase the energy and software efficiency as well as reduce the cost of data centers through several recent innovations including, 64-bit ARM cores, 3D heterogeneous silicon-on-silicon integration, FD SOI process technology, and new software techniques for efficient resource management. The Euroserver project is a European collaboration between Commissariat à l'énergie atomique et aux énergies alternatives (France), STMicroelectronics (France), ARM Ltd (United Kingdom), EUROTECH SPA (Italy), Technische Universitaet Dresden (Germany), Barcelona Supercomputing Center (Spain), Foundation for Research & Technology - Hellas (Greece), Onapp Ltd (Gibraltar), and Chalmers Tekniska Högskola AB (Sweden) [1].

Memory compression is included in Chalmers contribution to the Euroserver project with the goal of being implemented in future generations of the Euroserver.

1.2 Aim of the project

The implementation of a memory compression scheme in today's computers is a challenging task. In this thesis we aim to tackle this task and have divided the thesis project into two parts: evaluate the Huffman compression with respect to scale-out workloads as well as to implement a prototype of a computer system employing data compression. The compression results will give an indication on how the Huffman compression behaves compared to the other algorithms and what memory compression factors one can expect for scale-out workloads.

The system implementation phase aims to implement compression algorithms between the LLC and the main memory on a Zynq processing system [4] which includes among other things, an FPGA, two ARM A9 cores, and a memory hierarchy with caches and main memory. In addition, the system will be evaluated with respect to performance, area, and power consumption. The main goal in this phase is to implement a proof-of-concept for the Huffman compression algorithm but this phase also includes implementation of other compression algorithms.

1.3 Limitations

This thesis focuses on compression of data in main memory for scale-out systems running applications commonly found in data centers. To better understand the behaviour in computing systems of all sizes, more research is needed.

Virtual machines used were limited to four cores and a memory size of 5 GB due to resource limitations. To accurately represent a scale-out system one would prefer to virtualise a system implementing more cores and a much larger main memory as encountered in real-world data centers.

This thesis limits its analysis to a comparison between four different compression algorithms, Huffman, base delta immediate (BΔI), frequent pattern compression (FPC), and DupDict. These four compression schemes have already been implemented in a C++ application which we have used throughout this project. The five compression algorithms were chosen since they represent a large set of compression schemes available.

Hardware implementations of a Huffman compressor and decompressor have been given. Although, some optimizations have been made.

1.4 Contribution

Previous work has touched upon the topic of implementing compression in the memory hierarchy of computer systems. Examples of these papers include [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15] although they do not have the same focus or in depth analysis of Huffman compression as this thesis does. This thesis is, to the knowledge of the authors, the only work in the area that focuses on compression for scale-out systems although [8] includes some evaluations for scale-out benchmarks.

Papers [7, 8, 12, 13, 14] all suggest using simple or fast compression techniques that generate lower compression values than what more advanced algorithms can. Papers [7, 14] center around the compression possible when exploiting the high amount of zeroes found in the memory hierarchy of computers. Papers [8, 12, 13] centers around the usage of BΔI or FPC compression, two schemes that are evaluated further in this paper. This thesis wants to examine the possibility of using Huffman compression.

Papers [9, 10, 11] use more advanced techniques with the potential of higher compression, but they all include software implementations at the OS level. This thesis wants to implement a fairly advanced algorithm, the Huffman algorithm, in hardware to gain both high compression and high speed.

This thesis stands out as one of the few contributions that not only evaluates hardware compression, but also creates a functional hardware prototype for feasibility testing. Although, complete systems that implement memory compression in hardware have been developed both at the University of Michigan [6] and by IBM [15] prior to this work.

The hardware developed at the University of Michigan [6] is an embedded processor, the Phoenix Processor, that implements compression with the focus of reduced energy consumption and memory usage. The compression algorithm used by the Phoenix Processor is Huffman coding. Compared to the design we are evaluating the Phoenix Processor's Huffman tree is static and will experience lower compression rates than by using a dynamic Huffman tree based on the memory traffic.

The most complete work in this area of memory compression is probably made by IBM and ServerWorks which have created a commercially available single chip memory controller called Pinnacle [15] which is compatible with several Intel processors. This compression scheme is called Memory Expansion Technology (MXT) and claims to give compression factors between two and six. The MXT implements the Ziv-Lempel compression algorithm to compress and decompress data sent between main memory and the cache hierarchy.

1.5 Thesis outline

Chapter 2 contains related theory that a reader may find helpful to fully understand the content in the report. The chapter starts with a description of scale-out systems in section 2.1. Following this, a description of CloudSuite and some of its benchmarks in section 2.2. In 2.3, the reader will find theory related to setting up and running virtual machines. Section 2.4 expands on the theory of this thesis by explaining the different compression schemes used throughout this thesis. Section 2.5 describes the hardware used in this project and section 2.6 describes the design flow of taking a hardware description all the way to finished hardware.

In Chapter 3 we present the compression analysis. Section 3.1 contains the set up and the results from the dataset analysis. The set up and results from the CloudSuite benchmark analysis are presented in 3.2 with 3.2.1 and 3.2.2 containing information from the Data analytics and Graph analytics benchmarks. Section 3.2.3 contains a VFT size analysis and section 3.2.4 a VFT sensitivity analysis. A block and page alignment analysis can be seen in 3.2.5 and in 3.2.6 we present an evaluation of the length overhead.

Chapter 4 starts with presenting descriptions of hardware implementations for the BΔI and FPC compression algorithms in sections 4.1 and 4.2. Section 4.3 contains a description of a proof-of-concept system that is implemented on an evaluation and development board. The section includes hardware descriptions, software created as well as necessary changes to the datapath between the processors and memory of the board. Section 4.4 contains an evaluation of the different implementations in terms of speed, required area and power consumption.

A discussion is presented in Chapter 5. In this chapter, decisions made during the thesis is explained such as why conclusions was not drawn from some results. This chapter also includes discussion on the compression behaviour seen in the compression analysis results as well as the implications of this behaviour.

Chapter 6 contains our conclusions of the knowledge gathered during the course of this project. Here we present conclusions drawn from the results of the project, including assessment of Huffman compression compared to other compression schemes such as BΔI and FPC. In 6.1 we make some suggestions on how to continue development of the proof-of-concept hardware is presented, including both optimizations of existing hardware as well as suggestions on new functionality.

2

Theory

This chapter provides the theoretical background that is necessary to understand the compression analysis and implementation of the prototype system. First, scale-out systems are explained. This is followed by information related to the benchmarks' virtual machine set up. Later on, the evaluated compression schemes are explained and lastly, theory related to the implementation of the prototype is presented.

2.1 Scale-out

In today's data centers, scale-out applications such as cloud computing have emerged as the dominating type of application to deliver scalable online services to hundreds of millions of users [16]. Corporations such as Amazon, Google, Facebook, and many more all have invested in their own data center infrastructure to deliver their cloud computing services [17, 18].

Scale-out workloads are workloads that operate on large datasets split across a large number of machines. The workloads typically serve large numbers of completely independent requests that do not share any state and have application software designed specifically for the cloud infrastructure. Inter-machine connectivity is only used for high-level task management and coordination [16].

Scale-out systems and their applications have workloads that are different to workloads of traditional systems. Below is a list with some of the differences between traditional and scale-out systems.

- Scale-out systems execute a much lower rate of floating point operations [19].

- The L1 instruction cache misses are much more frequent in scale-out applications due to their large instruction sets [16, 19].
- Applications on scale-out systems exhibit a low amount of memory-level and instruction parallelism [16].
- The memory bandwidth is not strained in scale-out applications [16].
- The rate of memory instructions versus computational instructions is much higher in scale-out systems [19].
- There is a much higher runtime overhead from the memory management unit (MMU) and a higher amount of page walks for scale-out workloads [17].
- Prefetchers have proven to be unproductive for scale-out workloads [16].
- Ferdman et. al. conclude that most execution time in scale-out systems is dominated by stalls and that stalls in a scale-out system arise due to long-latency memory accesses. Traditional benchmarks such as SPECint and PARSEC do not account for these long-latency memory accesses and thus are not suitable to simulate scale-out systems [16].

Because of the differences in workload from more traditional computing systems, scale-out systems must also be evaluated differently [16]. As such, a benchmarking suite must be chosen with care to emulate, test, and simulate scale-out systems accurately.

2.2 CloudSuite

Several traditional benchmark suites exist today and are easily available from the web. These include, amongst other, the HPCC, PARSEC, and SPECCPU benchmarks. While they are good when serving their purpose, they do not emulate the workload in scale-out systems in a manner that is accurate enough for research in the field of data centers.

To stimulate the research in the field of data centers, CloudSuite was introduced [17, 20]. CloudSuite is a benchmark suite for emerging scale-out applications that represents an application set that is found in the vast majority of today's data centers [21].

The CloudSuite benchmark includes eight different benchmarks that simulate the workload of the most popular types of applications found in scale-out systems. The eight benchmarks are data analytics, data caching, data serving, graph analytics, media streaming, software testing, web search, and web serving. An overview the different benchmark cases of CloudSuite can be found in Figure 2.1.

Data analytics - Uses Hadoop and Mahoop to, with the help of machine learning, extract useful information from vast amounts of human generated data. In this case, Wikipedia pages.

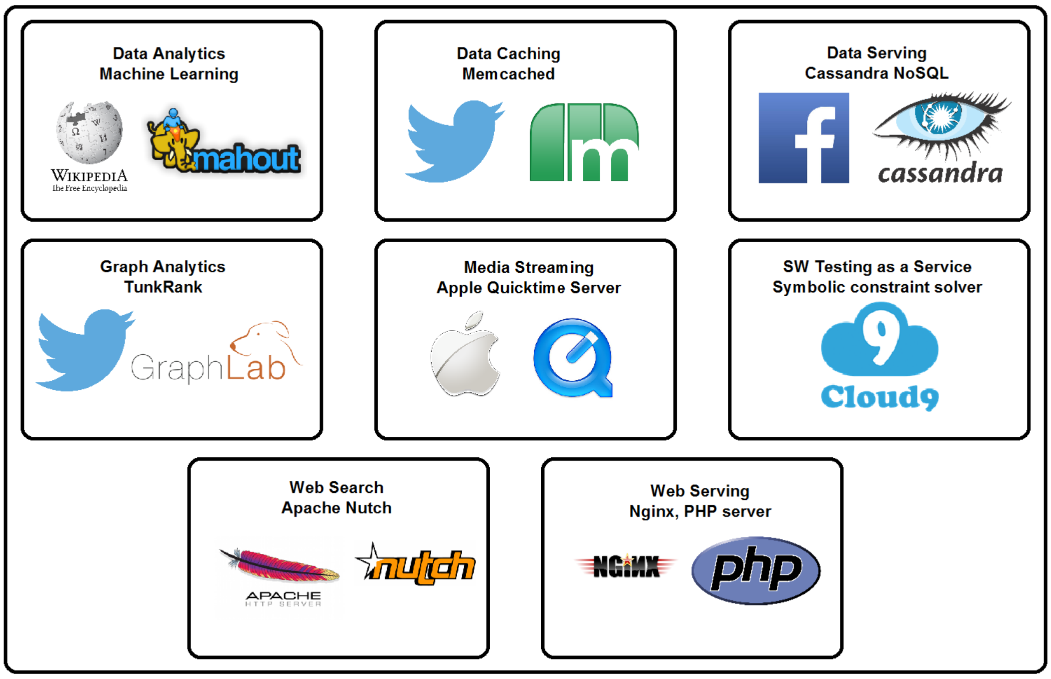

Figure 2.1: Overview of the different set ups for the CloudSuite benchmarking suite [22]

Data caching - With the help of Memcached software, Twitter data is cached into memory from the disk. Twitter users are then emulated to generate realistic memory and disk accesses.

Data serving - NoSQL systems split hundreds of TB of data onto data clusters. Cassandra, an open source NoSQL is used and receives stimulus for read and write requests from the Yahoo! Cloud Serving Benchmark.

Graph analytics - Runs TunkRank on the GraphLab framework to analyse graphs that are distributed across several nodes. Graphs could include social network graphs over users or web graphs.

Media streaming - An Apple Darwin Streaming Server is set up with a dataset that consists of pre-encoded videos. A user is then emulated that downloads the video through the Faban traffic generator.

Software testing - A cloud service offering software testing through Cloud9 to a user is set up. Then a cloud master allocates computing resources among several nodes.

Web search - The Nutch/Lucend is used to set up a web search server that sends requests to independent index serving nodes. Then Faban traffic generator is set up to generate search request to the Nutch/Lucent search server.

Web serving - The Faban traffic generator is used to send traffic in form of requests to an Nginx web server running PHP. The web server accesses a database server running MySQL for accessing images.

2.2.1 Data analytics

In recent years massive increase in human-generated data, often referred to as big data, on the web has made it necessary for automated analytical processing to classify and filter this data [22]. The analytical process can be used to predict user behaviour, opinions, and preferences with an example being to give user recommendations, which the user is likely to appreciate, such as book and movie recommendations. The analytic process can also be used for security reasons with an example being spyware detection on web sites. The analytic process often uses machine learning which has grown to be an important solution for data centers to analyse big data.

CloudSuite's Data Analytics (DA) benchmark is running Apache's machine learning library Mahout which is designed to run data analytics on big data. Mahout runs with Hadoop which is an open-source implementation of the MapReduce paradigm [22]. The DA benchmark uses a text-based dataset of Wikipedia articles. When the benchmark is running, the master node sends Wikipedia documents for classification to the slave nodes. The slave nodes classify the documents using a, previously created, model and send the results back to the master node. The model used by the slave nodes is created from a set of Wikipedia training articles.

2.2.2 Graph analytics

Internet based social network sites such as Facebook, Twitter, and Google Plus have created a high demand on large scale graph analyses. The information obtained from these analyses can be used to provide different services for the user, including friend suggestions and providing the user with relevant news feeds [23].

The Graph Analytics benchmark uses GraphLab to analyse an input graph. GraphLab is a high performance, distributed computation framework that has been developed for machine learning and other data-mining tasks. GraphLab can be configured for processing in two modes, either it uses a single-machine setup or it performs distributed processing. In distributed processing, the graph is distributed across several nodes that communicate for adjacent vertices. The processing is overseen by a master node that collects the results. In a single-machine setup, all computations are made in a single node [23].

Cloudsuite's Graph Analytics benchmark implements the TunkRank algorithm to run on the GraphLab framework. The TunkRank algorithm's input is a graph containing directed edges describing relationships between users in a social network. The algorithm recursively calculates each user's influence on the network based on different user properties such as number of relations [24].

2.3 Virtualization

Virtualization is the act of creating a virtual representation of something. This something could be, but is not limited to, hardware resources, OS, and network devices. In virtualization of a computer system, a new layer of software is added. This software is called the hypervisor or virtual machine monitor (VMM) and its main task is to separate access to the hardware resources in a host system so that the resources may be shared between several OSs. The set of virtual platform resources and interfaces that are presented by the VMM to an OS is an environment that is called a virtual machine (VM) [25, 26].

A VM is an efficient and isolated duplicate of a real machine. To ensure this, the VMM classically must follow three characteristics. First, it must provide an environment that is essentially identical to the original machine. A piece of software should not recognize that it is not executed directly on a host system. Secondly, programs must run with only a minor decrease in speed. This means that a majority of instructions must be executed directly on the host processor. This excludes many emulators that operate mainly in software from being called VMs. Finally, the VMM must be in total control of the hardware resources [25, 26].

There exist many reasons why virtualization of a system may be an appealing option for users. Firstly, OSs and software may be tested on different hardware architectures without the need of the actual hardware. Secondly, since programs within a VM cannot reach outside of its environment, it provides a layer of safety between the program and the host system. Thirdly, a VM can easily be stopped and its state can be examined while stopped. Lastly, VMs can be migrated from one host system to another, sometimes even during execution.

On today's market, there exist several VM environments including Hyper-V, kernel-based virtual machine (KVM), QEMU, Virtual Box, Virtual PC, and many more.

2.3.1 QEMU and KVM

Quick EMUlator (QEMU) is a hosted VM manager that performs fast machine emulation and emulates a set of different CPUs on a range of different hosts, including the Intel x86 architecture. QEMU can run a full system emulation where a system is fully simulated, including a running OS. To translate the target instructions into host compatible instructions, QEMU uses dynamic translation during runtime. Moreover, in contrary to many other emulators with dynamic translation, QEMU is easily portable to other host machines [27].

Kernel-Based Virtual Machine (KVM) is a virtualization infrastructure for the Intel x86 architecture on Linux that uses virtualization extensions to create a VMM capability to Linux. When used with KVM and virtualizing the same architecture that can be found in the host, QEMU can instead of doing a system emulation, create a VM that operates at near-native speeds for the intel x86, server/embedded PowerPC, and S390 guest architectures. This is accomplished by accelerated execution by running guest code directly on the system's hardware. When KVM is not available, or if the guest architecture simulated is different from the host hardware, QEMU falls back on software emulation.

2.4 Compression algorithms

The goal of data compression is to store data in less space than the original representation. There are several ways to achieve this. For instance, by analysing data and searching for certain patterns or relationships within the data, one can represent data using less space than originally required [28]. Decompression is the opposite of compression and is the act of reversing a compression so that the data may be used again.

Data compressing can either be lossless or lossy. In a lossy approach, it is allowed for some information to be lost in the compression and decompression phases to gain a higher compression rate or faster computing times [28]. However, when loss of data is not acceptable, a lossless compression is required.

Data compression and decompression are not free actions, they will impose some sort of overhead to the user. There is often a trade-off between compression rate and speed, since both compression and decompression demand time and hardware resources [12, 28].

2.4.1 Huffman algorithm

The Huffman algorithm is an encoding technique developed by D. A. Huffman which compresses data into a minimal form where values get an encoding depending on the value's frequency. Frequent values get a shorter encoding and infrequent values get a longer encoding. For a given dataset, the algorithm creates a table of the encountered values and their frequency. From this VFT, the Huffman algorithm generates individually defined bit codes for the different values. These bit codes are of variable length depending on the value's frequency [29].

To guarantee that decompression will be possible, no encoding is allowed to be encoded so that it is a prefix to any other encoding. Also, its prefixes may not be used as an encoding for any other values [29]. The Huffman algorithm creates a binary tree structure from the bottom up, using the frequencies to decide which nodes to combine into a parent. Once the tree is built, binary encodings are assigned to the different nodes from the top down.

The following example shows how the Huffman algorithm can be used to compress a simple dataset. The dataset has been evaluated and four values were found in the dataset. We call these four values: A, B, C, and D. The dataset's VFT can be seen in Table 2.1.

Table 2.1: VFT of example dataset

Value	Frequency
A	0.40
B	0.35
C	0.20
D	0.05

From the VFT, the binary tree seen in Figure 2.2 is created. The steps included in the creation of the tree structure are listed below.

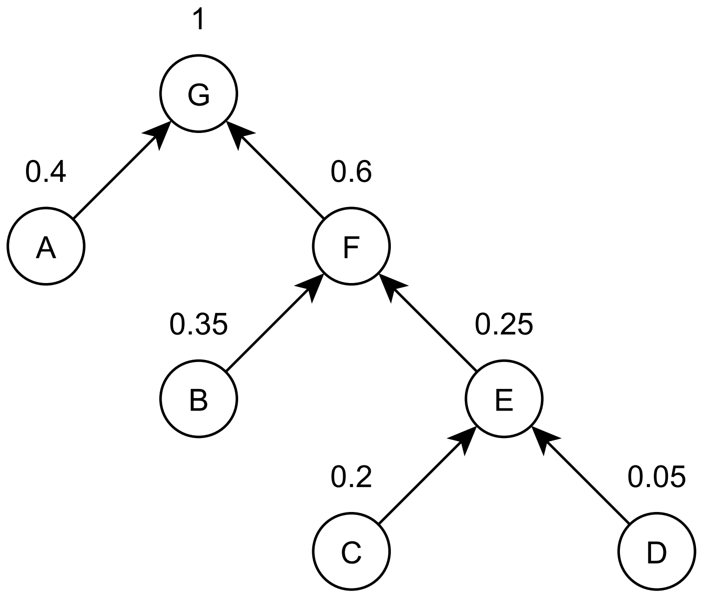

Figure 2.2: Creating a binary tree from the bottom-up. The two nodes with lowest frequencies are combined into a single parent node

1. The two nodes with the lowest frequencies, C and D, are combined to create a new node, E. The frequency of node E is the sum of node C and D. Node C and D are no longer considered when building the rest of the tree.
2. Step one is repeated with the nodes with the lowest frequency, B an E, and the

new node F is created. The B and E nodes are no longer considered.

3. Step one is once more repeated with the nodes A and F, and node G is created. All nodes are now included in a binary tree and the tree is completed.

Once the tree is complete, encodings are assigned to the nodes, starting at the top of the tree. Whenever a step is taken to the left, a zero is added to the prefix and whenever a step is taken to the right, a one is added to the prefix. This process is depicted in Figure 2.3 and a dictionary is generated shown in Table 2.2. As can be seen in Figure 2.3, no encoding is the prefix of another encoding.

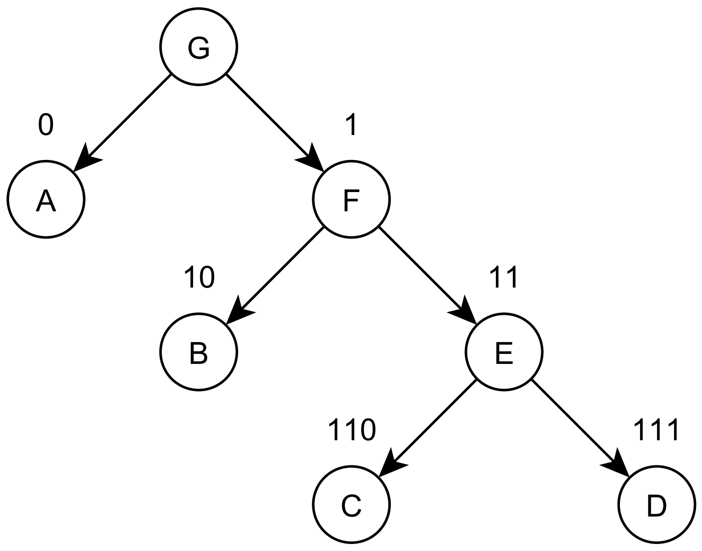

Figure 2.3: Encodings are assigned from the top-down. Every step to a left child represents a zero and every step to a right child represents a one

Table 2.2: New Huffman encodings given to the encountered Values

Value	Encoding
A	0
B	10
C	110
D	111

The example dataset contains four values. Encoding these four values requires a minimum of two bits per value with a traditional encoding. The Huffman encoding created for the example dataset has an encoding that varies from one to three bits per value. The average per value encoding length for the traditional encoding is two bits whereas the avarage per value encoding for the Huffman encoding is $0.4 \cdot 1 + 0.35 \cdot 2 + (0.20 + 0.05) \cdot 3 = 1.85$ bits per value.

2.4.2 ZCA

The Zero-Content Augmented (ZCA) scheme, as suggested by Dusser et al. [14], relies on the following observation made by several people including Dusser et al., Ekman and Stenström, and Alameldeen and Wood: Applications often have a large proportion of their memory containing null and zero data and that this data often have a high spatial locality [7, 13, 14]. Ekman and Stenström for example showed that for some benchmarks, null data or zero blocks make up for up to 75% of the data stored in main memory.

In a memory implementing ZCA, the normal memory is augmented with an adjunct, zero-content memory that only is made up from address tags and validity bits. In this memory, the null blocks or zero blocks are represented using their address tag and N validity bits. These N validity bits can be used to represent several adjacent blocks of zero content and thus exploit the high spatial locality often experienced [14].

When a processor issues a read instruction to a ZCA memory, both the traditional memory as well as the new zero-content memory are searched in parallel and the result is brought to the processor. Upon a hit in the zero content memory, no calculations are needed to convert the fetched data into its original value of zero. Thus, no extra latency is incurred on the processor during accesses to the memory as long as the zero-content memory is kept to a responsible size [14].

A positive side-effect of this no-extra latency property of the ZCA is that it can be applied to all levels of the memory hierarchy if needed. Thus, a zero-content memory can be added in parallel to the L1 cache, L2 cache, L3 cache, and to the main memory, increasing the hit rate of all levels in the memory hierarchy. But it could also only be added to a single level in the hierarchy, such as in the main memory if the designer wishes to limit the number of disc accesses [14].

Since the traditional part of a ZCA memory still remains all the functionality as implemented without the ZCA implementations, it is possible to implement control logic that turns off the zero-content memory when the number of accesses to the zero-content memory reaches below a set threshold. This could be used to reduce a system's energy consumption [14].

2.4.3 BΔI

Pekhimenko et al. have developed a compression technique called Base Delta Immediate (BΔI) that relies on the key observation that the value difference between words in a cache line is usually relatively small [12]. Therefore, one could represent the values within a cache line with the help of a base value and let the different words be represented by a delta value that is the difference between the original word's value and the selected base value. In cases where the dynamic range between the different words of a cache line are small, one could therefore represent the differences, or deltas, with small values. This means that the combined size of the base values and the deltas could be smaller than the original representation.

Pekhimenko et al. have shown that BΔI compression not only compresses cache lines where dynamic differences are small [12], but also many well-known data patterns that are usually found in data systems such as repeated zeros, repeated patterns, and narrow values. This gives BΔI an advantage compared to schemes that only aim to compress some of these well-known data patterns.

To optimize compressibility of this scheme, not only one, but two bases are used for this compression scheme. The first base is selected as the first word in the cache line that is to be compressed and the second base is an implicit base of value zero. This means that BΔI could be used on more data structures such as arrays of structs that contain one big value and one small [12].

To maximize the chance of a successful compression of a cache line, several different word lengths as well as the delta values are tested simultaneously. A cache line can either be treated as eight 8-byte words, sixteen 4-byte words or 32 2-byte words. All these versions are tested and finally, the version that requires least amount of space is selected. This can be done in parallel to ensure no performance loss.

One huge benefit with BΔI compared to many other compression schemes is that decompression of a cache line, which lies on the critical path when fetching data from memory, can be made in parallel. Pekhimenko et al. have shown that compression rates of roughly 1.53 can be expected across different platforms and applications [12].

2.4.4 FPC

Frequent pattern compression (FPC) is a significance-based compression scheme suggested by Almeldeen and Wood that is based on the observation that large portions of stored data do not need all its bits in a word [13]. For instance, an integer value of 150 only needs 8 bits to be represented but is often represented as a 32-bit value. Almeldeen and Wood suggest that a cache line should be compressed by dividing it into 32-bit words. For each word, a series of tests are made to check if the word can be fitted into any of the following cases:

- If the data consist of some 4-bit sign extended data, then the last four bits are stored.
- If the original data consist of a one byte sign extended value, then the last byte is stored.

- If the original data consist of a sign extended of a half-word, then just the half-word is stored.
- If the original data consist of a half-word that is padded with zeros, then the last half-word is stored.
- If the original data consist of two half-words that both consists of a sign extended byte, then the last bytes of those two half-words are stored.
- If the word consists of four repeating bytes, then just one of these bytes is stored.
- In the special case that one or more consecutive zeros are detected, then a value that indicates the number of consecutive zeros is stored.
- If the original data may not fit into any of these categories, then the data must be stored uncompressed.

The word is compressed as described above and the word gets a prefix added. These prefixes should according to Almeldeen and Wood be stored in the cache tags for easy access when decompressing the data [13].

2.4.5 DupDict

The DupDict compression scheme is a scheme proposed by Angelos Arelakis at Chalmers University of Technology that is similar to an earlier scheme proposed by Chen et al. [30]. The key observation made is that words found within memory is often replicated several times and often replicated within a close proximity to the original value. To exploit this fact, DupDict stores a set of indexes instead of the original values.

DupDict performs compression on data in a memory on a block-by-block granularity. Each time a value is replicated within a block, the replicated instance is replaced by an index pointing to the first occurrence of the value within the block. This compression is performed by storing data in three separate fields. The first field is a table where every value that occurs at least once within a data block is stored. The second field is a mask with a single bit for every value within a memory block. This bit indicates whether the value that it maps to is the first occurrence of a value or a replicated occurrence. The last field is a set of indexes that shows the table index found in the first field of the original value that the replicated data was representing.

A description of a hardware implementation of DupDict can be found in a thesis presented by Giannopoulos [31].

2.5 Development platform

To test and evaluate a hardware design, as well as to be able to do a thorough proof-of-concept development, the design should be implemented on actual hardware. A development hardware containing programmable logic is preferably chosen when the hardware description is likely to change several times. Several solutions for development hardware with programmable logic exist on the market today as of-the-shelf hardware. These include, but are not limited to, The Atlys Circuit Board, the MicroBlaze Development Kit board, and the ZedBoard.

2.5.1 ZedBoard

The ZedBoard is an evaluation and development board based on the Xilinx Zynq-7000 All Programmable System on a Chip (SoC). As can be seen in Figure 2.4, the ZedBoard contains a mixture of on-board peripherals, including a 512 MB DDR3 memory, as well as the Zynq XC7Z020 SoC including a processing system (PS) as well as programmable logic that are tightly coupled to each other [32].

Zynq processing system

The PS of the ZedBoard is centered around a dual ARM Cortex-A9 MPCore seen in the top right corner of Figure 2.5. The CPUs can each execute two instructions per cycle and are allowed to perform Out-of-Order execution. Each of the two processors have 4-way set-associative, separate, 32-kB L1 instruction and data caches. The cache line lengths for both L1 caches are 32 bytes or eight words of four bytes each. The L2 cache is 512 kB of shareable 8-way set-associative cache with 32-byte line size used by the PS as last-level cache (LLC) [4].

The processing system uses 32-bit addressing to access data in memory or to access other parts of the processing system as depicted in Table 2.3. By examining this table, it is possible to see from the CPUs point of view, which addresses correspond to which physical location on the processing system. Two address ranges dominate the majority of addresses reachable through 32-bit addressing. The first of these ranges are the addresses that map to system memory, ranging from hexadecimal values of 0000 0000 to 3FFF FFFF. The second range are addresses that map data to the programmable logic through two 32-bit general purpose AXI interconnections, ranging from hexadecimal values of 4000 0000 to BFFF FFFF.

Programmable logic

A Z-7020 FPGA is included on the ZedBoard as the programmable logic. This is where user created hardware may be implemented for testing and development. The PL is

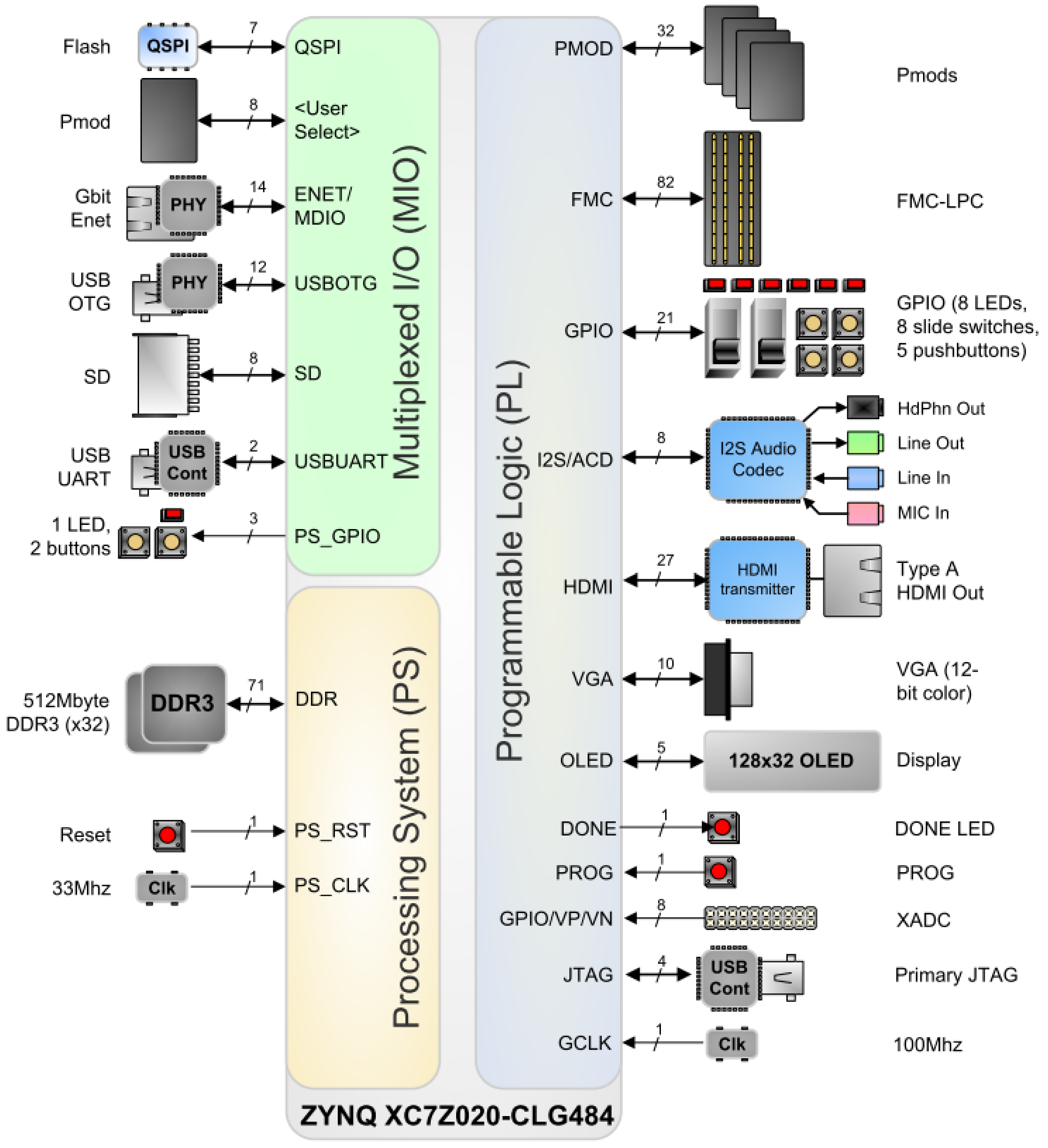

Figure 2.4: Block Diagram of the ZedBoard [32]

accessible from the PS through a series of ports such as general purpose AXI master ports. The characteristics of the FPGA can be found in Table 2.4 [4].

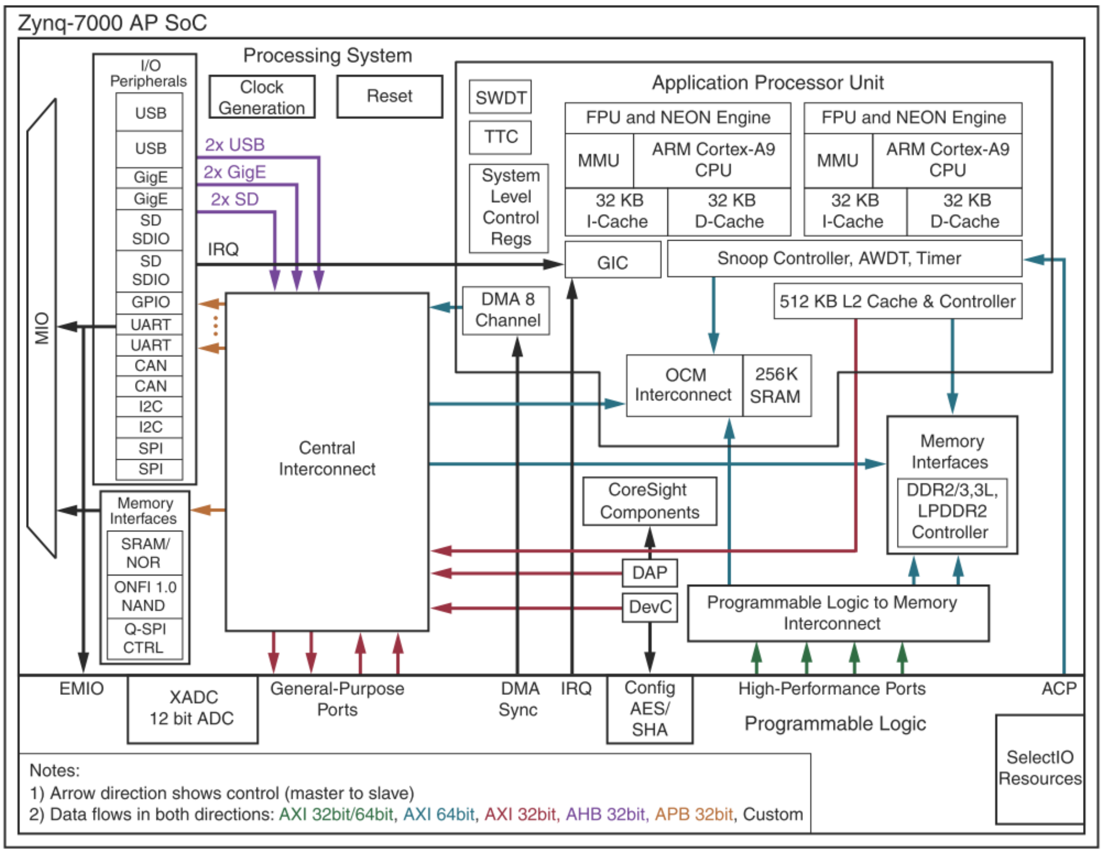

Figure 2.5: Zynq PS block diagram [4]

2.5.2 AMBA AXI4 interface protocol

The AMBA is a set of open standard, on-chip interconnect specification protocols for connection and management of functional blocks on a SoC. The AMBA specification includes several interfaces, including AHB, AXI, and APB.

The AXI4 specification was released by ARM in 2010 and is targeted at high performance systems with short clock cycles. It is implemented as a bidirectional communications protocol that use a master-slave configuration with a two-way handshaking mechanism to send information [33].

As can be seen in Figure 2.6 and Figure 2.7, the AXI4 specification divides communication into five independent transaction channels. These five channels all use a two-way handshaking mechanism to ensure successful transactions. The address and control channel contains information that specifies the destination of following data as well as the nature of the following data transfer. The data channels are used to send data that will be used by the receiver. An answer channel is used only when writing from the master and is used to acknowledge a successful (or unsuccessful) data transfer.

Table 2.3: Address Map

Address Range	From CPUs and ACP	Notes
0000_0000 to 0003_FFFF	OCM	Address not filtered by SCU and OCM is mapped low
0000_0000 to 0003_FFFF	DDR	Address filtered by SCU and OCM is mapped low
0000_0000 to 0003_FFFF	DDR	Address filtered by SCU and OCM is not mapped low
0000_0000 to 0003_FFFF		Address not filtered by SCU and OCM is not mapped low
0004_0000 to 0007_FFFF	DDR	Address filtered by SCU
0004_0000 to 0007_FFFF		Address not filtered by SCU
0008_0000 to 000F_FFFF	DDR	Address filtered by SCU
0008_0000 to 000F_FFFF		Address not filtered by SCU
0010_0000 to 3FFF_FFFF	DDR	Accessible to all interconnect masters
4000_0000 to 7FFF_FFFF	PL	General Purpose Port #0 to the PL M_AXI_GP0
8000_0000 to BFFF_FFFF	PL	General Purpose Port #1 to the PL M_AXI_GP1
E000_0000 to E02F_FFFF	IOP	I/O Peripheral register
E100_0000 to E5FF_FFFF	SMC	SMC Memories
F800_0000 to F800_0BFF	SLCR	SLRC registers
F800_1000 to F880_FFFF	PS	PS System registers
F890_0000 to F8F0_2FFF	CPU	CPU Private registers
FC00_0000 to FDFF_FFFF	Quad-SPI	Quad-SPI linear address for linear mode
FFFC_0000 to FFFF_FFFF	OCM	OCM is mapped high
FFFC_0000 to FFFF_FFFF		OCM is not mapped high

Table 2.4: FPGA Characteristics

Xilinx 7 Series Programmable Logic Equivalent	Artix-7 FPGA
I/O Pin Count	484
Available IOBs	200
Programmable Logic Cells (Approximate ASIC Gates)	85K Logic Cells (1.3M)
Look-Up Tables (LUTs)	53,200
Flip-Flops	106,400
Extensible Block RAM (# 36 Kb Blocks)	560 kB (140)
Programmable DSP Slices (18x25 MACCs)	220
Peak DSP Performance (Symmetric FIR)	276 GMACs

Figure 2.6 and Figure 2.7 also show required timing behaviour of the AXI protocol. The address and control channel is used first to set up a data transfer. Once both master and slave have agreed on a set of characteristics for the following data transfer, data is sent in the desired direction. If the data transfer was a write action from the master, then the slave also ends with a write answer once all data have been transfered.

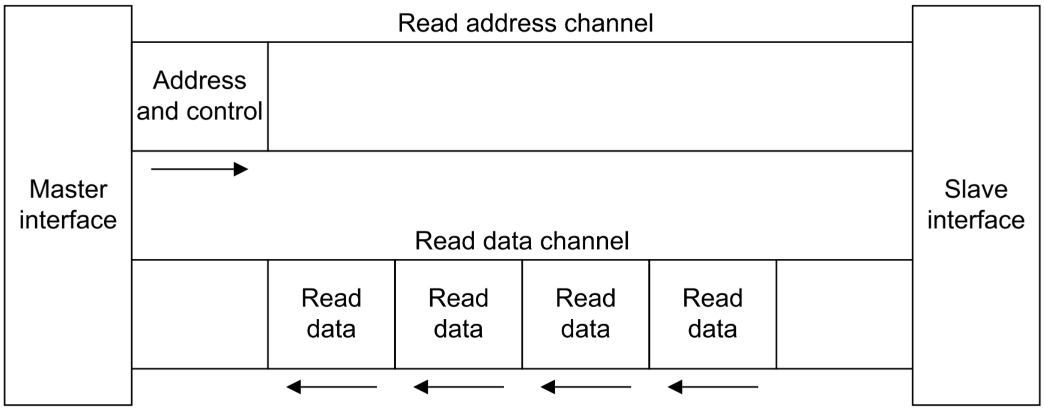

Figure 2.6: Channels and timing used to read data from slave to master [33]

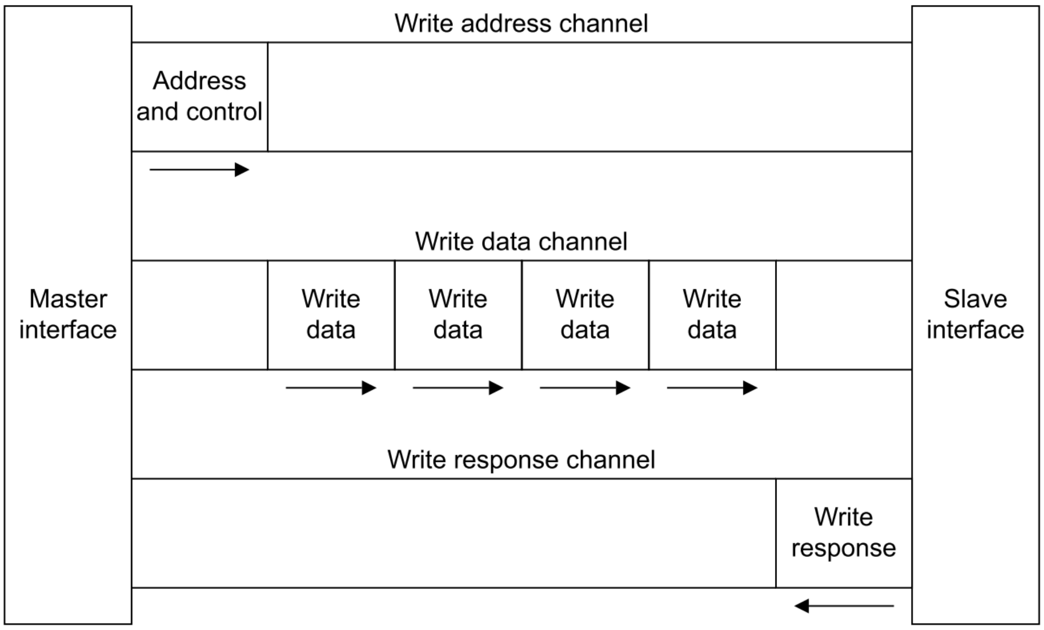

Figure 2.7: Channels and timing used to write data from master to slave [33]

2.6 Design flow

To turn a desired circuit description from an abstract description such as hardware description language (HDL) code into a physical circuit is not a simple task. It is a process that involves several steps which eventually produce the aspired end product. The design flow for FPGA devices and ASIC devices is similar but differs slightly on some points. These differences will be pointed out in the following text [34].

The starting point of the design flow is usually a HDL code description of the desired circuit. This description is usually made in VHDL or Verilog, although higher-level descriptions are becoming increasingly common. At this point, the designer is responsible to make sure that the design only uses constructs that are possible to implement on a physical circuit. If it is possible to implement the HDL code onto a physical circuit, then we call the source code register transfer level code (RTL-code) and say that it is synthesizable. An example of a construction that is unsynthesizable is a circuit that is clocked on both the rising edge and the falling edge of a signal. Such a construction would have to be removed by the designer before proceeding to next step [34].

Next, the RTL-code is synthesized. This means that the RTL-code is mapped onto a description of the actual hardware gates that will be used for the design. To do this mapping, a synthesis tool must also have information about what hardware the tool has at its disposal. For FPGA designing, this would mean information about which FPGA the design should be implemented on. For design of an ASIC, this would include a library of available gates that the synthesis tool may use. Design constraints are also required and typically include information such as expected operating frequencies and pin mapping [34].

During synthesis, a synthesis tool usually starts by performing architecture-independent optimizations before the actual technology mapping happens. These optimizations include techniques such as propagating constant values through the design, operations sharing hardware and redundancy removal. When technology mapping then starts, the tool focuses on performing architecture-specific optimizations such as using on-chip multipliers or creating adders with dedicated carry-chains. Once synthesis is completed, a gate netlist is created as output that describes the needed hardware [34].

Once the gate netlist has been produced, it will need to be placed and routed. In the place and route phase, location for each element in the mapped netlist will be determined. This will be followed by placement-driven optimizations. Once gates are placed, wires and signals are routed throughout the system. On an FPGA, the routing is more restricted than on an ASIC as only prefabricated routing resources may be used while the ASIC may create routing however it desires [34].

When place and route is completed, so is the design flow when designing ASICs. An ASIC design is now ready to be implemented as a physical circuit. An FPGA however, will need one more step before a hardware design can be implemented [34].

The final step in the design flow of an FPGA device is to generate a bit-stream and program a suitable target device with it. To generate the bit-stream, the placed and routed design from the previous step is used as an input. The FPGA is then programmed with the generated bit-stream, creating the desired hardware design on the FPGA [34].

3

Compression analysis

This chapter contains a detailed data compression analysis of several different datasets as well as two benchmarks from CloudSuite. The algorithms used in the compression analysis are Huffman, Huffman+LO, BΔI, FPC, and DupDict. Huffman+LO is Huffman compression with length overhead (LO). The LO is four bits added in front of every value to indicate the length of the following code word. The four bits allow for code lengths from one to sixteen bits and are intended to be used by the underlying hardware so that the values can easily be separated and decompressed in parallel.

During the compression analysis, parsed binary representation of datasets and benchmark memory samples were split into 4096 byte pages which were analysed one at a time with the compression algorithms.

Huffman compression relies on a value frequency table (VFT) to create the encoding for a tree structure where values closer to the root have higher expected frequency. In a computer system employing Huffman compression one has to sample the memory traffic over some time to create the VFT. When analysing the datasets and the memory samples from the benchmarks it was not possible to use this methodology. The methodology used in the compression analysis was to generate the VFT from the whole binary representation of the dataset or the benchmark memory sample. This will create the best possible VFT and this methodology will be referred to as an optimal VFT. The VFT sizes used in the compression analysis were 128, 1024, and 8192 for the datasets and 128, 1024, and 4096 for the benchmarks' memory samples.

The metric used to display the compressibility of the different compression algorithms is compression factor (CF). Equation 3.1 shows how the CF is calculated. In this thesis the CF for Huffman and Huffman+LO are always calculated for non-null 64 byte blocks and the Huffman CF displayed in tables and figures is the Huffman CF when an optimal VFT has been used. The Huffman CF is often used as a reference to show the impact on CF of different restrictions required in a hardware implementation of Huffman compression.

$$CF = \frac{Uncompressed\ size}{Compressed\ size} \tag{3.1}$$

3.1 Datasets

The dataset analysis evaluated two types of datasets: text-based and number-based. The text-based datasets were parsed with UTF-8 encoding, converted to binary representation, and stored as a file. The number-based dataset was read as consecutive values with four-byte granularity, converted to binary representation, and stored as a file. The datasets containing several files were concatenated into a single binary file.

The dataset analysis comprises two Twitter datasets, one Google Plus dataset, one Memetracker dataset, and two Wikipedia datasets. Both Twitter datasets are text-based with the first dataset containing over two million tweets from the Grammy awards 2013 and the second over eight million tweets from Superbowl 2014 [35, 36].

The first Wikipedia dataset (Wikipedia_NB) originally contained over 30 000 files including binary files such as images. When the Wikipedia_NB dataset was parsed, binary files were excluded and the remaining files were parsed as text-based files. The second Wikipedia dataset (Wikipedia_L) is based on a 50 GB XML file containing Wikipedia articles [37]. The Wikipedia_L dataset contains the first 10 GB of data from the XML file and was parsed as text-based. The Wikipedia_L dataset was limited to 10 GB due to time and resource limitations.

The Google Plus dataset consists of circles, profiles, and ego networks from the Google Plus social network. The dataset contains numbers and was parsed as a number-based dataset.

Memetracker tracks frequent quotes and phrases from news websites and blogs and the text-based Memetracker dataset contains a large number of meme-messages. The datset entries contain information such as text, URL of the author, and a time stamp.

3.1.1 Dataset analysis results

In the dataset analysis, three different VFT sizes were used for the Huffman compression: 128, 1k, and 8k entries. Table 3.1 shows the CF for the different compression algorithms. The number-based Google Plus dataset has higher CF for all algorithms compared to the text-based datasets. The BΔI, FPC, and DupDict compression factors are consistently low for all datasets. The datasets do not contain any null-blocks due to the parsing.

Table 3.1: Compression results for the datasets

	Huffman			**Huffman+LO**			**BΔI**	**FPC**	**Null Blocks**	**DupDict**
Data Set	**128**	**1K**	**8K**	**128**	**1K**	**8K**				
Google Plus	2.076	2.172	2.232	1.378	1.407	1.497	1.095	1.258	0.00%	1.260
Memetracker	1.136	1.317	1.641	1.043	1.149	1.367	1.001	1.000	0.00%	1.009
Twitter Grammy	1.296	1.466	1.790	1.143	1.248	1.465	1.000	1.000	0.00%	1.052
Twitter Superbowl	1.422	1.598	1.891	1.257	1.344	1.532	1.000	1.000	0.00%	1.146
Wikipedia_NB	1.231	1.515	2.057	1.109	1.284	1.637	1.000	1.005	0.00%	1.023
Wikipedia_L	1.085	1.241	1.564	1.019	1.098	1.312	1.000	1.002	0.00%	1.015

3.2 Cloudsuite benchmarks

The DA and Graph Analytics (GA) benchmarks were set up to run on VMs. The benchmark VMs were set up with the QEMU hypervisor and utilised KVM which made it possible for the VMs to maintain near-native performance. The computer running the benchmark VMs, the host, had some of its resources allocated to the benchmark VMs. Table 3.2 shows the configuration and resources allocation for the host and the benchmark VMs.

Table 3.2: System details for host and benchmark VMs

	Host	**GA**	**DA master/slave**
Linux OS version	Mint 17 Cinnamon 64-bit	Mint 17.1 Rebecca 64-bit	Ubuntu 14.04 LTS 64-bit
Linux kernel	3.13.0-24-generic	3.13.0-37-generic	3.13.0-39-generic
Processor	Intel Core i7-4790 CPU @ 3.60GHz x 4	Intel Core Processor (Haswell)	Intel Core Processor (Haswell)
Threads	8 (two threads per core)	4	4
Memory	8 GB	5 GB	2 GB / 5 GB

In order to test the compressibility of the memory used by the benchmarks we extracted the benchmarks' memory and analysed its content. Figure 3.1 shows the set up for the DA benchmark. The benchmark was set up using two VMs: a master and a slave. For the DA benchmark we analysed only the slave VM's memory. The reason for this is that the slave VM was running the memory intensive operations and thus is more interesting in the scope of this thesis.

The DA benchmark is running several Java virtual machines, called nodes, within the benchmark. The number of nodes depends on the Hadoop framework configuration but there is always at least a DataNode and a TaskTracker node running in a slave machine. Together with the DataNode and TaskTracker there is a number of Map and Reduce nodes running, called child nodes, which perform the actual work.

The compressibility analysis of the benchmarks' memory content consists of two phases: memory extraction and memory analysis. The memory extraction phase was carried out while the benchmark was running and the memory analysis was carried out afterwards.

The two phases are illustrated with dashed-boxes in Figure 3.1. In the memory extraction phase the VM's memory was extracted to a file with the help of the VM managing tool Virsh. At the same time as the VM's memory was extracted, the virtual-to-physical address translation was also stored.

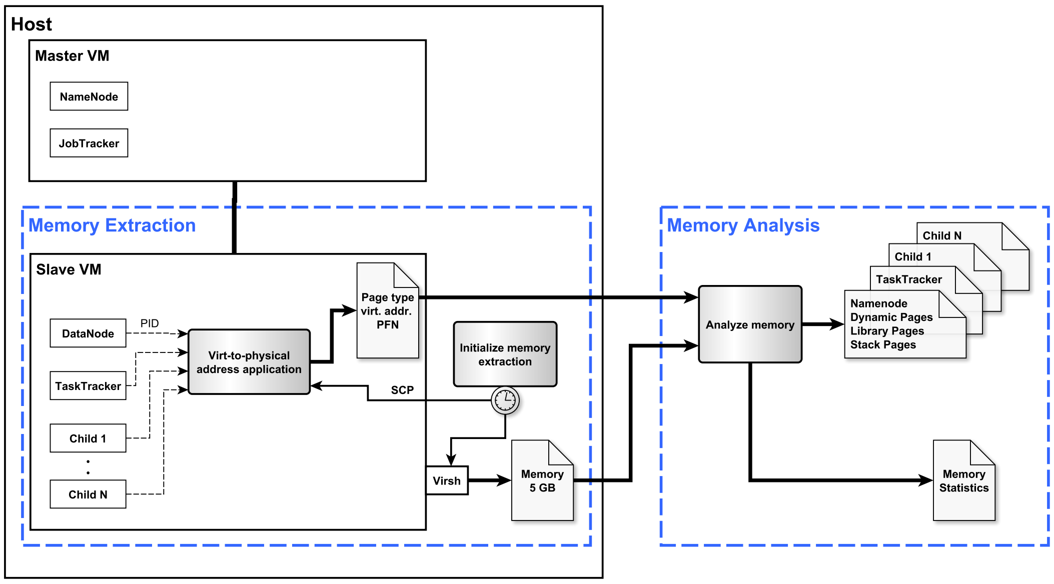

Figure 3.1: Compression analysis overview

Figure 3.2 shows the setup for the virtual-to-physical address translation performed by the Virt-to-physical address application. The initial step of the application was to fetch the process IDs (PIDs) of the Hadoop nodes running at that instance. The PID, for a running Hadoop process, was used to get the virtual address range and the page type from the /proc/PID/maps file. While analysing the maps file, pages were classified into three categories: dynamic pages (including heap), library pages, and stack pages. A process' virtual address range is at least the same size as the physical address space, but the virtual address space is usually substantially larger. This means that only a small portion of the virtual address range is stored in the physical memory at any given time.

In order to get the physical pages used by a process one needs to render the virtual-to-physical address translation. This was possible using the /proc/PID/pagemap file. If a virtual page exists in the physical memory its page frame number (PFN) was stored into a file.

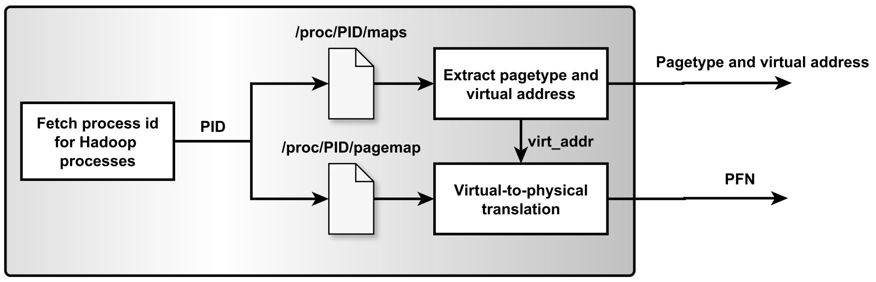
/proc/PID/maps
Fetch process id for Hadoop processes
PID
/proc/PID/pagemap
Extract pagetype and virtual address
virt_addr
Virtual-to-physical translation
Pagetype and virtual address
PFN

Figure 3.2: Virtual-to-physical address translation application

In the second phase of the compressibility analysis, the memory analysis phase, binary images were created corresponding to the memory used by the Hadoop processes running when the VM's memory was sampled. Each process had three images created, one for each page type, by concatenating pages from the memory sample. Figure 3.3 shows an example of how images were created for dynamic pages, library pages, and stack pages. The input to the Analyze memory application is a range of PFN for each page type. The dynamic page ranges from 100-102 and 106-109. These seven pages are selected in the application's second input file, the 5 GB memory sample. The selected pages were concatenated into an image. This methodology also applies to library and stack pages and the application output is three images per process.

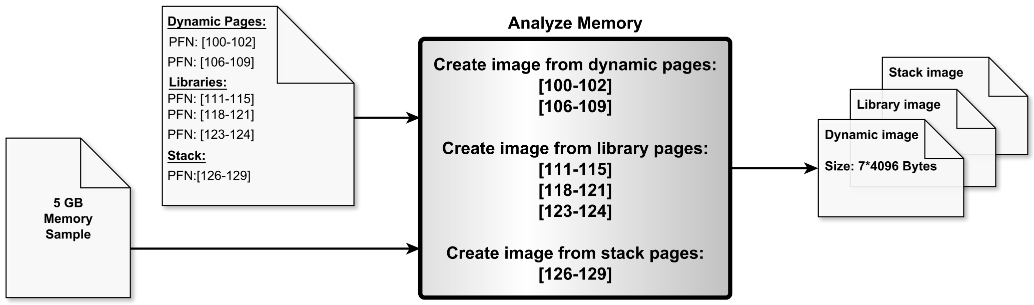

Figure 3.3: Memory extraction example

In this section the DA benchmark is used to describe the methodology used to extract and analyse the memory used by the two benchmarks. The DA benchmark is used because its set up is more complicated than the set up for the GA benchmark. The DA benchmark set up use two VMs with multiple processes running in the slave VM whereas the GA benchmark use only one VM with a single process, the TunkRank process. Even though the set up of the two benchmarks differ, the methodology of the memory extraction and analysis remains the same.

While analysing the memory samples taken from the VMs we discovered that one gigabyte of memory was left mostly unused. Figure 3.4 shows how the memory has changed over five hours of running the GA benchmark. Each page in a five-gigabyte memory sample is represented with a bar in the figure. Black bars are pages that have changed value over the five hours of running the benchmark, red bars are null pages that have not changed, and white bars are non-null pages that have not changed value. As the figure depicts, the pages between the third and fourth gigabyte in the memory sample are almost entirely null-pages that have not changed over the five hours of running the benchmark. We found that the source of this observation is related to the x86-64 architecture. The x86-64 architecture has a one-gigabyte region, located at the high-end of the physical address space, dedicated to I/O [38].

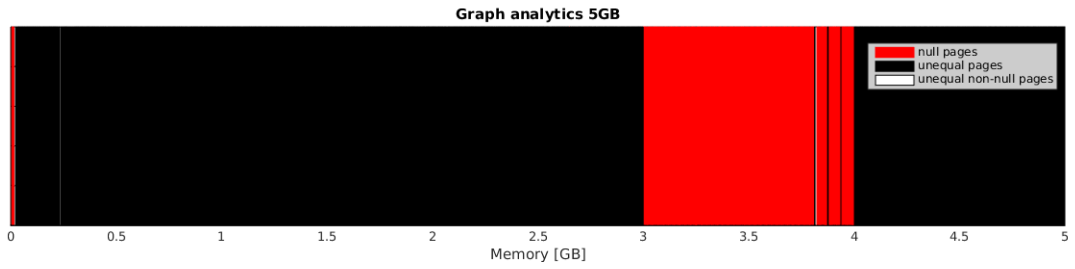

Figure 3.4: Memory changes over a five hours of running the GA benchmark

3.2.1 Data analytics

The DA benchmark was set up using two VMs: one master and one slave. The master VM running the NameNode and the JobTracker nodes and the slave VM running the DataNode, TaskTracker, Map, and Reduce nodes. The slave VM was set up to run two Map nodes and one Reduce node. The benchmark was running with a large Wikipedia XML file, around 50 GB, as input and was running for about 90 minutes before finishing. The benchmark was set to restart, with the same input file, when the benchmark finished running.

While the benchmark was running, four 5 GB memory samples of the slave VM's memory were taken. In Table 3.3 the sample time as well as sample names are shown. The sample names are used throughout the report in tables and graphs.

Table 3.3: DA memory sample details.

Sample number	Sample name	Sampled at time
1	DA_z0	1 minute
2	DA_z1	1 hour
3	DA_z2	6 hour
4	DA_z3	24 hour

Memory analysis

When the first sample was taken, DA_z0, the slave VM was running two Map tasks and no Reduce tasks. Along with the two Map tasks, Child1 and Child2, the slave VM was running the TaskTracker and the DataNode tasks. Table 3.4 shows the tasks running and their memory usage when the memory dump was taken. The two Map tasks consume a combined value of 95.94% of the total memory used by the benchmark, whereas the memory consumed by the TaskTracker and DataNode is much less, 2.46% and 1.6%, respectively. The table also shows that 98.85% of the pages in the memory sample are dynamic pages.

When the DA_z0 sample was taken there was no Reduce task running. However, when the DA_z1 and DA_z2 memory samples were taken, two Map tasks and one Reduce tasks were running. The memory usage of the Reduce task was much less than the Map tasks, less than 2% of the total memory usage. Detailed memory information for the DA_z1 and DA_z2 memory samples can be seen Appendix B.

The observations that the Map tasks consume at least 90% of the total memory and that more than 95% of the pages are dynamic pages are also true for the DA_z1, DA_z2, and DA_z3 memory samples.

Table 3.4: DA_z0 node memory details.

Node	Page type	Pages	% of total pages	Physical memory size (MB)	Swap size (MB)
TaskTracker	DYNAMIC	25881	2.28%	15.18	27.33
	LIBRARIES	1513	0.13%	3.81	1.61
	STACK	559	0.05%	1.80	0.12
DataNode	DYNAMIC	16377	1.44%	82.54	23.47
	LIBRARIES	1322	0.12%	4.29	1.83
	STACK	469	0.04%	1.48	0.81
Child (Map)	DYNAMIC	531120	46.77%	1643.86	531.61
	LIBRARIES	3147	0.28%	8.64	4.25
	STACK	1405	0.12%	5.33	0.42
Child (Map)	DYNAMIC	549246	48.36%	2047.70	202.01
	LIBRARIES	3147	0.28%	8.53	4.36
	STACK	1477	0.13%	5.51	0.54

Table 3.5 shows the sum of the nodes' dynamic, library, and stack pages for each memory sample. A large percent of the library pages are shared between the tasks and very few of the dynamic and stack pages are shared. The table also shows that for the DA_z0, DA_z1, and DA_z2 memory dumps, around 50% of the dynamic pages are null pages. For the fourth sample, DA_z3, proportion of null pages was less, 21.54%.

Table 3.5: DA memory details

Sample	Page type	Pages w/o duplicates	% Duplicates	% of total pages	Physical memory size (MB)	Swap size (MB)	% null pages
DA_z0	DYNAMIC	1114718	0.70%	99.30%	3787.92	777.97	45.43%
	LIBRARIES	3927	56.98%	0.35%	10.54	5.54	0.51%
	STACK	3900	0.26%	0.35%	14.09	1.88	2.82%
DA_z1	DYNAMIC	1049458	0.00%	99.25%	3582.83	715.75	56.41%
	LIBRARIES	3974	64.12%	0.38%	8.31	7.96	5.12%
	STACK	3953	0.00%	0.37%	8.72	7.47	4.13%
DA_z2	DYNAMIC	1001449	0.01%	95.80%	3415.80	686.13	56.47%
	LIBRARIES	3678	58.96%	0.35%	7.58	7.48	2.54%
	STACK	40247	0.00%	3.85%	135.93	28.92	80.14%
DA_z3	DYNAMIC	852570	0.27%	97.59%	2797.38	694.74	21.53%
	LIBRARIES	1888	55.48%	0.22%	4.70	3.03	31.62%
	STACK	19159	0.01%	2.19%	30.03	48.45	61.98%

Compression analysis

The VFT coverage improves when increasing the size of the VFT as shown in Table 3.6. The VFT coverage for VFT sizes from 128 to 32k for the four memory dumps can be seen in Appendix A. Appendix A also shows the frequency of the 25 most frequent values in the VFT. The by far most frequent value in the memory sample is zero. The percent of zero values ranges from 16.81% to 32.79% of the total number of values in the memory sample with null pages excluded.

Table 3.6: VFT coverage (including null blocks)

Sample	128 entries	1k entries	4k entries
DA_z0	83.39%	86.65%	87.99%
DA_z1	88.33%	90.06%	90.71%
DA_z2	88.25%	90.91%	91.67%
DA_z3	77.65%	82.37%	83.69%

Table 3.7 shows the compression results for the different compression algorithms. Huffman compression gives the best compression results and will still give better results than the other compression algorithms when the LO is considered. Table 3.8 shows the loss in compression when Huffman with the LO is considered. The loss in compression is between 21% and 26% and a slight increase in compression loss can be detected with a larger VFT size.

Figure 3.5 and Figure 3.6 illustrate the CF for the different compression algorithms. The first graph shows the Huffman CF and the second graph shows the Huffman+LO CF.

Table 3.7: DA compression results

Application	Huffman 128	Huffman 1K	Huffman 4K	Huffman+LO 128	Huffman+LO 1K	Huffman+LO 4K	BΔI	FPC	Null Blocks	DupDict
DA_z0	2.388	2.597	2.675	1.878	1.995	2.025	1.457	1.763	3.53%	1.663
DA_z1	2.522	2.680	2.719	1.960	2.044	2.062	1.414	1.760	8.46%	1.665
DA_z2	2.500	2.765	2.819	1.938	2.082	2.107	1.243	1.743	3.58%	1.538
DA_z3	2.535	2.783	2.854	1.952	2.092	2.126	1.469	1.801	1.81%	1.696

Table 3.8: Loss in compression for Huffman+LO with respect to Huffman.

Sample	128 entries	1k entries	4k entries
DA_z0	21.36%	23.18%	24.30%
DA_z1	22.28%	23.73%	24.16%
DA_z2	22.48%	24.70%	25.26%
DA_z3	23.00%	24.83%	25.51%

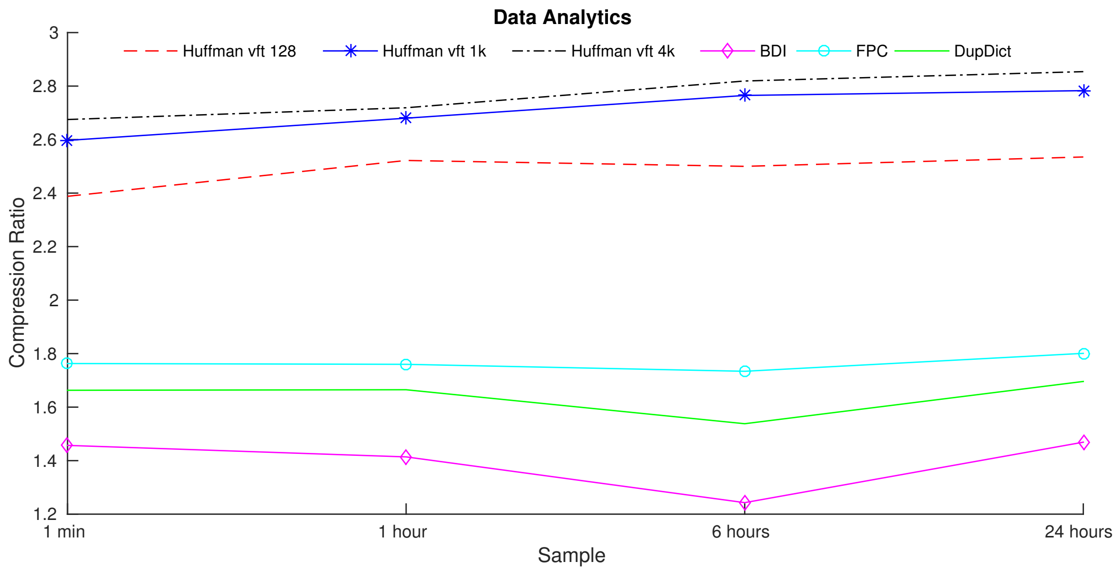

Figure 3.5: DA compression results.

3.2.2 Graph analytics

The GA benchmark was running a single VM set up with the GraphLab framework software. On top of this framework, the TunkRank software algorithm was built.

The GA benchmark is very memory intensive and requires a large memory to run input

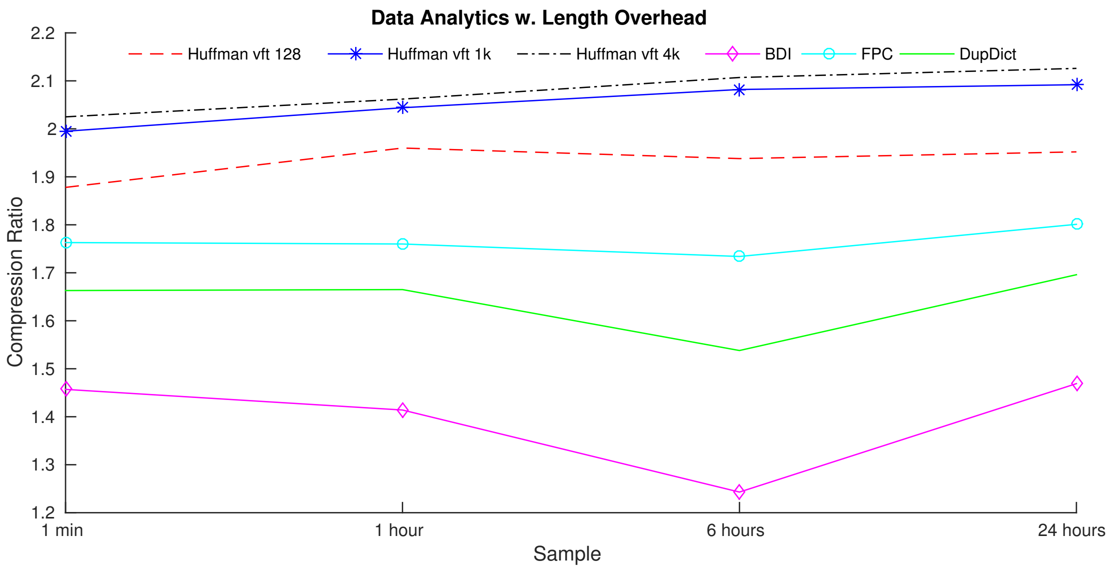

Figure 3.6: DA compression results - Huffman compression with LO

data with moderate size. Since the VM running the benchmark had limited resources, the VM's swap space was set to 20 GB. The GA benchmark was set up to analyse two different input graphs. The first graph was a 4.3 GB file containing a subset of a Twitter relationship graph between Twitter users. The second graph was a 9 GB file based on relationships between Google Plus users.

The Twitter input graph took more than 24 hours for the GA benchmark to analyse while the Google Plus graph finished within two hours. Four memory samples were taken while running the Twitter graph, and three memory samples were taken while running the Google Plus graph. Table 3.9 and Table 3.10 show the memory sample details.

Table 3.9: GA Twitter dataset details

Sample number	Sample name	Sampled at time	Benchmark phase
1	GA_z0	1 minute	Read phase
2	GA_z1	1 hour	Analyse phase
3	GA_z2	6 hour	Analyse phase
4	GA_z3	24 hour	Analyse phase

Twitter graph

Cloudsuite's GA benchmark comes with a 26 GB Twitter graph containing ordered numerical data. These numerical data represents the edges of a graph showing the

Table 3.10: GA Google Plus dataset details

Sample number	Sample name	Sampled at time	Benchmark phase
1	GA_g_0	30 minutes	Read Graph
2	GA_g_100	60 minutes	Analyse Graph
3	GA_g_130	90 minutes	Analyse Graph

relationships between a large portion of active Twitter users in 2009 [39]. The graph consists of two columns of user IDs. The two columns indicate that the user ID in the first column follows the user ID in the second column. Because of privacy reasons, the real user IDs have been hidden and instead, aliases are used starting from user ID 1 and going upwards.

The TunkRank algorithm was designed with this graph in mind. Moreover, when using the full 26 GB twitter file, the Tunkrank algorithm requires roughly 50 GB of memory and takes several days to execute on the set up benchmark VM. Because of the size of the input file's relation with the memory usage, a 4.3 GB subset of the original graph was created by extracting every sixth row in the graph input file.

Google Plus graph

A 1.3 GB graph containing user relationships between Google Plus users was also analysed. This graph did not come with the GA benchmark and had three problems that had to be addressed for TunkRank to be able to analyse the graph. Firstly, the edges had to be ordered in numerical order, otherwise the TunkRank would not execute. Secondly, the graph included descriptions of the same edges several times. Lastly, the user IDs were represented using 21-digit numbers which cause problems with TunkRank due to their sheer size.

To solve these problems, the graph had to be reconfigured. The file had its rows sorted in numerical order, duplicated rows were removed, and aliases were given to all user IDs in the file. Starting from the lowest user ID which was given the value of one and then assigning successive numbers to the rest of the user IDs. Parsing the graph accordingly decreased its size to 173 MB. This caused TunkRank finishing its execution in mere seconds.

To increase file size and thus execution time, the data in the parsed file was duplicated with a fixed offset value given to all duplicated user IDs with the result being several independent copies of the original graph. By creating 40 duplicated graphs in the file, it was possible to create a 9 GB file describing Google Plus users.

These changes lessen the relevance of the input data. The input, while big, have a low complexity. Also, the file consists of several small, independent graphs and it was shown by Cha et al. that more than 90 percent of nodes in a social network are connected [39].

Memory analysis

While running the GA benchmark we saw that the benchmark has two phases. In the first phase, the benchmark reads the input graph from memory. During this phase, the first memory sample was taken for both input graphs. When studying Table 3.11 and Table 3.12, we see that the first samples of the Twitter graph (GA_z0) and the Google Plus graph (GA_g_0) have a high amount of memory usage and that they are almost exclusively dynamic pages.

Table 3.11: Graph Analytics, Twitter

Sample	Page type	Pages	% of total pages	Physical memory size (MB)	Swap size (MB)	% null pages
GA_z0	DYNAMIC	1162415	99.99%	3886.40	874.73	9.57%
	LIBRARIES	97	0.01%	0.34	0.05	0.00%
	STACK	8	0.00%	0.03	0.00	0.00%
GA_z1	DYNAMIC	1070592	99.98%	3596.68	788.46	0.07%
	LIBRARIES	218	0.02%	0.71	0.18	0.00%
	STACK	26	0.00%	0.07	0.04	0.00%
GA_z2	DYNAMIC	1082868	99.98%	3676.31	759.12	0.06%
	LIBRARIES	200	0.02%	0.65	0.17	0.00%
	STACK	16	0.00%	0.03	0.04	0.00%
GA_z3	DYNAMIC	1089731	99.98%	3693.75	769.79	0.01%
	LIBRARIES	194	0.02%	0.55	0.24	0.00%
	STACK	16	0.00%	0.03	0.04	0.00%

During the benchmark's second phase, the graph analyse phase, the TunkRank algorithm starts to analyse the input graph. All the remaining memory samples were taken during this phase. The memory statistics are consistent for the remaining memory samples with the exception on the number of null pages that decrease in the case where the Twitter graph was used.

From Table 3.11 and Table 3.12 we see several patterns that hold true for all memory samples. The memory content is almost exclusively dynamic pages. Null pages are almost non-existent where the exception is the first memory sample when using the Twitter graph as input.

Table 3.12: Graph Analytics, Google plus

Sample	Page type	Pages	% of total pages	Physical memory size (MB)	Swap size (MB)	% null pages
GA_g_0	DYNAMIC	1152984	99.99%	3816.25	906.38	0.07%
	LIBRARIES	54	0.00%	0.19	0.03	0.00%
	STACK	4	0.00%	0.02	0.00	0.00%
GA_g_100	DYNAMIC	1087365	99.98%	3639.64	814.21	0.06%
	LIBRARIES	195	0.02%	0.58	0.22	0.00%
	STACK	21	0.00%	0.07	0.02	0.00%
GA_g_130	DYNAMIC	1157351	99.98%	3884.27	856.24	0.05%
	LIBRARIES	181	0.02%	0.52	0.22	0.00%
	STACK	16	0.00%	0.05	0.02	0.00%

Compression analysis

Table 3.13 and Table 3.14 summarize the VFT coverage for the Twitter and Google Plus graphs. The coverage for all VFT sizes from 128 to 32k can be seen in Appendix A. The first memory dump taken for both the Twitter and Google Plus graph, in the benchmark's graph reading phase, has much lower VFT coverage than the later samples from the benchmark's graph analyse phase. When comparing the VFT coverage between the Twitter graph and the Google Plus graph, one sees that the VFT coverage is substantially higher for the Twitter graph. Appendix A also shows the 25 values with the highest frequency in the VFTs. When examining the VFTs for the memory dumps with very high VFT coverage, one can see that they contain a few values with very high frequency. The memory dumps GA_z0, GA_z1, and GA_z2 contain 45.72%, 43.61%, and 44.98% zero values, respectively.

Table 3.13: GA Twitter dataset VFT coverage (including null blocks)

Sample	128 entries	1k entries	4k entries
GA_z0	31.34%	35.66%	37.87%
GA_z1	70.16%	71.85%	72.60%
GA_z2	67.27%	69.42%	70.56%
GA_z3	65.88%	68.05%	69.19%

Table 3.15 shows the compression results for the Twitter graph. Most noticeable is the leap in compression seen when going from the graph reading phase to the graph analysing phase. Figure 3.7 depicts the compression results for Huffman and Figure 3.8 depicts the compression results for Huffman+LO.

Table 3.16 shows the loss in compression for Huffman+LO with respect to Huffman.

Table 3.14: GA Google Plus dataset VFT coverage (including null blocks).

Sample	128 entries	1k entries	4k entries
GA_g_0	0.28%	1.01%	2.76%
GA_g_100	10.87%	11.76%	13.64%
GA_g_130	7.16%	8.05%	10.03%

Table 3.15: Compression results for GA with the Twitter dataset.

	Huffman			Huffman+LO			BΔI	FPC	Null Blocks	DupDict
Application	128	1K	4K	128	1K	4K				
GA_z0	1.255	1.299	1.315	1.136	1.168	1.179	1.886	1.013	0.15%	1.893
GA_z1	2.725	2.789	2.805	2.155	2.189	2.198	1.546	1.666	0.80%	2.012
GA_z2	2.552	2.625	2.641	2.062	2.102	2.109	1.320	1.635	0.11%	1.942
GA_z3	2.480	2.547	2.567	2.016	2.052	2.062	1.301	1.925	0.07%	1.925

The loss in compression for GA_z1 -GA_z3 range from 18.71% to 21.64% and the same pattern applies as for DA, the loss in compression is slightly increased with a larger VFT size.

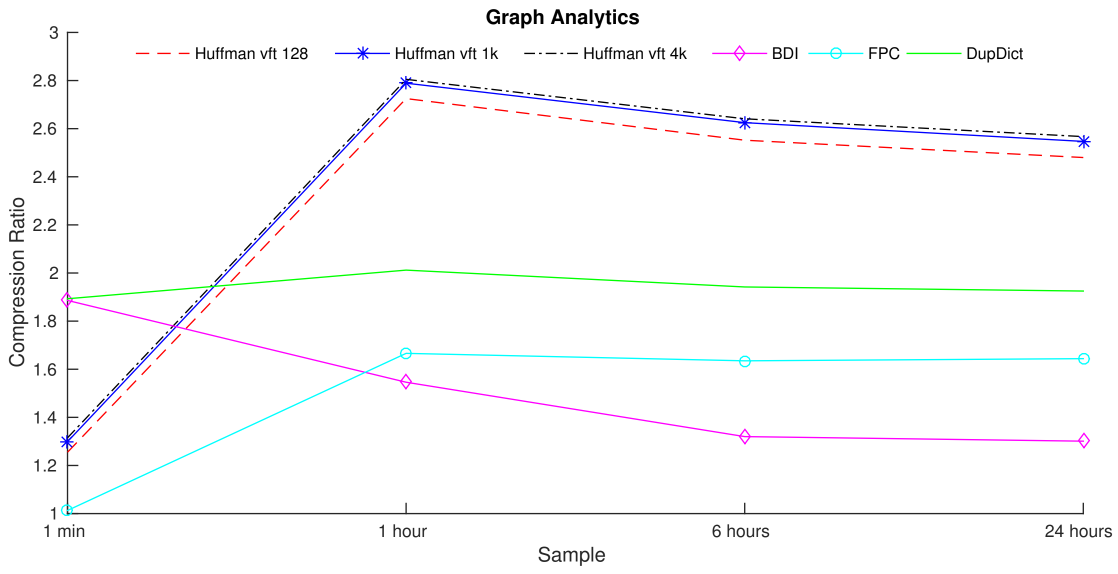

Figure 3.7: GA compression results

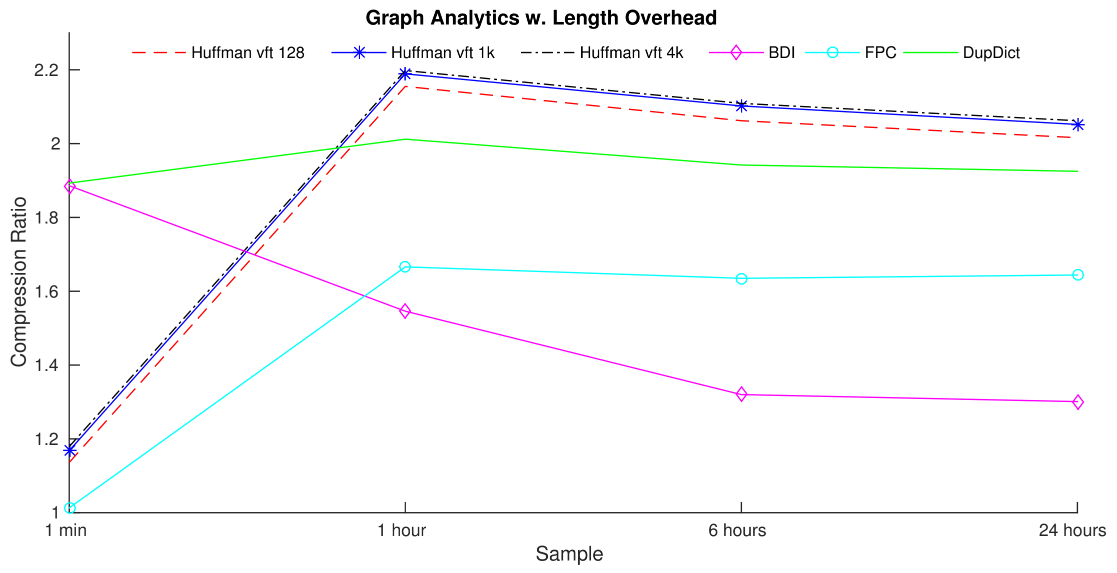

Figure 3.8: GA compression results with LO.

Table 3.16: Loss in compression for Huffman+LO with respect to Huffman

Sample	128 entries	1k entries	4k entries
GA_z0	9.48%	10.08%	10.34%
GA_z1	20.92%	21.51%	21.64%
GA_z2	19.20%	19.92%	20.14%
GA_z3	18.71%	19.43%	19.67%

Table 3.17 shows the compression results for the Google Plus graph. The compression values are substantially lower than the compression values for the Twitter graph. Interesting to note is the high compression value for the BΔI algorithm.

Table 3.17: Compression results for GA with the Google plus dataset

	Huffman			Huffman+LO			BΔI	FPC	Null Blocks	DupDict
Application	128	1K	4K	128	1K	4K				
GA_g_0	1.000	1.001	1.011	1.000	1.001	1.000	5.353	1.000	0.02%	2.015
GA_g_100	1.123	1.125	1.127	1.051	1.052	1.052	1.194	1.013	0.06%	1.038
GA_g_130	1.095	1.095	1.098	1.045	1.045	1.045	1.244	1.012	0.11%	1.034

3.2.3 VFT size improvement analysis

The VFT contains the most frequent values and their encoding. A larger VFT can fit more values and will always give a better CF than a smaller VFT. However, if the CF improvement is very small when comparing two VFT sizes one might want to use the smaller VFT since it will be faster, consume less energy, and use less area.

Table 3.18 shows a summary of the improvement in CF for the DA and GA benchmarks when comparing three different VFT sizes, namely 128, 1k, and 4k entries. The DA sample in the table shows the geometric mean (GM) of the DA_z0 - DA_z3 samples. The same goes for the GA and GA with GA_z0 excluded.

Table 3.18: GM of improved CF with increased VFT size.

Sample	**VFT SIZE 128 ->1k CF improvement**	**VFT SIZE 128 ->4k CF improvement**	**VFT SIZE 1k ->4k CF improvement**
DA	8.68%	11.08%	2.16%
GA	2.82%	3.62%	0.76%
GA (GA_z0 excluded)	2.63%	3.30%	0.65%

3.2.4 VFT sensitivity analysis

So far we have only discussed Huffman CFs when optimal VFTs have been used. The VFTs have been generated from the entire binary image which results in the best possible encoding. In a hardware implementation this will not be possible, instead one has to sample the memory traffic over some time and then generate the VFT depending on the frequency of the values seen in the sampling window.

In our benchmark set up it is not possible to access the memory traffic and we can not generate VFTs in the same way that they would be generated in a hardware implementation. However, if we use a small portion of the binary image to represent the sampling window we can get an idea of how this would impact the Huffman CF. Four different sampling lengths of consecutive values have been analysed: 100k, 1M, 10M, and 100M values. The values have been taken with 250M values offset into the binary image. Using values with four-byte granularity, the sampling windows correspond to 0.38 MB, 3.81 MB, 38.15 MB, and 381.47 MB of data with an offset of 1 GB.

Table 3.19 shows the CF as well as the loss in CF with respect to the Huffman CF for the DA benchmark with the four different sampling window lengths. The VFT was generated from the DA_z0's binary image and was used when calculating the CF for all samples. With a 100M values sampling window the loss in CF is between 1.67% and 8.27%.

Table 3.19: DA VFT sensitivity analysis - sampled at DA_z0.

Sample	**Huffman**	**CF w. sampled VFT**				**CF loss wrt. Huffman CF**			
	CF	**100E3**	**1M**	**10M**	**100M**	**100k**	**1M**	**10M**	**100M**
DA_z0	2.388	1.973	2.060	2.291	2.348	17.37%	13.74%	4.04%	1.67%
DA_z1	2.522	2.065	2.108	2.378	2.420	18.12%	16.41%	5.71%	4.04%
DA_z2	2.500	1.907	2.001	2.236	2.293	23.72%	19.96%	10.54%	8.27%
DA_z3	2.535	2.116	2.214	2.447	2.485	16.55%	12.65%	3.48%	1.96%

Table 3.20 shows the CF results for GA when GA_z0 was used to create the VFT. The low CF results for GA_z0 will affect the other samples' CF. The CF loss for the GA_z1-GA_z3 ranges from 32.81% to 38.75%.

Table 3.20: GA VFT sensitivity analysis - sampled at GA_z0.

Sample	**Huffman**	**CF w. sampled VFT**				**CF loss wrt. Huffman CF**			
	CF	**100k**	**1M**	**10M**	**100M**	**100k**	**1M**	**10M**	**100M**
GA_z0	1.255	1.212	1.223	1.210	1.227	3.39%	2.55%	3.54%	2.22%
GA_z1	2.725	1.693	1.572	1.646	1.669	37.88%	42.31%	39.59%	38.75%
GA_z2	2.552	1.648	1.538	1.605	1.629	35.42%	39.73%	37.11%	36.17%
GA_z3	2.480	1.686	1.568	1.634	1.666	32.00%	36.77%	34.10%	32.81%

Table 3.21 shows the results when the GA_z1 sample was used to generate the VFT. The loss in CF for the GA_z1-GA_z3 ranges from 0.51% to 1.31% which is significantly lower compared to when the GA_z0 sample was used to generate the VFT.

Table 3.21: GA VFT sensitivity analysis - sampled at GA_z1.

Sample	**Huffman**	**CF w. sampled VFT**				**CF loss wrt. Huffman CF**			
	CF	**100k**	**1M**	**10M**	**100M**	**100k**	**1M**	**10M**	**100M**
GA_z1	2.725	2.209	2.491	2.721	2.711	18.93%	8.60%	0.15%	0.51%
GA_z2	2.552	2.100	2.336	2.545	2.536	17.71%	8.46%	0.27%	0.63%
GA_z3	2.480	2.099	2.343	2.452	2.447	15.34%	5.51%	1.11%	1.31%

3.2.5 Alignment analysis

In a traditional computer system blocks and pages have fixed sizes of 64 and 4096 bytes, respectively. In a computer system utilising compression, blocks and pages must be allowed to vary in size. In this section we present an analysis evaluating the impact on the Huffman CF by adding additional block and page addressing.

Block alignment analysis

In the computer system we are evaluating, each block consists of sixteen values with four-byte granularity and with successful compression the block size will be less than 64-byte. In order for a system to utilise the compression, the system needs to be able to address blocks smaller than 64-byte. In this section we evaluate two different block alignment schemes: CF alignment and size alignment.

The CF alignment scheme uses a block's CF as alignment. Each block can have a CF of between one and eight where an uncompressed block has a CF of one. The block size ranges for the eight different CFs in the CF alignment can be seen in Table 3.22. Equation 3.2-3.4 show how the maximum block size, minimum block size, the maximum slack are calculated for a CF. The highest CF, a CF of eight, has to cover all remaining block sizes and thus is given the minimum range of one.

Table 3.22: Block ranges and slack for CF alignment.

Block CF	**Block size / CF**	**CF block range**		**Max slack**
		Max	Min	
1	64	64	33	31
2	32	32	22	10
3	21.33	21	17	4
4	16	16	13	3
5	12.80	12	11	1
6	10.67	10	10	0
7	9.14	9	9	0
8	8	8	1	7

$$blkrange\_max_i = \left\lfloor \frac{Block\ size}{i} \right\rfloor \tag{3.2}$$

$$blkrange\_min_i = \left\lceil \frac{Block\ size}{i+1} \right\rceil \tag{3.3}$$

$$max\_slack_i = blkrange\_max_i - blkrange\_min_i \tag{3.4}$$

The size alignment scheme uses the block's size as alignment and uses sixteen evenly spread block sizes from 4 to 64 bytes as can be seen in Table 3.23.

Table 3.24 shows the Huffman CF and the CF loss with respect to Huffman CF for both block alignment schemes. The block size alignment scheme outperforms the CF alignment scheme and one can also see that the loss in CF for the size alignment scheme is quite consistent for the three different VFT sizes.

Table 3.23: Block ranges and slack for size alignment.

Block	**CF block range**		**Max slack**
size	Max	Min	
4	4	1	3
8	8	5	3
12	12	9	3
16	16	13	3
20	20	17	3
24	24	21	3
28	28	25	3
32	32	29	3
36	36	33	3
40	40	37	3
44	44	41	3
48	48	45	3
52	52	49	3
56	56	53	3
60	60	57	3
64	64	61	3

Table 3.24: GM of block alignment results

Sample	**VFT size**	**Huffman CF**	**CF w. CF alignment**	**CF w. Size alignment**	**CF loss wrt Huffman CF w. CF alignment**	**CF loss wrt Huffman CF w. Size alignment**
DA	128	2.486	1.995	2.372	19.66%	4.57%
GA	128	2.157	1.889	2.047	11.38%	4.32%
GA (GA_z0 excluded)	128	2.584	2.333	2.421	9.69%	6.23%
DA	1k	2.705	2.199	2.565	18.72%	5.15%
GA	1k	2.218	1.941	2.101	11.93%	4.57%
GA (GA_z0 excluded)	1k	2.652	2.380	2.479	10.24%	6.47%
DA	4k	2.766	2.246	2.615	18.80%	5.43%
GA	4k	2.236	1.949	2.115	12.20%	4.74%
GA (GA_z0 excluded)	4k	2.669	2.392	2.494	10.39%	6.52%

Figure 3.9 and Figure 3.10 shows the block distribution for the CF and size alignment schemes with a 128 entry VFT. The block distribution is more evenly spread for the DA than for the GA and GA with GA_z0 excluded for both the alignment schemes. Around 85% of the blocks are covered with four of the sixteen block sizes for GA with size alignment and GA_z0 excluded. Appendix C contains additional block alignment information.

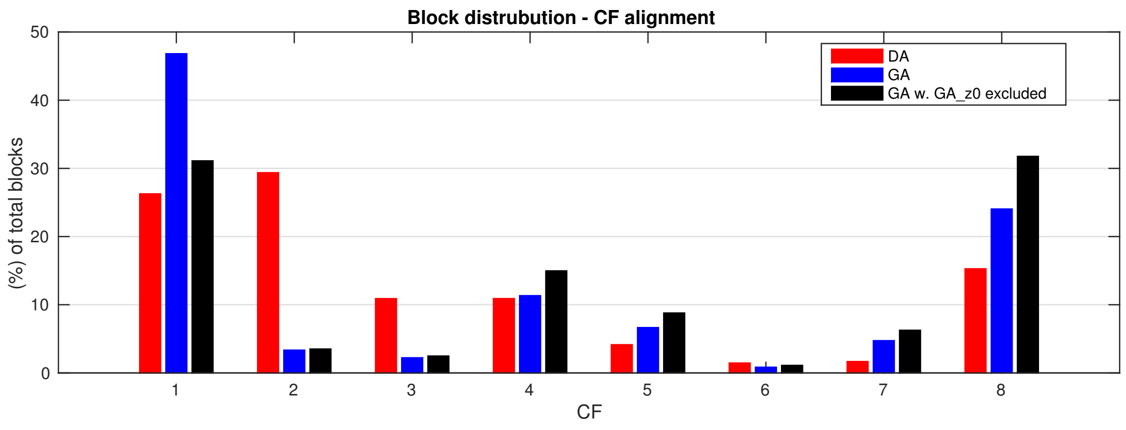

Figure 3.9: Distribution of blocks with a CF of 1-8 (uncompressed blocks have a CF of 1)

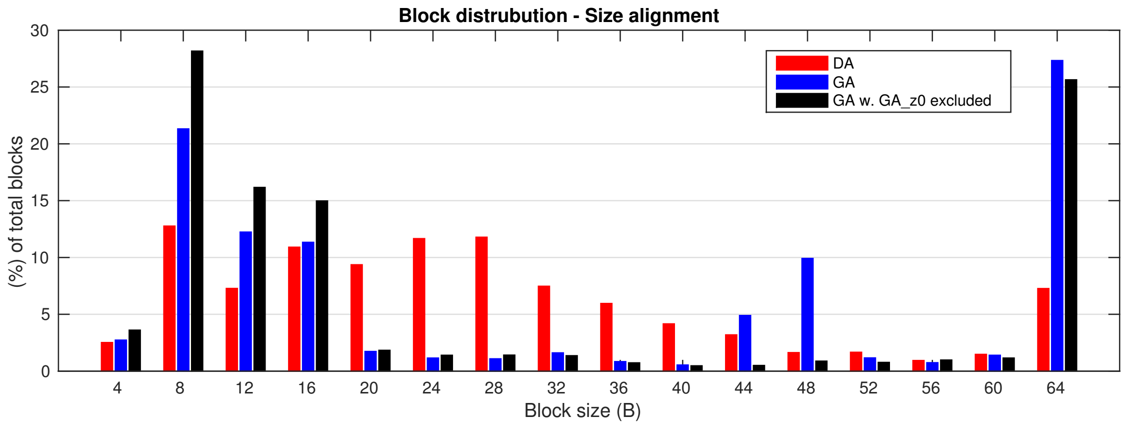

Figure 3.10: Distribution of blocks with a size of 4, 8, .. , 64 byte (uncompressed blocks have a size of 64 bytes)

Page alignment

The reason for a more detailed page alignment is required is similar to that of the block alignment. Pages in a traditional computer system have a fixed size of 4096-bytes. In a computer system utilising compression, compressed pages will have a size smaller than 4096-bytes.

The page alignment analysis in this section is similar to the size alignment scheme for the block alignment. The difference being that instead of only evaluating four added addressing bits we evaluate a range from one to six added addressing bits.

Table 3.25 shows the CF results and the loss in CF with respect to optimal Huffman compression for one, two, four, and six added page addressing bits. With six added bits, the loss in CF for the DA benchmark is 1.83% - 2.07% and for the GA benchmark with

GA_z0 excluded the loss is 1.4% - 1.5%.

Table 3.25: GM of page alignment results

Sample	**VFT size**	**Huffman CF**	**CF w. added page addressing bits**				**CF loss wrt Huffman CF w. added page addressing bits**			
			1	2	4	6	1	2	4	6
DA	128	2.486	1.648	1.862	2.313	2.440	33.64%	25.05%	6.95%	1.83%
GA	128	2.157	1.374	1.755	2.010	2.131	33.72%	15.48%	6.26%	1.11%
GA (GA_z0 excluded)	128	2.584	1.515	1.995	2.376	2.547	41.31%	22.77%	8.04%	1.40%
DA	1k	2.705	1.769	1.977	2.503	2.651	34.59%	26.91%	7.47%	2.01%
GA	1k	2.218	1.389	1.787	2.062	2.190	34.85%	16.77%	6.53%	1.21%
GA (GA_z0 excluded)	1k	2.652	1.526	2.031	2.431	2.613	42.43%	23.41%	8.32%	1.46%
DA	4k	2.766	1.784	1.994	2.547	2.709	35.47%	27.89%	7.88%	2.07%
GA	4k	2.236	1.387	1.795	2.074	2.207	35.76%	17.60%	6.80%	1.26%
GA (GA_z0 excluded)	4k	2.696	1.527	2.044	2.444	2.629	42.76%	23.43%	8.43%	1.49%

Figure 3.11 and Figure 3.12 shows the page distribution for the DA benchmark with four and six added page addressing bits (VFT size of 128 entries). When six added bits are used around half of the available page sizes have less than 1% of the total number of pages.

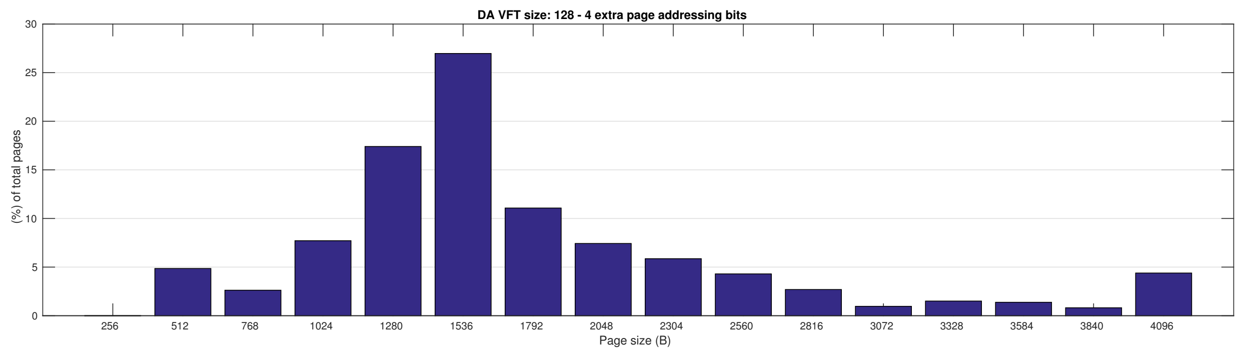

Figure 3.11: Page distribution for DA - 4 added addressing bits.

Figure 3.13 and Figure 3.14 show the page distribution for the GA benchmark with four and six added page addressing bits (VFT size of 128 entries). The page distribution is very different from the page distribution for the DA benchmark. There are only three different page sizes with more than 5% of the total pages with six added addressing bits. When removing the GA_z0 sample, Figure 3.15 and Figure 3.16, the peak at the 2944-byte page size disappears. Appendix C contains additional page alignment information.

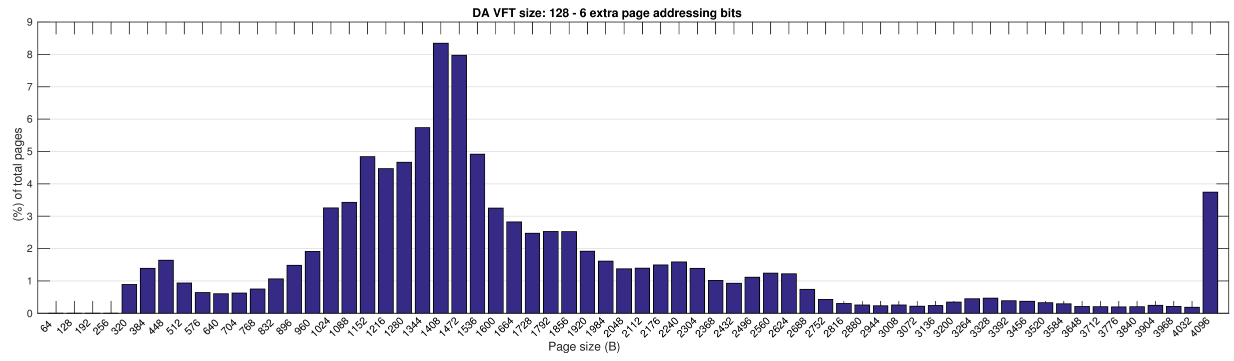

Figure 3.12: Page distribution for DA - 6 added addressing bits.

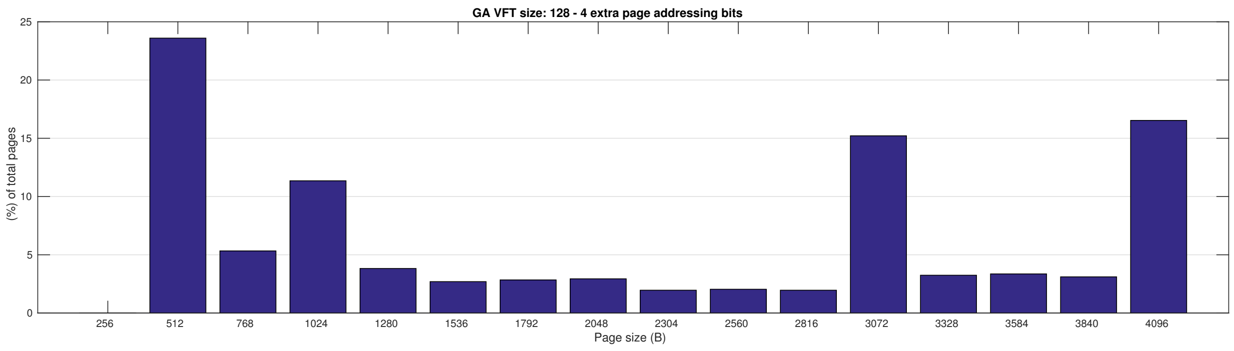

Figure 3.13: Page distribution for DA - 4 added addressing bits.

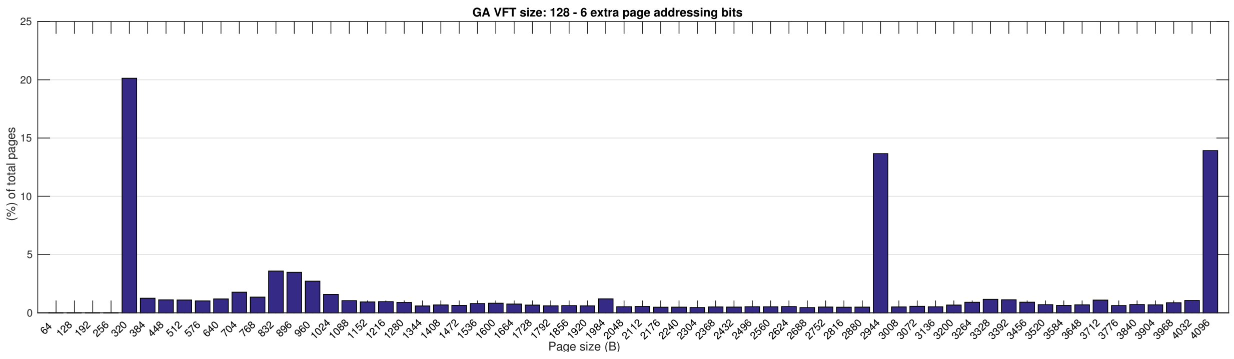

Figure 3.14: Page distribution for GA - 6 added addressing bits.

Combined block and page alignment

In this section we evaluate the impact of combining the block alignment, the page alignment, and the LO. Four and six added addressing bits have been evaluated for the page alignment scheme and the blocks are aligned with the size alignment scheme since it

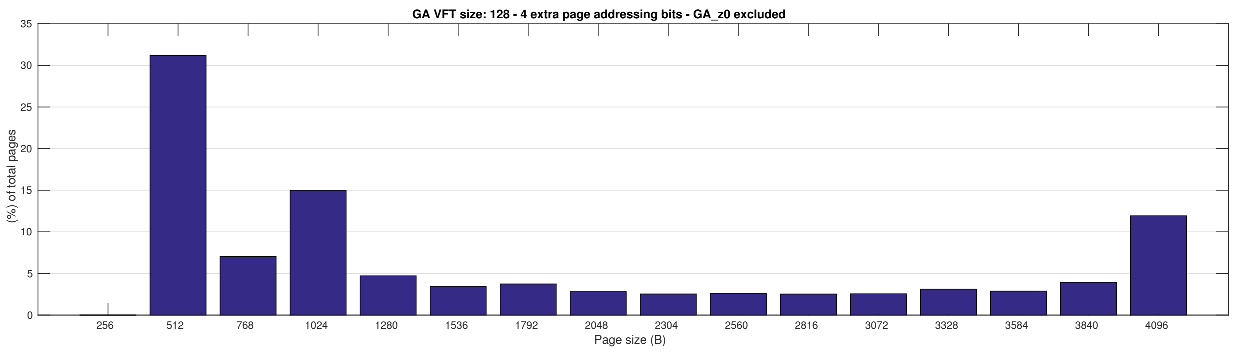

Figure 3.15: Page distribution for GA - 4 added addressing bits - DA_z0 excluded.

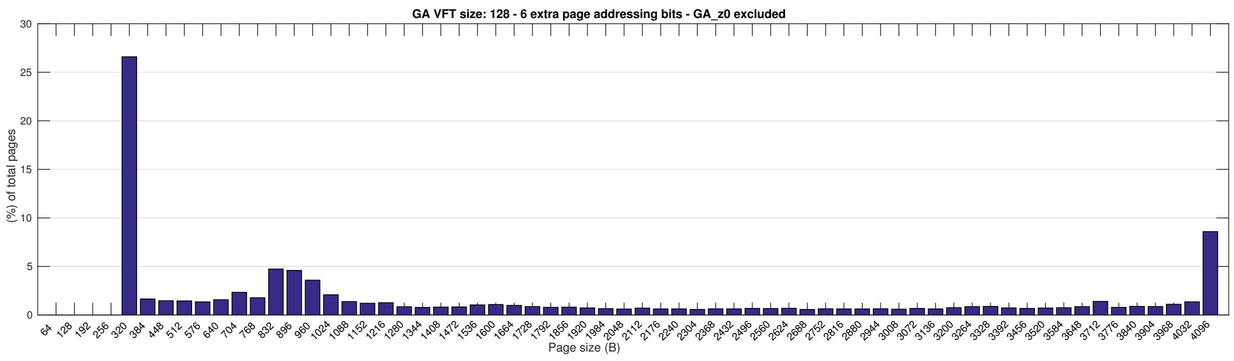

Figure 3.16: Page distribution for DA - 6 added addressing bits - GA_z0 excluded.

produced a better result than the CF alignment scheme. Table 3.26 shows the results when combining block alignment and page alignment. The loss in CF with respect to optimal Huffman CF ranges from 5.87% to 7.83% for the DA and GA benchmarks.

Table 3.26: GM of CF with block and page alignment.

Sample	VFT size	Huffman CF	CF w. block size alignment & 4 added page addressing bits	CF w. block size alignment & 6 added page addressing bits	loss in CF wrt Huffman CF w. block size alignment & 4 added page addressing bits	loss in CF wrt Huffman CF w. block size alignment & 6 added page addressing bits
DA	128	2.486	2.222	2.333	10.61%	6.13%
GA	128	2.157	1.959	2.025	8.22%	5.31%
GA (GA_z0 excluded)	128	2.584	2.299	2.390	11.00%	7.47%
DA	1k	2.705	2.380	2.518	11.99%	6.92%
GA	1k	2.218	2.009	2.077	8.55%	5.62%
GA (GA_z0 excluded)	1k	2.652	2.354	2.446	11.23%	7.73%
DA	4k	2.766	2.431	2.567	12.11%	7.17%
GA	4k	2.236	2.021	2.090	8.86%	5.92%
GA (GA_z0 excluded)	4k	2.669	2.367	2.460	11.31%	7.81%

Table 3.27 shows the result when page alignment, block alignment, and the LO are combined.

Table 3.27: GM of CF with LO, block and page alignment.

Sample	VFT size	Huffman CF	Huffman CF w. LO	CF w. block size alignment & 4 added page addressing bits	CF w. block size alignment & 6 added page addressing bits	loss in CF wrt Huffman CF w. block size alignment & 4 added page addressing bits	loss in CF wrt Huffman CF w. block size alignment & 6 added page addressing bits
DA	128	2.486	1.932	1.770	1.840	28.76%	25.97%
GA	128	2.157	1.786	1.654	1.698	21.91%	19.97%
GA (GA_z0 excluded)	128	2.584	2.077	1.893	1.953	26.70%	24.36%
DA	1k	2.705	2.053	1.857	1.939	31.33%	28.29%
GA	1k	2.218	1.822	1.685	1.730	22.76%	20.81%
GA (GA_z0 excluded)	1k	2.652	2.114	1.924	1.984	27.42%	25.15%
DA	4k	2.766	2.080	1.886	1.967	31.78%	28.87%
GA	4k	2.236	1.832	1.691	1.737	23.21%	21.21%
GA (GA_z0 excluded)	4k	2.669	2.122	1.931	1.991	27.63%	25.37%

3.2.6 Alternative LO

The LO is a fixed number of bits appended in front of every value which allows for faster, parallel, decompression. These added bits indicate the code length of the following code word. Up to this point we have considered a four-bit LO which allows for code lengths ranging from one to sixteen bits.

In this section we evaluate the impact of altering the LO from four bits down to a single bit. Reducing the LO length will impact the number of values that can be stored in the VFT. In this analysis LO_bits bits correspond to 2^{LO_bits} unique code lengths instead of code lengths ranging from 1 to 2^{LO_bits}. This could easily be implemented using a LUT and is due to the fact that depending on the VFT encoding some code lengths might be omited.

The results from the LO analysis can be seen in Table 3.28. The four-bits LO column is the same as the previously reported Huffman+LO values for a VFT size of 128. Values in the table with green background indicate an increase in CF with respect to a four-bit LO. A three-bit LO will yield better CF for all the samples even though the VFT may contain as few as only fourteen values. The GA_z1 - GA_z3 samples' CF for a three-bit LO is better than for a VFT size of 4k with four-bit LO even though the number of VFT entries is only between fourteen and eighteen for the three-bit LO and 4096 for the four-bit LO.

Another interesting detail is that with an one-bit LO we cover two code words: the most common value's code word and the code word for uncompressed values. The most common value in all of the samples is zero as can be seen in Appendix A. The CF with a two-bit LO is still high, between 1.480 and 1.812 and shows that it is often the case that a single value represents a very large portion of the total compression.

Table 3.28: CF for alternative LO lengths.

Sample	**Huffman**	**CF**				**Values in the VFT**			
	CF	4 bits LO	3 bits LO	2 bits LO	1 bits LO	4 bits CL	3 bits CL	2 bits CL	1 bits CL
DA_z0	2.388	1.878	1.967	1.690	1.577	128	109	10	2
DA_z1	2.522	1.960	2.029	1.630	1.614	128	66	5	2
DA_z2	2.500	1.938	1.973	1.492	1.480	128	73	4	2
DA_z3	2.535	1.952	1.985	1.643	1.648	128	53	4	2
GA_z0	1.255	1.136	1.163	1.166	1.183	128	128	4	2
GA_z1	2.725	2.155	2.238	2.215	1.812	128	14	7	2
GA_z2	2.552	2.062	2.123	2.113	1.759	128	18	8	2
GA_z3	2.480	2.016	2.083	2.007	1.807	128	18	6	2

4

System implementation

This chapter starts with detailed descriptions on how several different compression schemes have been implemented as well as an evaluation of these implementations. The algorithms implemented are Huffman, BΔI and FPC. The Huffman compression algorithm was used on a proof-of-concept implementation that has been implemented on a small running system. The system has had software programmed on it that utilises the compression hardware to test the concept of memory compression. At the end of the chapter, an evaluation of the different hardware implementations has been made.

The evaluation was performed by firstly dividing the different schemes into their compression and decompression logic. The evaluation was performed by synthesizing the different designs onto a 28 nm library targeting a cycle time as low as possible. The synthesized netlists were then evaluated for their different characteristics.

4.1 BΔI

Our VHDL implementation of the BΔI scheme is similar to the BΔI scheme suggested by Pekhimenko et al. from an overall perspective, however, some minor details have been changed and some optimizations have been made [12].

Figure 4.1 shows an overview of our modified version of the BΔI compression algorithm. An input cache line is sent to all compression modules in parallel and the compression modules output compressed data and a valid bit to a selector. The valid bit signals whether the compression was successful or not. The selector chooses data from the compression module which generated valid data with the highest compression. The selector outputs the compressed data with a prefix indicating which module was used to compress the data.

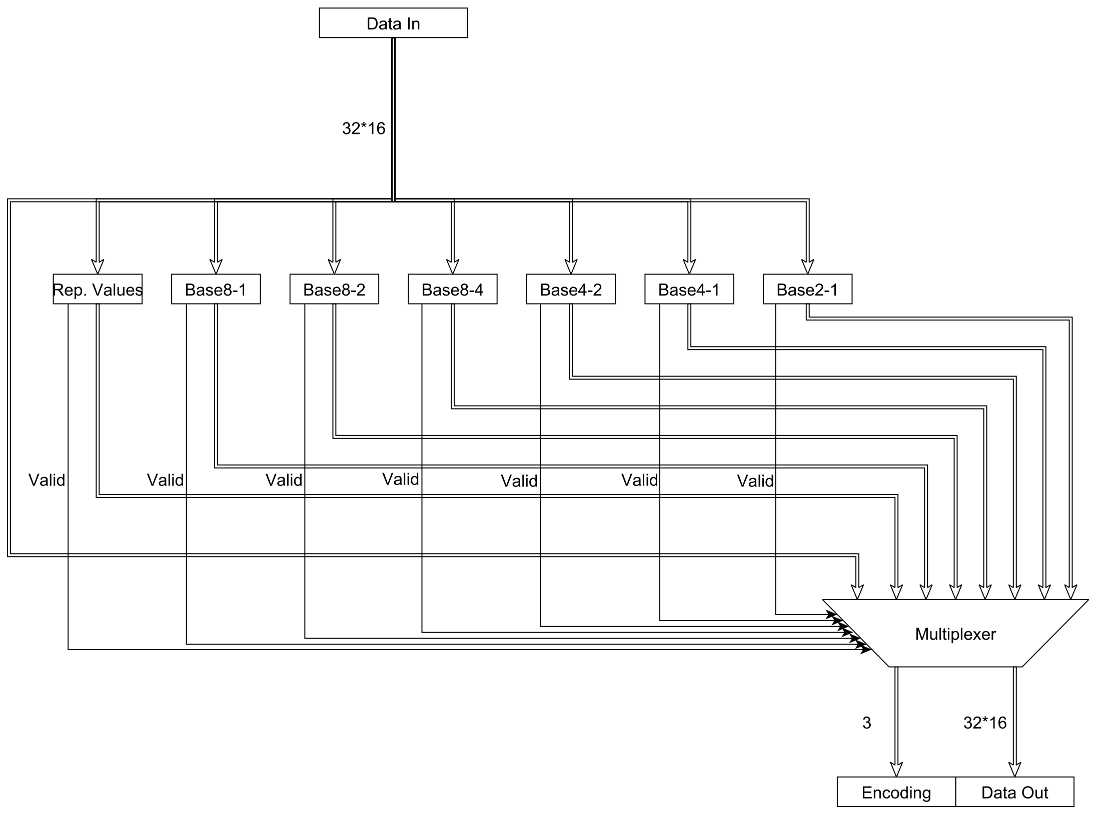

Figure 4.1: Compressor of the modified BΔI compression implementation

One of the differences between our implementation of BΔI the original scheme is that in our implementation, no module have been implemented to handle the special case of a cache line full of zeros. The reason for this is that the zero values will already have been compressed by other compression algorithm that will be inserted previous to the BΔI algorithm in future prototype systems. Since we now only have eight different encodings (one encoding for each of the seven compression modules and one encoding for uncompressed data), we are also able to use prefixes of three bits instead of four as used by Pekhimenko et al. [12].

Another more significant difference lies in the implementation of the compression and decompression modules. One difference lies in how an appropriate base is chosen. Pekhimenko et al. does this by adopting the first word in a cache line as the base [12]. However, in our implementation, the base is described as the first word in a cache line that cannot be represented using a delta and zero as base. This allows for a larger set of input data to be compressed by this algorithm. As can be seen in Figure 4.2, the respective deltas are calculated in the same manner as in the original description. At the end, a shifter is inserted. This shifter is not implemented in the original implementation and is used to remove previously unused information in the compressed cache line.

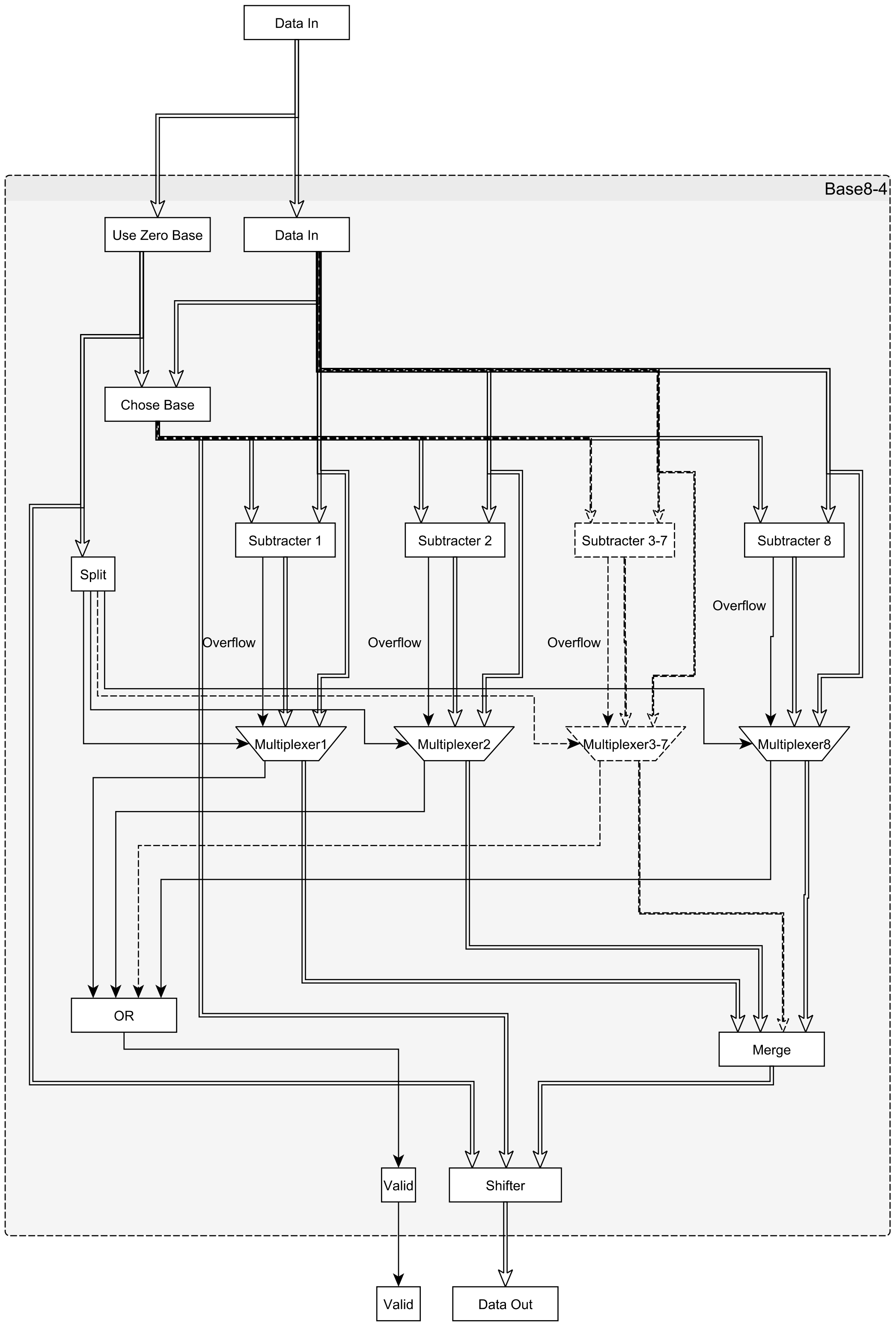

Figure 4.2: Detailed view of a BΔI single compression component

Figure 4.3 shows an example of how data is output from a Base4-1 compression module is organised. In this compression module, an input cache line of a size of 512 bits is divided into 16 sections of 4 bytes. A base of 4 bytes is used with 16 deltas that are 1 byte wide to represent the original sections. In total, the output data consists of 3 bits of encoding to indicate that base4-1 compression is used. 16 bits indicate for each of the 16 deltas whether they use zero or the second base value as base. A 4-byte second base value then follows and lastly, 16 deltas represented with 1-byte values are included.

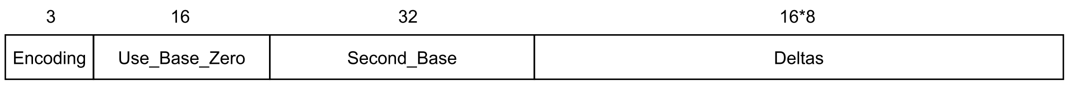

Figure 4.3: Output from the Base4-1 compression component

After a cache line has been compressed, the data have been compressed in one of two patterns. In the first case, all words in the line can be represented using a delta value from the zero base. In this case, we can shift the bits out that would be used to represent the second base value. This makes it possible to save as much as 64 bits for some compression modules. In the second case, the second base is chosen from one of the words in the cache line. This means that there will always be a delta with value zero and we also know the position of this delta as the first occurrence of a word that cannot be described with a delta and a base of zero. Therefore, it is possible to shift this delta out when compressing the data and later just shift in the base value at that position when decompressing the data. The shifter will hence lead to a saving of up to 32 bits, depending on which compression module that is used, when compressing.

4.2 FPC

Our VHDL implementation of the FPC compression algorithm is very similar to the original description made by Alameldeen and Wood. There is one difference though. For this project, we have chosen not to store the prefixes in the tag-field of the cache for simplicity reasons. This decision causes side effects on the compression and decompression algorithms. The first side effect is that since we do not have bits dedicated to prefixes, we can sometimes use fewer bits to describe the prefixes. This is because the number of prefixes is not a constant value due to the zero run prefix. When fewer bits are used to describe the prefixes, it is possible to shift the data bits towards the prefixes and therefore in total, use fewer bits to describe the prefixes plus data than if a memory tag solution is used for the prefixes.

A shifted data start can cause problems when decompressing a cache line. Since we no longer know for certain where the prefixes end and where the data starts, we can no longer decompress the data without using very complex and time consuming algorithms.

To tackle this problem, prefixes have to be stored directly in front of the compressed data that they point to.

A memory containing compressed data, compressed with our modified FPC algorithm, will have the following patter:

$$prefix_1 + data_1 + prefix_2 + data_2 + prefix_3 + data_3 + ... + prefix_{16} + data_{16}$$

4.3 Proof-of-concept

During this thesis, a first prototype of a data compressing memory system was developed as a proof-of-concept. The original idea of implementing a Huffman algorithm to compress and decompress data between the LLC and main memory was enforced by the compression analysis made earlier in this thesis. This is because the Huffman algorithm generated the best compression numbers. The prototype was developed and implemented on the ZedBoard where the Processing System was programmed with a small, bare-metal software that first writes and then reads a few values to and from the main memory. A top-down approach will be used in this section to describe the implemented prototype.

As can be seen in the simplified block diagram of the SoC in Figure 4.4, the datapath between the LLC and the main memory needs to be altered as suggested by the EuroServer Team at FORTH institute in Greece [40]. The standard datapath is shown using dashed, turquoise arrows. This path goes directly to the main memory from the LLC. However, for data to be altered by compression/decompression hardware, data needs to be guided through the programmable logic of the SoC. For data to go through the programmable logic, the data will have to be routed via a central interconnect and a 32-bit AXI interface before entering the programmable logic. To send data from the programmable logic to the processing system's main memory, the data needs to be routed through a 32-bit AXI interface via programmable memory interconnect logic before reaching main memory. This whole new path from processing system to programmable logic is represented by solid, red arrows.

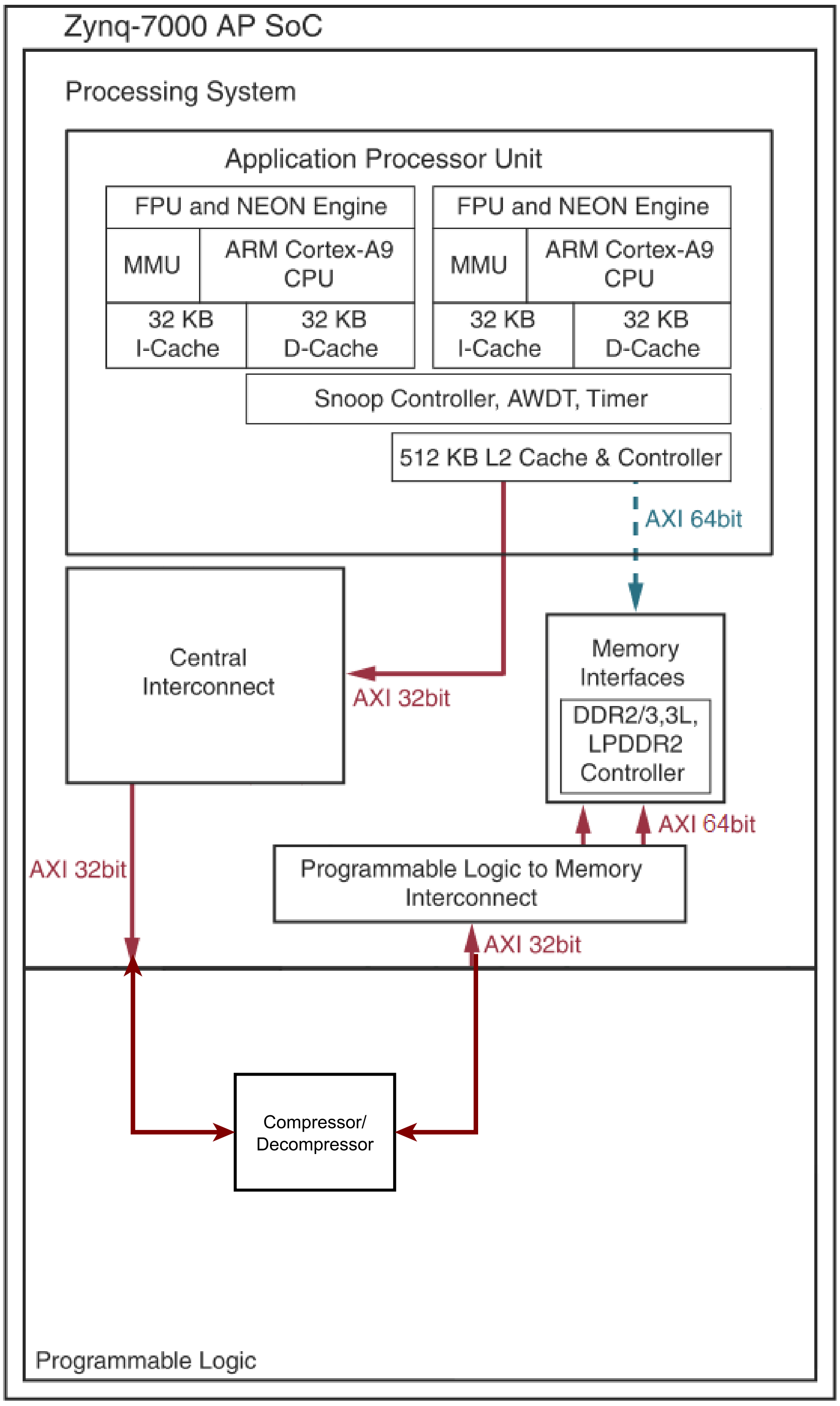

Figure 4.4: Prototype block diagram

Figure 4.5 shows a detailed version of the custom logic. The AXI bus is divided into two independent components, one write component and one read component. This enables full duplex communication between the processing system and main memory, enabling parallel reads and writes.

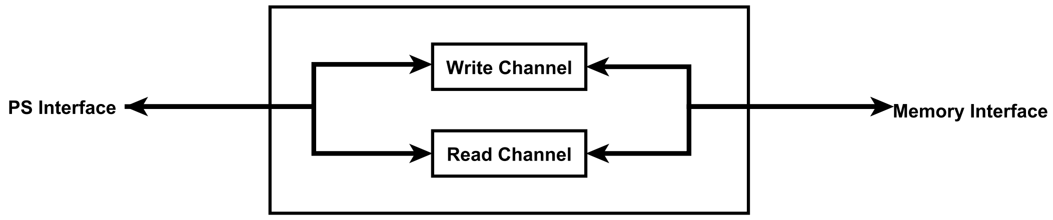

Figure 4.5: Independent read and write components for full duplex communication

4.3.1 Write component

The in-depth view of the write component in Figure 4.6 shows how the write signals are divided into its three independent channels, the address channel, the data channel, and the answer channel. The address channel and the answer channel are less complex than the data channel.

To send data to the programmable area, the processing system must write to an address that is not mapped onto the main memory. The address channel of the write component has two jobs, to change the address to one that is mapped onto the main memory and to indicate that a cache line with ten words of data is being sent instead of eight. This is because a cache line of eight words that is sent from the processing system to the main memory may increase in size to ten words if the compressor logic is fed with uncompressible data.

The job of the answer channel is to signal whether a write action was performed successfully or not. The write component will always receive write actions successfully so it is hard-wired to always signal a successful answer to the processing system. Likewise, on the other side of the component, the address channel is always ready to receive an answer from the memory.

The data channel is the most advanced channel as it involves seven stages of travel for data before compression is completed. The only component of the data channel that does not handle the data directly is the initaliser component. This initaliser component has a static LUT that describes a constant, pre-generated Huffman-tree structure that is read to the compressing component immediately after a system reset. Future prototypes will not use such an initializing component as they will not use a static table. Instead they will write a Huffman-tree structure to the compressor that will be created in software using sampling from the actual content in main memory.

Data that is written through the data channel will first encounter the AXI slave logic. This logic ensures that the write component is compatible with the AXI interface communication from the processing system. The AXI slave also checks the status of the two FIFOs in the write component and halts input data if necessary to avoid overflowing of data.

Next, data is sent to a Speed Adapting FIFO. The job of the Speed Adapting FIFO is to ensure that data accepted from the AXI Slave logic will be presented to the Compressor component for at least four cycles. This is a requirement from the Compressor. Also, it acts a buffer to the processing system so that it may write an entire line to the write component and does not need to wait for the slower compressor during single write requests.

The Compressor component was created by Dimitris Giannopoulos and is the component that performs the actual compression. To perform the compression, the compressor component has implemented a Huffman compressing algorithm. This component executes Huffman by comparing input data with data in a LUT that was initalised by the initaliser. If a match is found, then the compressor outputs a compressed code word with padding of zeroes instead of the original data. If the input data is not found in the internal LUT, then the original data is output without any modification. This component can only accept data at an input rate of one word every fourth cycle and is the bottleneck of writing data to main memory [31].

When compressed data have been replaced by its shorter Huffman encoding, the Prefix Adder is the next component in line. The Prefix Adder lets encoded data pass through without altering it. However, data that is uncompressed by the Compressor component is given a unique prefix that will indicate to future decompressor hardware that the next word is uncompressed data.

At this point, Huffman encoding of input data is complete. However, the Huffman coded data may contain padding with zero-bits. The Output Shifter component is responsible to ensure that encodings are packed together as closely as possible by removing all bits that are used as padding. The data is now compressed and packed as closely as possible and will be sent to a FIFO in 32-bit chunks.

The FIFO is a simple buffer placed in the end to store data until a signal is raised from the AXI Master logic that the main memory is ready to accept new data. The AXI Master logic not only outputs the compressed data when appropriate, but it also ensures that the write component is compliant with the AXI communication specification used by the processing system.

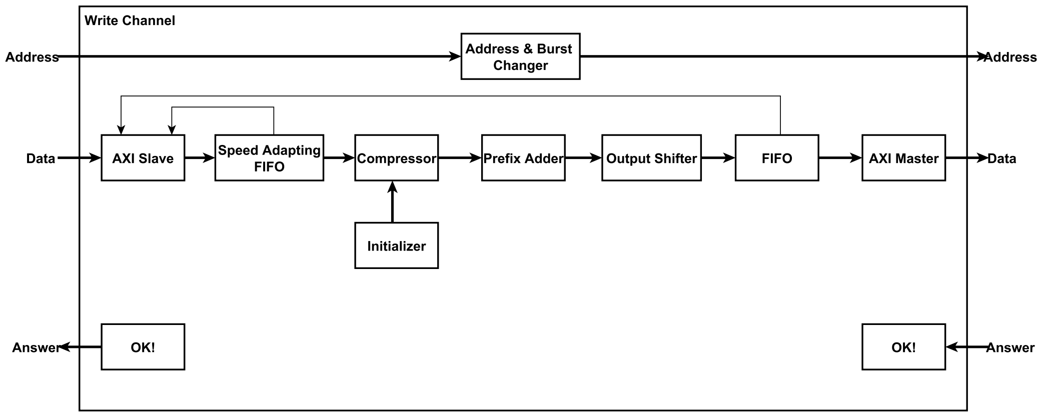

Figure 4.6: Block diagram of write component

4.3.2 Read component

The read component block diagram in Figure 4.7 shows how the input signals, just as for the write component, are divided into their independent channels. The read component uses two of these channels, an address channel and a data channel. The data channel is the more complex of the two channels although they are both equally important.

From the processor's point of view, read requests have to be made to addresses that are mapped to the programmable logic rather than mapped to main memory. If this was not done, then the processor would read from memory directly without going through the programmable logic. The first job of the address channel in the read component is therefore to remap the requested address once a read request has reached the programmable logic to a new address that is mapped onto main memory. Secondly, the address channel must change the number of requested words that are sent from the processing system. From the processing system's point of view, a cache line is made up from eight words and therefore a request of eight words is sent when the processing system wants to receive a cache line from memory. However, when compressing data a cache line may grow to a size of ten words, therefore the read request must be changed to ten words before the request is sent to main memory.

The process of uncompressing data through the read components involves four stages that the data must pass through. An initaliser component is inserted that does not handle data directly, instead, it initialises the decompressor component with Huffman coding information needed to decompress data. This information lies inside the initaliser component as a constant, pre-generated LUT. This initaliser was created as a standalone component to make it easier for future implementations to initialise the decompressor logic using data derived from sampling main memory content.

Data read from memory will firstly pass through the AXI Master logic. This logic is included to create an interface on the read component that is able to successfully communicate with main memory. Once the AXI Master has accepted a cache line of data the AXI Master halts inputs to the read component until the decompressor signals that it has successfully decoded a cache line and sent it to the FIFO stage.

The decompressor component is the most important of the read component's hardware. The decompressor used here was created by Li Kang and is described in great detail in his thesis [41]. The decompressor is organised into three stages. The first stage is a simple FIFO where input data is stored to avoid starvation to following decompressor logic. The second stage is a code detection stage where the length of the next compressed word in line is detected. The third and last stage of the decompressor is a value retriever stage. In this stage, the next compressed word in line is used to retrieve its uncompressed counterpart from a LUT. The decompression component is pipelined and requires a total of five cycles to output the first uncompressed word once compressed data has been inputted. This means that a cache line of eight words can be decompressed in as few as twelve cycles if the decompressor does not suffer from starvation. Ten words are read to the decompressor whether the compressed size of a cache line is smaller than that or not. This means that once a compressed line has been decompressed, some values may remain inside the decompressor that belong to other cache lines. Before a new cache line can start decompression, the decompressor must flush out the unwanted data, a process that requires two cycles.

Data that has been successfully decompressed is then sent to the FIFO component. This component acts as a buffer and stores decompressed values until the processing system is ready to accept data. Lastly, data is sent to the AXI Slave logic, which communicates with the processing system. The AXI Slave interface makes sure that the read component is compatible with the AXI communication protocol used by the processing system.

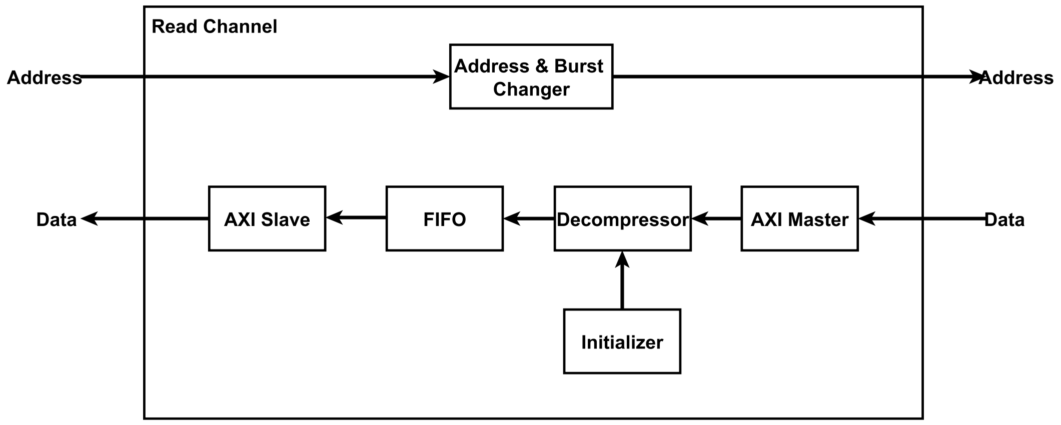

Figure 4.7: Block diagram of the read component

4.3.3 Software

The software consists of two programs created to be run on the proof-of-concept. These programs were given the task to initalise the ZedBoard and to test the data compression. The job of initializing the board was given to a first stage bootloader (FSBL) program created using a template provided by Vivado SDK release 2014.4 [42].

The FSBL performs a wide array of operations, some of them include initializing the system clock, flushing caches so that they are ready to be used, doing small tests of the hardware such as reading and writing tests to the main memory, and enabling voltage level shifters between the processing system and programmable logic so that the programmable logic may be used. The FSBL is also responsible for loading any OS that is to be run on the processing system. This early prototype used no OS, although future prototypes will.

The second program that is run once the FSBL is completed is a test program that tests the writing and reading of cache lines through compression logic. This program includes the following steps for completion:

1. Tell the processing system that address ranges covering AXI communication to the programmable logic should be cached. This step is necessary so that entire cache lines are sent between the LLC and main memory in a single AXI transfer. If the cache is not set up, then an AXI transfer will be set up for the transfer of every individual word, a process that would generate unnecessary overhead.

2. Enter an empty for-loop and stay there until compression initalisers are finished. Since the LUT of the compression and decompression components are initalised from the initaliser components, we have to halt execution until those initalisers are finished.

3. Write zeroes to the memory addresses that will be used in the test program. This step is made so that when examining memory content, it will be easier to distinguish saved, valid, compressed data from the original random data that fill a memory on startup.

4. Perform eight stores of values that are compressible to a cache line whose address makes sure that the line is sent to programmable logic once the line is evicted from the cache.

5. Tell the system to flush the cache line with newly stored values. This will send the compressible data to memory via compression logic in the programmable logic area.

6. Read the flushed addresses and compare if the read values are equal to the values that were stored in step 4. This read request reads data from memory via the decompression logic implemented on the FPGA.

7. Perform steps 4-6 once again using eight other values to compress.

8. Print the number of errors detected to the terminal, then exit the program.

Once both programs are completed, the memory content is examined manually to see if data stored in main memory actually is stored in its compressed form.

4.4 Evaluation

One of the goals of the implementational part is to create a fully working proof-of-concept using data compression between the LLC and main memory. To test if this has been reached successfully, the ZedBoard was implemented with the custom logic developed for this thesis and the testing C-programs were programmed to an ARM CPU as described in chapters above. The test program was ran on the ZedBoard several times testing different combinations of compressible and uncompressible numbers sending a variable amount of cache lines to the memory for different runs.

Figure 4.8 shows one example where two cache lines where sent to memory and compressed from 32 bytes each to 6 bytes and 12 bytes. The Xs in the figure represents saved memory space. Once the program was completed, the memory was manually inspected to validate that compressed data was stored in memory. To further validate that the CPUs fetched data from memory and not from the cache which could give unnoticeable errors and to manually validate a correct compression and decompression, logic analyzers was inserted before and after the compression logic. From the inspections noted above and from the correct executions of the test runs, it was agreed that the proof-of-concept implementation was fully working.

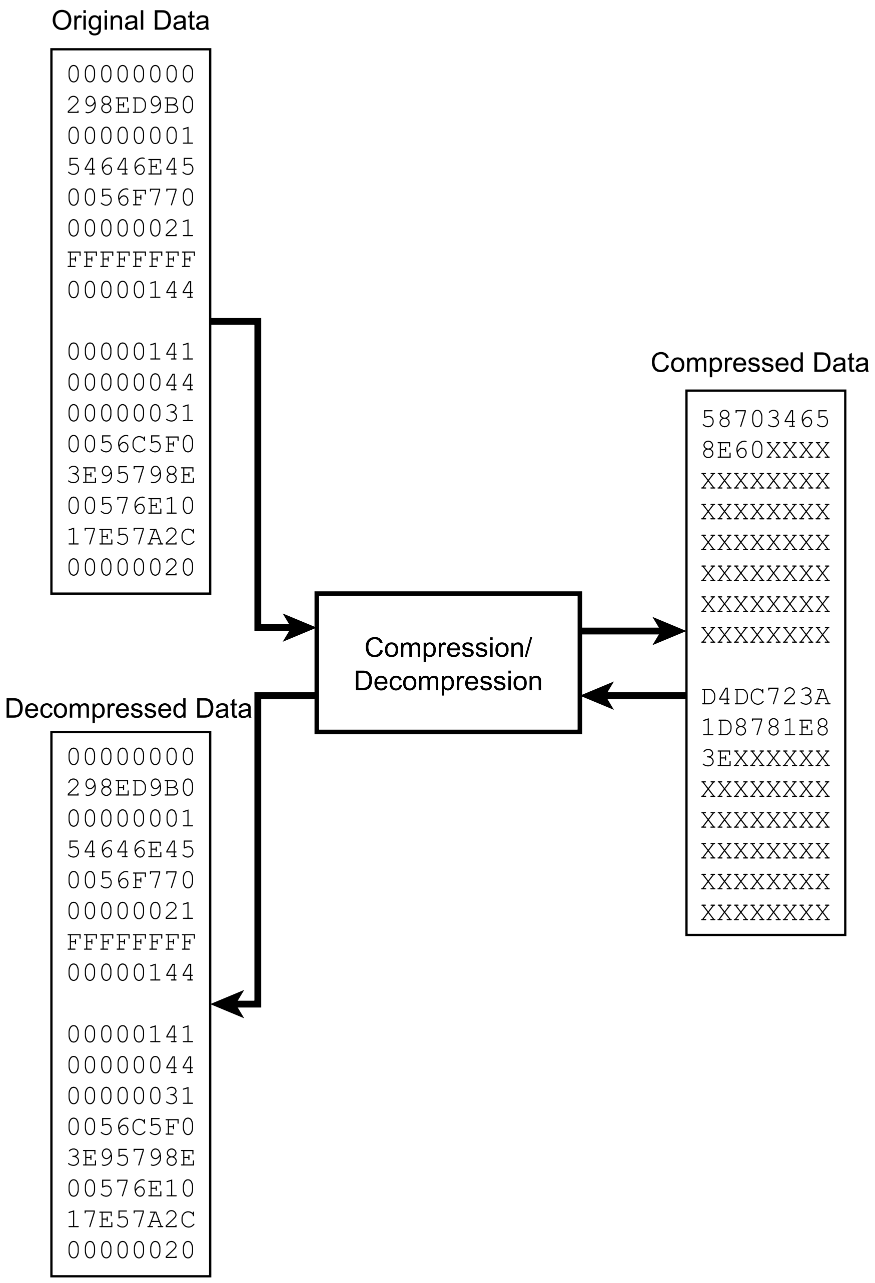

Figure 4.8: Test run of compressible vectors

Once the proof-of-concept had been completed and verified, the focus shifted to evaluate the design and compare it to the BΔI and FPC compression schemes in terms of timing, area, and power consumption. For the evaluational part, the compression logic and the decompression logic were evaluated individually for each scheme.

To enable fair comparisons between the fully implemented Huffman compression that has surrounding logic to be able to run on the ZedBoard (such as AXI connections) with the BΔI and FPC compression schemes, two versions of the Huffman compression were evaluated. The smaller version was tested using only the compression and decompression logic, these are the numbers that should be used to compare with BΔI and FPC. The larger version is including all surrounding logic such as AXI Master/Slave logic and FIFOs. The bigger components are labelled write component and read component.

The Synopsys Design Compiler version 2011.09-SP3 was used when synthesizing the components. They were synthesized and evaluated targeting a 28 nm library that emulates a circuit run with 1.1 V at -40°C. This case of high voltage and low temperature is unrealistic and gives us very good timing properties. The nominal case with 1.0 V at 25°C or the worst case with 0.9 V at 125°C would have given a more realistic evaluation of timing properties. The performance-optimistic case of high voltage and low temperature was used because it was the library in the predecessor project [41].

The components included in a Huffman based compression or decompression were evaluated using a VFT table with 1024 entries (creating equally sized LUTs in the components) and the other components were set to be evaluated when using a 16-word cache line size. The VFT and encoding tables used in the Huffman schemes was synthesized using logic cells. In future prototypes, these tables will be implemented using SRAM which will lead to lower area demands and lower power usage. Therefore, area and power usage by the VFT tables will be presented independently as well. All components were evaluated using as strict timing requirements as possible. The produced netlist was evaluated for timing and area and the results can be seen in Table 4.1.

In the columns ”Cycles required to handle a cache line” as well as ”Total time required”, there is a parenthesised, variable number for some components. This is because the number of needed cycles varies with the input data. For instance, the write component may vary significantly between different input sets. This is due to the fact that output data is output only when compressed data of at least 32 bits have been derived. If several highly compressible words are input to the write channel component, then it will take many cycles before 32 bits of compressed data is filled up.

The area and power demands of the read channel as well as the write channel include the area and power needed to use the initaliser components. The initaliser components contain large tables of constants that will not be included in future systems and should be ignored. Therefore, their numbers have been presented individually as well in Table 4.1 and Table 4.2.

Several points are worth mentioning in Table 4.1. The BΔI and FPC components are all asynchronously implemented, thereby the value of zero for number of required cycles. This must be considered when comparing minimum cycle time between different components. For instance, the BΔI compressor uses a cycle time that is more than 200

Table 4.1: Timing and area comparison between different compression schemes

Component	Cycles required to handle a cache line	Cycle time (ns)	Total time required (ns)	Area (micrometer^2)
FPC compressor	0	55.000	55.000	45036.5082
FPC decompressor	0	1.510	1.510	22029.8784
BΔI compressor	0	0.583	0.583	47821.1907
BΔI decompressor	0	0.355	0.355	38869.9971
Huffman compressor	-1 + 4 per word in a line	0.155	(-0.155 to 0.930) + 0.620 per word in a line	138863.6166
Compressor table				138177.8502
Huffman decompressor	(4 to 6) + 1 per word in a line	0.370	(1.480 to 2.220) + 0.370 per word in a line	135537.7649
Decompressor table				130416.8754
Write component	(5 to 12) + 4 per word in a line	0.360	(1.800 to 4.320) + 1.440 per word in a line	143031.9089
Write initaliser				426.7680
Read component	(8 to 10) + 1 per word in a line	0.375	(3.000 to 3.750) + 0.375 per word in a line	148238.4773
Read initaliser				2413.4016

ps longer than the Huffman decompressor but executes overall faster since it uses fewer clock cycles. Also worth noticing is the long cycle time of the FPC compressor. When examining the worst path, it is concluded that this long delay is an effect of the sequential nature of the FPC scheme and the fact that every stage must be able to perform several shifts of uncertain size. It should also be noted how there is an increase in size when moving into Huffman-based logic, this is almost in its entirety due to the LUTs used by those components and is expected to decrease with a SRAM implementation of the LUTs in future prototypes. It should be pointed out again that the timing results are from a library synthesized in the ideal case of 1.1 V and -40°C which gives underestimated delays compared to a nominal or worst case scenario.

The cycle times for the write component and the read component are only the required time to transfer information on their data channels, meaning that time needed to transfer address and control data as well as information on the answer channel is not included. This is because it would mean that setup time for memory on reads and similar overheads that are impossible for the compression/decompression logic to affect would be included. This overhead is also highly variable since components in AXI protocol communications may pause communication for as long as they wish. For the write component, these overheads were usually kept small at around 2-3 cycles in total. For read action however, an average of 10-15 extra cycles were inserted due to overheads such as memory set up time.

Components in Table 4.1 are affected differently by different kinds of scaling. When increasing cache line size, the FPC component will suffer from longer cycle time due to its sequential nature. The BΔI will not suffer much from longer execution time due to its parallel nature, but will use a larger area and consume more power. The Huffman based components will not increase in area or power consumption, but will use up more cycles to process an entire cache line. Lastly, the Huffman based components' area demands depends to a very large extent on their LUTs, this area demand will naturally vary with the number of entries in their LUTs, meaning that large VFT tables require a relatively high amount of area.

Once the different components had been evaluated for timing and area, the designs were synthesized with the Cadence Encounter RTL compiler for power evaluations. The designs were synthesized using a 65nm low-power library. The produced netlists were then evaluated for average switching behaviour by stimulating it with over 1000 test vectors using the simulation engine NC Sim. The test vectors used to stimulate the components where taken as a sample from test vectors used by Li Kang who had extracted those vectors from a test bench simulation [41]. From this stimulus, the Encounter RTL Compiler is able to create a SAIF-file with information regarding average gate switching activity. The produced SAIF-file in combination with the netlist is finally used to perform accurate power estimations of the circuit. The information from the power evaluations can be found in Table 4.2.

The Huffman-based components differ from the BΔI and FPC components in that before compression or decompression is performed, the internal LUTs must be initalised. The numbers presented in Table 4.2 do not include power evaluations for the initializational phase. This is because initializations will be active for a much smaller time than operation in compression/decompression mode.

To enable a fair power comparison between the different components, all components were synthesized with a timing requirement of 250 MHz. The exception is the FPC compressor, which requires a much slower speed than the other components, leading to a timing requirement of 10 MHz.

In Table 4.2, it can clearly be seen that Huffman compression consumes more energy than the other compression schemes. This is almost in its entirety due to the tables used by Huffman and this number is expected to drop in future prototypes when these tables will be implemented in SRAM rather than logic cells. When scaling, the different components scale differently with cache line length. The Huffman based components are sequential and will as such not demand a higher power to compress/decompress data as only more time is required to perform this action on a longer cache line. Parallel components such as the BΔI will experience a higher power demand though since their compression and decompression will require more circuits configured in parallel.

Table 4.2: Power comparison between different compression schemes

Component	**Static Power (nW)**	**Dynamic Power (nW)**	**Total Power (nW)**
FPC compressor	31386.614	3855152.884	3886539.498
FPC decompressor	5470.725	32532724.906	32538195.630
B∆I compressor	10174.935	48027626.610	48037801.545
B∆I decompressor	5868.509	29455867.910	29461736.419
Huffman compressor	27689.830	104756000.543	104783690.373
Compressor table	27490.893	93039478.883	93066969.778
Huffman decompressor	28392.924	118673244.507	118701637.431
Decompressor table	27611.963	103763655.199	103791267.162
Write component	28838.013	106389380.602	106418218.615
Write initaliser	90.108	272869.421	272959.529
Read component	30611.493	145903854.787	145934466.280
Read initaliser	634.259	516567.854	517202.112

5

Discussion

Table 3.1 shows that the CF for the text-based datasets is quite low. Text-based, UTF-8 encoded datasets have common letters encoded with one byte and uncommon letters encoded with a string of bytes. Most symbols in English text are likely to be in the English alphabet (we simplify and do not consider symbols such as escape characters, punctuation etc.). When we are generating the VFT we consider 32-bit values which correspond to four characters from the English alphabet. This gives us a total of $26^4 = 456976$ different combinations of four letters (again we simplify and do not consider that some letters have higher expected frequency). A VFT of size 128 or 1024 can only capture a small fraction of these combinations and the low VFT coverage is likely to result in a low CF.

The four memory samples taken from the DA benchmark have a large amount of null pages, between 21.53% and 56.41% as can be seen in Table 3.5. The evaluated compression algorithms are likely to give a very high CF for these pages which would skew the total CF for the memory sample. In this thesis we have reported the CF for the non-null pages since null-pages are often referred to as unused memory.

The GA benchmark ran with two different datasets: a Twitter dataset and a Google Plus dataset. The Google Plus dataset had to be transformed into a format accepted by the GA benchmark. This transformation affected the dataset which lost several attributes that characterise big graphs. Therefore, no conclusions were drawn from the analysis of the Google Plus graph as we deemed its results very unreliable. However, we chose to present our results so that interested readers can draw their own conclusions.

In the block alignment analysis we saw that the size alignment scheme gave a lower loss in CF than the CF alignment scheme. The CF alignment scheme has eight sub-blocks whereas the size alignment scheme has sixteen. Since the size alignment has twice as many sub-blocks it is hard to evaluate which is the best alignment scheme. The loss in CF for the CF alignment scheme would be reduced with more sub-blocks added for compression factors between one and two. Table 3.22 shows the block ranges and the slack for the CF alignment scheme. The slack for a CF of one is 31 bytes. This means that a block compressed to a size of 33 bytes will have the same size in the memory as uncompressed blocks.

In the block and page alignment analysis we saw that the distribution of pages and blocks was rather uneven between the different sub-block and sub-page sizes. It might be possible to select some of these sub-blocks and sub-page sizes and without losing too much of the CF. However, one would have to reduce the number of sub-blocks or sub-pages to half in order to remove one of the added addressing bits.

Even though an extensive compression analysis has been performed, and a proof-of-concept has been developed, it is still hard to accurately predict the gains from performing memory compression. The architecture of computing systems has developed over the years, making modern day systems highly complex machines. This means that to achieve accurate readings on the CF of a system implementing memory compression, one might have to fully implement such a system.

6 Conclusion

The analysis made in this thesis confirms that memory compression is an interesting technique for systems benefiting from larger memories or from reduced power consumption. In this thesis we have evaluated large-scale out systems that are running memory intensive applications requiring large memories. Battery-powered devices would also benefit from the possibility of using smaller memories with comparable performance as this will reduce the power consumption.

CloudSuite's Data Analytics and Graph Analytics scale-out benchmarks have been evaluated and we have shown that the memory used by these benchmarks can be compressed by a factor of about 2.7 using Huffman compression. The Huffman compression factor was calculated for non-null blocks and we have also shown that the Huffman algorithm yields better result than other compression algorithms including BΔI and FPC.

Huffman compression makes use of a value frequency table (VFT) which consists of values and their expected frequency. A larger VFT will increase compression but will have implication on hardware in terms of speed, energy consumption, and area. We have seen that VFT sizes over 1024 entries give very small improvements in compression. The Data Analytics and Graph Analytics benchmarks' Huffman compression factors are increased by on average 2.23% and 0.8%, respectively, when increasing the VFT size from 1024 to 4096 entries.

We have also shown that it is possible to maintain good Huffman compressibility even though the Huffman VFT has been generated from a small portion of data. The compressibility for the Graph Analytics and Data Analytics benchmarks does not seem to deteriorate over time, although, it is very important when the VFT is generated and changes in memory-intensive applications may affect the Huffman compressibility since the content in the VFT may no longer be relevant.

Traditional computer systems have memory blocks and pages that are fixed in size. In a computer system employing compression, blocks and pages must be allowed to have varying sizes. We have shown two different block alignment schemes and seen that the loss in compression when adding four additional block addressing bits is between 4% and 7%. We have also shown in a similar page alignment analysis that the loss in compression due to additional page alignment is roughly 6%-9% and 1%-2%, respectively, for four and six added page addressing bits.

The successful creation of a proof-of-concept hardware makes us conclude that it is feasible to create a system that implements memory compression. However, there are still problems that need to be solved before a fully implemented memory compression scheme can be used. To solve these questions and to improve upon existing solutions, more research in the area is required. A research that we feel is validated with the compression results presented in this project and with the successful creation of a proof-of-concept.

6.1 Future work

The hardware created in this project can be used as a base for future prototypes of memory compressing systems. Although it works as a basic proof-of-concept for memory compression, there are still many questions that need to be answered and problems to be solved. Problems that need to be solved for a fully working prototype include both hardware and software development. Static tables are used to initialize the compression and decompression units. To optimize compressibility, a VFT should be created dynamically from sampling the memory traffic. The VFT should then be used to create the Huffman tree which can be used by the compressor and decompressor.

At the moment, a compressed line is saved at the same starting point in memory as an uncompressed line would be. If a 32-byte cache line that normally is stored in byte addresses 0-31 has a compressed size of 8 bytes, stored in addresses 0-7, then the next cache line should start at address 8. However, in the developed hardware, the second cache line is still stored at starting address 32. An address translation needs to be created in the future to pack compressed lines closely together in memory. The hardware implemented in the proof-of-concept assumes that all values sent from a cache line are valid. Future prototypes should set invalid values to zero as this maximizes compressibility. Also, by adding the additional LO bits in the compression stage it is possible to perform faster decompression. This is not used by the proof-of-concept and should be implemented in the future.

6.1.1 Compressor improvements

There are currently two known opportunities for improvement on the Huffman compression hardware. The first improvement targets the bottleneck of the entire compression logic, the fact that the compression logic requires a steady input for four cycles limits compression speed. The Huffman compression stage of the compression hardware should be made pipelined to enable compression of one word per cycle. The second improvement is to include the AXI interface logic into the FIFOs located at input and output of the compression component.

6.1.2 Decompressor improvements

The decompressor logic for the hardware implemented Huffman algorithm has two known possibilities to improvement. Firstly, the requirement to flush the decompressor after a read cache line severely affects performance when reading several consecutive lines of data. An easy solution to this problem would be to implement two decompressor units and issue every other read to the units. The output could also be alternated between the two decompressor units, making the two units transparent to surrounding logic. The second improvement is to include the AXI interface logic into the FIFOs located at input and output of the compression component.

Bibliography

[1] Y. Durand, P. M. Carpenter, S. Adami, A. Bilas, D. Dutoit, A. Farcy, G. Gaydadjiev, J. Goodacre, M. Katevenis, M. Marazakis, E. Matus, I. Mavroidis, and J. Thomson, “EUROSERVER: Energy Efficient Node for European Micro-servers,” in *Digital System Design (DSD), 17th Euromicro Conference on*, pp. 206–213, HPCA, IEEE, Aug 2014.

[2] A. Arelakis and P. Stenström, “SC2: A Statistical Compression Cache Scheme,” in *Proceeding of the 41st annual international symposium on Computer Architecture*, pp. 145–156, ISCA, IEEE, NJ, USA, June 2014.

[3] A. Beloglazov, J. Abawajy, and R. Buyya, “Energy-aware resource allocation heuristics for efficient management of data centers for Cloud computing,” in *Future Generation Computer Systems*, vol. 28 of *5*, (Amsterdam, The Netherlands), p. 755–768, Elsevier Science Publishers B. V., May 2012.

[4] Xilinx, Inc, “Zynq-7000 All Programmable SoC Technical Reference Manual.” `http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf`, Feb 2015.

[5] M. Kjelsù, M. Gooch, and S. Jones, “Performance evaluation of computer architectures with main memory data compression,” in *Journal of Systems Architecture*, vol. 45, pp. 571–590, Elsevier Science B.V., Feb 1999.

[6] M. Seok, S. Hanson, Y.-S. Lin, Z. Foo, D. Kim, Y. Lee, N. Liu, D. Sylvester, and D. Blaauw, “The Phoenix Processor: A 30pW Platform for Sensor Applications,” in *Symposium on VLSI Circuits Digest of Technical Papers*, (Honolulu, HI, USA), pp. 188 – 189, IEEE, Jun 2008.

[7] M. Ekman and P. Stenström, “A robust main-memory compression scheme,” in *ISCA ’05: Proceedings of the 32nd annual international symposium on Computer Architecture*, p. 74–85, IEEE Computer Society, Jun 2005.

[8] G. Pekhimenko, V. Seshadri, Y. Kim, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Linearly Compressed Pages: A Low-Complexity, Low-Latency Main Memory Compression Framework," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, (New York, NY, USA), pp. 172–184, ACM, Dec 2013.

[9] L. Yang, H. Lekatsas, and R. P. Dick, "High-Performance Operating System Controlled Memory Compression," in *Design Automation Conference*, vol. 43, (San Francisco, CA, USA), pp. 701–704, Jul 2006.

[10] I. C. Tuduce and T. Gross, "Adaptive Main Memory Compression," in *USENIX Annual Technical Conference*, pp. 237–250, 2005.

[11] H. Franke, B. Abali, L. M. Herger, D. E. Poff, R. A. S. Jr., and T. B. Smith, "METHOD FOR OPERATING SYSTEM SUPPORT FOR MEMORY COMPRESSION," in *United States Patent no. 6,681,305* (IBM, ed.), Jan 2004.

[12] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Base-Delta-Immediate Compression: A Practical Data Compression Mechanism for On-Chip Caches," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pp. 377–388, ACM, Sep 2012.

[13] A. R. Alameldeen and D. A. Wood, "Frequent Pattern Compression: A Significance-Based Compression Scheme for L2 Caches," tech. rep., Computer Sciences Department, University of Wisconsin-Madison, Apr 2004.

[14] J. Dusser, T. Piquet, and A. Seznec, "Zero-Content Augmented Caches," in *ICS '09 Proceedings of the 23rd international conference on Supercomputing*, pp. 46–55, ACM, Jun 2009.

[15] R. B. Tremaine, P. A. Franaszek, J. T. Robinson, C. O. Schulz, T. B. Smith, M. E. Wazlowski, and P. M. Bland, "IBM Memory Expansion Technology (MXT)," in *IBM Journal of Research and Development* (IBM, ed.), vol. 45, (Riverton, NJ, USA), pp. 271–285, Mar 2001.

[16] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the Clouds - A Study of Emerging Scale-out Workloads on Modern Hardware," in *ASPLOS XVII Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, pp. 37–48, ACM, ACM New York, NY, USA, Mar 2012.

[17] V. Karakostas, O. S. Unsal, M. Nemirovsky, A. Cristal, and M. Swift, "Performance Analysis Of The Memory Management Unit Under Scale-Out Workloads," in *International Symposium on Workload Characterization*, IEEE, Oct 2014.

[18] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, O. Kocberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Ozer, and B. Falsafi, "Scale-Out Processors," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, pp. 500–511, ISCA, IEEE Computer Society Washington, DC, USA, Jun 2012.

[19] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu, "BigDataBench: a Big Data Benchmark Suite from Internet Services," in *High Performance Computer Architecture, IEEE 20th International Symposium on*, pp. 488–499, HPCA, IEEE Orlando, FL, USA, Feb 2014.

[20] EPFL PARSA, "CloudSuite." `http://parsa.epfl.ch/cloudsuite/cloudsuite.html`, November 2014.

[21] T. Jiang, Q. Zhang, R. Hou, L. Chai, S. A. McKee, Z. Jia, and N. Sun, "Understanding the Behavior of In-Memory Computing Workloads," in *International Symposium on Workload Characterization (IISWC)*, IEEE, 2014.

[22] O. Kocberber, "Rigorous and Practical Server Design Evaluation." `http://parsa.epfl.ch/cloudsuite/docs/CloudSuite2.0-on-Flexus-ispass14.pdf`, March 2014.

[23] EPFL PARSA, "CloudSuite Overview." `http://parsa.epfl.ch/cloudsuite/overview.html`, November 2014.

[24] D. Tunkelang, "A Twitter Analogue to PageRank." `http://thenoisychannel.com/2009/01/13/a-twitter-analog-to-pagerank.html`, January 2015.

[25] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Communications of the ACM*, vol. 17, pp. 412–421, Jul 1974.

[26] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. Martins, A. V. Anderson, S. M. Bennett, A. Kägi, F. H. Leung, and L. Smith, "Intel Virtualization Technology," *Computer*, vol. 38, pp. 48–56, May 2005.

[27] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in *USENIX Annual Technical Conference*, pp. 41–46, USENIX, Apr 2005.

[28] K. Sayood, *Introduction to Data Compression.* 225 Wyman Street, MA, USA: Elsevier, Inc, fourth ed., 2012.

[29] D. A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," in *Proceedings of the IRE*, vol. 40, pp. 1098–1101, IEEE, Sep 1952.

[30] X. Chen, L. Yang, R. Dick, L. Shang, and H. Lekatsas, "C-pack: A high-performance microprocessor cache compression algorithm," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 18, pp. 1196–1208, Aug 2010.

[31] D. Giannopoulos, "Hardware Implementations of Memory Compression Schemes and Timing, Space and Power Analysis," bachelor thesis, Chalmers University of Technology and University of Crete, 2015.

[32] Avnet, "ZedBoard (Zynq™Evaluation and Development) Hardware User's Guide." `http://zedboard.org/sites/default/files/documentations/ZedBoard_HW_UG_v2_2.pdf`, Jan 2014.

[33] ARM, "AMBA® AXI™ and ACE™ Protocol Specification," Oct 2011.

[34] D. Chen, J. Cong, and P. Pan, "FPGA Design Automation: A Survey," in *Foundations and Trends in Electronic Design Automation*, vol. 1, p. 195–330, now publishers inc, Nov 2006.

[35] TECHTUNK, "TECHTUNK Entertainment :: Social and Gaming Development." `http://www.techtunk.com/index.php?dove=downloads`, April 2015.

[36] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection." `http://snap.stanford.edu/data`, June 2014.

[37] Wikipedia, "Index of /enwiki/latest/." `http://dumps.wikimedia.org/enwiki/latest/enwiki-latest-pages-articles.xml.bz2`, Dec. 2014.

[38] J. Gandhi, A. Basu, M. D. Hill, and M. M. Swift, "Efficient memory virtualization: Reducing dimensionality of nested page walks," pp. 178–189, IEEE, 2014.

[39] M. Cha, H. Haddadi, F. Benevenuto, and K. P. Gummadi, "Measuring User Influence in Twitter: The Million Follower Fallacy," in *Proceedings of the 4th International AAAI Conference on Weblogs and Social Media (ICWSM)*, (Washington DC, USA), May 2010.

[40] FORTH, "Foundation for research and technology - hellas." `http://www.forth.gr/`, June 2015.

[41] L. Kang, "Design and implementation of a decompression engine for a Huffman-based compressed data cache," master thesis, Chalmers University of Technology, Jan 2014.

[42] Xilinx, "Xilinx Software Development Kit (SDK)." `http://www.xilinx.com/tools/sdk.htm`, April 2015.

A

Appendix A

A.1 Data analytics, DA_z0

Table A.1: Data analytics, DA_z0, summary

Application Details	
Application	**Data analytics (1min)**
Data Set	Wikipedia
Size (GB)	3.79

VFT Size Table	
VFT Size	**% of Total Values in VFT**
128	83.39%
256	84.70%
512	85.81%
1K	86.65%
2K	87.35%
4K	87.99%
8K	88.62%
16K	89.16%
32K	89.62%

Sample Values		
Num Pages	924785	
Null Pages	420126	45.43%
Non-Null Pages	504659	54.57%
Null Blocks	28029567	
Page Size (Byte)	4096	
VFT Entry Size	4	

Table A.2: Data analytics - DA_z0, VFT content

VFT Content Table					
Rank Value Frequency	**Endian**		**Frequency**	**Frequency % of total Values**	**Frequency of % Total Values Without Null Pages**
	Big	**Little**			
1	0	0	641848832	67.78%	22.35%
2	1	16777216	28496104	3.01%	5.51%
3	3879999628	2350400743	6335488	0.67%	1.23%
4	3879993535	3204465895	6324879	0.67%	1.22%
5	5	83886080	6274432	0.66%	1.21%
6	2	33554432	6003253	0.63%	1.16%
7	7	117440512	5961139	0.63%	1.15%
8	9	150994944	3913307	0.41%	0.76%
9	3	50331648	3776611	0.40%	0.73%
10	11	184549376	3092392	0.33%	0.60%
11	121	2030043136	2567001	0.27%	0.50%
12	256	65536	2504535	0.26%	0.48%
13	4	67108864	2456227	0.26%	0.48%
14	6	100663296	2322151	0.25%	0.45%
15	4294967295	4294967295	2309078	0.24%	0.45%
16	16777216	1	2089089	0.22%	0.40%
17	16843009	16843009	2073376	0.22%	0.40%
18	65536	256	1936888	0.21%	0.37%
19	8	134217728	1890893	0.20%	0.37%
20	15	251658240	1582912	0.17%	0.31%
21	16	268435456	1512435	0.16%	0.29%
22	257	16842752	1505488	0.16%	0.29%
23	16777472	65537	1494217	0.16%	0.29%
24	3879993890	570574055	1482830	0.16%	0.29%
25	16777217	16777217	1473329	0.16%	0.29%

A.2 Data analytics, DA_z1

Table A.3: Data analytics, DA_z1, summary

Application Details	
Application	**Data analytics (1h)**
Data Set	Wikipedia
Size (GB)	3.58

VFT Size Table	
VFT Size	**% of Total Values in VFT**
128	88.33%
256	89.01%
512	89.62%
1K	90.06%
2K	90.40%
4K	90.71%
8K	91.04%
16K	91.38%
32K	91.73%

Sample Values		
Num Pages	874715	
Null Pages	493464	56.41%
Non-Null Pages	381251	43.59%
Null Blocks	33645497	
Page Size (Byte)	4096	
VFT Entry Size	4	

Table A.4: Data analytics DA_z1, VFT content

VFT Content Table					
Rank Value Frequency	**Endian**		**Frequency**	**Frequency % of total Values**	**Frequency of % Total Values Without Null Pages**
	Big	**Little**			
1	0	0	683719485	76.33%	19.92%
2	1	16777216	19319314	2.16%	4.95%
3	9	150994944	5281244	0.59%	1.35%
4	5	83886080	4711529	0.53%	1.21%
5	2	33554432	3929824	0.44%	1.01%
6	3	50331648	3589940	0.40%	0.92%
7	7	117440512	3323895	0.37%	0.85%
8	11	184549376	3300766	0.37%	0.85%
9	17	285212672	2642362	0.30%	0.68%
10	3879993535	3204465895	2539253	0.28%	0.65%
11	3879999628	2350400743	2493344	0.28%	0.64%
12	4294967295	4294967295	2345796	0.26%	0.60%
13	16843009	16843009	1994145	0.22%	0.51%
14	3879993890	570574055	1937288	0.22%	0.50%
15	3879993748	2483111143	1929114	0.22%	0.49%
16	3879993677	1291928807	1823722	0.20%	0.47%
17	256	65536	1789507	0.20%	0.46%
18	15	251658240	1735932	0.19%	0.44%
19	4	67108864	1677082	0.19%	0.43%
20	13	218103808	1398552	0.16%	0.36%
21	16777216	1	1286374	0.14%	0.33%
22	31	520093696	1228435	0.14%	0.31%
23	6	100663296	1220589	0.14%	0.31%
24	8	134217728	1142513	0.13%	0.29%
25	65536	256	1112936	0.12%	0.29%

A.3 Data analytics, DA_z2

Table A.5: Data analytics, DA_z2, summary

Application Details	
Application	**Data analytics (6h)**
Data Set	Wikipedia
Size (GB)	3.42

VFT Size Table	
VFT Size	**% of Total Values in VFT**
128	88.25%
256	89.40%
512	90.26%
1K	90.91%
2K	91.33%
4K	91.67%
8K	92.00%
16K	92.31%
32K	92.62%

Sample Values		
Num Pages	833936	
Null Pages	470928	56.47%
Non-Null Pages	363008	43.53%
Null Blocks	30970691	
Page Size (Byte)	4096	
VFT Entry Size	4	

Table A.6: Data analytics, DA_z2, VFT content

Rank Value Frequency	**Endian**		**Frequency**	**Frequency % of total Values**	**Frequency of % Total Values Without Null Pages**
	Big	**Little**			
1	0	0	625799897	73.28%	16.81%
2	1	16777216	15641070	1.83%	4.21%
3	5	83886080	8372327	0.98%	2.25%
4	3879993535	3204465895	5289407	0.62%	1.42%
5	9	150994944	5181475	0.61%	1.39%
6	4	67108864	5012378	0.59%	1.35%
7	3	50331648	4112225	0.48%	1.11%
8	3879999628	2350400743	3867541	0.45%	1.04%
9	7	117440512	3535982	0.41%	0.95%
10	6	100663296	3455874	0.41%	0.93%
11	4294967295	4294967295	3363252	0.39%	0.90%
12	8	134217728	3080539	0.36%	0.83%
13	11	184549376	3026768	0.35%	0.81%
14	2	33554432	2845848	0.33%	0.77%
15	256	65536	2479646	0.29%	0.67%
16	3879993748	2483111143	2404020	0.28%	0.65%
17	16843009	16843009	1677476	0.20%	0.45%
18	121	2030043136	1525146	0.18%	0.41%
19	3879993890	570574055	1521214	0.18%	0.41%
20	15	251658240	1481391	0.17%	0.40%
21	3879993677	1291928807	1457928	0.17%	0.39%
22	17	285212672	1442109	0.17%	0.39%
23	1071644672	57407	1344753	0.16%	0.36%
24	1070176665	2576992575	1344727	0.16%	0.36%
25	2576980378	2593757593	1344510	0.16%	0.36%

A.4 Data analytics, DA_z3

Table A.7: Data analytics, DA_z3, summary

Application Details	
Application	**Data analytics (24h)**
Data Set	Wikipedia
Size (GB)	2.80

VFT Size Table	
VFT Size	**% of Total Values in VFT**
128	77.65%
256	79.60%
512	81.32%
1K	82.37%
2K	83.03%
4K	83.69%
8K	84.38%
16K	85.00%
32K	85.52%

Sample Values		
Num Pages	682955	
Null Pages	147043	21.53%
Non-Null Pages	535912	78.47%
Null Blocks	10031581	
Page Size (Byte)	4096	
VFT Entry Size	4	

Table A.8: Data analytics, DA_z3, VFT content

VFT Content Table					
Rank Value Frequency	**Endian**		**Frequency**	**Frequency % of total Values**	**Frequency of % Total Values Without Null Pages**
	Big	**Little**			
1	0	0	379855750	54.32%	32.79%
2	1	16777216	36514330	5.22%	6.65%
3	5	83886080	9061993	1.30%	1.65%
4	2	33554432	7255949	1.04%	1.32%
5	3879993535	3204465895	7087794	1.01%	1.29%
6	3879999628	2350400743	7058551	1.01%	1.29%
7	7	117440512	5696732	0.82%	1.04%
8	9	150994944	5113240	0.73%	0.93%
9	3	50331648	4315928	0.62%	0.79%
10	4	67108864	3023922	0.43%	0.55%
11	256	65536	2727923	0.39%	0.50%
12	16777216	1	2462390	0.35%	0.45%
13	11	184549376	2444911	0.35%	0.45%
14	4294967295	4294967295	2413348	0.35%	0.44%
15	65536	256	2335588	0.33%	0.43%
16	6	100663296	2326264	0.33%	0.42%
17	16843009	16843009	2178648	0.31%	0.40%
18	16	268435456	2152227	0.31%	0.39%
19	8	134217728	1963128	0.28%	0.36%
20	3880474706	1381518311	1864058	0.27%	0.34%
21	17	285212672	1794985	0.26%	0.33%
22	257	16842752	1737745	0.25%	0.32%
23	16777472	65537	1702123	0.24%	0.31%
24	16777217	16777217	1700302	0.24%	0.31%
25	65537	16777472	1635051	0.23%	0.30%

A.5 Graph analytics, GA_z0

Table A.9: Graph analytics, GA_z0, summary

Application Details	
Application	**Graph analytics (1min)**
Data Set	Twitter
Size (GB)	3.89

VFT Size Table	
VFT Size	**% of Total Values in VFT**
128	31.34%
256	32.99%
512	34.65%
1K	35.66%
2K	36.66%
4K	37.87%
8K	39.37%
16K	41.18%
32K	43.42%

Sample Values		
Num Pages	948827	
Null Pages	90776	9.57%
Non-Null Pages	858051	90.43%
Null Blocks	5891343	
Page Size (Byte)	4096	
VFT Entry Size	4	

Table A.10: Graph analytics, GA_z0, VFT content

Rank Value Frequency	**Endian**		**Frequency**	**Frequency % of total Values**	**Frequency of % Total Values Without Null Pages**
	Big	**Little**			
1	8835584	13796864	171263067	17.63%	19.49%
2	0	0	103274819	10.63%	8.06%
3	4294967295	4294967295	3616724	0.37%	0.41%
4	2470119	3887080704	499602	0.05%	0.06%
5	19058681	4191101441	497550	0.05%	0.06%
6	522239	4294379264	344851	0.04%	0.04%
7	428333	763954688	340985	0.04%	0.04%
8	677075	3545500160	340389	0.04%	0.04%
9	19397785	2583439105	332493	0.03%	0.04%
10	18617047	3608288257	332490	0.03%	0.04%
11	334246	2786657536	326627	0.03%	0.04%
12	16190898	2987259648	314318	0.03%	0.04%
13	1591947	2336888832	313295	0.03%	0.04%
14	2256645	91169280	307424	0.03%	0.03%
15	831319	1471089664	307329	0.03%	0.03%
16	1302597	1172312832	304938	0.03%	0.03%
17	13284411	1001703936	298472	0.03%	0.03%
18	16409683	1399126528	285178	0.03%	0.03%
19	13339129	4186557184	281998	0.03%	0.03%
20	18220175	2399409665	276207	0.03%	0.03%
21	9976334	238721024	275213	0.03%	0.03%
22	19554706	2455841281	271768	0.03%	0.03%
23	17461978	3664906753	270464	0.03%	0.03%
24	15846407	130871552	266272	0.03%	0.03%
25	8626708	346194688	254907	0.03%	0.03%

A.6 Graph analytics, GA_z1

Table A.11: Graph analytics, GA_z1, summary

Application Details	
Application	**Graph analytics (1h)**
Data Set	Twitter
Size (GB)	3.60

VFT Size Table	
VFT Size	**% of Total Values in VFT**
128	70.16%
256	70.95%
512	71.49%
1K	71.85%
2K	72.28%
4K	72.60%
8K	72.93%
16K	73.29%
32K	73.71%

Sample Values		
Num Pages	878096	
Null Pages	616	0.07%
Non-Null Pages	877480	99.93%
Null Blocks	490321	
Page Size (Byte)	4096	
VFT Entry Size	4	

Table A.12: Graph analytics, GA_z1, VFT content

VFT Content Table					
Rank Value Frequency	**Endian**		**Frequency**	**Frequency % of total Values**	**Frequency of % Total Values Without Null Pages**
	Big	**Little**			
1	0	0	411065192	45.72%	45.65%
2	6305648	1882677248	51214082	5.70%	5.70%
3	1065353216	32831	29217439	3.25%	3.25%
4	32512	8323072	25611797	2.85%	2.85%
5	1483198464	13657944	25606886	2.85%	2.85%
6	5023488	10963968	25502977	2.84%	2.84%
7	1	16777216	17849224	1.99%	1.99%
8	2	33554432	9346828	1.04%	1.04%
9	4294967295	4294967295	7026800	0.78%	0.78%
10	3	50331648	5487270	0.61%	0.61%
11	4	67108864	3465020	0.39%	0.39%
12	5	83886080	2240048	0.25%	0.25%
13	6	100663296	1477742	0.16%	0.16%
14	7	117440512	1008810	0.11%	0.11%
15	8	134217728	973153	0.11%	0.11%
16	8835584	13796864	834624	0.09%	0.09%
17	9	150994944	600805	0.07%	0.07%
18	10	167772160	436283	0.05%	0.05%
19	11	184549376	357429	0.04%	0.04%
20	8781824	34304	316430	0.04%	0.04%
21	12	201326592	298426	0.03%	0.03%
22	13	218103808	253804	0.03%	0.03%
23	65536	256	249620	0.03%	0.03%
24	677075	3545500160	238131	0.03%	0.03%
25	2470119	3887080704	229835	0.03%	0.03%

A.7 Graph analytics, GA_z2

Table A.13: Graph analytics, GA_z2, summary

Application Details	
Application	**Graph analytics (6h)**
Data Set	Twitter
Size (GB)	3.68

VFT Size Table	
VFT Size	**% of Total Values in VFT**
128	67.27%
256	68.26%
512	68.94%
1K	69.42%
2K	69.99%
4K	70.56%
8K	71.18%
16K	71.87%
32K	72.68%

Sample Values		
Num Pages	897536	
Null Pages	549	0.06%
Non-Null Pages	896987	99.94%
Null Blocks	96080	
Page Size (Byte)	4096	
VFT Entry Size	4	

Table A.14: Graph analytics, GA_z2, VFT content

Rank Value Frequency	**Endian**		**Frequency**	**Frequency % of total Values**	**Frequency of % Total Values Without Null Pages**
	Big	**Little**			
1	0	0	400792046	43.61%	43.55%
2	6305648	1882677248	50732303	5.52%	5.52%
3	1065353216	32831	30812272	3.35%	3.35%
4	32512	8323072	25371628	2.76%	2.76%
5	1483198464	13657944	25366091	2.76%	2.76%
6	5023488	10963968	21930307	2.39%	2.39%
7	1	16777216	17377757	1.89%	1.89%
8	2	33554432	8538687	0.93%	0.93%
9	4294967295	4294967295	5766618	0.63%	0.63%
10	3	50331648	5072874	0.55%	0.55%
11	4	67108864	3205950	0.35%	0.35%
12	5	83886080	2079567	0.23%	0.23%
13	6	100663296	1375229	0.15%	0.15%
14	7	117440512	942161	0.10%	0.10%
15	65536	256	898639	0.10%	0.10%
16	256	65536	805370	0.09%	0.09%
17	16777216	1	761343	0.08%	0.08%
18	8	134217728	735192	0.08%	0.08%
19	9	150994944	595950	0.07%	0.06%
20	8835584	13796864	418495	0.05%	0.05%
21	10	167772160	409116	0.05%	0.04%
22	11	184549376	335238	0.04%	0.04%
23	677075	3545500160	310973	0.03%	0.03%
24	2470119	3887080704	306947	0.03%	0.03%
25	12	201326592	279941	0.03%	0.03%

A.8 Graph analytics, GA_z3

Table A.15: Graph analytics,GA_z3, summary

Application Details	
Application	**Graph analytics (24h)**
Data Set	Twitter
Size (GB)	3.69

VFT Size Table	
VFT Size	**% of Total Values in VFT**
128	65.88%
256	66.88%
512	67.59%
1K	68.05%
2K	68.58%
4K	69.19%
8K	69.87%
16K	70.62%
32K	71.45%

Sample Values		
Num Pages	901795	
Null Pages	550	0.06%
Non-Null Pages	901245	99.94%
Null Blocks	77656	
Page Size (Byte)	4096	
VFT Entry Size	4	

Table A.16: Graph analytics, GA_z3, VFT content

VFT Content Table					
Rank Value Frequency	**Endian**		**Frequency**	**Frequency % of total Values**	**Frequency of % Total Values Without Null Pages**
	Big	**Little**			
1	0	0	415353956	44.98%	44.92%
2	6305648	1882677248	51533817	5.58%	5.58%
3	32512	8323072	25770280	2.79%	2.79%
4	1483198464	13657944	25766779	2.79%	2.79%
5	1	16777216	17923136	1.94%	1.94%
6	1065353216	32831	11278467	1.22%	1.22%
7	2	33554432	8324206	0.90%	0.90%
8	5023488	10963968	8134984	0.88%	0.88%
9	4294967295	4294967295	5865324	0.64%	0.64%
10	3	50331648	4898767	0.53%	0.53%
11	65536	256	3179427	0.34%	0.34%
12	4	67108864	3088534	0.33%	0.33%
13	256	65536	3085368	0.33%	0.33%
14	5	83886080	2000029	0.22%	0.22%
15	16777217	16777217	1616926%	0.18%	0.18%
16	16777216	1	1532130	0.17%	0.17%
17	6	100663296	1326846	0.14%	0.14%
18	7	117440512	912766	0.10%	0.10%
19	8	134217728	756376	0.08%	0.08%
20	9	150994944	583279	0.06%	0.06%
21	8835584	13796864	455821	0.05%	0.05%
22	10	167772160	399972	0.04%	0.04%
23	1065772646	1717995071	382649	0.04%	0.04%
24	11	184549376	328372	0.04%	0.04%
25	677075	3545500160	307752	0.03%	0.03%

A.9 Graph analytics, GA_g_0

Table A.17: Graph analytics, GA_g_0, summary

Application Details	
Application	**Graph analytics (1min)**
Data Set	Google Plus
Size (GB)	3.82

VFT Size Table	
VFT Size	**% of Total Values in VFT**
128	0.28%
256	0.41%
512	0.63%
1K	1.01%
2K	1.65%
4K	2.76%
8K	4.60%
16K	7.58%
32K	12.23%

Sample Values		
Num Pages	931701	
Null Pages	641	0.07%
Non-Null Pages	931060	99.93%
Null Blocks	55701	
Page Size (Byte)	4096	
VFT Entry Size	4	

Table A.18: Graph analytics, GA_g_0, VFT content

Rank Value Frequency	**Endian Big**	**Endian Little**	**Frequency**	**Frequency % of total Values**	**Frequency of % Total Values Without Null Pages**
1	0	0	1049237	0.11%	0.041%
2	3552044	741553664	21199	0.00%	0.002%
3	3659655	2279028480	19934	0.00%	0.002%
4	2798767	2947820032	19673	0.00%	0.002%
5	3767266	3799726336	19557	0.00%	0.002%
6	3336822	1995059712	18826	0.00%	0.002%
7	3982488	2562997248	18655	0.00%	0.002%
8	3444433	3515757568	18339	0.00%	0.002%
9	3013989	1711090944	18319	0.00%	0.002%
10	3874877	1025522432	18059	0.00%	0.002%
11	2906378	173616128	18058	0.00%	0.002%
12	4090099	4083695104	17608	0.00%	0.002%
13	3229211	457584896	16879	0.00%	0.002%
14	4197710	1309491200	16180	0.00%	0.002%
15	1507435	1795168000	16113	0.00%	0.002%
16	3551266	573584896	15144	0.00%	0.002%
17	3121600	3231788800	15054	0.00%	0.002%
18	2691156	1410345216	15043	0.00%	0.002%
19	3551258	439367168	14811	0.00%	0.002%
20	2905592	4166331392	14548	0.00%	0.002%
21	3443647	3213571072	14513	0.00%	0.002%
22	2797981	2645633536	14401	0.00%	0.002%
23	3766480	3497539840	14212	0.00%	0.001%
24	3443655	3347788800	14185	0.00%	0.001%
25	3658877	2111059712	14034	0.00%	0.001%

A.10 Graph analytics, GA_g_100

Table A.19: Graph analytics, GA_g_100, summary

Application Details	
Application	**Graph analytics (1h)**
Data Set	Google Plus
Size (GB)	3.64

VFT Size Table	
VFT Size	**% of Total Values in VFT**
128	10.87%
256	11.05%
512	11.33%
1K	11.76%
2K	12.47%
4K	13.64%
8K	15.54%
16K	18.60%
32K	23.32%

Sample Values		
Num Pages	888583	
Null Pages	538	0.06%
Non-Null Pages	888045	99.94%
Null Blocks	69055	
Page Size (Byte)	4096	
VFT Entry Size	4	

Table A.20: Graph analytics, GA_g_100, VFT content

Rank Value Frequency	**Endian**		**Frequency**	**Frequency % of total Values**	**Frequency of % Total Values Without Null Pages**
	Big	**Little**			
1	0	0	65423260	7.19%	7.13%
2	6305648	1882677248	6885719	0.76%	0.76%
3	32512	8323072	3443614	0.38%	0.38%
4	3507781632	8393937	3442882	0.38%	0.38%
5	1	16777216	2240176	0.25%	0.25%
6	1065353216	32831	2072532	0.23%	0.23%
7	4294967295	4294967295	1908442	0.21%	0.21%
8	5023488	10963968	1671419	0.18%	0.18%
9	196608	768	1112752	0.12%	0.12%
10	327680	1280	985030	0.11%	0.11%
11	393216	1536	676650	0.07%	0.07%
12	8835584	13796864	610038	0.07%	0.07%
13	262144	1024	584293	0.06%	0.06%
14	2	33554432	481281	0.05%	0.05%
15	3	50331648	414638	0.05%	0.05%
16	4	67108864	313180	0.03%	0.03%
17	5	83886080	270337	0.03%	0.03%
18	8781824	34304	237530	0.03%	0.03%
19	6	100663296	237052	0.03%	0.03%
20	8	134217728	230120	0.03%	0.03%
21	131072	512	229444	0.03%	0.03%
22	7	117440512	220693	0.02%	0.02%
23	9	150994944	180303	0.02%	0.02%
24	10	167772160	164706	0.02%	0.02%
25	11	184549376	145517	0.02%	0.02%

A.11 Graph analytics, GA_g_130

Table A.21: Graph analytics, GA_g_130, summary

Application Details	
Application	**Graph analytics (6h)**
Data Set	Google Plus
Size (GB)	3.88

VFT Size Table	
VFT Size	**% of Total Values in VFT**
128	7.16%
256	7.34%
512	7.61%
1K	8.05%
2K	8.79%
4K	10.03%
8K	12.08%
16K	15.45%
32K	20.73%

Sample Values		
Num Pages	948308	
Null Pages	509	0.05%
Non-Null Pages	947799	99.95%
Null Blocks	63970	
Page Size (Byte)	4096	
VFT Entry Size	4	

Table A.22: Graph analytics, GA_g_130, VFT content

VFT Content Table					
Rank Value Frequency	**Endian**		**Frequency**	**Frequency % of total Values**	**Frequency of % Total Values Without Null Pages**
	Big	**Little**			
1	0	0	44796481	4.61%	4.56%
2	6305648	1882677248	6936845	0.71%	0.72%
3	3507781632	8393937	3468510	0.36%	0.36%
4	32512	8323072	3468246	0.36%	0.36%
5	4294967295	4294967295	1227056	0.13%	0.13%
6	196608	768	1126427	0.12%	0.12%
7	327680	1280	998149	0.10%	0.10%
8	1	16777216	752454	0.08%	0.08%
9	393216	1536	694199	0.07%	0.07%
10	262144	1024	559963	0.06%	0.06%
11	2	33554432	328393	0.03%	0.03%
12	3	50331648	282876	0.03%	0.03%
13	4	67108864	213253	0.02%	0.02%
14	8	134217728	184872	0.02%	0.02%
15	5	83886080	184365	0.02%	0.02%
16	131072	512	183915	0.02%	0.02%
17	6	100663296	160216	0.02%	0.02%
18	7	117440512	152982	0.02%	0.02%
19	8835584	13796864	125777	0.01%	0.01%
20	9	150994944	121989	0.01%	0.01%
21	10	167772160	111392	0.01%	0.01%
22	11	184549376	98047	0.01%	0.01%
23	12	201326592	96145	0.01%	0.01%
24	13	218103808	92038	0.01%	0.01%
25	14	234881024	85642	0.01%	0.01%

A.12 Dataset Google Plus

Table A.23: Dataset Google Plus, summary

Application Details	
Application	**Google Plus Small**
Files	791
Size (GB)	2.9318389893
Parsed Size (GB)	3.1
Compressed Size (GB)	
Character Size	1

VFT Size Table	
VFT Size	**% of Total Values in VFT**
128	79.44%
256	80.47%
512	81.99%
1K	83.51%
2K	84.67%
4K	86.17%
8K	88.11%
16K	90.48%
32K	93.18%
64K	95.97%
128K	98.30%

Sample Values		
Num Pages	768564	
Non-Null Pages	199173	25.91%
Null Pages	569391	74.09%
Page Size (Byte)	4096	
VFT Entry Size	4	
Num Values	787009536	
Null Values From Pages	583056384	
Non-Null Values From Pages	203953152	

Table A.24: Dataset Google Plus, VFT content

VFT Content Table					
Rank Value Frequency	**Endian**		**Frequency**	**Frequency % of total Values**	**Frequency of % Total Values Without Null Pages**
	Big	**Little**			
1	0	0	600576467	76.31%	8.59%
2	83886080	5	7841794	1.00%	3.84%
3	100663296	6	5670541	0.72%	2.78%
4	1	16777216	443622	0.06%	0.22%
5	65536	256	370761	0.05%	0.18%
6	256	65536	240124	0.03%	0.12%
7	16777216	1	206566	0.03%	0.10%
8	186	3120562176	117355	0.02%	0.06%
9	151	2533359616	116034	0.02%	0.06%
10	241	4043309056	112936	0.01%	0.06%
11	5	83886080	105339	0.01%	0.05%
12	17	285212672	102853	0.01%	0.05%
13	129	2164260864	100697	0.01%	0.05%
14	6	100663296	100319	0.01%	0.05%
15	158	2650800128	98720	0.01%	0.05%
16	26	436207616	98119	0.01%	0.05%
17	128	2147483648	97505	0.01%	0.05%
18	34	570425344	97008	0.01%	0.05%
19	89	1493172224	96900	0.01%	0.05%
20	30	503316480	95971	0.01%	0.05%
21	64	1073741824	93235	0.01%	0.05%
22	50	838860800	92652	0.01%	0.05%
23	61	1023410176	92516	0.01%	0.05%
24	327680	1280	91297	0.01%	0.04%
25	171	2868903936	91160	0.01%	0.04%

A.13 Dataset Twitter, Grammy

Table A.25: Dataset Twitter, Grammy, summary

Application Details	
Application	**Twitter Grammy**
Files	1
Size (GB)	0.3798
Parsed Size (GB)	0.3798
Compressed Size (GB)	
Character Encoding	UTF-8

VFT Size Table	
VFT Size	**% of Total Values in VFT**
128	32.24%
256	36.34%
512	41.20%
1K	47.25%
2K	54.33%
4K	62.03%
8K	70.38%
16K	78.13%
32K	84.47%

Sample Values		
Num Pages	92735	
Null Pages	0	0.00%
Non-Null Pages	92735	100.00%
Null Blocks	0	
Page Size (Byte)	4096	
VFT Entry Size	4	

Table A.26: Dataset Twitter, Grammy, VFT content

VFT Content Table					
Rank Value Frequency	**Endian**		**Frequency**	**Frequency % of total Values**	**Frequency of % Total Values Without Null Pages**
	Big	**Little**			
1	813445244	2083552304	3874136	4.08%	4.08%
2	2083552304	813445244	3232073	3.40%	3.40%
3	2083532336	808333436	1294354	1.36%	1.36%
4	774929456	813445166	1294038	1.36%	1.36%
5	808333436	2083532336	1293926	1.36%	1.36%
6	813445166	774929456	1290669	1.36%	1.36%
7	858861618	842019123	684335	0.72%	0.72%
8	540225840	808530720	672623	0.71%	0.71%
9	825242159	791818289	652528	0.69%	0.69%
10	791818364	2083533359	651842	0.69%	0.69%
11	825176624	808595249	651580	0.69%	0.69%
12	170949680	813445130	646059	0.68%	0.68%
13	1835884914	1918987629	603454	0.64%	0.64%
14	2037214561	1634561401	599158	0.63%	0.63%
15	807416625	825434160	455630	0.48%	0.48%
16	1835102791	1198678381	438906	0.46%	0.46%
17	825306930	841953585	378796	0.40%	0.40%
18	791753007	791753007	378105	0.40%	0.40%
19	841953585	825306930	378006	0.40%	0.40%
20	808595249	825176624	377942	0.40%	0.40%
21	1937337709	1835891059	334745	0.35%	0.35%
22	1701344288	544499813	325127	0.34%	0.34%
23	1075860562	1381244992	312254	0.33%	0.33%
24	1634879264	541553249	293876	0.31%	0.31%
25	543516788	1952998688	292998	0.31%	0.31%

A.14 Dataset Wikipedia_NB

Table A.27: Dataset Wikipedia_NB, summary

Application Details	
Application	**Wikipedia (No bin)**
Files	29853
Size (GB)	1.2
Parsed Size (GB)	0.4604
Compressed Size (GB)	
Character Encoding	UTF-8

VFT Size Table	
VFT Size	**% of Total Values in VFT**
128	32.24%
256	36.34%
512	41.20%
1K	47.25%
2K	54.33%
4K	62.03%
8K	70.38%
16K	78.13%
32K	84.47%

Sample Values		
Num Pages	112413	
Null Pages	0	0.00%
Non-Null Pages	112413	100.00%
Null Blocks	0	
Page Size (Byte)	4096	
VFT Entry Size	4	

Table A.28: Wikipedia_NB, VFT content

VFT Content Table					
Rank Value Frequency	**Endian**		**Frequency**	**Frequency % of total Values**	**Frequency of % Total Values Without Null Pages**
	Big	**Little**			
1	538976288	538976288	3844196	3.34%	3.34%
2	538970686	1040850976	1052844	0.92%	0.92%
3	538976266	169877536	911501	0.79%	0.79%
4	1008738336	538976316	907955	0.79%	0.79%
5	757932348	1008807213	618754	0.54%	0.54%
6	1818584109	761554284	525117	0.46%	0.46%
7	1802398815	1600941675	525098	0.46%	0.46%
8	1701063981	757949541	524589	0.46%	0.46%
9	1852596076	1818193006	524589	0.46%	0.46%
10	1043148139	1798122814	524589	0.46%	0.46%
11	1818192997	1701601132	524268	0.46%	0.46%
12	757951342	1852517677	524265	0.46%	0.46%
13	1680682273	556608868	524265	0.46%	0.46%
14	1600939364	1684368479	524248	0.46%	0.46%
15	762015340	1819175725	524156	0.46%	0.46%
16	757152800	540811565	502971	0.44%	0.44%
17	540945709	757939744	418502	0.36%	0.36%
18	1701344288	544499813	374088	0.33%	0.33%
19	543516788	1952998688	342254	0.30%	0.30%
20	1852795252	1953066862	330966	0.29%	0.29%
21	1851879539	1936744814	317566	0.28%	0.28%
22	1634493216	543386721	313718	0.27%	0.27%
23	1936941420	1818325875	313430	0.27%	0.27%
24	1935764579	1668047219	313024	0.27%	0.27%
25	1030976353	1634956093	310502	0.27%	0.27%

A.15 Dataset Twitter, Superbowl

Table A.29: Dataset Twitter, Superbowl, summary

Application Details	
Application	**Twitter Superbowl**
Files	28
Size (GB)	1.5
Parsed Size (GB)	1.39
Compressed Size (GB)	
Character Encoding	UTF-8

VFT Size Table	
VFT Size	**% of Total Values in VFT**
128	41.35%
256	44.71%
512	48.44%
1K	53.24%
2K	58.73%
4K	64.88%
8K	71.67%
16K	78.22%
32K	83.71%

Sample Values		
Num Pages	339120	
Null Pages	0	0.00%
Non-Null Pages	339120	100.00%
Null Blocks	0	
Page Size (Byte)	4096	
VFT Entry Size	4	

Table A.30: Dataset Twitter, Superbowl, VFT content

Rank Value Frequency	**Endian**		**Frequency**	**Frequency % of total Values**	**Frequency of % Total Values Without Null Pages**
	Big	**Little**			
1	2003791467	1802399607	9882769	2.85%	2.85%
2	1869507438	1852534383	9865807	2.84%	2.84%
3	1853321070	1852798830	9855493	2.84%	2.84%
4	1802392956	2085973611	9853078	2.84%	2.84%
5	1852534357	1433299822	9847696	2.84%	2.84%
6	2087614319	1870098044	7890299	2.27%	2.27%
7	1434218103	2003729493	7854169	2.26%	2.26%
8	1851096174	1853642094	7843054	2.26%	2.26%
9	808333436	2083532336	3923510	1.13%	1.13%
10	774929456	813445166	3922301	1.13%	1.13%
11	2083532336	808333436	3921284	1.13%	1.13%
12	875638834	842019124	2049398	0.60%	0.60%
13	540291376	808530976	2035339	0.59%	0.59%
14	825242159	791818289	2011909	0.58%	0.58%
15	808399408	808595248	2010058	0.58%	0.58%
16	791818364	2083533359	2009195	0.58%	0.58%
17	813445244	2083552304	1964747	0.57%	0.57%
18	813445166	774929456	1964342	0.57%	0.57%
19	1434202158	774929493	1963191	0.57%	0.57%
20	1851096112	813454702	1960026	0.56%	0.56%
21	175011695	1870097930	1960005	0.56%	0.56%
22	807416881	825499696	1521292	0.44%	0.44%
23	1886680168	1752462448	1059601	0.31%	0.31%
24	808595251	858731056	1023015	0.30%	0.30%
25	858795826	841953331	1022381	0.29%	0.29%

A.16 Dataset Memetracker

Table A.31: Dataset Memetracker, summary

Application Details	
Application	**Memetracker**
Files	1
Size (GB)	10.9
Parsed Size (GB)	10.93
Compressed Size (GB)	
Character Encoding	UTF-8

VFT Size Table	
VFT Size	**% of Total Values in VFT**
128	18.92%
256	24.95%
512	31.47%
1K	38.74%
2K	47.18%
4K	56.65%
8K	66.73%
16K	76.29%
32K	84.27%

Sample Values		
Num Pages	2668988	
Null Pages	0	0.00%
Non-Null Pages	2668988	100.00%
Null Blocks	0	
Page Size (Byte)	4096	
VFT Entry Size	4	

Table A.32: Dataset Memetracker, VFT content

VFT Content Table					
Rank Value Frequency	**Endian**		**Frequency**	**Frequency % of total Values**	**Frequency of % Total Values Without Null Pages**
	Big	**Little**			
1	1886680168	1752462448	24670869	0.90%	0.90%
2	792359028	1953511983	24099682	0.88%	0.88%
3	980448372	1953787962	24087133	0.88%	0.88%
4	791624304	1882861359	24086912	0.88%	0.88%
5	1953785865	157840500	23801180	0.87%	0.87%
6	1952975180	1275684980	19981318	0.73%	0.73%
7	1745439754	172755304	19969067	0.73%	0.73%
8	1836016430	778268525	15698638	0.57%	0.57%
9	795701091	1668246831	14281996	0.52%	0.52%
10	997223777	1634562107	9960493	0.36%	0.36%
11	1886216486	643919216	9717061	0.36%	0.36%
12	1701344288	544499813	8777113	0.32%	0.32%
13	543516788	1952998688	7326726	0.27%	0.27%
14	959459378	842018873	6878787	0.25%	0.25%
15	1735355490	1651273575	6221948	0.23%	0.23%
16	1836345390	778597485	4634756	0.17%	0.17%
17	1819112552	1752460652	4481193	0.16%	0.16%
18	544175136	544501536	4480881	0.16%	0.16%
19	543649385	1768843040	4429198	0.16%	0.16%
20	1852795252	1953066862	4123265	0.15%	0.15%
21	543452769	1634624544	4082539	0.15%	0.15%
22	758722608	808466733	3947579	0.14%	0.14%
23	758394925	758133805	3905531	0.14%	0.14%
24	808270128	809053488	3898331	0.14%	0.14%
25	875572537	959262772	3889653	0.14%	0.14%

A.17 Dataset Wikipedia_L

Table A.33: Dataset Wikipedia_L, summary

Application Details	
Application	**Wikipedia large**
Files	1
Size (GB)	50
Parsed Size (GB)	10.00
Compressed Size (GB)	
Character Encoding	UTF-8

VFT Size Table	
VFT Size	**% of Total Values in VFT**
128	13.45%
256	18.36%
512	24.72%
1K	32.71%
2K	42.07%
4K	52.68%
8K	63.55%
16K	73.64%
32K	82.06%

Sample Values		
Num Pages	2668988	
Null Pages	0	0.00%
Non-Null Pages	2668988	100.00%
Null Blocks	0	
Page Size (Byte)	4096	
VFT Entry Size	4	

Table A.34: Dataset Wikipedia_L, VFT content

VFT Content Table					
Rank Value Frequency	**Endian**		**Frequency**	**Frequency % of total Values**	**Frequency of % Total Values Without Null Pages**
	Big	**Little**			
1	538976288	538976288	38374973	1.43%	1.43%
2	1701344288	544499813	13955413	0.52%	0.52%
3	543516788	1952998688	12645565	0.47%	0.47%
4	538970686	1040850976	9202570	0.34%	0.34%
5	543584032	544171552	8929654	0.33%	0.33%
6	538976266	169877536	8309314	0.31%	0.31%
7	1008738336	538976316	8254451	0.31%	0.31%
8	1953461617	1903521652	6938344	0.26%	0.26%
9	997486453	1970238523	6860148	0.26%	0.26%
10	1869967654	644969839	6858349	0.26%	0.26%
11	543452769	1634624544	6831935	0.26%	0.26%
12	1852795252	1953066862	6471528	0.24%	0.24%
13	997484326	644314171	6375821	0.24%	0.24%
14	997485606	644641851	6361315	0.24%	0.24%
15	1684955424	543256164	6223995	0.23%	0.23%
16	543649385	1768843040	5175970	0.19%	0.19%
17	544106784	543780384	5115067	0.19%	0.19%
18	544175136	544501536	4320257	0.16%	0.16%
19	1869182049	1635019119	3829554	0.14%	0.14%
20	544108393	1768910368	3518670	0.13%	0.13%
21	1952917094	1713792884	2945814	0.11%	0.11%
22	644244850	1919247910	2943891	0.11%	0.11%
23	1730569829	1701193319	2942318	0.11%	0.11%
24	1701667182	1851878757	2760550	0.10%	0.10%
25	543516756	1416127776	2718996	0.10%	0.10%

B

Appendix B

B.1 Data analytics, Wikipedia DA_z0

Table B.1: DA_z0 node statistics

Sample	Page type	Pages	% of total pages	Physical memory size (MB)	Swap size (MB)	% in physical memory	Virt address space (MB)
TaskTracker	DYNAMIC	25881	2.28%	15.18	27.33	35.70%	70086.71
	LIBRARIES	1513	0.13%	3.81	1.61	70.27%	
	STACK	559	0.05%	1.80	0.12	93.60%	
DataNode	DYNAMIC	16377	1.44%	82.54	23.47	77.86%	70074.66
	LIBRARIES	1322	0.12%	4.29	1.83	70.13%	
	STACK	469	0.04%	1.48	0.81	64.76%	
Child 1	DYNAMIC	531120	46.77%	1643.86	531.61	75.56%	4726.58
	LIBRARIES	3147	0.28%	8.64	4.25	67.05%	
	STACK	1405	0.12%	5.33	0.42	92.67%	
Child 2	DYNAMIC	549246	48.36%	2047.70	202.01	91.02%	4726.58
	LIBRARIES	3147	0.28%	8.53	4.36	66.16%	
	STACK	1477	0.13%	5.51	0.54	91.13%	

Table B.2: DA_z1 node statistics

Sample	Page type	Pages	% of total pages	Physical memory size (MB)	Swap size (MB)	% in physical memory	Virt address space (MB)
TaskTracker	DYNAMIC	42359	3.98%	157.27	16.23	90.64%	70077.67
	LIBRARIES	639	0.06%	1.32	1.29	50.55%	
	STACK	590	0.06%	1.76	0.66	72.88%	
DataNode	DYNAMIC	13043	1.23%	40.95	12.47	76.65%	70088.72
	LIBRARIES	1034	0.10%	1.09	3.15	25.63%	
	STACK	336	0.03%	0.75	0.63	54.46%	
Child 1	DYNAMIC	14842	1.39%	41.21	19.58	67.79%	4986.59
	LIBRARIES	3108	0.29%	5.93	6.80	46.59%	
	STACK	323	0.03%	0.37	0.95	27.86%	
Child 2	DYNAMIC	493043	46.32%	1688.03	331.47	83.59%	4726.58
	LIBRARIES	3147	0.30%	6.03	6.86	46.77%	
	STACK	1266	0.12%	2.56	2.62	49.45%	
Child 3	DYNAMIC	486176	45.67%	1655.39	335.99	83.13%	4726.58
	LIBRARIES	3147	0.30%	6.64	6.25	51.48%	
	STACK	1438	0.14%	3.28	2.61	55.63%	

Table B.3: DA_z2 node statistics

Sample	**Page type**	**Pages**	**% of total pages**	**Physical memory size (MB)**	**Swap size (MB)**	**% in physical memory**	**Virt address space (MB)**
TaskTracker	DYNAMIC	31649	3.01%	102.05	27.59	78.72%	70079.68
	LIBRARIES	771	0.07%	0.96	2.20	30.35%	
	STACK	532	0.05%	1.28	0.90	58.83%	
DataNode	DYNAMIC	7113	0.68%	25.55	3.58	87.71%	70088.72
	LIBRARIES	574	0.05%	0.68	1.67	29.09%	
	STACK	339	0.03%	1.03	0.36	74.04%	
Child 1	DYNAMIC	16403	1.56%	36.86	30.33	54.86%	5114.59
	LIBRARIES	1324	0.13%	1.55	3.87	28.63%	
	STACK	795	0.08%	0.85	2.40	26.16%	
Child 2	DYNAMIC	475724	45.28%	1578.89	369.68	81.03%	4726.58
	LIBRARIES	3147	0.30%	5.59	7.30	43.34%	
	STACK	20885	1.99%	63.15	22.40	73.82%	
Child 3	DYNAMIC	470619	44.79%	1672.62	255.03	86.77%	4726.58
	LIBRARIES	3147	0.30%	6.44	6.45	49.95%	
	STACK	17696	1.68%	69.62	2.86	96.06%	

C

Appendix C

C.1 Block alignment analysis

Table C.1: Block alignment analysis results.

	VFT size	Huffman CF	CF w. CF alignment	CF w. Size alignment	CF loss w. CF alignment wrt Huffman CF	CF loss w. Size alignment wrt Huffman CF
DA_z0	128	2.388	1.897	2.287	20.56%	4.23%
DA_z1	128	2.522	2.064	2.405	18.16%	4.64%
DA_z2	128	2.500	2.046	2.380	18.16%	4.80%
DA_z3	128	2.535	1.977	2.418	22.01%	4.62%
GA_z0	128	1.255	1.024	1.237	18.41%	1.43%
GA_z1	128	2.725	2.451	2.527	10.06%	7.27%
GA_z2	128	2.552	2.320	2.386	9.09%	6.50%
GA_z3	128	2.480	2.233	2.353	9.96%	5.12%
DA_z0	1k	2.597	2.113	2.473	18.64%	4.77%
DA_z1	1k	2.680	2.189	2.547	18.32%	4.96%
DA_z2	1k	2.765	2.261	2.608	18.23%	5.68%
DA_z3	1k	2.783	2.234	2.637	19.73%	5.25%
GA_z0	1k	1.299	1.054	1.278	18.86%	1.62%
GA_z1	1k	2.789	2.497	2.582	10.47%	7.42%
GA_z2	1k	2.625	2.372	2.451	9.64%	6.63%
GA_z3	1k	2.547	2.276	2.407	10.64%	5.50%
DA_z0	4k	2.675	2.149	2.538	19.66%	5.12%
DA_z1	4k	2.719	2.215	2.579	18.54%	5.15%
DA_z2	4k	2.819	2.302	2.652	18.34%	5.92%
DA_z3	4k	2.854	2.321	2.695	18.68%	5.57%
GA_z0	4k	1.315	1.055	1.291	19.77%	1.83%
GA_z1	4k	2.805	2.512	2.595	10.45%	7.49%
GA_z2	4k	2.641	2.380	2.464	9.88%	6.70%
GA_z3	4k	2.567	2.288	2.425	10.87%	5.53%

Table C.2: Block alignment analysis results with Length Overhead.

	VFT size	Huffman CF	CF w. LO	CF w. LO and CF alignment	CF w. LO and Size alignment	CF loss w. LO and CF alignment wrt Huffman CF	CF loss w. LO and Size alignment wrt Huffman CF
DA_z0	128	2.388	1.878	1.442	1.817	39.61%	23.91%
DA_z1	128	2.522	1.960	1.538	1.890	39.02%	25.06%
DA_z2	128	2.500	1.938	1.487	1.867	40.52%	25.32%
DA_z3	128	2.535	1.952	1.500	1.883	40.83%	25.72%
GA_z0	128	1.255	1.136	1.009	1.122	19.60%	10.60%
GA_z1	128	2.725	2.155	1.911	2.031	29.87%	25.47%
GA_z2	128	2.552	2.062	1.856	1.955	27.27%	23.39%
GA_z3	128	2.480	2.016	1.839	1.934	25.85%	22.02%
DA_z0	1k	2.597	1.995	1.544	1.923	40.55%	25.95%
DA_z1	1k	2.680	2.044	1.620	1.967	39.55%	26.60%
DA_z2	1k	2.765	2.082	1.632	1.993	40.98%	27.92%
DA_z3	1k	2.783	2.092	1.601	2.009	42.47%	27.81%
GA_z0	1k	1.299	1.168	1.014	1.152	21.94%	11.32%
GA_z1	1k	2.789	2.189	1.920	2.061	31.16%	26.10%
GA_z2	1k	2.625	2.102	1.872	1.991	28.69%	24.15%
GA_z3	1k	2.547	2.052	1.841	1.963	27.72%	22.93%
DA_z0	4k	2.675	2.025	1.557	1.947	41.79%	27.21%
DA_z1	4k	2.719	2.062	1.625	1.981	40.24%	27.14%
DA_z2	4k	2.819	2.107	1.645	2.014	41.65%	28.56%
DA_z3	4k	2.854	2.126	1.639	2.039	42.57%	28.56%
GA_z0	4k	1.315	1.179	1.011	1.159	23.12%	11.86%
GA_z1	4k	2.805	2.198	1.926	2.068	31.34%	26.27%
GA_z2	4k	2.641	2.109	1.879	1.997	28.85%	24.38%
GA_z3	4k	2.567	2.062	1.850	1.972	27.93%	23.18%

Table C.3: Slack for block CF alignment.

Block CF	**DA_z0**	**DA_z1**	**DA_z2**	**DA_z3**	**GA_z0**	**GA_z1**	**GA_z2**	**GA_z3**	**Max slack**	**Arithmetic mean**		
										DA	**GA**	**GA (GA_z0 excluded)**
1	15.740	13.460	14.400	17.510	11.860	3.000	2.240	3.300	31	15.28	5.10	2.85
2	5.400	5.850	5.640	5.720	3.330	4.240	5.040	4.680	10	5.65	4.32	4.65
3	2.070	1.990	2.080	2.080	1.120	1.950	1.910	1.960	4	2.06	1.74	1.94
4	1.150	1.360	1.460	1.200	1.970	2.250	2.200	2.280	3	1.29	2.18	2.24
5	0.670	0.700	0.560	0.680	0.170	0.010	0.000	0.030	1	0.65	0.05	0.01
6	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0	0.00	0.00	0.00
7	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0	0.00	0.00	0.00
8	2.180	1.840	1.740	2.220	0.610	3.000	0.170	3.230	7	2.00	1.75	2.13

Table C.4: Slack for block size alignment.

Block size	**DA_z0**	**DA_z1**	**DA_z2**	**DA_z3**	**GA_z0**	**GA_z1**	**GA_z2**	**GA_z3**	**Max slack**	**Arithmetic mean**		
										DA	**GA**	**GA (GA_z0 excluded)**
4	0.180	0.280	0.380	0.170	0.000	0.170	0.150	0.000	3	0.25	0.08	0.11
8	1.740	1.480	1.360	1.780	0.610	3.000	0.090	2.960	3	1.59	1.67	2.02
12	1.430	1.540	1.450	1.430	1.860	0.690	1.850	0.120	3	1.46	1.13	0.89
16	1.150	1.360	1.460	1.200	1.970	2.250	2.200	2.280	3	1.29	2.18	2.24
20	1.330	1.410	1.430	1.410	0.130	1.540	1.650	1.700	3	1.40	1.26	1.63
24	1.160	1.230	1.220	1.170	1.900	1.830	1.960	1.910	3	1.20	1.90	1.90
28	1.450	1.500	1.570	1.410	0.410	1.380	1.430	1.430	3	1.48	1.16	1.41
32	1.440	1.670	1.410	1.380	2.270	1.260	1.560	1.440	3	1.48	1.63	1.42
36	1.250	1.270	1.280	1.220	2.760	1.870	2.040	1.670	3	1.26	2.09	1.86
40	0.930	0.980	1.050	0.820	1.640	1.770	1.730	1.460	3	0.95	1.65	1.65
44	0.780	0.660	0.740	0.840	0.990	1.730	1.890	1.100	3	0.76	1.43	1.57
48	0.990	1.210	1.240	1.060	1.080	1.470	1.620	1.340	3	1.13	1.38	1.48
52	0.820	0.690	1.070	1.040	0.020	1.260	1.090	1.090	3	0.91	0.87	1.15
56	1.460	1.390	1.580	1.360	0.950	1.660	1.350	1.560	3	1.45	1.38	1.52
60	1.180	0.790	1.150	1.340	1.000	1.870	1.080	1.440	3	1.12	1.35	1.46
64	0.090	0.100	0.230	0.100	0.010	0.120	0.100	0.090	3	0.13	0.08	0.10

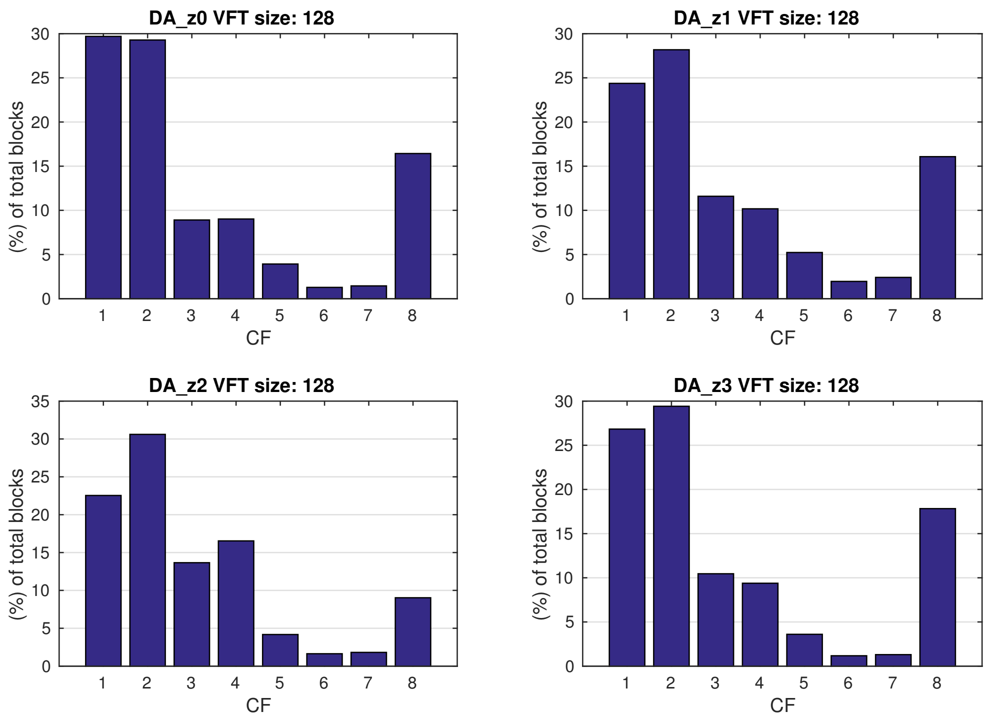

Figure C.1: Block CF distribution for DA_z0 - DA_z3.

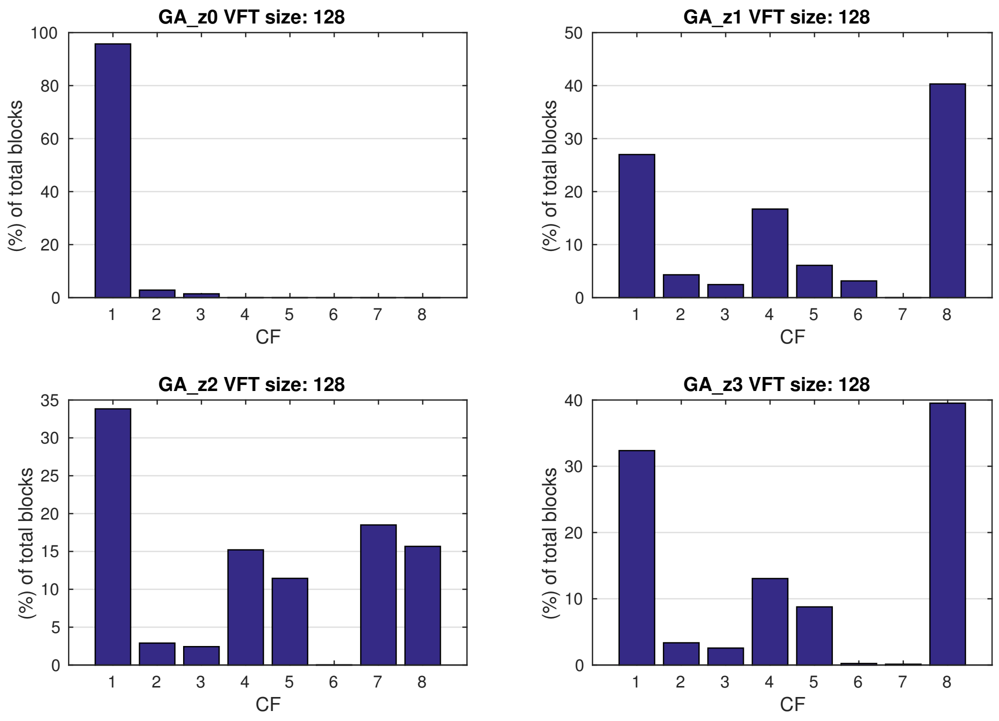

Figure C.2: Block CF distribution for GA_z0 - GA_z3.

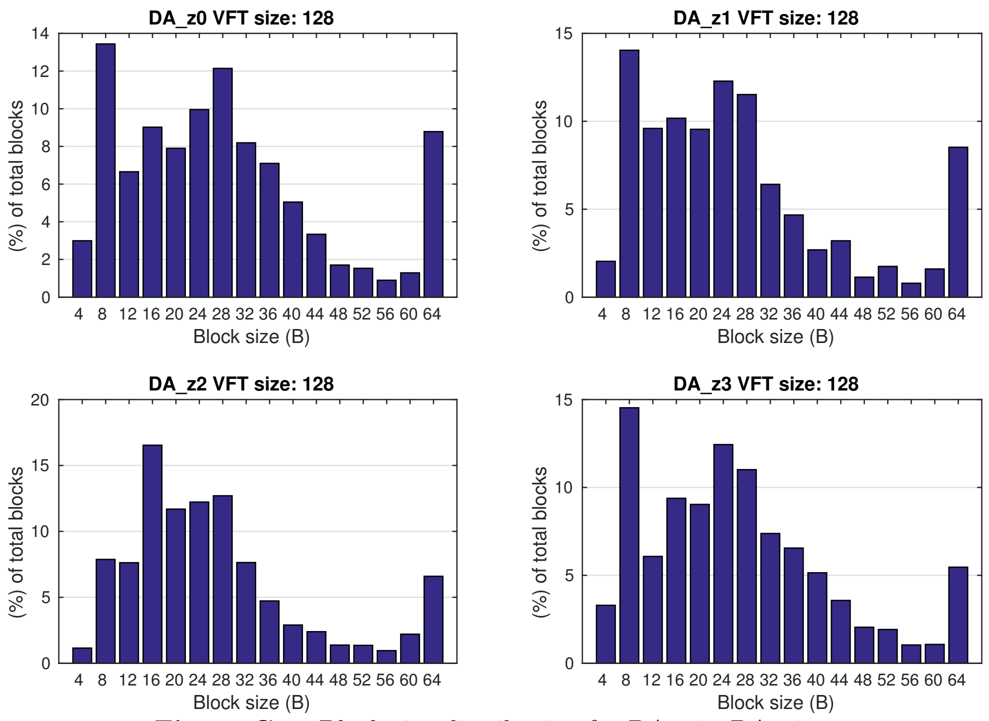

Figure C.3: Block size distribution for DA_z0 - DA_z3.

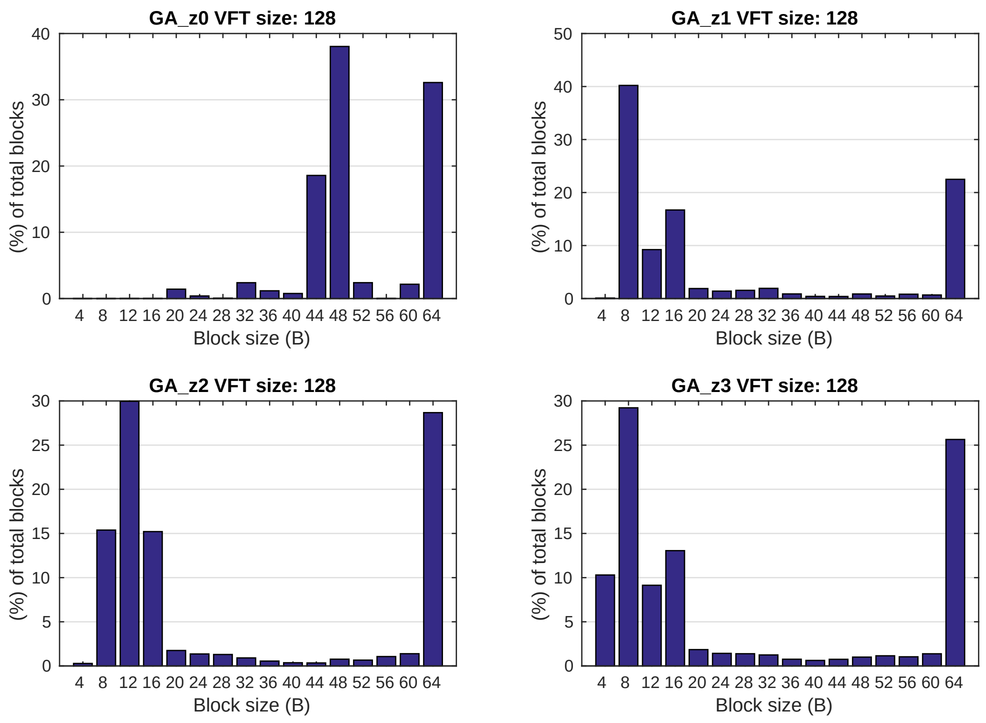

Figure C.4: Block size distribution for GA_z0 - GA_z3.

C.2 Page alignment analysis

Table C.5: Page alignments analysis results.

			Extra page addressing bits						
Sample	VFT size	Huffman CF	0	1	2	3	4	5	6
DA_z0	128	2.388	1.000	1.586	1.827	2.121	2.234	2.306	2.347
DA_z1	128	2.522	1.000	1.715	1.895	2.205	2.341	2.430	2.475
DA_z2	128	2.500	1.000	1.666	1.834	2.169	2.325	2.411	2.453
DA_z3	128	2.535	1.000	1.629	1.893	2.220	2.352	2.442	2.488
GA_z0	128	1.255	1.000	1.025	1.194	1.212	1.218	1.246	1.248
GA_z1	128	2.725	1.000	1.543	2.085	2.412	2.487	2.622	2.693
GA_z2	128	2.552	1.000	1.516	1.976	2.252	2.355	2.459	2.515
GA_z3	128	2.480	1.000	1.487	1.928	2.190	2.289	2.388	2.440
DA_z0	1k	2.597	1.000	1.740	1.943	2.266	2.417	2.499	2.547
DA_z1	1k	2.680	1.000	1.766	1.956	2.317	2.478	2.578	2.627
DA_z2	1k	2.765	1.000	1.775	1.983	2.362	2.545	2.651	2.707
DA_z3	1k	2.783	1.000	1.794	2.025	2.385	2.575	2.670	2.726
GA_z0	1k	1.299	1.000	1.048	1.219	1.248	1.258	1.288	1.290
GA_z1	1k	2.789	1.000	1.549	2.115	2.453	2.535	2.677	2.753
GA_z2	1k	2.789	1.000	1.549	2.115	2.453	2.535	2.677	2.753
GA_z3	1k	2.547	1.000	1.500	1.966	2.242	2.345	2.450	2.506
DA_z0	4k	2.675	1.000	1.759	1.962	2.326	2.479	2.571	2.622
DA_z1	4k	2.719	1.000	1.776	1.968	2.340	2.513	2.613	2.664
DA_z2	4k	2.819	1.000	1.793	2.003	2.401	2.587	2.702	2.759
DA_z3	4k	2.854	1.000	1.809	2.044	2.431	2.613	2.734	2.793
GA_z0	4k	1.315	1.000	1.040	1.217	1.255	1.268	1.298	1.305
GA_z1	4k	2.805	1.000	1.551	2.134	2.469	2.547	2.693	2.769
GA_z2	4k	2.641	1.000	1.529	2.026	2.320	2.428	2.540	2.600
GA_z3	4k	2.567	1.000	1.501	1.974	2.257	2.360	2.467	2.524

Table C.6: Page alignments CF loss with respect to Huffman CF.

		Extra page addressing bits					
Sample	VFT size	1	2	3	4	5	6
DA_z0	128	33.58%	23.49%	11.18%	6.45%	3.43%	1.72%
DA_z1	128	32.00%	24.86%	12.57%	7.18%	3.65%	1.86%
DA_z2	128	33.36%	26.64%	13.24%	7.00%	3.56%	1.88%
DA_z3	128	35.74%	25.33%	12.43%	7.22%	3.67%	1.85%
GA_z0	128	18.33%	4.86%	3.43%	2.95%	0.72%	0.56%
GA_z1	128	43.38%	23.49%	11.49%	8.73%	3.78%	1.17%
GA_z2	128	40.60%	22.57%	11.76%	7.72%	3.64%	1.45%
GA_z3	128	40.04%	22.26%	11.69%	7.70%	3.71%	1.61%
DA_z0	1k	33.00%	25.18%	12.75%	6.93%	3.77%	1.93%
DA_z1	1k	34.10%	27.01%	13.54%	7.54%	3.81%	1.98%
DA_z2	1k	35.80%	28.28%	14.58%	7.96%	4.12%	2.10%
DA_z3	1k	35.54%	27.24%	14.30%	7.47%	4.06%	2.05%
GA_z0	1k	19.32%	6.16%	3.93%	3.16%	0.85%	0.69%
GA_z1	1k	44.46%	24.17%	12.05%	9.11%	4.02%	1.29%
GA_z2	1k	44.46%	24.17%	12.05%	9.11%	4.02%	1.29%
GA_z3	1k	41.11%	22.81%	11.97%	7.93%	3.81%	1.61%
DA_z0	4k	34.24%	26.65%	13.05%	7.33%	3.89%	1.98%
DA_z1	4k	34.68%	27.62%	13.94%	7.58%	3.90%	2.02%
DA_z2	4k	36.40%	28.95%	14.83%	8.23%	4.15%	2.13%
DA_z3	4k	36.62%	28.38%	14.82%	8.44%	4.20%	2.14%
GA_z0	4k	20.91%	7.45%	4.56%	3.57%	1.29%	0.76%
GA_z1	4k	44.71%	23.92%	11.98%	9.20%	3.99%	1.28%
GA_z2	4k	42.11%	23.29%	12.15%	8.07%	3.82%	1.55%
GA_z3	4k	41.53%	23.10%	12.08%	8.06%	3.90%	1.68%