Problem Description

Mobile devices such as iPads or iPhones have a relatively little amount of RAM, but a comparably powerful CPU and increasing amounts of low-latency flash memory storage (128 GiB on the last generation iPad). Increased computational power and storage, combined with progressively more energy efficient devices, allows one to pursuit new areas of research including exploiting available hardware in unique ways.

The Trondheim-based company "Atbrox" are involved with an EU project focusing on search on mobile devices. An important component of search is the representation of the inverted index, and within, the per term postings list – a compressed list of URIs, where each URI represents a document containing the search term. Compression and decompression of the posting list can be handled in a number of different ways.

This thesis will focus on implementation, tuning (e.g. chunk size) and benchmarking of Variable byte encoding, a simple but efficient way to store the posting list. A survey of additional algorithms such as Elias gamma coding will also be included.

The work carried out in this project should provide insight to the performance of reading and writing from an SSD on a mobile device, streaming data and random access, and the balance between SSD and CPU utilization.

This work will be implemented in Objective-C, utilizing acceleration frameworks provided on the iOS platform.

Abstract

Recent years has seen a tremendous increase in both the performance of handheld devices and the use cases they are required to fullfil. Indeed, operations previously reserved for handling on personal computers have begun being executed on smart phones and tablets instead. This revolutionary development allows one to exploit handheld device hardware in novel applications.

Trondheim-based start-up "Atbrox" is engaged in an EU project where Atbrox' focus is search on mobile devices. An important component of search is the inverted index, and within, the per term postings list – an encoded list of Unified Resource Identifiers (URI). Decoding of a postings list must be fast in order to not comprimise the user experience, but is also required to hold a small storage footprint. As the first to our knowledge, this thesis attempts to identify the properties of postings list encoding and decoding on handheld devices.

Variable-byte coding, *Group Varint coding*, and *Elias γ coding* are implemented in Objective-C. Performance is surveyed by benchmarking three devices out of Apple: A 5th generation iPod, a 4th generation iPad, and an iPad Air. Executions are run from disk-to-disk, *i.e.* by reading a block of data, applying either encoding or decoding, and writing the result to permanent storage. Block sizes are varied. In addition, multithreading is applied during both encoding and decoding and compared to serial executions in an attempt to identify the properties under which each coding scheme performs best.

This thesis provides valueable insight to the properties of coding schemes on handheld devices. Among its findings is the varying degree of performance and compression ratio between coding schemes: Group Varint proves to outperform the two others in terms of speed, however, is lacking in terms of compression. Elias γ code provides the best compression ratio, but is the slowest in both encoding and decoding. Results also prove a strong correspondance between block size and performance, although a point of saturation is reached at 512 KiB. Additionally, block sizes below 512 KiB display an inability to take advantage of multithreading.

Sammendrag

De senere årende har sett en rivende utvikling innen ytelsen i håndholdte enheter og deres bruksområder. Gjøremål tidligere reservert for PC har blitt flyttet over til smarttelefoner og nettbrett. Denne revolusjonerende utviklingen tillater utnyttelse av tilgjengelig maskinvare på nye, spennende måter.

Det Trondheims-baserte selskapet "Atbrox" er engasjert i et EU-prosjekt, der Atbrox fokuserer på søk på mobile enheter. En viktig komponent i søk er den inverterte indeksen, med tilhørende postings-liste – en komprimert liste av Unified Resource Identifier-er (URI). Dekomprimering av postings-listen må være rask for ikke å forkludre brukerfølelsen, men samtidig tilordne seg lite lagringsplass. Til vår kjennskap er denne masteroppgaven det første forsøket på å kartlegge en postingslistes egenskaper på mobile enheter.

Variable-byte-, *Group Varint-* og *Elias* γ-komprimering er implementert i Objective-C. Ytelsen er kartlagt ved å benytte følgende enheter fra Apple: En femtegenerasjons iPod Touch, en fjerdegenerasjons iPad og en iPad Air. Kjøringer er gjort ved å lese en datablokk, anvende komprimering eller dekomprimering på datablokken og skrive resultatet til disk. Blokkstørrelser varieres mellom kjøringer. Flere tråder er benyttet under både komprimering og dekomprimering og sammenlignet med serielle kjøringer.

Masteroppgaven gir verdifull kunnskap om egenskapene ved de nevnte komprimeringsmetodene på håndholdte enheter. Blant funnene er variasjonen i ytelse og komprimeringsgrad metodene i mellom. Group Varint gir best ytelse sett i forhold til hastighet, men betaler med dårligst komprimeringsgrad, mens Elias γ har egenskaper invertert av disse. Resultatene viser også en sterk sammenheng mellom hastighet og blokkstørrelse. Et metningspunkt oppstår dog ved 512 KiB. I tillegg virker applikering av flere tråder mot sin hensikt for blokkstørrelser under 512 KiB.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Recent years has seen a tremendous increase in both the performance of handheld devices and the use cases they are required to fullfil. Indeed, operations previously reserved for handling on personal computers have begun being executed on smart phones and tablets instead. This revolutionary development allows one to exploit handheld device hardware in novel applications. In addition, while handheld devices have become increasingly faster in terms of computation, they have also received storage space on par with Solid State Drive (SSD) based notebooks. The recent iPad Mini and iPad Air out of Apple each hold storage capacities of 128 GiB. With flash memory as the storage technology, such devices promise high Input/Output (I/O) performance.

Trondheim-based technology start-up "Atbrox" specialize in search technologies and novel applications of search. They are currently engaged in an on-going EU project where Atbrox' focus is on search on mobile devices. An important component of search is the inverted index, and within, the per term postings list – an encoded list of Unified Resource Identifiers (URI), each identifying a document in the document storage system. Decoding of a postings list must be fast in order to not comprimise the user experience, but is also required to hold a small storage footprint as the device storage space is shared with numerous other applications.

This thesis is motivated by recent hardware developments in the handheld device market, with a major potential for utilizing the available performance in unique ways. Three different coding schemes commonly used in encoding and decoding postings lists will presented and implemented: *Variable-byte coding*, *Group Varint coding*, and *Elias γ coding*. Each coding will be applied using a 5th generation iPod, a 4th generation iPad, and an iPad Air, with the performance of each being

recorded. In addition, tests with a varying block size will be conducted, and multi-threading will be attempted in order to investigate the potential for parallel encoding and decoding.

1.1 Project Context

This master's thesis is the end product of a five year Master of Science (M.Sc.) education in Computer Science conducted at the Norwegian University of Science and Technology (NTNU). It is one of several projects organized by the High Performance Computing (HPC) group at the Department of Computer and Information Science (IDI) [16]. Further, the project is under advisement from Dr. Anne C. Elster from NTNU[1], and Dr. Amund Tveit on behalf of atbrox[2].

1.2 Project Contributions

While novel ways of coding or structuring postings lists are not part in the this projects contributions, it provides valuable insight to the behaviour of coding schemes on mobile platforms, the performance, and optimal ways of use. In addition, problems related to blocked reading of data to encode or encoded data are assessed and handled. These are situations that have not been found described in other litterature. Implicitly, the relative performance between devices is also measured, both in terms of CPU performance and disk performance.

1.3 Thesis Outline

The remainder of this thesis is structured as follows:

Chapter 2: Contains an introduction to the current handheld device market, as well as a more thorough presentation of Apple *i*-series of devices and the hardware found within. In addition, a quick walkthrough of developing on the iOS platform will be given.

Chapter 3: Gives a review of parallel concepts such as multi-core, parallel data models, and theoretical models of parallelization.

[1] http://www.idi.ntnu.no/~elster/
[2] http://www.atbrox.com

Chapter 4: Presents the concept of a postings list in general, how they are commonly organized, as well as encoding and decoding methods relevant to postings lists. The chapter will end with a review of work related to this thesis.

Chapter 5: Discloses the implementation details of the work performed, with information on the work flow of created applications and the processing of data from disk into encoded or decoded form. Additionally, the internals of implemented coding schemes are presented. Towards the end, a description on how benchmarks were executed is given.

Chapter 6: Begins with an overview of the performance and compression ratio of implemented coding schemes. At the same time, a comparison with generalized compression methods is made. Following, results of the flash memory reading benchmarks are presented. The remainder of the chapter is dedicated to presenting and discussing results found testing each coding scheme under various parameters on the benchmarked devices. In the end, results are summarized and given a critical review.

Chapter 7: Contains a recap of the findings in the thesis, an assessment of these findings, and conclusions that have been made. Suggestions for future work is presented at the end.

Chapter 2

Handheld Devices and Development

This chapter will introduce the reader to the handheld device market, and development on such devices. First off, common hardware configurations and their properties are presented. Further, Apple's *i*-series, *i.e.* the iPod Touch, iPhone, and iPads, are given a more thorough introduction. Due to Atbrox' involvement in iOS, important contributors and large hardware manufacturers such as Samsung, HTC, and LG, have not been covered in this thesis.

Towards the end, a walkthrough of development on iOS will be given, introducing the Objective-C programming language and important programming frameworks.

2.1 Central Processing Unit

While limitations in power consumption, generation of heat and physical die space are all factors hampering the performance of a Central Processing Unit (CPU) running on a desktop computer, compromises made on mobile platforms are more rigorous. Consumers demand high performance coupled with maximized battery life. In addition, the tight body of mobile devices force manufactures to shrink the CPU's die and forego active cooling solutions such as fans. Indeed, even passive cooling solutions must be trimmed to fit within the shell of a handheld device. An increased awareness of battery life, the absense of active cooling and less real estate for the CPU have forced a new market of mobile CPU manufacturing with roots in the embedded market.

2.1.1 ARM Architecture

ARM's embedded past and early initiative with low-powered Graphics Processing Units (GPUs) have made them a dominant entity in the handheld chip market. Their list of Intellectual Property (IP) licensee's include manufacturers such as Apple, Qualcomm, Samsung and others [10].

The architecture has been designed to be small and simple, allowing for a low power consumption. In essence, it follows the design of a Reduced Instruction Set Computer (RISC), incorporating several typical RISC features such as a *load/store*-centric architecture and a large uniform register file. Coupled with additional enhancements to the traditional RISC architecture, particularly towards the use of the Arithmetic Logic Unit (ALU), ARM processors achieve a good balance between performance, power consumption and die size [50]. ARM's first 64 bit processor was introduced in 2011.

ARM has been present in the handheld market since the early 1990s, first introduced with the Apple Newton [23]. Since then, the company has become a dominant actor within the handheld chip market. In 2006, research estimated an ARM designed core was present in 98 % of all mobile phones [40]. Table 2.1 list common ARM processors and their features found in current mobile phones and tablets [7–9].

Table 2.1 Core specifications of common ARM processors.

	Cortex-A8	Cortex-A9	Cortex-A15
Clock Frequency:	600 MHz – 1 GHz	800 MHz – 2 GHz	1.0 GHz - 2.5 GHz
# Cores:	1	1 – 4	1 – 4
L1 Cache (I/D):	32 KiB /32 KiB	32 KiB /32 KiB *per core*	32 KiB /32 KiB *per core*
L2 Cache:	-	128 KiB – 8 MiB	128 KiB – 8 MiB
64 bit:	-	-	-
SIMD Extensions:	NEON	NEON	NEON

2.1.2 IA–32/Intel x86–64 Architecture

Intel's IA–32 and x86–64 architectures are architectures based on and backwards compatible with Intel's x86 architecture. In comparison to the previously mentioned ARM architecture, x86 is a Complex Instruction Set Computer (CISC) architecture. A characteristic of CISC is the inherit complexity of instructions, enabling them to perform several operations per instruction, for instance loading a value from memory and dispatching it to the ALU. Later generations of x86 introduced

a decoding step and a RISC-like core in the processor. X86/CISC instructions are decoded into *micro-operations* and executed in a RISC-like manner [33, 57].

In handheld devices, Intel's *Atom* series of processors has been the most prevalent x86-based CPU. Intel Atom is a family of Ultra-Low-Voltage (ULV) processors, created to establish Intel in the embedded and handheld device market [32, 34]. As with ARM, Intel Atom sports a notably lower clock frequency compared to processors designed for desktop and server use. In addition, the physical size of the chip is significantly less than its desktop counterparts.

2.2 Storage

In order to keep device size at a minimum, but still provide sufficient storage, handheld devices resort to the use of flash memory instead of traditional, mechanical hard drives.

2.2.1 Flash Memory

Flash memory is a type of non-volatile electronic storage medium, commonly found within hardware devices such as mobile phones, tablets and Solid State Drives (SSD). The technology promises significant performance gains, while being more dense and power efficient. However, flash memory does not come without idiosyncrasies. It is known to be less durable, as well as having data integrity issues [20]. Indeed, the current most common type of flash memory, Negated AND (NAND) memory, has been predicted a bleak future since both the increase in performance and reliability have stagnated as the density has risen [21]. Despite mentioned idiosyncrasies, flash memory's low power consumption and performance compared to that of mechanical hard drives makes it a favourable candidate when choosing the storage medium for a device. Being both electric and non-volatile, flash memory combines the properties of technology found in use as main memory, *i.e.* high random access performance, without data loss when the memory is unpowered. In addition, some issues, such as wear-leveling and integrity, may be countered through the use of an intelligent flash controller or a flash translation layer.

2.3 Memory

With the advent of more and more mobile devices, with a higher demand for prolonged battery life as well as less device real estate, a new type of Random Access Memory (RAM) was introduced: Low Power Double Data Rate RAM (LPDDR) or

Mobile DRAM (mDDR). LPDDR was first standardized by JEDEC[1] in 2007, then as a minor modification to the existing DDR standard, specifying lower operating voltage, a new deep power down mode as well as a smaller physical size [38]. Devices such as the first generation of iPad and Samsung Galaxy Tab adopted the new type of memory [19].

As new devices have been released, so has the LPDDR standard. JEDEC announced LPDDR2 in 2009, further lowering the operating voltage in addition to representing a more dramatic change from conventional DDR [36, 53]. LPDDR2 was quickly adopted by the industry, and is represented in devices such as the iPhone 5 and Samsung Galaxy S3 [30, 51]. LPDDR3 was announced in May 2012, promising a higher data rate, improved bandwidth and power efficiency, as well as higher memory densities than its predecessor [37, 54].

2.4 Apple Devices

Apple released their first modern, handheld *i*-device in 2007 with the introduction of the iPhone[2] [25]. Since inception, the devices have had a strong emphasis on preserving battery life. Indeed, an emphasis to the extent that processors within *i*-devices have been underclocked[3]. Each generation of devices from Apple have possessed the previously mentioned hardware characteristics: An ARM based CPU, flash drive for storage and a variant of Low Power DDR RAM. Table 2.2, Table 2.3, and Table 2.4 lists the different generations, as well as key hardware components.

Table 2.2 Specifications of iPod Touch devices.

	System on Chip	Memory	Storage
1st gen. iPod Touch:	Samsung S5L8900 (412 MHz)	128 MiB	8 GiB – 32 GiB
2nd gen. iPod Touch:	Samsung S5L8720 (533 MHz)	128 MiB	8 GiB – 32 GiB
3rd gen. iPod Touch:	Samsung S5L8920 (600 MHz)	128 MiB	32 GiB – 64 GiB
4th gen. iPod Touch:	Apple A4 (800 MHz)	256 MiB	8 GiB – 64 GiB
5th gen. iPod Touch:	Apple A5 (1 GHz)	512 MiB	16 GiB – 64 GiB

[1] JEDEC is a standardization body and independant semiconductor trade organization. The DDR SDRAM standards are a product of JEDEC. More information is available at http://www.jedec.org.

[2] Only generations of iPod running iOS are considered in this project, that is, the iPod Touch-series of devices.

[3] Running a CPU on a clock frequency lower than specified. As opposed to overclocking, where the clock frequency is increased beyond specification., iphone2008underclocked, iphone2009underclocked

Table 2.3 Specifications of iPhone devices.

	System on Chip	Memory	Storage
iPhone:	Samsung S5L8900 (412 MHz)	128 MiB LPDDR	4 GiB – 16 GiB
iPhone 3G:	Samsung S5L8900 (412 MHz)	128 MiB LPDDR	8 GiB – 16 GiB
iPhone 3GS:	Samsung S5PC100 (600 MHz)	256 MiB LPDDR	8 GiB – 32 GiB
iPhone 4:	Apple A4 (800 MHz)	512 MiB LPDDR2	8 GiB – 32 GiB
iPhone 4s:	Apple A5 (800 MHz)	512 MiB LPDDR2	8 GiB – 64 GiB
iPhone 5:	Apple A6 (1.3 GHz)	1 GiB LPDDR2	16 GiB – 64 GiB
iPhone 5C:	Apple A6 (1.3 GHz)	1 GiB LPDDR2	16 GiB – 32 GiB
iPhone 5S:	Apple A7 (1.3 GHz)	1 GiB LPDDR3	16 GiB – 64 GiB

Table 2.4 Specifications of iPad devices.

	System on Chip	Memory	Storage
1st gen. iPad:	Apple A4 (1 GHz)	256 MiB LPDDR	16 GiB – 64 GiB
2nd gen. iPad:	Apple A5 (1 GHz)	512 MiB LPDDR2	16 GiB – 64 GiB
3rd gen. iPad:	Apple A5X (1 GHz)	1 GiB LPDDR2	16 GiB – 64 GiB
4th gen. iPad:	Apple A6X (1.4 GHz)	1 GiB LPDDR2	16 GiB – 128 GiB
1st gen. iPad Mini:	Apple A5 (1 GHz)	512 MiB LPDDR2	16 GiB – 64 GiB
2nd gen. iPad Mini:	Apple A7 (1.3 GHz)	1 GiB LPDDR3	16 GiB – 128 GiB
iPad Air:	Apple A7 (1.4 GHz)	1 GiB LPDDR3	16 GiB – 128 GiB

An interesting observation to make is the close relationship between hardware configurations in generations of iPod and iPhone.

2.4.1 Apple A-Series System on Chip

The A-series family of System on Chips (SoC) integrate one or several ARM-based CPU cores, an arbitrary GPU, cache memory and additional electronic equipment required for mobile computing functions. Its first official debute was in the release of Apple's *iPad*, then represented by the Apple A4. Apple themselves design the package, while manufacturing is out-sourced to external contractors such as Samsung [56]. A-series SoCs are found in nearly all electronic equipment produced by Apple; iPad, iPod Touch, iPhone as well as the Apple TV.

As mentioned, Apple A4 first introduced the series. Then followed the Apple A5 and A5X the consecutive year and lastly Apple A6 and Apple A6X. In 2013, Apple released its first 64 bit mobile SoC aptly named A7. Properties of each generation is summarized in Table 2.5 [41, 43–45, 56].

Table 2.5 Properties of the different generations of A-series System on Chips.

	Apple A4	Apple A5	Apple A5X	Apple A6	Apple A6X	Apple A7
Processor:	ARM Cortex-A8	ARM Cortex-A9	ARM Cortex-A9	Apple Swift	Apple Swift	Apple Cyclone
Clock Frequency:	800 MHz – 1 GHz	800 MHz – 1 GHz	1 GHz	1.3 GHz	1.4 GHz	1.3 GHz – 1.4 GHz
# Cores:	1	1 – 2	2	2	2	2
L1 Cache (I/D):	32 KiB /32 KiB	32 KiB /32 KiB	32 KiB /32 KiB	32 KiB /32 KiB	32 KiB /32 KiB	64 KiB /64 KiB
L2 Cache:	512 KiB	1 MiB	1 MiB	1 MiB	1 MiB	1 MiB
64 bit:	-	-	-	-	-	Yes
GPU:	PowerVR SGX 535	PowerVR SGX543	PowerVR SGX543	PowerVR SGX543	PowerVR SGX554	PowerVR G6430

Apple recently made a move from making use of ARM designed cores, to cores designed by Apple themselves, *i.e.* the Apple Swift found in Apple's A6 and A6X and Apple Cyclone found in A7.

2.4.2 Storage

The amount of available storage on devices made by Apple varies from generation to generation. From the first iPhone having options between 4 and 16 Gigabytes (GB), to the latest iPhone 5S having been made available with 16, 32 or 64 GB of storage. Common to all is the use of NAND flash Memory as the electronic storage medium. It is unknown whether Apple employ the use of an additional Input/Output (I/O) controller to facilitate access to the flash memory or if the CPU has direct access. Several investigations into the internals of *i*-devices make no mention of a controller [27–29, 31]. Indeed, reverse engineering attempts show atleast earlier versions of iPhone make use of dynamic wear leveling implemented in software as a proprietary flash translation layer (FTL) [35]. However, in 2012 Apple acquired Anobit Technologies, Ltd, an Israeli flash memory controller manufacturer [52]. Articles commenting on the acquisition mention Anobit technology as already present in iPhones and iPads [17].

2.4.3 Memory

Main memory in Apple's devices is not bundled alongside the CPU and GPU on the same SoC, but rather attached as a Package on Package (PoP). That is, RAM and SoC are stacked and unified through a standard routing interface. Apple makes use of one of the standards of Low Power DDR RAM (LPDDR), a physically smaller type of DDR RAM operating on lower than normal voltages. The original LPDDR was used up until the release of the second generation iPad (iPad 2) and iPhone 4, providing bandwidths up to 1600 Megabytes per second (MB/s) depending on memory clock rate and width of the memory bus. iPad 2, iPhone 4 and later generations of the two use LPDDR2, achieving theoretical bandwidths up to 12800 MB/s. In iPhone 5S, the Apple A7 is packaged with LPDDR3, further pushing the theoretical bandwidth limit.

2.5 Mac OS X and iOS Development

The following sections will introduce the programming languages and tools available for development on Apple's platforms.

2.5.1 Programming Languages

Three different, although related, programming languages are available when developing for Mac OS X and iOS: *C*, *C++*, and *Objective-C*, with the latter being the primary language of use.

Objective-C

Objective-C is the primary programming language of use when developing software for Mac OS X and iOS. It is a strict superset of the C programming language, defining several powerful extensions. Among these are Object-oriented capabilities by adopting Smalltalk-like messaging and a dynamic runtime [2, About Objective-C].

Syntax Being a thin layer on top of C, any Objective-C compiler is able to compile a C program. In addition, a developer is permitted to freely include C code in an Objective-C class. It follows Smalltalk's syntax for sending messages, *i.e.* calling an object's method or function. Syntax for associating variables with values, arithmetics, conditional constructs, and other non-object oriented behaviour follows the same convention as C.

Classes and Objects As with other object-oriented languages, objects in Objective-C are made to encapsulate and package related data. A class is a description of an object. It acts as a blueprint, defining properties and behaviour of objects that belong to this specific class. For instance, an array object may contain functionality for storing, expanding, and contracting data. However, one does not need to know the interals of such an object, as described in the class, only how one is expected to interact with the object and how the object will respond [2, Defining Classes].

Class inheritance is an important feature of an object-oriented design, and Objective-C is no exception. Although not required, almost all classes used in relation to Mac OS X or iOS inherit from `NSObject`. `NSObject` corresponds to Java's `Object` class or Python's `object` class. That is, a root class[4] with a basic interface to the runtime system, enabling child classes to behave as objects of their respective programming languages.

Interface An important focus in Objective-C is to define the behaviour of objects of a class, and hide implementation details. To facilitate such an architecture, classes are required to define public methods and properties in what is called an *interface*. Listing 2.1 illustrate a bare class interface definition named `MyPersonClass`, inheriting methods and properties from `NSObject`.

Listing 2.1 An example of an Objective-C class interface definition.

```
1 #import <Foundation.h>
2
3 @interface MyPersonClass : NSObject
4
5 @end
```

In Objective-C, the properties of a class are public class instance variables. These are variables that should be easily referenced and require no additional source when accessed, other than setting or getting their value. For instance, consider if one was to extend the class created in Listing 2.1 to contain information to better describe a person. This could be achieved by creating properties to contain a person's name and birth date as demonstrated in Listing 2.2. Note the asterisk preceding the variable names. This is due to `NSString` and `NSDate` being Objective-C objects and the language's thin layer around C: objects must be represented by their pointers [2, Defining Classes].

[4] A class which inherits from no other class and defines a common interface shared by all objects in the hierarchy below it.

Listing 2.2 Creating properties to contain information abouth a specific person.

```
#import <Foundation.h>

@interface MyPersonClass : NSObject

@property NSString *firstName;
@property NSString *lastName;
@property NSDate *birthDate;

@end
```

As mentioned earlier, objects primarily communicate with each other through messages. These messages are defined through method declarations. While the concept is similar to how one would assume C method declarations to behave, the syntax is quite different, as demonstrated in Listing 2.3a and Listing 2.3b. In Listing 2.3b, `first` and `second` are the variable names that must be referred to in the implementation of the method, while `anotherParameter` is a descriptive name of the parameter. It is important to note, however, that Objective-C does not support named parameters, as for instance Python does. Both the order of parameters and the descriptive name is part of the method declaration and must match when the method is called or is to be implemented. In other words, the declaration in Listing 2.3b is not the same as the one displayed in Listing 2.3c [2, Defining Classes]. In general, a method declaration in Objective-C follows the following pattern:

```
- (ReturnType)methodName:(FirstParameterType)firstParameter
nextParameterDescription:(NextParameterType)nextParameter
```

The leading - (dash) creates an instance method. Replacing it with a + (plus) will create a class method.

Implementation With the properties and methods defined in an interface, the implementation of a class' behaviour is written inside an `implementation` directive. As illustrated in Listing 2.4, this is done by importing the header file containg the interface declaration and writing the implementation for each defined method in the interface. Here, the interface example from earlier has been extended with an additional method, `secondsSinceBirthDate:(NSDate *)date` [2, Defining Classes].

Listing 2.3 Comparison of method definitions in C and Objective-C.

```c
void someMethod(int first,
                int second);
```

(a) Method definition in C.

```
- (void)someMethod:(int)first
   anotherParameter:(int)second;
```

(b) Method definition in Objective-C.

```
- (void)someMethod:(int)first
    secondParameter:(int)second;
```

(c) A second method definition in Objective-C.

2.5.2 Fundamental Frameworks and APIs

When developing with Objective-C for Mac OS X or iOS, there are a set of essential frameworks and Application Programming Interfaces (API). These are the Foundation framework and Cocoa and Cocoa Touch APIs.

Foundation

Foundation is a framework providing a base layer of primitive object classes and utility classes that are not covered by the Objective-C language. Among these are `NSString`, `NSArray`, `NSDictionary` and the previously referenced `NSDate`. Each of these are defined in separate header files. However, they can all be included by importing Foundation's primary header file, `Foundation.h` [5]:

```
#import <Foundation.h>
```

Cocoa and Cocoa Touch

Cocoa and Cocoa Touch are Apple's native object-oriented APIs for Mac OS X and iOS, respectively. Developers are encouraged to use Apple's Xcode Integrated Development Environment (IDE) when interfacing with Cocoa Touch, as the IDE tightly incorporates both APIs [3, 4].

Cocoa includes the previously introduced Foundation framework, in addition to the AppKit framework [3]. The latter includes classes for handling a program's User Interface (UI), as well as handling events when a user interacts with UI components. Cocoa Touch is based on Cocoa, with a an AppKit modified to suit iOS as well as additions for handling touch gestures and an animation framework [4].

Listing 2.4 Example of a defined interface and its implementation.

```objc
#import <Foundation.h>

@interface MyPersonClass : NSObject

@property NSString *firstName;
@property NSString *lastName;
@property NSDate *birthDate;

- (float)secondsSinceBirthDate:(NSDate *)date;

@end
```

(a) Interface for MyPersonClass.

```objc
#import "MyPersonClass.h"

@implementation MyPersonClass

- (float)secondsSinceBirthDate:(NSDate *)date {
    float seconds = [date timeIntervalSinceDate:self.birthDate];
    return seconds;
}

@end
```

(b) Implementation of MyPersonClass

Common to both is the use of the Model-View-Controller (MVC) development pattern for coupling user engaged events in the UI and corresponding data [3, 4].

Chapter 3

Multi-core and Parallel Computing

This chapter is a summary of the litterature studied at the beginning of the process leading up to this thesis. Within, the principles, technologies and hardware that define and set the bounds of the project is presented.

3.1 Parallel Programming

This section will give a quick overview of the architectures and memory models of parallel programming.

3.1.1 Architectural Definitions

Two different models are commonly used to define parallel architectures. The first uses the relationship between memory and compute units to describe its modes, while the other, often identified as Flynn's taxonomy, describe the relationship between instruction and data stream.

The former defines three different architectures:

Shared Memory: In shared memory, every compute unit shares the same unified memory location. This allows for relatively easy development and for fast communication between compute units. However, this architecture does generally not scale well because of race conditions. In addition, memory is often cached locally by compute units, which in turn raises a cache-coherency issue.

Distributed Memory: Distributed memory describes an architecture where each compute unit has its own private memory. This makes computation on local data very fast. However, computation on external data enforces communication between compute units beforehand, which in turn results the need for some interconnect and message passing interface between each compute unit. Distributed memory eliminates race conditions.

Distributed Shared Memory: In distributed shared memory, the memory is not shared among compute units, but it is addressed logically as it was shared. This means that the extra communication between compute units in distributed memory is abstracted away from the programmer.

Flynn's taxonomy defines four different models:

Single Instruction, Single Data (SISD): A single instruction is executed which does not exploit any data parallelism, either in the instruction or in the data.

Single Instruction, Multiple Data (SIMD): A single instruction which exploits multiple data streams. This is typical for graphics processing units (GPU).

Multiple Instruction, Single Data (MISD): Multiple instructions operate on a single data stream. This architecture is fairly uncommon.

Multiple Instruction, Multiple Data (MIMD): Multiple instructions simultaneously execute on multiple data streams. This architecture is typical for distributed systems, either organized as a shared memory system or a distributed memory system.

3.2 Parallel Scaling

It is important to note that the speedup gained by exploiting the parallelism in a program is highly dependent on the properties of each individual program, *i.e.* the fraction of the program's execution which is parallelizable. This rather pessimistic assumption is stated in Amdahl's law [1], and has later been revised by Mark D. Hill and Micheal R. Marty [24].

Parallel speedup is defined as the quotient after dividing the sequential execution time of an algorithm, T_1, by the parallel execution time of the algorithm, executed with p processors, T_p [39]:

$$S_P = \frac{T_1}{T_p}$$

Further, Amdahl's law states that if 90% of your program is sequential and the remaining 10% is parallelizable, the minimum execution time can not be lower than the 90% spent in the sequential part of the program. In other words, if P is the proportion of the program that can be made parallel, and $1-P$) is the remaining serial proportion, the maximum speedup given N processors is given by the following equation [1]:

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}}$$

John L. Gustafson and Edwin H. Barsis later reevaluated Amdahl's assertion in what has been known as "Gustafson's law" or "Gustafson-Barsis' law" [22]. Gustafson's law has a more optimistic take on parallel computing, stating that computations involving arbitrary large datasets can efficiently be parallelized. This has made Gustafson's statement a counter-part to Amdahl's law, which presents an upper bound for fixed size datasets. Gustafson's assumption was that software developers set the problem size based on the available hardware. Therefore, if more parallel hardware and powerful hardware is available, the problem size would increase. Inherently, as the problem size increases, the ratio of parallel-to-serial tasks also sees change. That is, the serial portion will become smaller in proportion to the total execution. Gustafson called his metric "scaled speedup" and defined it as such [22]:

$$S(P) = P - \alpha \times (1-P)$$

Where S is the speedup, P is the number of processors and α is the serial fraction of any parallel process.

With the advent of multi-core processors, Mark D. Hill and Michael R. Marty revised Amdahl's law in 2008, providing new insight to how multi-core processors should be designed in relation to Amdahl's law [24]. The authors deduced three equations for three different types of models:

3.2.1 Symmetric Multi-core Chips

Every core on a chip has equal cost, *i.e.* exploit the same amount of Base Core Equivalents (BCE)[1]. For instance, a symmetric multi-core with a budget of $n = 16$ BCEs and $r = 1$ BCE per core would give a 16 core symmetric, multi-core chip. In general, the number of cores is decided by the quotient of $\frac{n}{r}$, *i.e.* the BCE budget and number of BCEs per core. In addition, Hill and Marty defined a parameter $perf(r)$, which is equivalent to the performance of a core with r BCEs.

[1] A generic unit of cost depending on context, *e.g.* power, design effort or money.

The symmetric multi-core architecture uses one core to execute the serial part, with performance $perf(r)$, and applies all $\frac{n}{r}$ cores to execute the parallel part. As a result, the speedup is given by the formulae:

$$S_{symmetric}(f,n,r) = \frac{1}{\frac{1-f}{perf(r)} + \frac{f \times r}{perf(r) \times n}}$$

3.2.2 Asymmetric Multi-core Chips

The second architecture explored by Hill and Marty is that of the asymmetric multi-core chip. Here, the relation between BCEs and cores is not linear. Instead, several BCEs are combined into one large, more powerful core, while the remainder is divided among a set of smaller, less powerful cores. For instance, an asymmetric multi-core chip could have a budget of $n = 16$ BCEs at its disposal. Of these, four could be combined into one large, single core, with 12 small cores with one BCE each. In general, an asymmetric chip can have $1 + n - r$ cores. The large core allocates r BCEs, while the remainder, $n - r$, is distributed to the rest of the cores.

In the asymmetric architecture, the serial part is executed on the powerful core, and every core executes the parallel part. The speedup is modelled by the equation:

$$S_{asymmetric}(f,n,r) = \frac{1}{\frac{1-f}{perf(r)} + \frac{f}{perf(r)+n-r}}$$

With the advent of General Purpose Programming on GPUs (GPGPU), this architecture has become increasingly more relevant lately. If one thinks of a Central Processing Unit (CPU) as the large core, performing the serial portion of the program, the GPU can be thought of as the set of smaller, less performant cores, executing the parallel portion of the program. The result is a heterogeneous architecture.

3.2.3 Dynamic Multi-core Chips

The third and last architecture is that of dynamic multi-core chips. In this architecture, resources are dynamically allocated depending on where they are needed. When a program executes its serial fraction, all of the BCEs are combined into one large core. During the parallel execution of a program, the BCEs are distributed evenly among all cores, utilizing all base cores. This behaviour is modelled by:

$$S_{dynamic}(f,n,r) = \frac{1}{\frac{1-f}{perf(r)} + \frac{f}{n}}$$

For a more thorough explanation of the three architectures, and a modelled review of their performance, the reader is urged to investigate Hill and Marty's paper from 2008, "Amdahl's Law in the Multi-core Era" [24].

Chapter 4

Postings Lists in Inverted Indexes

This chapter will start with a description of the most common structure a postings list is given. Following sections will introduce techniques commonly employed to reduce the storage footprint. Due to the postings list often being organized as a list of sorted integers, more effective methods than generalized compression, such as Bzip2 and Zlib, have been developed. Coding schemes included ahead are *Variable-byte coding*, *Group Varint coding*, and *Elias γ coding*.

The chapter will end with an overview of work related to what has been performed in this thesis.

4.1 Structure

An inverted index is an index data structure used to map between arbitrary content, such as words or terms, to locations in a database or document storage. It is conventionally used in search engines to provide fast full text search, at the cost of expensive processing when the database or document storage system is updated. A common structure of an inverted index is to keep a *dictionary* of terms and pair each term with the individual IDs of documents the term occurs in. The resulting list of such IDs, or *postings*, is described as a *postings list* [46, p. 6]. Listing 4.1 displays a simple inverted index for the following sentences:

1. Hakuna Matata.
2. It is our motto.
3. What is a motto?

4. Nothing. What is a motto with you?

Below, terms occur on the left hand side, while separate postings lists are encapsulated in curly braces on the right hand side.

Listing 4.1 An inverted index for the four sentences above. Each number in the postings list represents the sentence a term occurs in.

```
a:       {3, 4}
hakuna:  {1}
is:      {2, 3, 4}
it:      {2}
matata:  {1}
motto:   {2, 3, 4}
nothing: {4}
our:     {2}
what:    {3, 4}
with:    {4}
you:     {4}
```

In its simplest form, a postings list constitute a number of Uniform Resource Identifiers (URI), sorted in ascending order. Each URI identifies a location in an arbitrary document storage system. Listing 4.2 illustrates one of form a postings list can take, a comma delimited series of sorted integers.

Listing 4.2 A simple example of a postings list for the term "motto" in the previously listed sentences.

```
motto:   {2, 3, 4}
```

A common addition to storing the URI is to also bundle the location of a word in each document together with the number of occurences. Again for the term "motto", a postings list more rich in information is demonstrated in Listing 4.3.

Listing 4.3 A postings list also containing the location of the word "motto" in each sentence, where the location is the *n*th position of the first character in "motto" in the sentence (whitespace included).

```
motto:  {<2, 11>, <3, 11>, <4, 20>}
```

4.2 Compression and Decompression

With an increasingly larger abundance of information being generated and indexed [18], the necessity of efficient schemes to minimize an inverted index' storage footprint is paramount. In addition, such schemes must enable fast retrieval of data stored in an index. Search engines implement a number of optimizations to reduce index size and provide better indexing and retrieval of data. Among these are specialized handling of extremely common terms which incur little benefit in providing a better search experience, so called "stop words" [46, p. 27], but also compression of the postings list. In addition to reducing the disk space, compression provides two additional benefits [46, p. 85]: *a*) Increased use of cache, as common terms can be stored in memory, rather than read from disk; and *b*) faster transfer between disk and memory. Indeed, compression schemes are known to have an efficiency level that surpass the time to transfer uncompressed data from disk to memory [46, p. 85].

A third, more subtle benefit is the ability to cache more data in memory, as an inverted index' size is decreased. For an uncompressed index, the cost of retrieval is equal to the sum of locating, *i.e.* seeking, for the index on disk, transferring it to main memory, and further caching it on the CPU. In order to deem a compression scheme successful, the reduction in retrieval time plus the time spent decompressing retrieved data, should not surpass the time cost of an uncompressed index [49].

The following sections will introduce techniques which facilitate the process of index minimization, each with a different granularity on their representation. Only *lossless* compression schemes that are effective for integer compression are presented.

4.2.1 Δ-Coding

Δ-*Coding* is the process of recording the difference or *delta* between sequential data, rather than the data values themselves. Algorithm 4.1 and 4.2 illustrate a serial, naive approach to encoding and decoding.

```
1: function DELTA(A : array[1..n])
2:     D ← array[1..n]                          ▷ Allocate result matrix D
3:     D[1] ← A[1]
4:     for i ← 2 to n do
5:         D[i] ← A[i] − A[i − 1]
6:     return D
```

Algorithm 4.1 Serial, naive approach to Δ-encoding.

```
1: function SUM(D : array[1..n])
2:     A ← array[1..n]                          ▷ Allocate result matrix A
3:     A[1] ← D[1]
4:     for i ← 1 to n − 1 do
5:         A[i + 1] ← A[i] + D[i + 1]
6:     return D
```

Algorithm 4.2 Serial, naive approach to Δ-decoding.

The effectiveness of the technique is influenced by the nature of data at hand. For an unsorted data set, Δ-encoding may yield little to no compression. However, for an evenly distributed data set of sorted values, results of compression may be significant. The difference in compression ratio is best demonstrated by an example: Consider the list of integers displayed in Listing 4.4. Assuming the result of encoding is stored as *ASCII* characters, Listing 4.5 and 4.6 illustrate the difference in length after encoding the list as unsorted data and sorted data. It can be seen that the former achieves a compression ratio of $\frac{40}{35} = 1.14$, while the latter achieves a compression ratio of $\frac{40}{22} = 1.89$.

Listing 4.4 An unsorted list of integers.

```
1 [177, 152, 171, 155, 170, 128, 163, 133, 143, 139]
```

Listing 4.5 List after Δ-encoding it without sorting.

```
1 [177, -25, 19, -16, 15, -42, 35, -30, 10, -4]
```

Listing 4.6 List after Δ-encoding it with sorting.

```
1 [128, 5, 6, 4, 9, 3, 8, 7, 1, 6]
```

4.2.2 Variable-byte Coding

As the name suggests, *Variable-byte Coding*, or *vByte*, is a byte-oriented coding scheme. It is popular in Information Retrieval (IR) systems because of its simplicity and balanced trade-off between speed and compression ratio [46, p. 96]. Variable-byte coding uses as an integral number of bits in a byte (7) to encode an integer's value (the *payload*), while the first bit (the *continuation bit*) denotes if a byte is the last byte of an encoded number. That is, the continuation bit is set to 1 if this is the last byte of an encoded number, otherwise it is set to 0 [46, p. 96]. Table 4.1 displays four integers and their respective binary representations after being encoded with Variable-byte coding.

Table 4.1 Example of Variable-byte coded integers. The continuation bit is highlighted in bold.

Integer	Encoded Bit String
1	**1**0000001
7	**1**0000111
9	**1**0001001
259	**0**0000010 **1**0000011

Decoding is done by reading a bytestream until the continuation bit is equal to 1. Payloads are then extracted from read bytes and concatenated into the resulting, decoded number [46, p. 96]. Algorithm 4.3 and Algorithm 4.4 illustrate the pseudocode for encoding and decoding, respectively. Decoding of variable-byte encoded numbers lends itself well to optimizations, as one is able to minimize the number of CPU cycles by use of bit shifts [12, p. 206].

During a keynote talk in 2009, Senior Google Fellow Jeff Dean introduced Google's modified variant of Variable-byte encoding, *Group VarInt*. A problem with traditional Variable-byte encoding is branch mispredictions. During decoding, the decoder must inspect every continuation bit and decide whether to continue decoding or concate the currently collected results and skip to the next set of data to decode. The decision is made via a branch instruction, and as such, a branch misprediction may occur. Group VarInt circumvents this by replacing the continuation bit with a two bit representation of an encoded posting's length. This allows for the

use of a lookup table or arithmetics to determine the number of bits to read for the posting currently being decoded [14]. Listing 4.7 illustrates the four previous values encoded with Group Varint. The vertical separator and the space between each encoded integer is present for readability only and not part of the encoded format.

Listing 4.7 An example of four integers encoded with Group Varint. Note that the length of the last encoded integer is first in bit mask representing the length of each encoded integer.

```
1 01 00 00 00 | 00000001 00000111 00001001 00000010 00000011
```

```
1: function ENCODE(numbers : array[1..n])
2:     bytestream ← array[]                    ▷ Allocate result array bytestream
3:     for i ← 1 to n do
4:         bytes ← array[]
5:         n ← numbers[i]
6:         while true do
7:             bytes ← [n mod 128 : bytes]     ▷ Prepend the result of n mod 128 to bytes
8:             if n > 128 then
9:                 break
10:            n ← n div 128
11:        bytes[len(bytes)] ← bytes[len(bytes)] + 128
12:        bytestream ← [bytestream : bytes]   ▷ Extend bytestream with bytes array
13:    return bytestream
```

Algorithm 4.3 Encoding a list of numbers with Variable-byte encoding.

```
1: function DECODE(bytestream : array[1..n])
2:     numbers ← array[]                          ▷ Allocate result array numbers
3:     n ← 0
4:     for i ← 1 to n do
5:         if bytestream[i] < 128 then
6:             n ← 128 × n + bytestream[i]
7:         else
8:             n ← 128 × n + (bytestream[i] − 128)
9:             numbers ← [numbers : n]   ▷ Append decoded number to numbers
   array.
10:            n ← 0
11:    return numbers
```

Algorithm 4.4 Decoding a bytestream of Variable-byte encoded numbers.

4.2.3 Elias γ Coding

Elias γ Coding is one of the first non-trivial coding schemes for positive integers, first described by Elias in 1975 [15, p. 193]. It is bit-oriented, dividing each encoded number into two components: *1*) the *selector*, a *unary* representation of the *body's* length; and *2*) the *body*, an integer's binary representation [12, p. 193]. Table 4.2 lists the encoded values of integers 1, 7, 9 and 21.

Table 4.2 Example of Elias γ encoded integers.

Integer	Selector	Body
1	1	1
7	110	111
9	1110	1001
21	11110	10101

With the unary segment of the encoded value having length $1 + \lfloor \log_2 x \rfloor$ and the binary representation having the same length, an integer encoded with Elias γ encoding will inhabit $2 \times \lfloor \log_2 x \rfloor + 2$ bits of space. By inverting the unary code, it can be observed that one is able to decrease the consumption of space, as the 1 bit between between the unary encoding and the integers binary representation is common. This is established from the fact that for an integer k with $selector(k) = j$, $2^{j-1} \leq k < 2^j$ is true. As such, the j-th least significant bit in k's binary representation, which happens to be the first bit in the encoded value's body, must be 1. With

this information in hand, the first bit in every encoded number's body is redundant and one can omit one bit per posting. The resulting storage foot print for an Elias γ encoded integer is thus $2 \times \lfloor \log_2 x \rfloor + 1$. [12, p. 193].

Elias γ coding is most effective when used together with postings lists of predominantly small gaps. However, for lists consisting of large gaps, it can be quite wasteful [12, p. 193]. Elias δ Coding is an attempt to improve the efficiency for larger values. Here, the length of the integer value, *i.e.* the values previously encoded in unary, is instead encoded using Elias γ encoding. This way of compression manages to represent an integer in $\lfloor \log_2 x \rfloor + 2 \times \lfloor \log_2 (\lfloor \log_2 x \rfloor + 1) \rfloor + 1$ bits [15]. However, Elias δ coding suffers from inefficiencies when encoding large values [49].

4.2.4 SIMD Accelerated Coding

The previously presented compression schemes are not trivially translated to SIMD instructions and accelerated in such a manner. This is in essence due to the variable nature of each codeword generated by the different techniques. For both Variable-byte coding and Elias γ coding, the byte position of an encoded value is unknown until the preceding codeword is decoded.

With some modifications to Elias' initial algorithm and storage format, Schlegel *et al.* are able to parallelize and produce a SIMD accelerated Elias γ coding scheme; *k-γ coding* [48].

Stepanov *et al.* use Variable-byte coding as a basis in their paper "SIMD-Based Decoding of Posting Lists", presenting SIMD accelerated compression techniques for variations of traditional Variable-byte coding, as well as the previously mentioned Group Varint.

Due to the unavailability of core SIMD instructions used in the implementation of the mentioned articles and time constraints, an implementation for ARM utilizing the NEON SIMD extensions have not been pursued.

4.3 Related Work

Research into how one can minimize a postings list's storage footprint, and compression of data in general, is a thorougly researched area. Common techniques for compressing an inverted index are found in book literature by Ian H. Witten *et al.* [58], as well as text books from Manning *et al.* [46] and Büttcher *et al.* [12]. Such techniques have further been revised and optimized, for instance by Falk Scholer *et al.* in "Compression of Inverted Indexes For Fast Query Evaluation" [49]. However, these authors are not concerned with the underlying hardware of the inverted in-

dex. “Fast integer compression using SIMD instructions” considers using Single Instruction Multiple Data (SIMD) instructions for a performance increase in integer compression and decompression, but does not address the storage medium [48].

Ahmed A. Aqrawi and Anne C. Elster considered compression performance in regards to SSD storage in 2011 [6]. However, their paper was concerned with the compression of seismic images and minimizing the size of data transferred across the bus between CPU and GPU. Microsoft, represented by Bojun Huang and Zenglin Xia attempted the use of flash memory as a replacement of expensive DRAM for caching frequently accessed data structures in a search engine [26].

To the best of this thesis' knowledge, none have previously investigated the nature of postings list encoding and decoding on a flash memory based storage medium in detail, and in addition, employed ultra-low-powered, *i.e.* handheld, hardware in the process.

Chapter 5

Postings List Coding on Mobile Devices

In order to perform the experiments required, two iOS applications have been created. The first is a tool to identify the flash memory read performance of benchmarked devices with a varied read buffer size. This was implemented due to an unavailability of similar applications for the platform. The first sections will be dedicated to the description of this.

The second is an application where one selects the compression scheme and buffer size, and measures the performance of a full disk-to-disk read-*encode*-write or read-*decode*-write. Below, test data, internal details of the application's implementation, concerns with developing for an embedded environment, and which coding schemes that have been implemented and the internals of these is disclosed. Problems one must handle when data is read blockwise and runs the risk of reading incomplete data is also described, together with how these problems have been solved.

Towards the end, details on how benchmarks are executed will be given.

5.1 Flash Memory Performance

Flash memory read performance is measured using a iOS application identified as "SSDPerformanceMapping". Via its User Interface (UI), displayed in Figure 5.1, a user is able to set the number of iterations for a performance measurement test, as well as the size of each block to be read and whether or not data should be read in a random access manner. During execution, a user will receive visual feedback on the

status of the benchmark through two progress bars. After a successful execution, data from the run may be saved by pushing the “Save Log” button. Data from the benchmark is then stored on the device and must be retrieved through iTunes. A sample log file is illustrated in Listing 5.1.

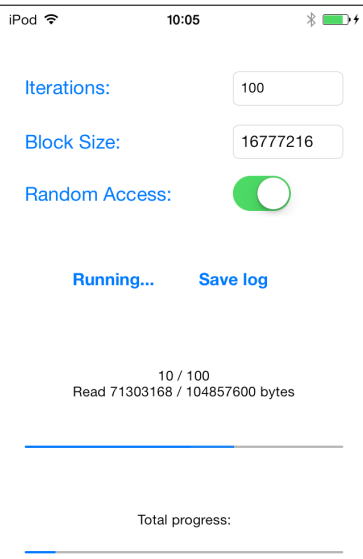

Figure 5.1 The User Interface of the Flash Memory benchmark app.

Listing 5.1 A sample of the output generated from a test run of the benchmark.

```
1 Test start: 2013-11-08 10:34:47 +0000
2 Test end: 2013-11-08 10:37:42 +0000.
3 Was Random: 0
4 Iterations: 10.
5 Total duration: 71.25 s.
6 Average iteration duration: 7.12 s.
7 Average transfer rate: 14.04 MB/s
```

5.1.1 Benchmark Details

A 100 Mebibytes (MiB) file of random data is bundled with the application[1]. The data file is read in blocks as configured by the executor, with each read block being discarded immediately. The time to read each block is summarized, with an average being calculated at the end, together with an estimate of the transfer rate.

If the benchmark is executed with several iterations one runs the risk of not achieving accurate results due to data being kept in cache. Restarting the device to clear memory is tedious and also prone to inaccurate results, and flushing the memory by reading another large file will cause the operating system to kill the application due to memory consumption. To counter caching of the data file, each iteration starts with moving and renaming the file. Tests have shown this to be a viable option for achieving accurate results.

If one is measuring random access performance, each read location, *i.e.* the jumps in the data file, are also randomized before each iteration.

[1] The file was generated by reading data from `/dev/urandom` as such: `dd if=/dev/urandom of=data.random bs=1024 count=\$((100*1024))`. The file size will be read as 105 MB (1×10^6), but is equal to 100 MiB (1×2^{20}).

5.2 Encode and Decode Performance

Benchmarking of encode and decode performance is similar to the flash memory measurements, in that a user is presented a UI with options, setting different parameters of the benchmark. These parameters include block size, number of iterations, number of threads, and the coding scheme to use. The option of setting the number of postings list terms to process is also present. However, current test data only includes one term and an associated postings list. Figure 5.2 displays a screenshot of the user interface.

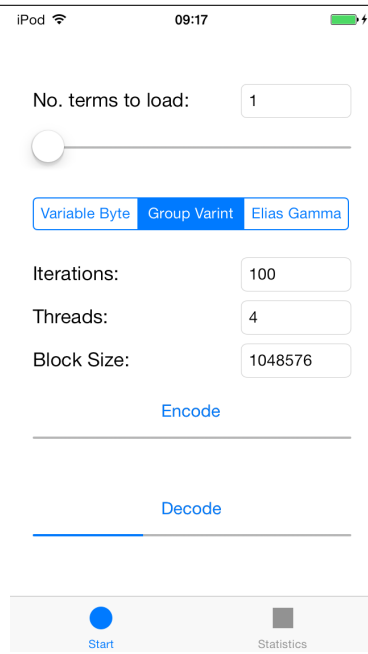

Figure 5.2 The user interface of the encoding and decoding benchmark app.

A summary of a benchmark run is displayed the user in the “Statistics” tab, where a user also is given the option to save a log of the execution. An example execution is displayed in Figure 5.3.

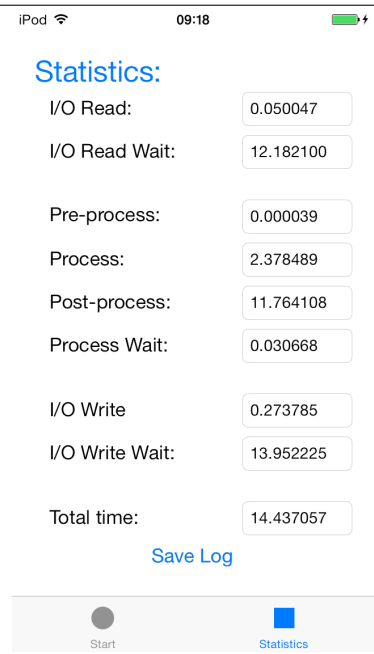

Figure 5.3 The user is presented the results of an execution in the encoding and decoding benchmark app.

5.2.1 Test Data Structure

A postings file is generated by a Python script and bundled with the application. In this project, the terms themselves are irrelevant. However, the length and distribution of postings in postings lists are not. As such, to uncomplicate the reading of data, terms are fixed in size and only represented as integers, prefixed with zeros to reach the fixed length. On the other hand, postings lists are generated according to Zipf's law [59]: The frequency of any word in a corpus of natural language utterances is inversely proportional to its rank in the frequency table. That is, the most frequent word in the corpus will occur approximately twice as often as the second most frequent word, three times as often as the third most frequent word, and so forth. For the test data structure of generated term and postings list pairs, this translates to the term simulated as being most popular having approximately twice the length as the second most popular term.

A file of test data is a simple structure of lines, where each line contains a term and postings list pair. The term and its associated postings list is separated by a *tab* character (`\t`), while each posting is separated by a comma (`,`). Listing 5.2 illustrates a sample of the test data set. In total, the data set consists of 4 862 476 postings.

Listing 5.2 A sample of the test data set.

```
1 00000001    17,60,86,92,107,119,126,129,145,167,170,172,175,179,186,218,238,269, ...
```

5.2.2 Disk-to-Disk Pipeline

The implementation is focused purely on benchmarking, and does not store intermediate results, *e.g.* encoded or decoded values of processed terms. Instead, term data is read from disk in blocks, processed by the CPU and written back to disk immediately.

Data is handled end-to-end in a five stage parallel pipeline, where each stage is notified of incoming work via the use of semaphores. The following paragraphs inform on the actions made during each stage. An illustration of the pipeline is displayed in 5.4.

Read block of data: Read a specific size of data into main memory. The size is constant throughout a run, except the last block if the number of bytes to reach end of data file is less than the set size.

Preprocess data: Data read during the earlier stage is preprocessed before encoding or decoding.

When reading unencoded data, a read block may split an integer. If this should occur, the tail of the buffer, *i.e.* the split integer, will be buffered and prepended to the consecutively read block. As such, while the block size of read data is constant, the actual size of the data to be processed in later stages may vary within a few bytes in size.

During encoding, incoming data is Δ-encoded in the pre-processing stage. As such, the last read data must also be buffered for use in the consecutive run to calculate the correct Δ-values for the complete set of integers.

Process data: The processing stage is where values are either encoded or decoded.

During decoding, a block of read data may not contain the required bytes to decode an integer correctly. All decoding implementations detect when a run is incomplete and record the position of the last completely decoded integer. The processing stage reads this value and buffers trailing data for the consecutive run.

Postprocess data: A stage where encoded or decoded data is postprocessed. During decoding, this stage calculates the prefix sum of incoming data. It also converts the array of decoded values to data that can be written to disk.

Write block of data: Receives postprocessed data and writes it to disk.

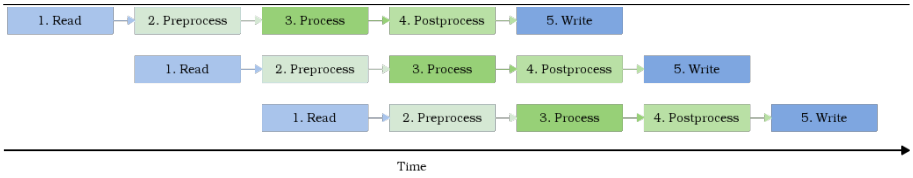

Figure 5.4 An illustration of the encoding/decoding pipeline.

An entity identified as a *Postings List Entry* wraps data passed between different stages. This entity has the following properties:

- `term`: The term the processed postings list belongs to.
- `range`: The start and end position of the postings list in the data file.
- `lastReadPosition`: The current position in the file containing the data set.
- `buffer`: The current block of read data.
- `bufferTail`: A buffer of any integers split during reading. This buffer is prepended to the formerly mentioned buffer during preprocessing.
- `toEncodeBuffer`: A buffer structured as an array with the current integers to be encoded.
- `toEncodeBufferTail`: An array of integers that were discarded during the current decode session. This buffer is only relevant during Group Varint encoding because of special requirements to the number of integers to be encoded.

- `lastReadValue`: The last unencoded value read in the currently buffered block of data. This property is critical in calculating the correct Δ-values for the complete set of integers.
- `encodedDataTail`: A tail of data to that was discarded during the previous write of encoded data. This buffer is only relevant during Elias γ coding it being a bitwise coding scheme.
- `haveDecodedBuffer`: An array of the currently decoded integers.
- `data`: A buffer containg the result data of the pipeline to be written to disk.
- `dataLength`: The length of the pipeline's result data.

The time spent in each stage, as well as the time each stage must wait for data to process is recorded:

- `READ`: The total time spent reading data.
- `READ_WAIT`: The total time spent waiting for the three process stages to complete.
- `PREPROCESS`: The total time spent preprocessing.
- `PROCESS`: The total time spent processing, *i.e.* encoding or decoding.
- `POST_PROCESS`: The total time spent postprocessing.
- `PROCESS_WAIT`: The total time spent waiting for data to be read from disk.
- `WRITE`: The total time spent writing data.
- `WRITE_WAIT`: The total time spent waiting for data from the processing stages.
- `TOTAL`: The total time spent on the benchmark.

5.2.3 Operating in an Embedded Environment

While *i*-devices, and similar hardware such as Android tablets and smart phones, become increasingly more powerful with each generation, they still feature traits found in embedded devices. One such trait is the limited available memory. For instance, the 5th generation iPod Touch only has 512 MiB of main memory. In addition, iOS does not allow one to freely allocate memory. If the total allocated

memory space of an application increases rapidly, the operating system will issue a warning. If the memory consumption continues to increase, the application will eventually be killed. This aggressiveness has proven difficult to overcome when operating with large files. As such, parts of the implementation may suffer in performance because one is forced release allocated memory in order to not provoke the operating system. In Objective C, this is handled through Automatic Reference Counting (ARC), a feature of Xcode where the burden of deallocating memory is placed on the compiler rather than the programmer. The compiler will investigate the source code and insert `release` and `retain` messages where it detects an object is no longer used. One can also force the insertion of calls to `release` and `retain` by wrapping source code in an `@autoreleasepool` block. This is a feature used heavily in this implementation to ensure memory is free between consecutive blocks of data being read into memory and data being processed. ARC is not to be confused with Garbage Collection, as found in other languages such as Java, as no background process or similar is running, collecting memory to be released.

5.2.4 Implemented Coding Schemes

All implemented schemes make use of Δ-encoding beforehand. The ones implemented are those presented in Chapter 4: *a*) Variable-byte Coding, *b*) Group Varint Coding, and *c*) Elias Gamma Coding.

Variable-byte Code

Of the three coding schemes, Variable-byte is the most straight forward to implement. It does not require the postings list's length to be divisable by a particular factor, and is also byte-oriented.

Encoding Encoding is executed by passing an array of unsigned integers to the encoding method. An initial pass is is made through the array to establish the amount of memory needed to hold the encoded values. This is done as to avoid having to resize the allocated memory area during encoding. The size of an integer in encoded form is equal to the number of bitwise right shifts by seven required to make the integer equal to zero. Listing 5.3 displays the source code used to establish an integer's encoded size. Another pass is made where each number is encoded and stored in an allocated `NSData` object.

Decoding A stream of bytes is supplied the decoding method, wrapped in an `NSData` object. Single bytes are read in turn and continuously decoded. The de-

Listing 5.3 Calculating the number of bytes required to store a Variable-byte encoded integer.

```
+ (NSUInteger)byteSize:(NSNumber *)number {
    NSUInteger numberAsInteger = [number integerValue];

    if (numberAsInteger <= 127)
        return 1;

    int byteSize = 0;
    do {
        ++byteSize;
        numberAsInteger >>= 7;
    } while (numberAsInteger);
    return byteSize;
}
```

coded value of a series of bytes or the length of the byte series is unknown until decoding is finished. This poses an issue when reading fixed size blocks, as a byte series representing an encoded integer may not be read completely. To counter this, a tail of the current read block, the size of the last decoded integer, is kept in memory and merged with the consecutive read block of data. This ensures all integers are decoded, and decoded into their correct, respective values.

The implementation is otherwise optimized with the use of bit shifts in replace of multiplications, divisions, and modulo operations.

Group Varint Code

Group Varint code incurs additional complexity compared to Variable-byte code. While being a byte-oriented coding scheme, the postings list is required to be a length divisable by four. This is due to how encoded integers are grouped and prefixed by their lengths.

Encoding Group Varint resembles Variable-byte coding in that integers are encoded by stripping leading zeros in the integers binary representation. Unlike, Variable-byte coding, however, Group Varint bundles several integers together and prefix the group with a bitmask representing the encoded byte length of each integer. During this project, a 32 bit version of Group Varint has been implemented. As a result, the prefix mask is one byte long, where each two bits signifies the size of an encoded integer: `00` represents a length of one byte, `01` represents a length of

two bytes, `10` represents a length of three bytes, and `11` represents a length of four bytes.

When encoding a group of integers, each integer has its value in the length bitmask calculated. This is done by first counting the number of leading zeros in an integer's binary representation. On ARM architectures, the implementation makes use of the `CLZ` instruction from the ARMv7 Instruction Set Architecture (ISA) language. Further, this value is divided by eight and subtracted from three to obtain the bitmask representing the number of bytes an integer occupies in encoded form.

Encoded integers are stored in a buffer equal to the sum of each integer's mask and the size of the length bitmask. Through the use of bitwise operations, integers are stored byte-by-byte in the buffer. In other words, an encoded integer occupying two bytes, will be split and stored in two continuous locations in the buffer. Listing 5.4 displays the source code of how the this operation is performed. The resulting buffer is wrapped in an `NSData` object and returned.

Because integers are grouped by four, it is important that four is a factor of the postings list length, unless one employs alternative coding schemes for trailing integers. In this implementation, an option is to ensure that the test data fulfills the length requirement of Group Varint coding. However, with the data being read in blocks of varying size for each benchmark run, it is not possible to ensure that each block contains a section of the postings list with a length also divisable by four. As such, read postings list sections have split to be divisable by four. The discarded data is collected and prepended to the consecutively read block. If the complete postings list itself has a length where four is not a factor, zeros are appended until it is. One would prefer to employ a secondary coding scheme for trailing values. However, with data being read in blocks, it is very difficult to detect when the alternative scheme is in use when decoding. The introduction of multithreaded encoding further adds to the complexity.

Decoding With the length bitmask being one byte in length, one can exploit that bitmask only can take 256 different values. In the decoding implementation, a lookup table was created where each index, *i.e.* entry, in the table points to an array with the length mask of each encoded integer. Listing 5.5 displays a sample of the lookup table. Having found the length of each integer in encoded form, the implementation jumps ahead in the read buffer and decodes each integer in sequence. For integers above a single byte in size, several sequential bytes are read from the buffer and combined through bitwise operations into the resulting 32 bit integer. The combining of integers is presented in Listing 5.6. As of this project, there exists no de-facto or officinal reference implementation of Group Varint, only details on

Listing 5.4 Stripping the leading zeros off of 32 bit integers, and storing them as discrete bytes in a buffer.

```
+ (void)stripLeadingZeros:(UInt8 *)buffer
         atBufferPosition:(NSUInteger)position
                forNumber:(UInt32)number
                  withKey:(UInt8)key {

    if (key == 0)
        buffer[position] = number & 0xFF;
    else if (key == 1) {
        buffer[position] = number & 0xFF;
        buffer[position + 1] = (number >> 8) & 0xFF;
    }
    else if (key == 2) {
        buffer[position] = number & 0xFF;
        buffer[position + 1] = (number >> 8) & 0xFF;
        buffer[position + 2] = (number >> 16) & 0xFF;
    }
    else {
        buffer[position] = number & 0xFF;
        buffer[position + 1] = (number >> 8) & 0xFF;
        buffer[position + 2] = (number >> 16) & 0xFF;
        buffer[position + 3] = (number >> 24) & 0xFF;
    }
}
```

the organization of bytes. As such, the implementation presented here may vary from other textbook or sample implementations.

Elias γ Code

Elias γ code is in principle a simple scheme. However, being a bitwise coding scheme, it incurs additional complexity during encoding.

Encoding As presented in Section 4.2.3, an encoded value consists of two parts: the length of an integer's bit representation written in unary, and the actual bit representation. While the algorithm itself is simple, the compression scheme's bitwise nature poses a problem when one encodes blocks of data and writes it to storage on completion. Often, the data after such a block encoding is not byte aligned. As the writing of data is done bytewise, this results in a trail of garbage bits at the end of an encoded block. Not only will such bits increase the file size, they will also force errornous data to be produced during decoding. Consider a block of encoded

Listing 5.5 A sample of the lookup table used during Group Varint decoding.

```
static const UInt8 MASK_LOOKUP_TABLE[256][4] = {
    {1, 1, 1, 1}, // 00 00 00 00
    {1, 1, 1, 2}, // 00 00 00 01
    {1, 1, 1, 3}, // 00 00 00 10
    {1, 1, 1, 4}, // 00 00 00 11

    {1, 1, 2, 1}, // 00 00 01 00
    {1, 1, 2, 2}, // 00 00 01 01
    {1, 1, 2, 3}, // 00 00 01 10
    {1, 1, 2, 4}, // 00 00 01 11

    ...

    {4, 4, 3, 1}, // 11 11 10 00
    {4, 4, 3, 2}, // 11 11 10 01
    {4, 4, 3, 3}, // 11 11 10 10
    {4, 4, 3, 4}, // 11 11 10 11

    {4, 4, 4, 1}, // 11 11 11 00
    {4, 4, 4, 2}, // 11 11 11 01
    {4, 4, 4, 3}, // 11 11 11 10
    {4, 4, 4, 4}  // 11 11 11 11
};
```

integers, 1021 bits in length. Assuming the block ended with the bits `10100`, an additional three bits of garbage, `000`, would be appended on write. A consecutive block of encoded data given the starting bits `00100` would result in the following representation in storage: `10000000100`, while the correct is `10000100`.

To handle such behaviour, encoded values of integers are first represented as arrays of boolean values. In Objective C, the type `BOOL` is an alias for C's `signed char` data type. These arrays are then truncated to the nearest length divisable by eight. Bits, *i.e.* `BOOL` elements, outside the new array are stored in memory and prepended to the next block of encoded data. The truncated array is then rewritten in binary and stored to disk. How the rewrite is performed is displayed in Listing 5.7.

Decoding Decoding is performed by continuously reading encoded bytes, where each byte is investigated bitwise. First, the length of the encoded integer is calculated by incrementing a counter until a set bit is encountered. Second, an amount of bits equal to the counter is read and combined into the resulting, decoded integer. Each number is appended to an array before they are summarized and written

Listing 5.6 Reading a byte buffer and decoding bytes into the original, 32 bit integer.

```
+ (NSUInteger)decodeBytes:(UInt8 *)buffer
             fromPosition:(NSUInteger)position
                   length:(UInt8)length {

    if (length == 1)
        return buffer[position];

    else if (length == 2)
        return buffer[position] |
               (buffer[position + 1] << 8);

    else if (length == 3)
        return buffer[position] |
               (buffer[position + 1] << 8) |
               (buffer[position + 2] << 16);

    else
        return buffer[position] |
               (buffer[position + 1] << 8) |
               (buffer[position + 2] << 16) |
               (buffer[position + 3] << 24);
}
```

to permanent storage.

It is important to properly ensure an individual decode is complete. As bits are continuously read and combined, errornous values may be produced if the complete encoded data of an integer is not present in the currently buffered block of data. As such, a decode in progress will not have its produced value added to the set of decoded values if its length bit length is found to surpass the current buffer length. Rather, this data is buffered and prepended to the next block to be decoded.

5.2.5 Parallelization Opportunities

The implementation has attempted to utilize the parallel resources available in Apple's devices.

Parallel Encoding

In the implementation, encoding is performed in two stages: First, the postings list is Δ-encoded. Second, the encoding scheme at hand is applied to the Δ-values. As the encoding of an integer is completely independent of all other integers, this

Listing 5.7 Writing an array of boolean bytes as bits.

```
+ (NSData *)rewriteByteArray:(NSData *)data {
    NSUInteger bitLength = (data.length >> 3) + 1;
    BOOL *bytes = (BOOL *)data.bytes;
    UInt8 bits[bitLength];


    for (NSUInteger i = 0; i < bytes.length; i++) {
        BOOL bit = bytes[i];
        if (bit) {
            bits[i >> 3] |= 1 << (i & 7);
        }
        else {
            bits[i >> 3] &= ~(1 << (i & 7));
        }
    }

    return [NSData dataWithBytes:bits length:bitLength];
}
```

latter operation can be performed in parallel. The implementation described here solves this by evenly dividing an incoming array of integers among a set number of threads. Each thread calculates its distinct set of data, which are later combined with the overall thread's data in sequence and returned. This three-step process may incur a penalty in performance if the amount of data to process is too small.

Parallel Decoding

Parallelization during decoding is not straightforward. Both Variable-byte coding and Elias γ coding are inherently serial. In the former, the length of a decoded integer is unknown until all but the last byte is read. In consequence, subsequent values can not be decoded until the preceding are finished. Stepanov *et al.* managed to utilize Single Instruction Multiple Data (SIMD) instructions with Variable-byte decoding by modifying the algorithm to prefix a set of values with their data individual data lengths, similar to Group Varint, and introducing an extra step of applying several masks and bitshifts [55]. However, their use of SIMD instructions did not result in a significant performance increase [55]. As such, this opportunity has not been investigated further.

The latter coding scheme suffers in a similar manner. However, unlike Variable-byte coding, the length of data associated with an integer is known before the actual

decoding is performed. This allows one to read the length, dispatch the remaining decoding operation to a separate thread, and continue to the next value. In this implementation, however, such a solution has not been taken advantage of. There's an overhead associated with dispatching a new thread, an overhead which is assumed to be larger than the increase in performance.

Concerning Group Varint, groups of four integers are completely independent. This allows one to read the first byte of a group to establish the group's total length and then dispatch a thread to perform the actual decoding. However, tests have shown the duration taken to dispatch a new thread surpasses the time required to decode a group by an order of six. In other words, there is no gain in such an implementation. Stepanov *et al.* managed to achieve a signicant speedup in their SIMD-based implementation of Group Varint [55]. However, the benchmark was performed on a Streaming SIMD Extensions 3 (SSE3) enabled system, and made use of an instruction identified as `PSHUFB`. This instruction receives a bitmask and a packed set of bytes and shuffles the bytes according to the bitmask. Unfortunately, an equivalent instruction does not exist in ARM's NEON SIMD extensions.

While each coding scheme is difficult to parallelize, there are opportunities in the surrounding implementation. After a block of data is decoded, the data is stored in an array. This array must then be converted to continuous bytes to enable writing it to permanent storage. It is possible to divide the array among several threads have the conversion happen in parallel.

5.3 Benchmark Execution

Benchmarks were executed on three devices for both the flash memory and encode and decode performance tests:

a) a 5th generation iPod Touch, *b*) a 4th generation iPad, *c*) and an iPad Air.

A summary of each device' specifications is displayed in Table 5.1.

Table 5.1 Concrete specifications of the devices employed during benchmarking.

	5th gen. iPod Touch	4th gen. iPad	iPad Air
SoC:	Apple A5	Apple A6X	Apple A7
Memory type:	LPDDR2	LPDDR2	LPDDR3
Memory amount:	512 MiB	1 GiB	1 GiB
Storage:	16 GiB	64 GiB	128 GiB

During the measurement of flash memory performance, a total of 30 configurations were executed per device. 15 different block sizes were investigated for both

sequential read performance and random access read performance[2]:

- 512 Bytes
- 1 KiB
- 2 KiB
- 4 KiB
- 8 KiB
- 16 KiB
- 32 KiB
- 64 KiB
- 256 KiB
- 512 KiB
- 1 MiB
- 2 MiB
- 4 MiB
- 8 MiB
- 16 MiB

A selection of the above mentioned block sizes were employed further in testing of encode and decode performance:

- 512 Bytes
- 1 KiB
- 4 KiB
- 512 KiB
- 1 MiB

[2] Block sizes were calculated in accorandance with IEEE standards, *i.e.* they are all values of 2^{nth}.

- 4 MiB

The reasoning behind these six being chosen is presented in Chapter 6.

In addition, encode and decode benchmarks were conducted with three different thread configurations:

- 1 Thread
- 2 Threads
- 4 Threads

With three different coding schemes to investigate in two different modi on three distinct devices, the number of configurations performed totals 324.

Chapter 6

Results and Discussion

Within this chapter are the results of the benchmarks described in Chapter 5. In addition, an overview of each coding scheme's respective performance and compression ratio will be presented.

The chapter will begin with an introduction of the different coding scheme's properties, before the results of the flash memory performance tests are presented. The results from encoding and decoding will be displayed towards the end. The latter results will introduce two new performance metrics: Encoded Integers Per Second (EIPS) and Decoded Integers Per Second (DIPS). This is a measure of the raw performance in each device for a given coding scheme. It is calculated by dividing the number of postings in the test data (4 862 476) by the time spent in the processing stage.

6.1 Coding Scheme Properties

Properties of each coding technique were researched by encoding and decoding the complete data set, recording the total time spent processing the data, and dividing the result by the number of postings processed. This gives an average number of seconds each coding scheme spends on encoding or decoding. Lastly, the resulting byte size after encoding was recorded to give insight to each technique's compression ratio. Table 6.1, Table 6.2 and Table 6.3 display encoding performance per posting, decoding performance per posting, and compression ratio respectively. As a demonstration of the viability of specialized encoding methods for postings lists, results after compressing the same data set with Bzip2 and Zlib are included. These are two well-known, general purpose compression methods. Both were applied to

the data set with the maximum compression level. While, all specialized encoding schemes provide worse compression than both Bzip2 and Zlib, the overhead associated with each general purpose method requires the uncompressed data to contain a significant amount of data, rendering such encodings non-applicable for short postings lists.

Table 6.1 Comparison of the time spent encoding the test data set.

Encoding	Time per posting	Total time
Variable-byte	822.6 ns	4.0 s
Group Varint	199.5 ns	0.97 s
Elias γ	674.6 ns	3.28 s

Table 6.2 Comparison of the time spent decoding the test data set.

Decoding	Time per posting	Total time
Variable-byte	43.2 ns	0.21 s
Group Varint	51.4 ns	0.25 s
Elias γ	125.5 ns	0.61 s

Table 6.3 Comparison of compression ratio on the test data set.

Scheme	Original File Size	Result File Size	Ratio
Variable-byte	43 221 384 B	4 871 192 B	8.87
Group Varint	43 221 384 B	6 078 117 B	7.11
Elias γ	43 221 384 B	4 558 525 B	9.48
Bzip2	*43 221 384 B*	*3 821 457 B*	*11.31*
Zlib	*43 221 384 B*	*4 486 009 B*	*9.63*

It is apparent that Group Varint provides the best performance both in terms of encoding and decoding. However, the gain in performance takes its toll on the ability to compress data. On the other hand, Elias γ coding provides excellent compression, but suffers in decoding speed. Variable-byte coding presents itself as the middle ground between compression ratio and performance in terms of speed.

In an addenda to the book "Information Retrieval: Implementing and Evaluating Search Engines", Büttcher *et al.* measure a relative performance difference between the techniques as presented here [13]. They pin Variable byte coding's

decoding results on the nature of their data set: The postings within the data set require seven bits or less, which in turn results in few branch mispredictions during Variable-byte decoding [13]. Indeed, low Δ-values is a trait in the test data applied in these benchmarks as well. Performing the same benchmark without prior Δ-encoding gives the following table (Table 6.4):

Table 6.4 Comparison of the time spent decoding the test data set without prior delta encoding.

Decoding	Time per posting	Total time
Variable-byte	65.8 ns	0.32 s
Group Varint	50.2 ns	0.24 s
Elias γ	353.7 ns	1.72 s

Branch mispredictions during Variable-byte decoding has placed Group Varint ahead in terms of decoding performance.

6.2 Flash Memory Performance

Following are the results of the flash memory read benchmarks. Devices are presented in descending order, sorted after their respective age.

6.2.1 iPod Touch, 5th Generation

From the results displayed in Figure 6.1 and Figure 6.2, it is apparent how the iPod favours a larger block size for both sequential access and random access. However, the performance increase appears to become saturated at a block size of 1 MiB.

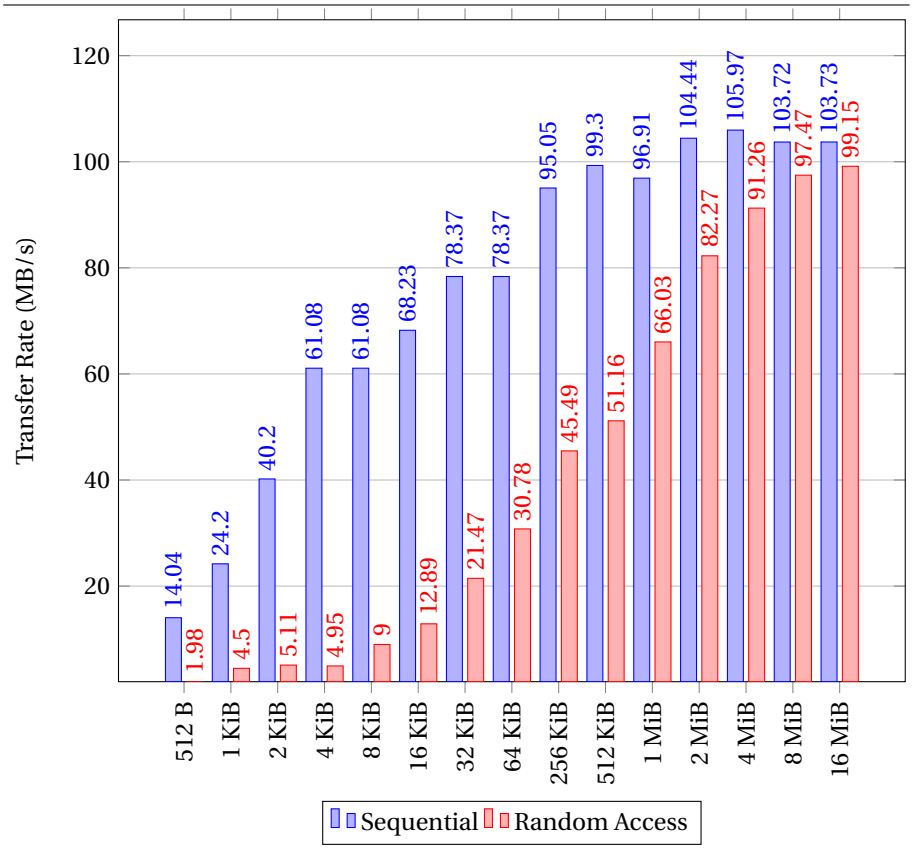

Figure 6.1 Results of flash memory benchmark on a 5th generation iPod Touch.

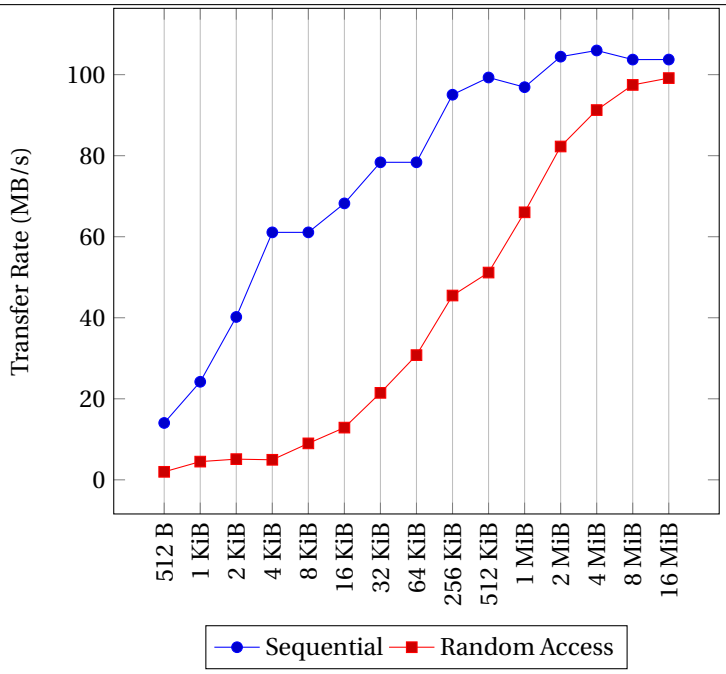

Figure 6.2 Results of flash memory benchmark on a 5th generation iPod Touch.

6.2.2 iPad, 4th Generation

The results from the flash memory perfomance measurement of the 4th generation iPad are presented in Figure 6.3 and Figure 6.4. Compared to the iPod, the iPad represents a much steeper curve before it reaches its peak transfer rate. Indeed, the highest transfer rate is measured as early as 16 KiB. It is not known what causes the 4th generation iPad to achieve such a transfer rate for blocks of this size. As will be presented further on, such a trait is only present in this device. Consecutive block sizes have a stable, albeit slowly decreasing transfer rate.

Concerning random access, the 4th generation iPad display similar traits as the iPod. Random access performance appear to have exponential growth, before stabilizing at a block size of about 2 MiB.

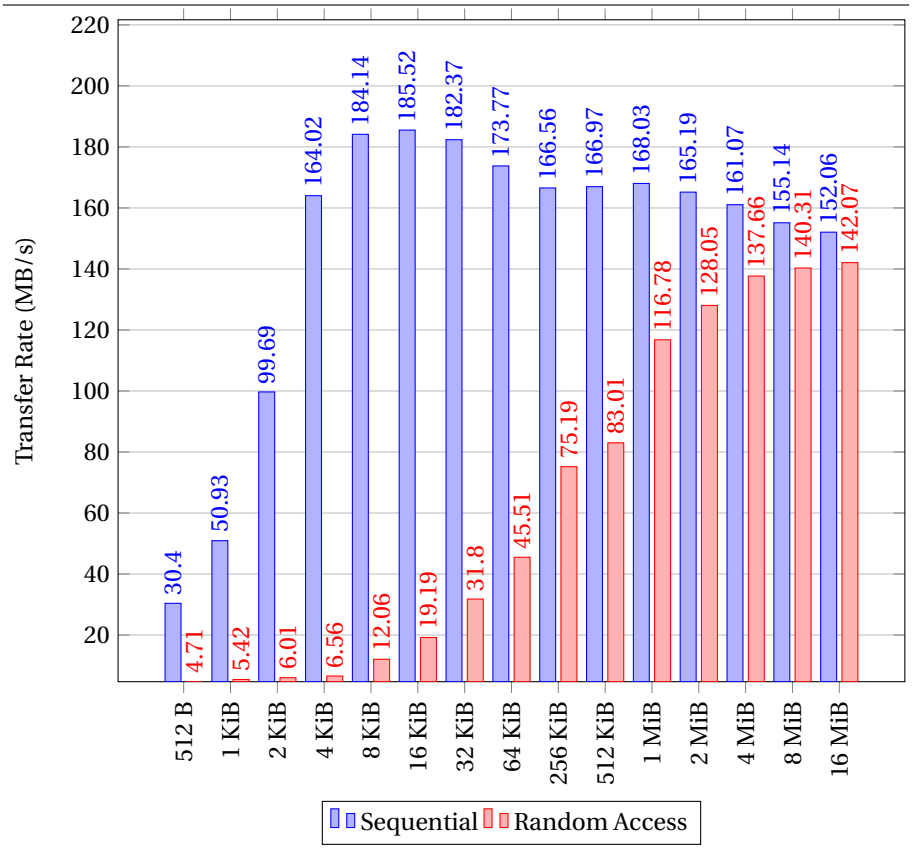

Figure 6.3 Results of flash memory benchmark on a 4th generation iPad.

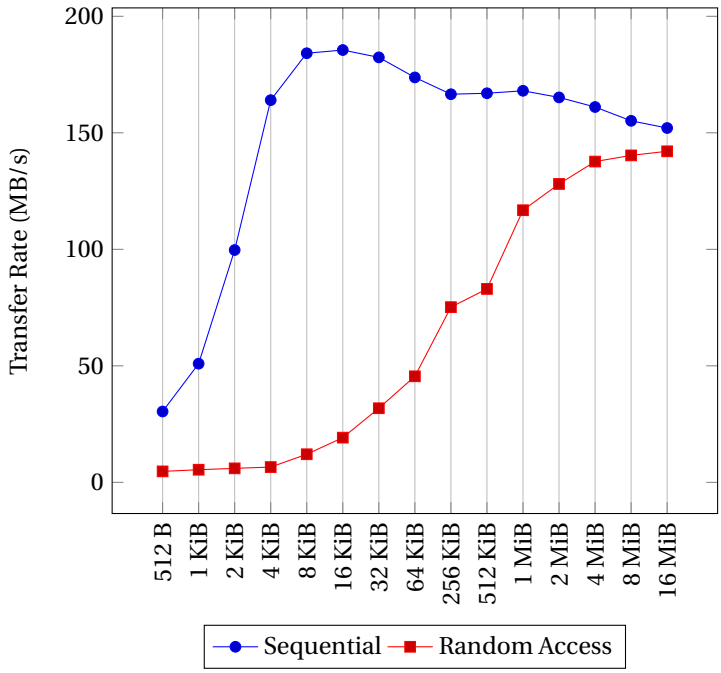

Figure 6.4 Results of flash memory benchmark on a 4th generation iPad.

6.2.3 iPad Air

Figure 6.5 and Figure 6.6 display the results from the flash memory benchmark on iPad Air. Both figures resemble the four previosly presented results in the large difference between sequential and random access for small to mid-range block sizes. The iPad Air presents itself as particularly similar to the iPod, with a steep increase up to 4 KiB – 8 KiB, and a more conservative performance growth towards larger block sizes.

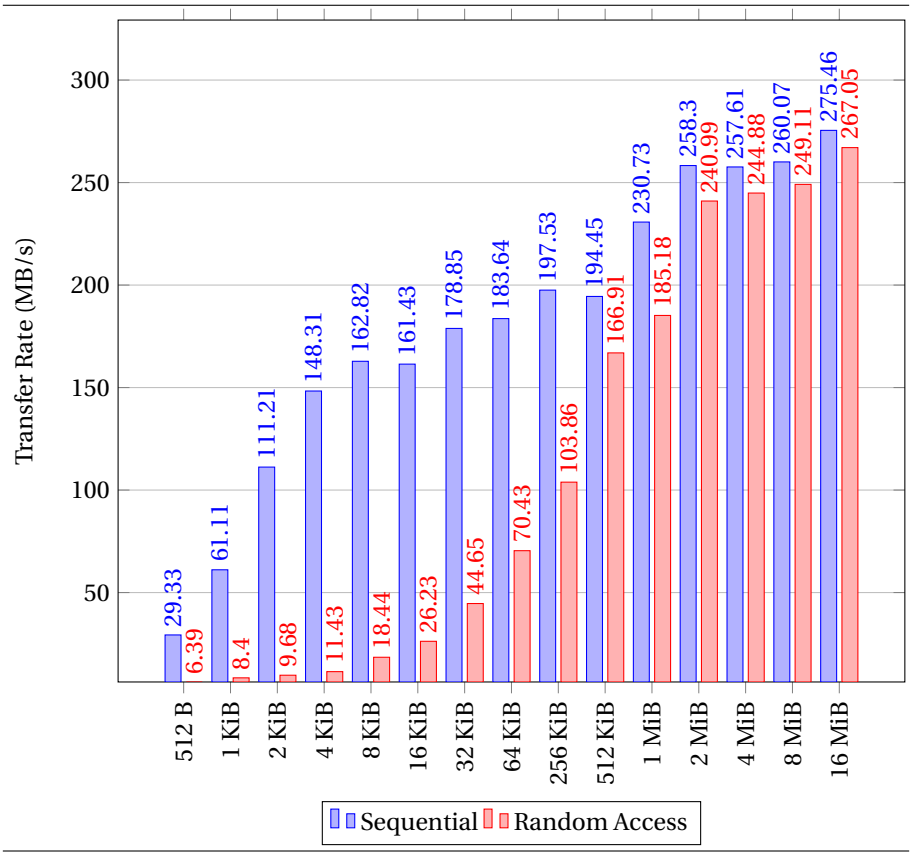

Figure 6.5 Results of flash memory benchmark on an iPad Air.

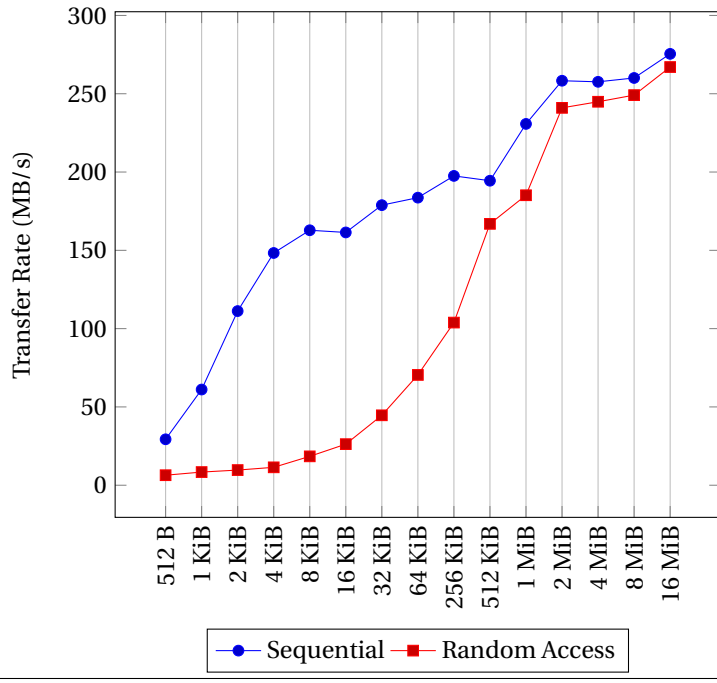

Figure 6.6 Results of flash memory benchmark on an iPad Air.

6.2.4 Summary

Figure 6.7 summarize the sequential performance of all three devices in a single graph. All three devices deliver impressive results, however, just as impressive is the relative performance increase between the 4th generation iPad and the new iPad Air: a near doubling in transfer rate from one iteration of the iPad to the next. With such a difference between two versions of the iPad, one might expect the performance delta between the iPod Touch and the 4th generation iPad to have been greater.

Little is known of the interface residing between the device's main memory and flash memory. An evolution in components in all areas of the devices is to be expected. In principle, the read performance measured should not be affected by properties in main memory or the CPU. However, the operating system may provide performance benefits by utilizing a wider memory bus or CPU cache. Indeed,

the Apple A7 found in iPad Air has double the memory bandwidth of its older sibling, in addition to a processor wide cache of 4 MiB [42]. These features may be culprits behind such an increase in perfomance.

Figure 6.8 displays a comparison of the random access read performance of the three devices. Each device' graph near echoes the shape of its comparands. Interestingly, a random access read pattern portraits properties similary to traditional, mechanical hard drives. For smaller block sizes, it appears that the seeks being performed to skip from location to location in the data file are quite costly. However, it is important to keep in mind the flash memory technology in use: Negated AND (NAND) flash memory. Indeed, NAND flash memory's organization of data and hardware interface makes random access for small block sizes a costly operation [11, 47]. Properties specific to NAND flash memory may also be to blame for the large difference between sequential and random access seen between block sizes of 1 KiB – 8 MiB.

The results off of the flash memory performance survey, sets the block sizes of which the encoding and decoding benchmarks will use. From the presented results, it is a clear distinction in the performance for lower block sizes, 512 B – 1 KiB, and higher block sizes, 512 KiB – 16 MiB. As such, block sizes from both regions have been chosen. In addition, the native block size of the file system, Hierarchical File System, case sensitive, (HFSX), is 4 KiB, which makes this a natural block size to investigate. This leads to the following block sizes being used further in encoding and decoding surveys:

- 512 Bytes
- 1 KiB
- 4 KiB
- 512 KiB
- 1 MiB
- 4 MiB

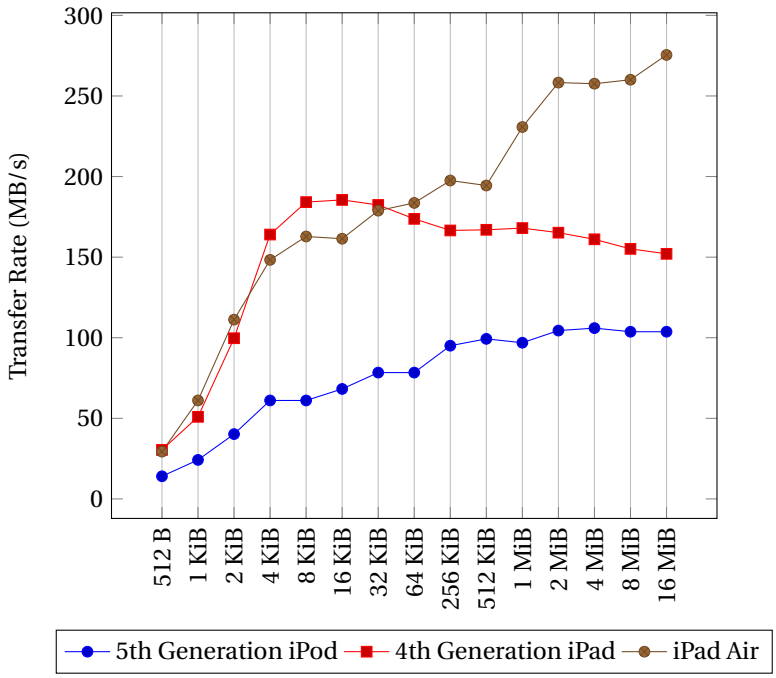

Figure 6.7 Summary of sequential flash memory performance.

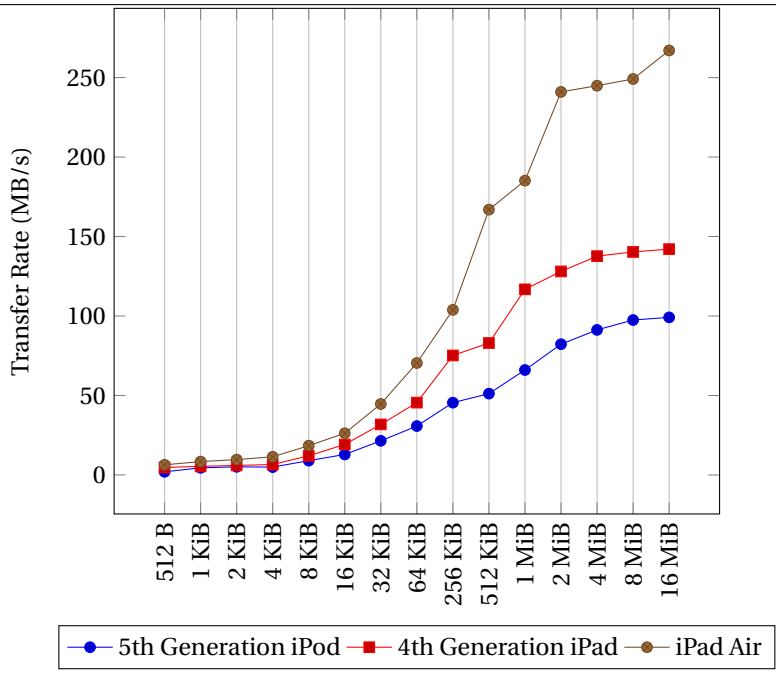

Figure 6.8 Summary of random access flash memory performance.

6.3 Performance Critical Applications in Objective-C

Before presenting concrete results, an important experience made during development of performance critical applications in Objective-C should be presented. With Objective-C being the language in use, it is tempting to employ the use of Objective-C objects to ease implementation. Initially, built-in objects such as `NSArray`, representing an array, and `NSNumber`, a container class for all numbers, were used. However, results were more than unsatisfactory. As such, investigations were made to modify the source code to employ bits of the Foundation fromework more close to pure C.

During both encoding and decoding, all occurences of `NSArray` were replaced by `CFArray`, a thinner encompassment of malloc, and unlike `NSArray`, capable of holding data not inheriting from `NSObject`. Results were significant. Figure 6.9 compare two single threaded executions of Variable-byte decoding on a 5th generation iPod, displaying results where the optimized implementation offers almost six times the performance of the preoptimization implementation.

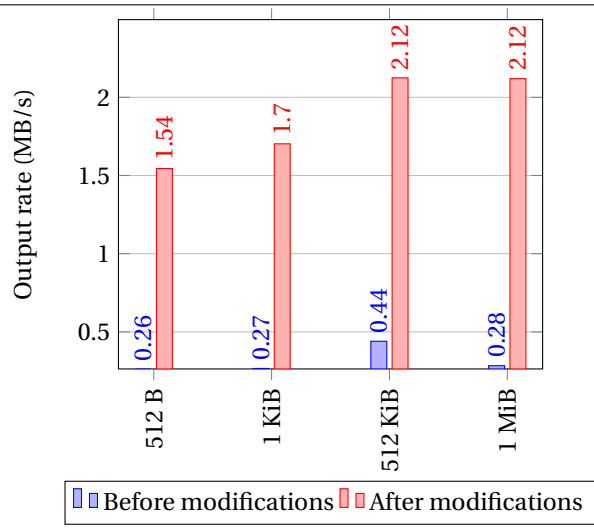

Figure 6.9 Disk-to-disk Variable-byte decoding before and after Objective-C optimizations.

6.4 Variable-byte Coding Performance

Initial tests of Variable-byte coding in Section 6.1 placed it as the slowest during encoding, but the nature of the test data set enabled it to claim the throne as the fastest during decoding. The following sections will further elaborate on the properties of Variable-byte coding on iOS platforms.

6.4.1 Encoding

The results from the encoding survey are divided into graphs per device, where each slope represents a different thread configuration. Coding schemes are presented in separate sections, with devices listed in the order as in Section 6.2.

iPod, 5th Generation

From Figure 6.10, it is quite apparent how multithreading hampers performance for a lower block size. The cost overhead of dispatching additional threads, as well as dividing the workload among said threads, is greater than the benefit. Not until providing the processing stages with 4 KiB blocks is the multithreaded implementation on par with single threaded processing, and the maximum provided speedup is only of about 19 %.

Table 6.5 displays detailed statistics, comparing a single threaded and a dual threaded run with a 4 MiB block size. Applying multithreading during encoding, *i.e.* the processing stage, has some benefit, however, not enough have a significant impact on overall performance. Circa 81 % of the computation takes place within the processing stage. As this is the section of the application one assumes to be parallelizable, one can apply Hill and Marty's formulae for symmetric multi-core chips with $r = 1$ and calculate a theoretical upper bound for expected speedup [24][1]:

- 2 threads: $Speedup_{symmetric}(n = 2, r = 1, p = 0.81) = \frac{1}{1-0.81+\frac{0.81}{2}} = 1.68$
- 4 threads: $Speedup_{symmetric}(n = 4, r = 1, p = 0.81) = \frac{1}{1-0.81+\frac{0.81}{4}} = 2.55$

A practical speedup of 1.19 is much lower than the theoretical.

Data in Table 6.5 also reveal that more time is spent waiting for the processing stage to finish than time spent doing actual processing, *i.e.* I/O fetches data faster than the CPU can process it. This is also reflected in the low *Process Wait* value,

[1] Hill and Marty's formulae is used as the processor within the iPod is a multi-core CPU. Assuming $r = 1$, this is equivalent to applying Amdahl's law [1].

and translates to efficient use of the CPU. From Figure 6.11 one can observe how the block size is strongly correlated with efficient use of the CPU. A smaller block size results in better efficiency on average per block, but a larger block size is more efficient overall.

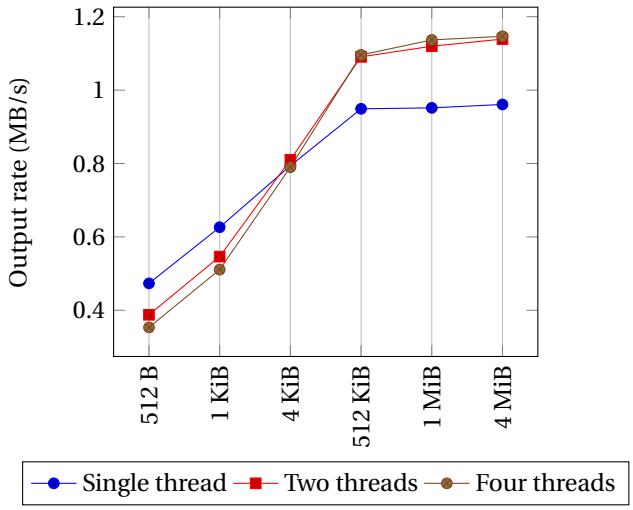

Figure 6.10 Disk-to-disk Variable-byte encoding for configurations of single thread, two threads, and four threads on a 5th generation iPod.

Table 6.5 Detailed comparison of a single threaded Variable-byte encoding and dual threaded Variable-byte encoding on a 5th generation iPod.

	Single Threaded (ms)	Dual Threaded (ms)	Ratio
I/O Read:	264.6	371.1	0.71
I/O Read Wait:	48320.2	40682.5	1.19
Preprocessing:	9143.9	9165.4	1.0
Processing:	40630.2	32764.3	1.24
Postprocessing:	3.3	3.1	1.006
Process Wait:	287.0	327.2	0.89
I/O Write:	40.1	46.8	0.86
I/O Write Wait:	50094.3	42244.9	1.19
Total Execution Time:	50185.5	42335.4	1.19

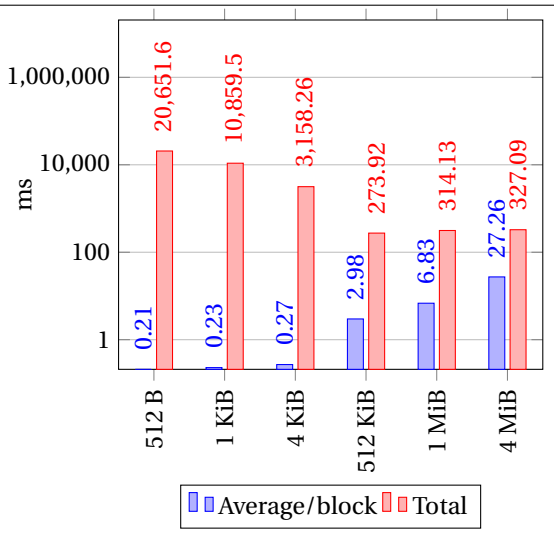

Figure 6.11 Correlation between block size and process waiting time on a 5th generation iPod (lower is better).

iPad, 4th Generation

Figure 6.12 contains the results from running Variable-byte encoding on the 4th generation iPad. A similar trend as the one observed earlier with the iPod is present here as well. For the attempted block sizes, multithreading has little to no effect, as the serial performance of the A6X System on Chip (SoC) present in the device encodes data faster than the overhead associated with dispatching a thread. With 85 % of the execution spent in the stage attempted parallelized, theoretical values of speedup would compare to the ones calculated for the iPod Touch. However, Table 6.6 displays a meager 5 % increase in performance.

While the performance is almost double compared to that of the iPod, the shape of each curve is almost identical to the previous ones. Backed by the results from the flash memory survey, one can assume the increased performance is due to improvements in the processor more than in the flash memory itself or the interface between processor and flash memory.

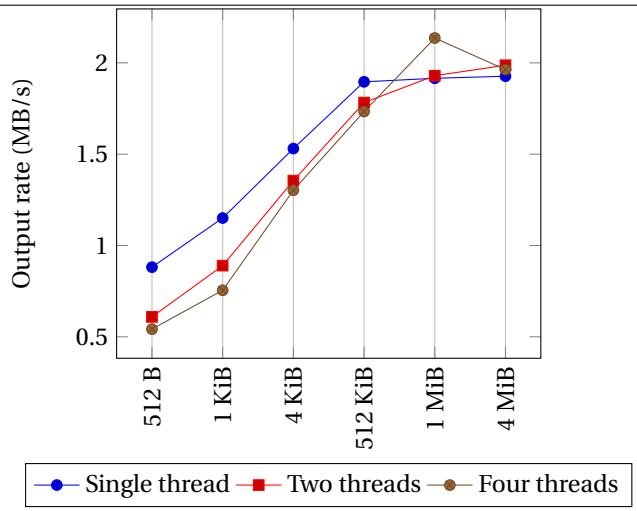

Figure 6.12 Disk-to-disk Variable-byte encoding for configurations of single thread, two threads, and four threads on a 4th generation iPad.

Table 6.6 Detailed comparison of a single threaded Variable-byte encoding and dual threaded Variable-byte encoding on a 4th generation iPad 4.

	Single Threaded (ms)	Dual Threaded (ms)	Ratio
I/O Read:	238.1	265.0	0.90
I/O Read Wait:	23994.8	23189.8	1.03
Preprocessing:	3656.6	3941.1	0.93
Processing:	**21092.5**	**20020.5**	**1.05**
Postprocessing:	1.4	1.5	0.93
Process Wait:	173.9	211.0	0.82
I/O Write:	20.4	23.8	0.86
I/O Write Wait:	24971.1	24204.1	1.03
Total Execution Time:	25027.4	24268.1	1.03

iPad Air

Results from Variable-byte encoding on the iPad Air are displayed in Figure 6.13. Usurprisingly, with an even faster CPU present in the iPad Air, the difference in performance between singel threaded and multithreaded implementations is neglible.

Table 6.7 contains a detailed comparison of a single threaded and dual threaded execution with the largest benchmarked block size. While the increase in performance is missing, statistics show an increase in efficient use of the CPU. Comparing the additional efficiency with corresponding values from benchmarks with the previous generation iPad and 5th generation iPod Touch, this is a unique trait in the iPad Air. However, it may also be due to an anomaly in the particular execution of the benchmark. Table 6.7 also display a raised I/O read value. This may cascade throughout the run and improve the usage of the CPU.

In terms of raw power, the iPad Air is more than twice as fast as the 4th generation iPad for large block sizes.

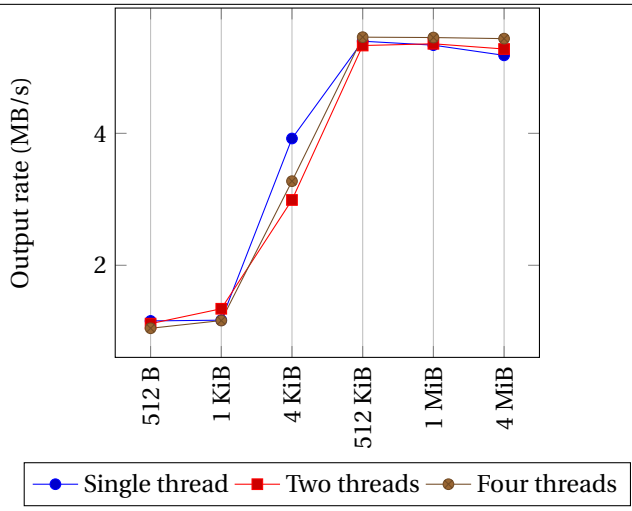

Figure 6.13 Disk-to-disk Variable-byte encoding for configurations of single thread, two threads, and four threads on an iPad Air.

Table 6.7 Detailed comparison of a single threaded Variable-byte encoding run and a dual threaded Variable-byte encoding run on an iPad Air.

	Single Threaded (ms)	Dual Threaded (ms)	Ratio
I/O Read:	**185.3**	**77.8**	**2.38**
I/O Read Wait:	8819.6	8765.0	1.01
Preprocessing:	1560.5	1512.8	1.03
Processing:	**7526.6**	**7512.4**	**1.0**
Postprocessing:	1.0	1.0	1.0
Process Wait:	148.4	35.3	4.20
I/O Write:	13.3	12.9	1.03
I/O Write Wait:	9259.5	9087.1	1.02
Total Execution Time:	9306.6	9136.6	1.02

Summary

Foregoing sections have particularly demonstrated one the features needed to efficiently utilize parallelization: An adequate amount of data to process. Only the 5th generation iPod Touch managed to gain a notable increase in performance when multithreading was applied. An additional observation is how a block size of 512 KiB appear to be a point of saturation. Encoding larger blocks at a time results in little to no gain. In the efficiency plot of the iPod Touch (Figure 6.11), a decrease in efficiency could indeed be spotted for block sizes of 1 MiB and 4 MiB. Table 6.8 demonstrates this is the case for all three devices. The following surveys of Group Varint code and Elias γ code will shed light on whether this is a feature provoked by Variable-byte encoding or the devices themselves.

Table 6.8 Correlation between block size and process wait time on the three tested devices (lower is better).

Block Size	5th Generation iPod Touch (ms)	4th Generation iPad (ms)	iPad Air (ms)
512 B	20666.6	10460.3	10310.3
1 KiB	10650.2	6906.4	8559.1
4 KiB	2649.6	2123.6	1180.8
512 KiB	240.1	48.8	80.3
1 MiB	273.6	88.6	87.8
4 MiB	287.0	173.9	148.4

Figure 6.14 plots the encoding speed of each device for comparison, that is, the processing stage. The significant difference in computational power is appar-

ent between the devices. An additional interesting observation to make is, while the increase in performance is moderate, every device enjoys processing few larger blocks rather than many small. This may be due to the cost of allocating memory, *i.e.* the cost allocating an area is not linearly correspondant to the size of the area. This is particularly visible in the iPad Air's slope.

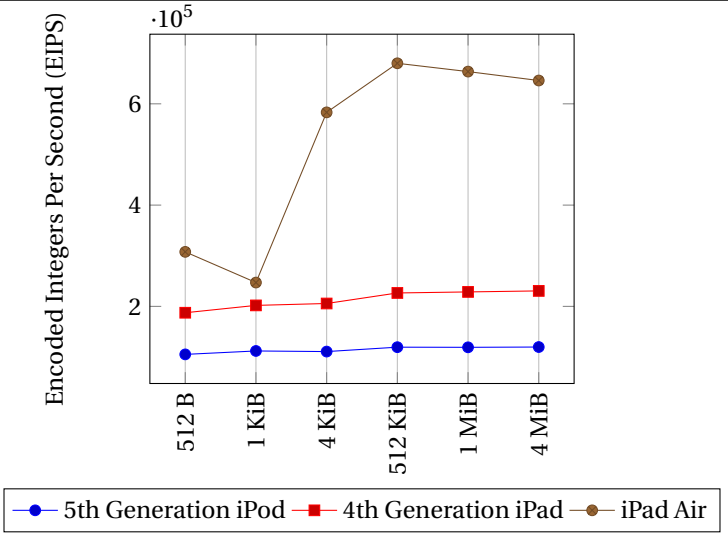

Figure 6.14 Variable-byte encoding performance of the three devices tested.

6.4.2 Decoding

Variable-byte decoding proved initially to be the fastest decoding scheme. The following subsections will more thoroughly investigate how decoding behaves on the Apple devices surveyed.

iPod Touch, 5th Generation

Figure 6.15 presents the disk-to-disk performance of Variable-byte decoding with a single thread implementation, as well as implementations with two threads and four threads. Multithreading is not applied to the actual decoding, but rather during postprocessing when the array holding decoded values is iterated through and converted to a continuous stream of bytes.

Applying multithreading during postprocessing pays dividence for the smallest block sizes, but one does not witness a significant improvement until a block of 512 KiB or larger is in use. Figure 6.15 illustrates a speedup of around 1.62 from a single thread execution to applying two threads for a block size of 1 MiB.

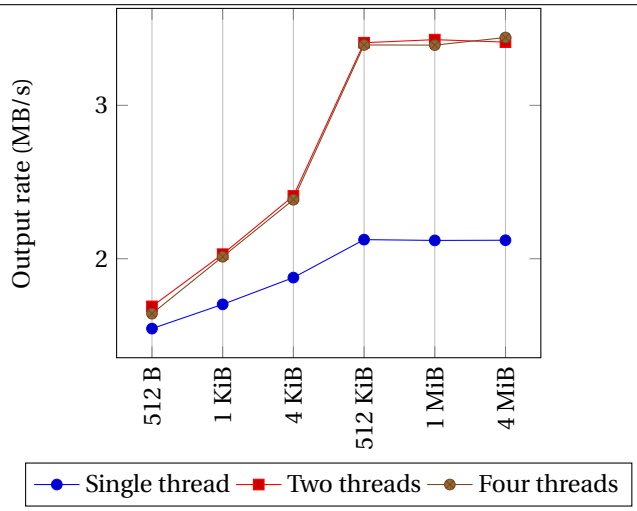

Figure 6.15 Variable-byte decoding performance on a 5th generation iPod Touch.

iPad, 4th Generation

It can be observed in Figure 6.16 how the trend from the 5th generation iPod benchmark continues. The performance benefit harvested during multithreaded decoding may be due an inefficient conversion step between an array of integers to a continuous series of bytes. If this step is slow, separating the workload between more than one thread may pay dividence. One must also take into account the amount of data being processed during postprocessing after data has been decoded. On average during Variable-byte decoding, a block of data will expand to over nine times its read size. Indeed, a block of encoded data, 4 MiB in size, will contain near the complete decoded postings list. As established earlier, for multithreading to be beneficial, one must supply enough data. Table 6.9 displays a detailed comparison of a decoding benchmark where the block size was set at 1024 MiB.

Applying Hill and Marty's formulae for a symmetric, multi-core processor, and assuming 90 % of the computation during a serial execution is within the parallelizable postprocessing stage, one achieves the following values for a theoretical speedup [24]:

- Two threads: $Speedup_{symmetric}(n = 2, r = 1, p = 0.9) = \frac{1}{1-0.90+\frac{0.90}{2}} = 1.81$
- Four threads: $Speedup_{symmetric}(n = 4, r = 1, p = 0.9) = \frac{1}{1-0.90+\frac{0.90}{4}} = 3.08$

It is apparent that for four threads, the practical speedup is not comparable to that of the theoretical. However, for two threads, a practical speedup of 1.65 is decent. This is also reflected in the total execution time.

In contrast to the results from the encoding survey, the increase performance during single threaded decoding is moderate in regards to block size. This is most probably due to the size of the total amount of encoded data. That is, the sheer number of reads required to decode a file of roughly 4.5 MiB in size is not high enough to have quite the impact it has during encoding.

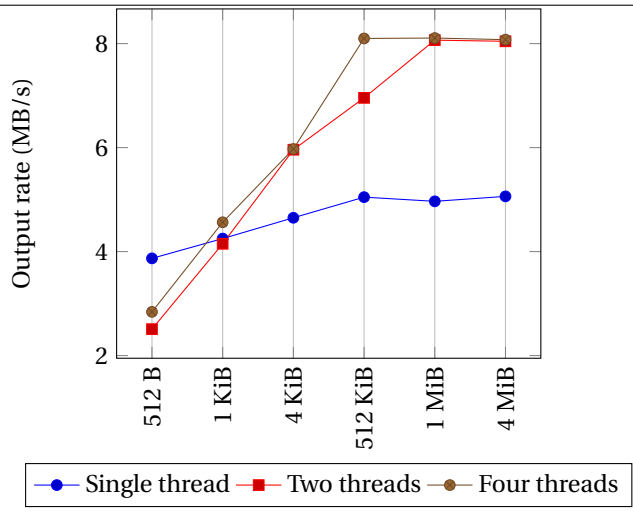

Figure 6.16 Variable-byte decoding performance on a 4th generation iPad.

Table 6.9 Detailed comparison of a single threaded Variable-byte decoding run and a dual threaded Variable-byte decoding run on a 4th generation iPad 4.

	Single Threaded (ms)	Dual Threaded (ms)	Ratio
I/O Read:	29.8	29.8	1.0
I/O Read Wait:	8159.0	4943.9	1.65
Preprocessing:	0.0	0.0	1.0
Processing:	790.1	843.1	0.94
Postprocessing:	**8655.7**	**4908.3**	**1.76**
Process Wait:	31.5	25.3	1.25
I/O Write:	67.7	69.5	0.97
I/O Write Wait:	9428.5	5728.5	1.65
Total Execution Time:	9708.4	5977.8	1.62

iPad Air

Figure 6.17 demonstrates how the iPad Air behaves in the same fashion the foregoing device. Applying multithreading for a lower block sizes is severly penalized. However, the slopes representing two threads and four threads are steep and surpass serial execution for a block size of 4 KiB. As during encoding, applying a block size larger than 512 KiB gives no significant benefit. Table 6.10 compares the three

executions of 512 KiB, 1 MiB, and 4 MiB. The difference may be small variations in the benchmark environment, however, it is interesting how the total time spent reading data increases for block sizes larger than 512 KiB. In addition, while decoding enjoys a smaller block size, the added data to process among threads for larger block sizes during postprocessing keeps the performance on an almost equal level.

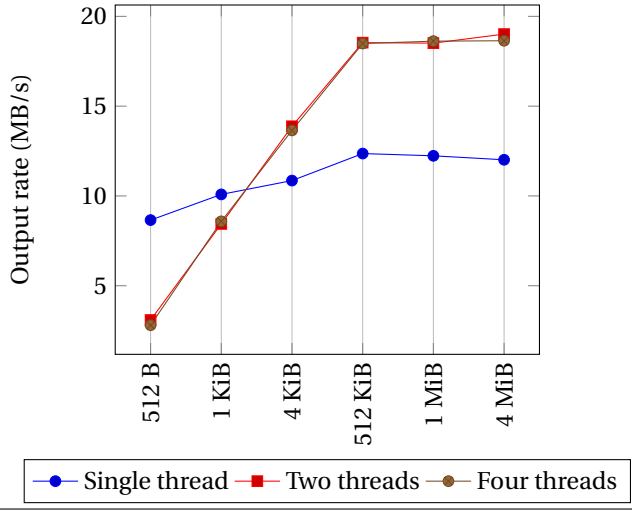

Figure 6.17 Variable-byte decoding performance on an iPad Air.

Table 6.10 Detailed statistics of an execution with two threads and block sizes of 512 KiB, 1 MiB, and 4 MiB.

	512 KiB (ms)	1 MiB (ms)	4 MiB (ms)
I/O Read:	8.9	9.0	10.9
I/O Read Wait:	2453.5	2143.1	2073.7
Preprocessing:	0.0	0.0	0.0
Processing:	349.0	347.0	370.9
Postprocessing:	2186.9	2146.3	2058.0
Process Wait:	10.0	13.4	7.7
I/O Write:	44.4	123.8	262.0
I/O Write Wait:	2501.8	2386.2	2184.4
Total Execution Time:	2601.6	2605.4	2536.5

Summary

A summary of the decoding performance of the three devices benchmarked is displayed in Figure 6.18. It is apparent that Apple's A7 SoC is a significant improvement from prior generations.

Each device plot demonstrate block size having a large role in improving decoding performance. The iPad Air displays an improvement of over 50 % from 512 bytes to 512 KiB, while the 4th generation iPad and the 5th generation iPod improve over 50 % and 30 % respectively. A point of saturation is reached at 512 KiB.

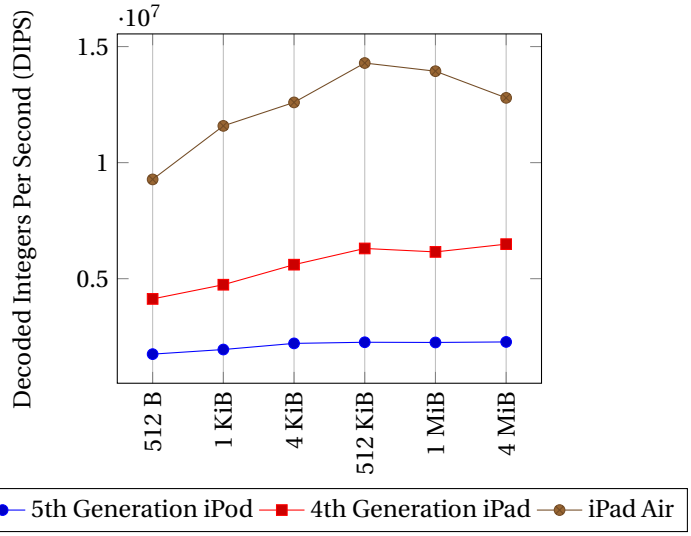

Figure 6.18 A comparison of the decoding performance of the three devices tested.

6.5 Group Varint Code

Group Varint coding proved to be the fastest encoding scheme, and also on par with Variable-byte coding during decoding.

6.5.1 Encoding

Variable-byte encoding struggled with making efficient use of multithreading due to the overhead associated with dispatching threads. With Group Varint encoding being a faster scheme, multithreading may not provide significant gain here either.

iPod, 5th Generation

Figure 6.19 displays signs of the initial assumption being wrong. For block sizes 512 KiB – 4 MiB, both applying two threads and four threads is beneficial to perfomance. While the improvement is modest, the plot illustrates potential for larger data sets.

One can also observe a steep improvement when increasing the block size. However, as previosly demonstrated during Variable-byte encoding, block sizes larger than 512 KiB achieve no gain in performance. Figure 6.19 illustrate a slight disfavour of applying block sizes of 1 MiB or 4 MiB.

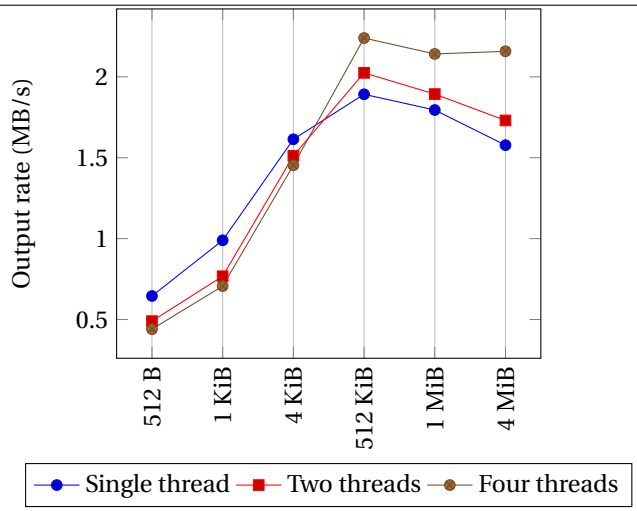

Figure 6.19 Encoding performance of Group Varint on a 5th generation iPod.

iPad, 4th Generation

Illustrated in Figure 6.20, similar trends as above are present when Group Varint encoding is applied with a 4th generation iPad. However, the optimistic results in regards to multithreading is not present. A detailed investigation, displayed in Table 6.11, reveals the multithreaded execution is slower during the parallelized part of the implementation. It is not known if this is a behaviour enforced by Group Varint encoding, the device itself, or an anomoly in the benchmark. The 5th generation iPod Touch also suffered a performance drop for a block size of 1 MiB, however, not as significant as this.

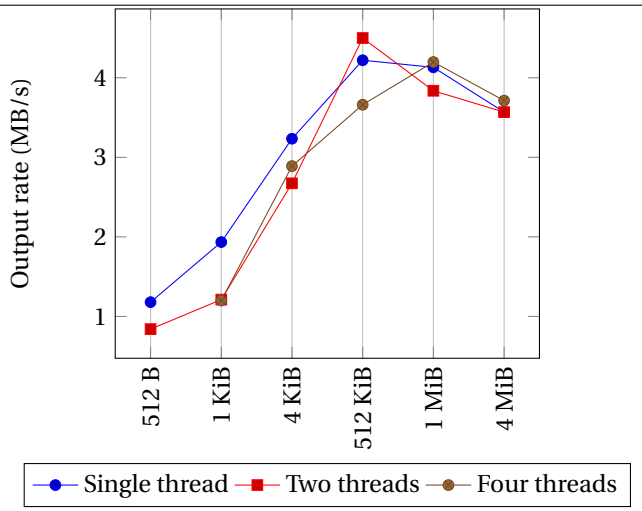

Figure 6.20 Encoding performance of Group Varint on a 4th generation iPad.

Table 6.11 Detailed statistics of a single threaded and dual threaded execution of Group Varint encoding on a 4th generation iPad 4, with 1 MiB block size.

	Single Threaded (ms)	Dual Threaded
I/O Read:	197.7	182.6
I/O Read Wait:	11395.3	12223.4
Preprocessing:	4486.8	4647.6
Processing:	**6946.5**	**7592.5**
Postprocessing:	0.0	0.0
Process Wait:	104.3	85.5
I/O Write:	101.9	34.4
I/O Write Wait:	11522.9	12406.8
Total Execution Time:	11672.4	12567.7

iPad Air

Figure 6.21 demonstrates a feature similar to the one just witnessed for the 4th generation iPad. When executing Group Varint encode with a block size of 1 MiB, the performance is severly hampered when two threads are applied to the computation. Both the single threaded and the four threaded results display lower values for this particular block size, however, not a difference as large as during the previ-

ous device' benchmark.

The result from the single threaded execution with a 4 MiB block size has most likely been disturbed during execution in some way, and should be rendered void.

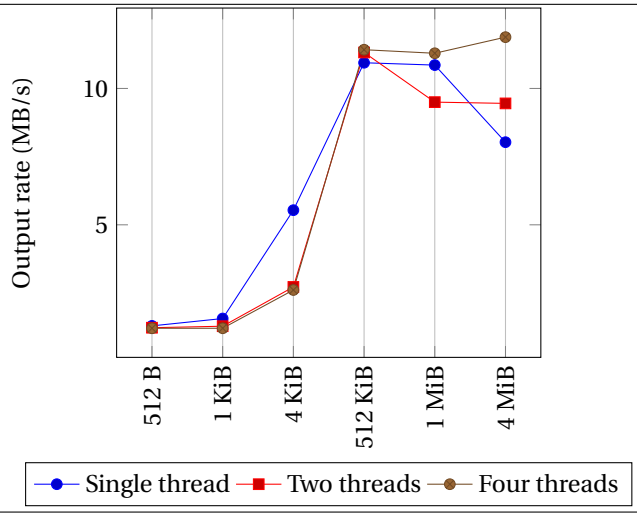

Figure 6.21 Encoding performance of Group Varint on an iPad Air.

Summary

Figure 6.22 contains a direct comparison of Group Varint encoding executed on the three devices.

The results are comparable to the ones displayed earlier from Variable-byte encoding (Figure 6.14). Particularly, the sharp increase in performance at a block size of 4 KiB during the iPad Air benchmarks is present here as well. Additionally, each slope is moderate in change, although interestingly, both the 4th generation iPad and 5th generation iPod degrade in performance as the block size increases. This is a surprising result, as the allocation of several small blocks of data proved was assumed to be more expensive than allocating few larger ones earlier (Section 6.4.1). The relationship between the cost of encoding and allocating an area of memory appears to have shifted during Group Varint encoding, slightly favouring small block sizes.

The sudden drop in performance for a block size of 4 MiB during execution on the iPad Air is thought to be due to a defective benchmark run.

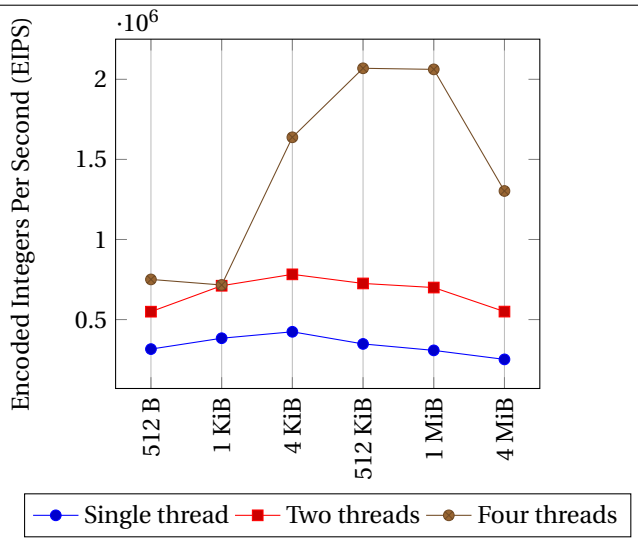

Figure 6.22 Comparison of Group Varint encoding performance on the devices tested.

6.5.2 Decoding

In Section 6.1, Group Varint decoding performed on par with Variable-byte decoding. As such, one would expect similar results in the following sections.

iPod, 5th Generation

Decoding using Group Varint promises a similar parallel speedup to the one achieved during Variable-byte decoding. Figure 6.23 illustrates a speedup of 1.61 for a block size of 512 KiB. This is expected as multithreading is applied to postprocessing and not the decoding, *i.e.* the processing stage, itself. Overall, the plot has features similar to previously presented results: A stabilized perfomance for a block size of 512 KiB and larger, as well as little additional gain in applying four threads to enhance the rate of output.

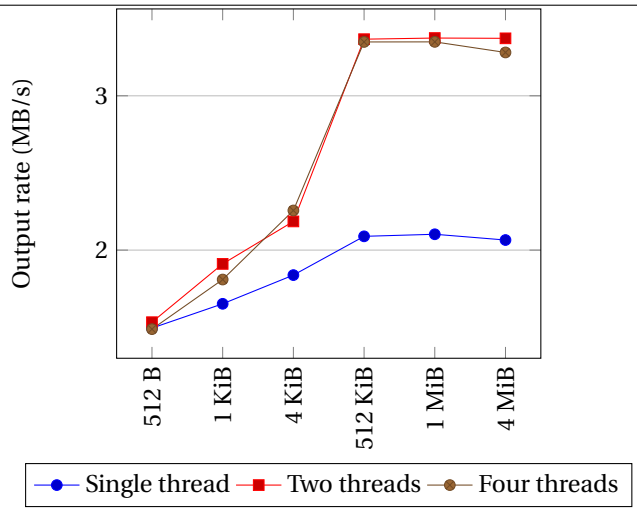

Figure 6.23 Decoding performance of Group Varint on a 5th generation iPod.

iPad, 4th Generation

Compared to Group Varint decoding using the 5th generation iPod, Figure 6.24 show multithreading needing additional data before improving the performance. While the 5th generation iPod surpassed serial execution at a block size of 1 KiB, the 4th generation iPad is trailing until 4 KiB of data is read per block. This is most probably due to the increased computational power available in the Apple A6 SoC. As such, more data must be supplied each thread to compensate the penalty of dispatching additional threads.

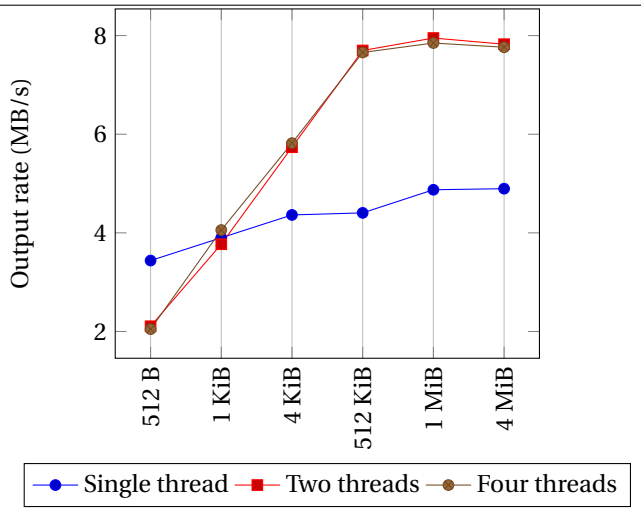

Figure 6.24 Decoding performance of Group Varint on a 4th generation iPad.

iPad Air

Figure 6.25 repeat the trend witnessed with the iPad Air during Variable-byte decoding (Figure 6.17), and is almost an exact replica of Group Varint decoding using the 4th generation iPad. The slope representing the single threaded execution has a moderate incline, while both multithreaded executions have a significant increase until reaching 512 KiB. At such a block size, multithreaded performance is seemingly exhausted.

While the 4th generation iPad and the iPad Air have similar performance patterns, the output rate from the latter device is more than double that of the former.

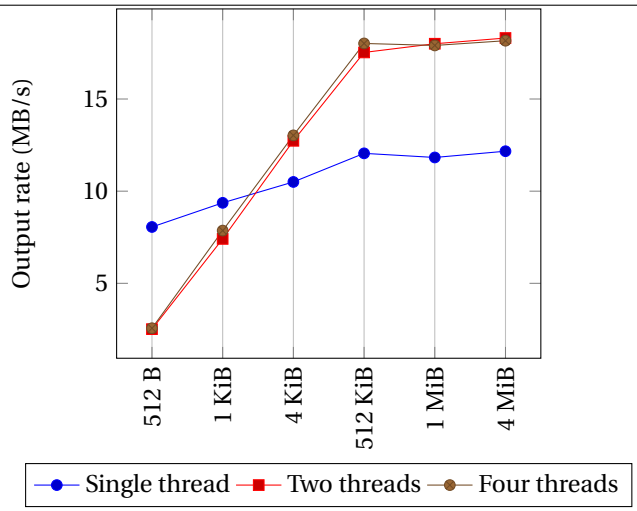

Figure 6.25 Decoding performance of Group Varint on an iPad Air.

Summary

Decoding results from all benchmarked devices are compared in Figure 6.26. In correspondence with the Variable-byte decoding summary, both the 4th generation iPad and the 5th generation iPod appear to have moderately increasing slopes. The former hold a difference in performance of almost 60 %, while the latter have a difference of about 35 %. The iPad Air differ almost 80 % between the lowest and highest measured value, leaving one to conclude that the block size plays a significant role also when decoding data present in memory.

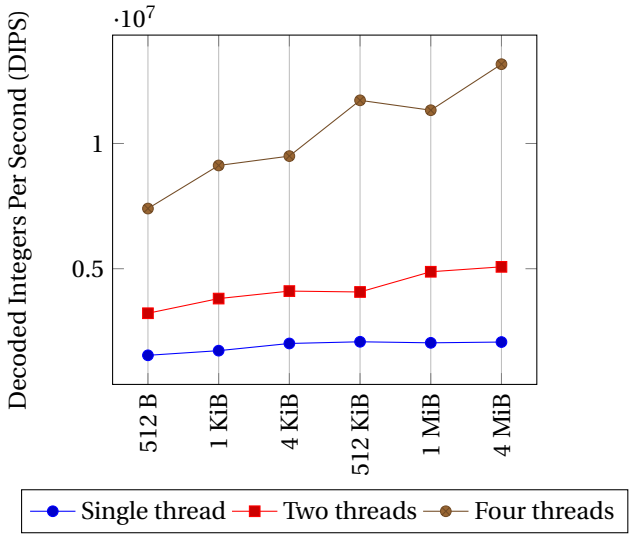

Figure 6.26 Comparison of Group Varint decoding performance on the devices tested.

6.6 Elias γ Code

While Elias γ code proved superior in compression ratio, it lacked an edge in perfomance both during encoding and decoding.

6.6.1 Encoding

Elias γ code is a bit-oriented coding scheme, and as such, extra measures must be taken when the result is to be written to permanent storage. During encoding, this results in an extra operation during postprocessing when the in-memory representation of the encoded data is converted from an array `BOOL` values to bits to be written.

iPod, 5th Generation

As displayed in Figure 6.27, encoding using Elias γ has a similar pattern as encoding using Variable-byte and Group Varint. An interesting trait is the significant decline in performance for a block size larger than 512 KiB. Particularly noticable is the

result after applying a block size of 4 MiB for a dual threaded execution. Several reexecutions with these distinct properties were performed, however, the poor result proved to be consistent. From Table 6.12, it is apparent that the additional time is spent during the processing stage, *i.e.* when the data is encoded. A tempting assumption to make is that the degrade in performance is due to an excessive amount of data to convert, however, this process is executed during postprocessing. It is not known what causes such a result, if it is a trait present in Elias γ encoding, the implementation, or a scheduling issue in the operating system.

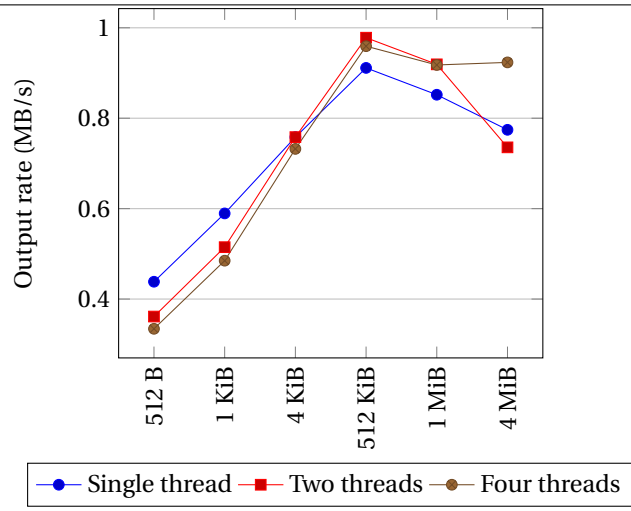

Figure 6.27 Encoding performance of Elias γ on a 5th generation iPod Touch.

Table 6.12 Detailed statistics of an execution on a 5th generation iPod with two threads and block sizes of 512 KiB, 1 MiB, and 4 MiB.

	512 KiB (ms)	1 MiB (ms)	4 MiB (ms)
I/O Read:	477.1	404.5	385.2
I/O Read Wait:	52144.3	51769.3	63300.8
Preprocessing:	9435.4	9417.9	9170.1
Processing:	39442.6	39090.8	52456.3
Postprocessing:	3509.9	3502.4	3490.1
Process Wait:	241.6	268.1	270.7
I/O Write:	41.8	38.5	30.9
I/O Write Wait:	52804.3	52333.8	65477.5
Total Execution Time:	52915.4	52461.2	65558.1

iPad, 4th Generation

While not as significant, executing on the 4th generation iPad (Figure **??**) present a similar degrade in performance as the 5th generation iPod for a setup of two threads and a block size of 4 MiB. Both from Figure 6.28 and 6.27, it can be seen how the results favour a block size of 512 KiB. Being present in both devices, this may point to involuntairy favouritism in the implementation or a trait in the operating system.

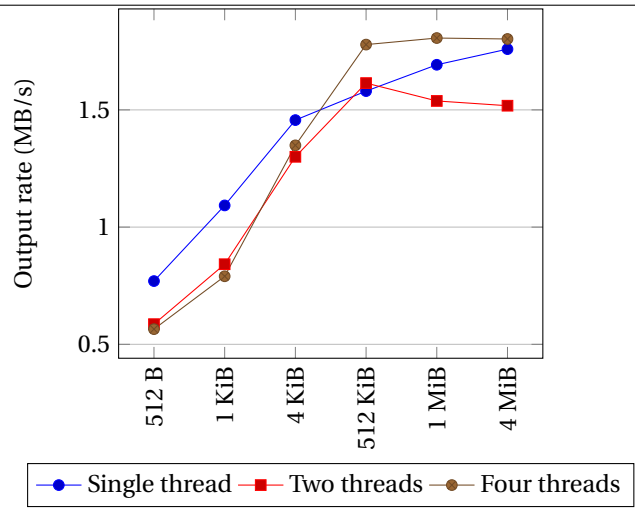

Figure 6.28 Encoding performance of Elias γ code on a 4th generation iPad.

iPad Air

Figure 6.29 maintains the trend presented in the two preceding paragraphs, the difference being the degrade in performance is present in all three threading variants. In addition, results are even more in favour of a block size of 512 KiB.

An interesting feature is the increase in performance from a block size of 1 KiB to that of 4 KiB. Table 6.13 compares serial executions with block sizes set to 512 B, 1 KiB, and 4 KiB. For a block size of 4 KiB, the time spent on processing is decreased with over 100 %. Consecutive executions displayed the same trait, however, the reason for the result is currently not identified.

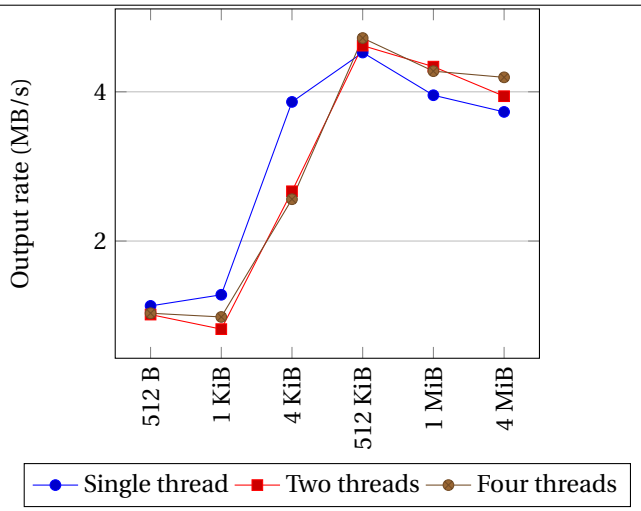

Figure 6.29 Encoding performance of Elias γ code on an iPad Air.

Table 6.13 Detailed statistics of a serial execution on an iPad Air with block sizes of 512 B, 1 KiB, and 4 KiB.

	512 B (ms)	1 KiB (ms)	4 KiB (ms)
I/O Read:	3828.7	2707.2	501.4
I/O Read Wait:	33674.2	31758.5	11482.3
Preprocessing:	7711.1	6221.7	1851.3
Processing:	**14033.3**	**16783.3**	**7859.3**
Postprocessing:	3254.7	3034.5	1011.7
Process Wait:	10388.3	7053.6	1055.2
I/O Write:	2682.2	1698.4	271.4
I/O Write Wait:	31234.8	31189.4	11495.1
Total Execution Time:	42639.5	37676.0	12477.8

Summary

Initially, Figure 6.30 may appear to contain invalid results. However, comparing the slopes of Elias γ encoding with those of Variable-byte encoding (Figure 6.14) and Group Varint encoding (Figure 6.22) reveals a similar pattern to be present. The significant gain in performance iPad Air achieves for a 4 KiB block size is a particularly noticable trait. As are the stable results recorded during tests with the

4th generation iPad and the 5th generation iPod.

It is not clear why the iPad Air looses momentum and declines in performance for block sizes 1 MiB and 4 MiB, however, the result is corresponding with what was presented in the individual encoding result graph (Figure 6.29).

While Elias γ encoding was measured as slightly faster in beginning of this chapter (Section 6.1), the values presented here are the lowest yet.

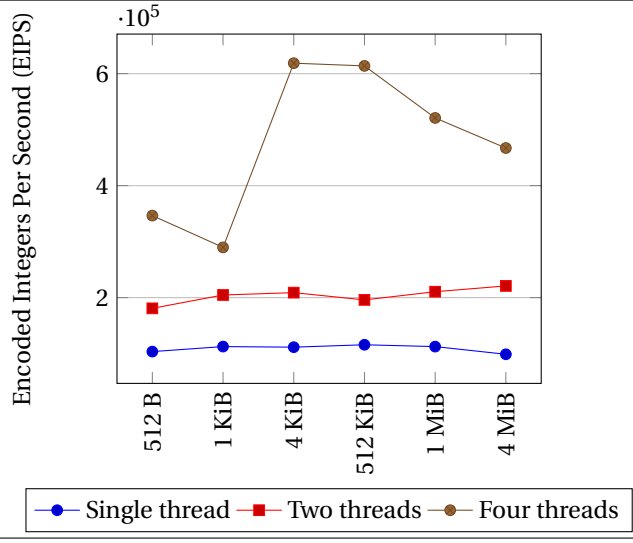

Figure 6.30 Comparison of Elias γ encoding performance on the devices tested.

6.6.2 Decoding

As mentioned in the introduction of Section 6.6.1, Elias γ encoding requires an additional operation when encoded data is to be written. This is relevant during decoding as well, as individual bits must be expanded to `BOOL` values and stored as an array in memory. The expansion is performed during the processing stage before decoding takes place.

iPod, 5th Generation

Figure 6.31 paints a picture of the performance, similar to the ones produced by Variable-byte decoding (Figure 6.15) and Group Varint decoding (Figure **??**): Multithreading is beneficial already at block size of 1 KiB, with the gain in perfomance saturated at a block size of 512 KiB.

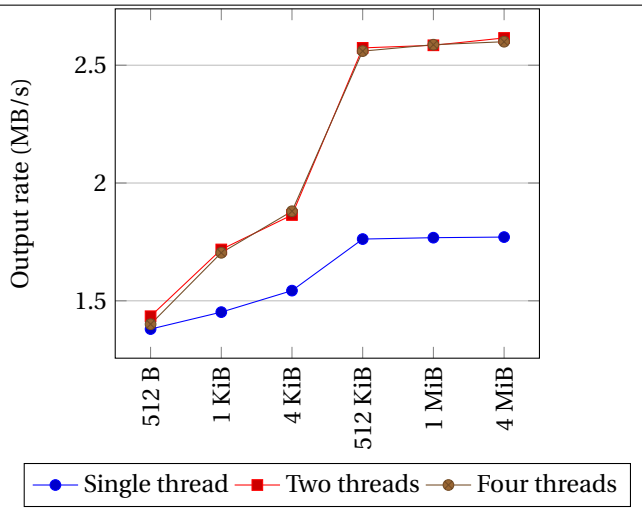

Figure 6.31 Decoding performance of Elias γ code on a 5th generation iPod Touch.

iPad, 4th Generation

In contrast to results from Variable-byte decoding (Figure 6.16) and Group Varint decoding (Figure 6.24), Figure 6.32 illustrates the 4th generation iPad 4 benefitting from multithreading already at a block size of 1 KiB. However, the speedup is a meager 14 %. Increasing the block size, the most significant increase is seen at 4 MiB, processed using four threads, with an improvement of about 50 %. As multithreading is applied during postprocessing, this result is low compared to those achieved during executions presented prior. For instance, the 4th generation iPad achieved a speedup of over 60 % during Variable-byte decoding. However, with an extra postprocessing operation, less of the total time is spent in the parallelized part of the implementation, which results in a lower, total achievable speedup. Indeed, Table 6.14 displays the speedup achieved during decoding is 1.76, however, the speedup reported by the total execution is lowered due to an increase in the amount of time spent in the serial processing stage.

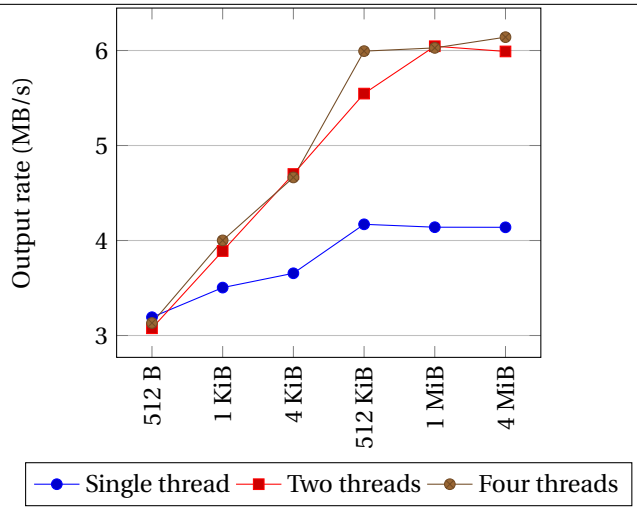

Figure 6.32 Decoding performance of Elias γ code on a 4th generation iPad.

Table 6.14 Detailed comparison of a single threaded Elias γ decoding and a four threaded Elias γ decoding with a block size of 4 MiB on a 4th generation iPad.

	Single Threaded (ms)	Four Threaded (ms)	Ratio
I/O Read:	35.5	37.4	0.95
I/O Read Wait:	10494.9	6931.8	1.51
Preprocessing:	0.0	0.0	1.0
Processing:	2870.0	2748.5	1.04
Postprocessing:	**8575.1**	**4864.1**	**1.76**
Process Wait:	34.4	38.2	0.9
I/O Write:	321.3	316.5	1.02
I/O Write Wait:	11170.1	7349.0	1.52
Total Execution Time:	11650.1	7853.2	1.48

iPad Air

As displayed in Figure 6.33, the iPad Air follows a pattern similar to that of the 4th generation iPad: Applying multithreading does achieve an increase in performance, however, not as significant as during Variable-byte decoding and Group Varint decoding. The execution sequence is equal for all devices, as such, the lowered parallel speedup is due to an increased presence in serial parts of the execution.

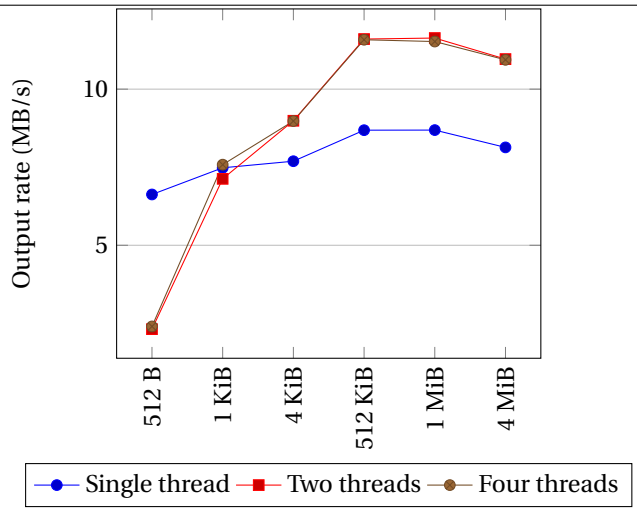

Figure 6.33 Decoding performance of Elias γ code on an iPad Air.

Summary

Elias γ decoding results are summarized and available for comparison in Figure **??**. The three graphs reflect previous results of Variable-byte decoding (Figure 6.18) and Group Varint decoding (Figure 6.26). As expected from Section 6.1, Elias γ code is significantly slower than the other coding schemes during decoding. Interestingly, the iPad Air is pictured as dominant performance-wise as earlier. Instead, the 4th generation iPad appears to enjoy decoding using Elias γ.

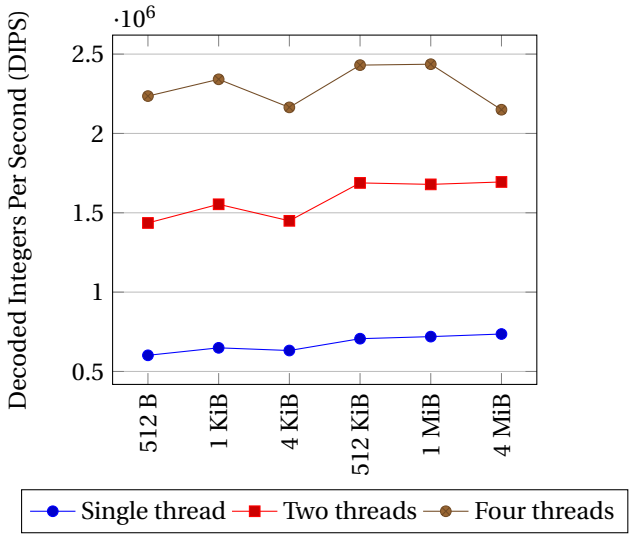

Figure 6.34 Comparison of Elias γ decoding performance on the devices tested.

6.7 Summary

The preceding sections have displayed detailed statistics of all coding schemes, executed on three different devices with varying parameters. Extracting common denominators from these executions is challenging, however, some traits are present in all executions.

6.7.1 Device Performance

The devices used during testing represent three generations from the Apple A-series of System on Chips (SoC), the A5, A6X, and the A7. Looking at the specifications of these chips within each device, the difference in performance is surprising.

The Apple A6X present within the 4th generation iPad used during testing is running at the same clock frequency as the Apple A7 within the iPad Air. The former is coupled with one GiB of Low Powered DDR2 (LPDDR2) memory, while the latter is interfaced with one GiB of LPDDR3 memory. In addition, the Apple Cyclone processor within the Apple A7 has double the level one cache, and an additional four MiB core global level three cache. Despite appearing near equal in specifications,

the iPad Air managed to perform more than double that of the 4th generation iPad on several occasions.

A surprising addition is the disk performance all three devices achieve during sequential reading. However, there is a significant difference comparing random access reading with sequential for smaller block sizes. It is important to remember that random access reading is an issue with Negated AND (NAND) based flash drives. As such, one should strive to read large blocks of data.

As a last, but important note is the overhead associated with employing Objective-C objects in performance critical applications. It is apparent from Section- 6.3 how wrapping during repeated tasks incur a significant performance penalty.

6.7.2 Optimum Parameters

Large variations occured during executions. However, should one pick a set of parameters in an attempt to maximize performance and cover most use cases, a block size of 512 KiB and execution using two threads appear achieve the best results. In each test performed, performance has been steadily rising until meeting a point of saturation at a 512 KiB block size. Preceding block sizes have either had a minor increase in performance or begun degrading. In addition, 512 KiB of data is sufficient to take advantage of multithreading.

All three devices tested are dual core. Tests were performed with both two threads and four threads sharing the load. Seldom did four threads gain a significant upper hand in comparison to a similar two-threaded execution. This may be due to an insufficient amount of data being processed to properly benefit from four threads, or it may be due to poor scheduling in the operating system. With the results presented earlier as a basis, two threads appear to be the better fit.

An additional note is worth making concerning multithreading and lower block sizes. During encoding, a block size of less than 512 KiB was not able to benefit from multithreading, leaving the single threaded execution as most performant alternative. Decoding benchmarks displayed similar results, but for a few executions. As such, should one wish to operate with a block size in area of 4 KiB or less, applying multithreading is discouraged.

6.8 Critique

Areas of the presented results are subject to some critique. For one, small variations within distinct executions may stack up and produce a seemingly significant variation between two results. Benchmarks are timed in a per block fashion. That is, the

time each block spends in for instance the processing stage or write stage is summarized with all timings gathered from overall blocks to produce the final result. This makes an execution vulnerable to outside influence, if for instance, a background process in operating system is executed and impact the resources available to the benchmark. On the other hand, results may not be as fine grained as one would wish for. An example is the processing stage. The timing may be affected by the time spent dividing resources among threads, allocating memory, and so forth, and not only the time spent encoding or decoding data.

A second point to make is the nature of the data set. Being one large postings list, it simulates a term found in almost all documents in a document storage system. One could compare the data set to the postings list of the word "the". This results in a postings list where calculated Δ-values are small, which in turn results in a high compression ratio and faster decoding. This is particularly the case for Variable-byte decoding and Elias γ decoding, where values are read byte-by-byte and bit-by-bit, respectively.

Additionally, encoding and decoding for midrange block sizes, *i.e.* 16 KiB – 64 KiB, were not performed. This has resulted in somewhat polarized results: If one is not privvy to or not in need of reading large chunks of data, use a block size of 4 KiB and do not apply multithreading. In the contrary use case, apply a block size of 512 KiB and divide the workload among two threads. With postings list varying significantly in size depending on the term, an additional inspection of midrange block sizes may be interesting.

Finally, one might inquire encoding results for block sizes of 16 MiB and larger. Previous sections pointed out the need for additional data to process to better make us of multithreading. However, the gain of dividing the workload among several threads appeared to be saturated already at 512 KiB, leaving one to conclude larger block sizes may not provide additional insight.

Chapter 7

Conclusion and Future Work

This thesis has focused on providing insight in the use of handheld devices for an uncommon and narrow use case: the encoding and decoding of postings list in inverted indexes. The focus has been to give an overview of suitable coding schemes, their properties, and how they perform on different devices under distinct conditions. To provide a broad basis of comparison, three different ways of coding were selected: *Variable-byte coding*, *Group Varint coding*, and *Elias γ coding*, with the latter being bit-oriented and the two former being byte-oriented. Benchmarks were applied to three different devices of Apple: A 5th generation iPod, a 4th generation iPad, and an iPad Air, each device sporting a distinct version of an Apple A-series SoC. Additionally, in the process of selecting parameters to use during the main tests, this thesis has identified the sequential and random access read performance of each device.

A fictional postings list was generated according to Zipf's law of term frequency, resulting in a data set of about 45 MiB in size and containing over 48 million postings.

We have developed two iOS applications in order to perform the executions in native environments: the "SSDPerformanceMapping" application, measuring sequential and random access reading of the flash memory present in the device, and the "PostingListApp", measuring encoding and decoding of the on-device postings list with the different schemes implemented. As all three devices tested are dual core, additional benchmarks were performed in attempt to utilize the capabilities presented in a multi-core CPU. As such, the latter application also has the ability to set the number of threads to use during execution. Postings list benchmarks were applied using one, two and four threads during both encoding and decoding.

Flash memory performance tests found the read performance present in Apple's *i*-series of devices impressive. For instance, the iPad Air peaked at over 275 MiB per second for sequential reading and about 267 MiB per second during random access reading. Overall results from these initial benchmarks selected the block sizes to apply in further surveying of postings list coding:

- 512 B
- 1 KiB
- 4 KiB
- 512 KiB
- 1 MiB
- 4 MiB

Blockwise reading of unencoded and encoded data presented problems not found described in previous litterature. During encoding, a block of read data may split the last read integer. Methods were developed to detect such an event, cache the split integer, and combine data on the consecutive reading. When reading encoded data, a similary situation may occur. Read data may not contain the required to bytes to perform a complete decode. Either, data is attempted decoded, but fails and data is discarded, or the decode produces the wrong result. We have developed methods per coding scheme to handle such situations.

With three devices benchmarked, three distinct coding schemes to survey in both encoding and decoding, three threading configurations, and an additional six different block sizes, the number of experiments performed counts to over 300. Extracting common denominators from these executions have been challenging, however, some traits were present in several results:

- Coding schemes vary significantly in terms of speed and compression ratio. Being bit-oriented, Elias γ coding achieves the better compression ratio, but falls short during both encoding and decoding. Variable-byte coding provides a middle-ground between compression ratio and performance. However, performance may degrade for sparse postings lists as decoding is subject to branch mispredictions. Group Varint eliminates Variable-byte's branch mispredictions, but suffers in compression ratio. It is, however, the fastest during both encoding and decoding. Group Varint incurs additional complexity in during if four is not a factor of the postings list's length. This

means one must either append data to extend the postings list or apply an alternative encoding for trailing values.

- For multithreading to be beneficial, one is required to supply sufficient data to process. There is an inherit overhead associated with dispatching additional threads. Results in this thesis proved this penalty to be quite expensive, usually not providing multithreading with a performance edge until the block size reached 512 KiB. As such, in the general case, a single threaded implementation is sufficient for block sizes in the area of 4 KiB. In addition, two threads proved to be sufficient in all experiments performed.

- Performance is strongly correlated with the block size. Although, a large block size is not synonymous with higher performance. Results indicated an increase in permance as the block size augmented. However, reaching a block size of 512 KiB, performance stalled. A minor decline could be witnessed in some experiments when the block size reached 1 MiB or 4 MiB. This correlation is mainly due flash memory appreciating reading few large blocks in contrast to several small ones. Isolating the values of encoding or decoding displayed a more modest relationship between block size and performance. Still, the difference between decoding a small black compared to that of a large block was over 50 % when measured on an iPad Air.

- One cannot determine the potential in devices by looking at the specifications. The SoCs present in benchmarked devices are similar specification-wise, but differ significantly in terms of performance. The difference between the Apple A6X and the Apple A7 is particularly surprising.

7.1 Future Work

To our knowledge, this is the first thesis investigating the potential for encoding and decoding postings lists on handheld devices. As such, the potential for future work is significant.

- One should strive to optimize current encoding and decoding implementations. Currently, bitwise operations replace modulo, multiplication, and division operations where available, however, as this is the first release of the source code, additional areas may not be sufficiently optimized. In addition, the current implementation makes use of prefix sum during decoding. This is a textbook parallelization problem, with the potential of contributing with additional speedup during decoding.

- Further research into ARM NEON, ARM's SIMD extensions should be performed. Stepanov *et al.* applied SIMD instructions to Group Varint using instructions (`PSHUFB`) lacking equivalents in the NEON instruction set [55]. However, investigations should be made to survey if combinations of instructions provide equal results. In addition, further revisions of NEON may provide what is required.

- More accurate timing should be provided, particularly during the processing stage of the pipeline, *i.e.* encoding and decoding. Currently, memory allocations and thread preparations are recorded. Such operations should not pollute the timing when one attempts to measure the performance of the coding scheme.

- Research should be made into more exotic data sets, that is, data sets with more sparsely populated and shorter postings lists. These are believed to particularly have an effect on the compression ratio, but also the performance of Variable-byte code and Elias γ code.

- One should look into taking this thesis one step further towards a mobile search engine. An opportunity is to investigate the viability of encoding or decoding several terms in parallel or the merging of postings from two or more postings lists.

- A survey of the memory consumption is recommended. During the creation of this thesis, memory consumption proved to be a challenge on several occasions. iOS is particularly strict in terms of the rate memory is allocated, in addition to the amount of memory currently in use. Memory is particularly an issue during decoding, as encoded data is initially small, but quickly deflated as decoding progresses. If one is to decode several terms in parallel, memory consumption will become an issue for terms occuring in an abundance of documents, *i.e.* having a very long postings list.

Bibliography

[1] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM. 18, 19, 67

[2] Apple. Programming with Objective-C. Web Site, 2012. URL https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html. Accessed October 31, 2013. 11, 12, 13

[3] Apple. Cocoa - OS X Technology Overview - Apple Developer. Web Site, 2013. URL https://developer.apple.com/technologies/mac/cocoa.html. Accessed November 2, 2013. 14, 15

[4] Apple. Cocoa Touch - iOS Technology Overview - Apple Developer. Web Site, 2013. URL https://developer.apple.com/technologies/ios/cocoa-touch.html. Accessed November 2, 2013. 14, 15

[5] Apple. Foundation Framework Reference. Web Site, 2013. URL https://developer.apple.com/library/mac/documentation/cocoa/reference/foundation/objc_classic/_index.html. Accessed November 2, 2013. 14

[6] Ahmed A Aqrawi and Anne C Elster. Bandwidth reduction through multithreaded compression of seismic images. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 1730–1739. IEEE, 2011. 31

[7] ARM. Cortex-15 processor - arm, 2013. URL http://www.arm.com/products/processors/cortex-a/cortex-a15.php. Accessed December 16, 2013. 6

[8] ARM. Cortex-a8 processor - arm, 2013. URL http://www.arm.com/products/processors/cortex-a/cortex-a8.php. Accessed December 16, 2013.

[9] ARM. Cortex-a9 processor - arm, 2013. URL http://www.arm.com/products/processors/cortex-a/cortex-a9.php. Accessed December 16, 2013. 6

[10] ARM. Processor Licensees - ARM. Web Site, 2013. URL http://www.arm.com/products/processors/licensees.php. Accessed October 8, 2013. 6

[11] Atmel. *NAND Flash Support in AT91SAM9 Microcontrollers*, 2006. URL http://www.atmel.com/dyn/resources/prod_documents/doc6255.pdf. Accessed December 11, 2013. 63

[12] Stefan Büttcher, Charles Clarke, and Gordon V Cormack. *Information retrieval: Implementing and evaluating search engines*. The MIT Press, 2010. 27, 29, 30

[13] Stefan Büttcher, Charles Clarke, and Gordon V Cormack. Information retrieval: Implementing and evaluating search engines - addenda for chapter 6: Index compression. Web Site, 2010. URL http://www.ir.uwaterloo.ca/book/addenda-06-index-compression.html. Accessed December 10, 2013. 54, 55

[14] Jeff Dean. Challenges in building large-scale information retrieval systems. Keynote, 2009. URL http://research.google.com/people/jeff/WSDM09-keynote.pdf. Accessed October 28, 2013. 28

[15] P. Elias. Universal codeword sets and representations of the integers. *Information Theory, IEEE Transactions on*, 21(2):194–203, 1975. ISSN 0018-9448. doi: 10.1109/TIT.1975.1055349. 29, 30

[16] Anne Cathrine Elster. HPC-Lab at IDI/NTNU. Web Site, 2013. URL http://research.idi.ntnu.no/hpc-lab/. Accessed June 5, 2013. 2

[17] Foresman, Chris. Half a billion dollars: why Apple's acquisition of Anobit matters. Web Site, 2012. URL http://arstechnica.com/apple/2011/12/apple-lays-down-half-a-billion-to-secure-its-flash-storage-future/. Accessed October 19, 2013. 10

[18] John F Gantz and Christopher Chute. The diverse and exploding digital universe: An updated forecast of worldwide information growth through 2011. IDC, 2008. 25

[19] Vivek Gowri. Samsung Galaxy Tab - The Anandtech Review. Web Site, 2010. URL http://www.anandtech.com/show/4062/samsung-galaxy-tab-the-anandtech-review. Accessed November 5, 2013. 8

[20] Laura M Grupp, Adrian M Caulfield, Joel Coburn, Steven Swanson, Eitan Yaakobi, Paul H Siegel, and Jack K Wolf. Characterizing flash memory: anomalies, observations, and applications. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 24–33. IEEE, 2009. 7

[21] Laura M. Grupp, John D. Davis, and Steven Swanson. The bleak future of NAND flash memory. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, FAST'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2208461.2208463. 7

[22] John L. Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31:532–533, 1988. 19

[23] Nick Heath. Inside ARM: The British success story taking the chip world by storm. Web Site, 2012. URL http://www.zdnet.com/uk/inside-arm-the-british-success-story-taking-the-chip-world-by-storm-7000008437 Accessed November 2, 2013. 6

[24] Mark D Hill and Michael R Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008. 18, 19, 21, 67, 75

[25] Honan, Mathew. Apple unveil iPhone. Web Site, 2007. URL http://www.macworld.com/article/1054769/iphone.html. Accessed October 10, 2013. 8

[26] Bojun Huang and Zenglin Xia. Allocating inverted index into flash memory for search engines. In *Proceedings of the 20th international conference companion on World wide web*, pages 61–62. ACM, 2011. 31

[27] iFixit. iPad Wi-Fi Teardown. Web Site, 2010. URL http://www.ifixit.com/Teardown/iPad+Wi-Fi+Teardown/2183/1. Accessed October 19, 2013. 10

[28] iFixit. iPod Touch 4th Generation Teardown. Web Site, 2010. URL http://www.ifixit.com/Teardown/iPod+Touch+4th+Generation+Teardown/3562/1. Accessed October 19, 2013.

[29] iFixit. iPhone 4s Teardown. Web Site, 2011. URL http://www.ifixit.com/Teardown/iPhone+4S+Teardown/6610/1. Accessed October 19, 2013. 10

[30] iFixit. Samsung Galaxy S III Teardown. Web Site, 2012. URL http://www.ifixit.com/Teardown/Samsung+Galaxy+S+III+Teardown/9391. Accessed November 5, 2013. 8

[31] iFixit. iPhone 5s Teardown. Web Site, 2013. URL http://www.ifixit.com/Teardown/iPhone+5s+Teardown/17383/1. Accessed October 19, 2013. 10

[32] Intel. New Intel Centrino Atom Processor Technology Ushers in 'Best Internet Experience in Your Pocket', 2008. 7

[33] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual. 1, 2011. 7

[34] Intel. Intel Atom Processor, 2013. 7

[35] Jean of Sogeti ESEC Lab. Low-level iOS forensics. Web Site, 2012. URL http://esec-lab.sogeti.com/post/Low-level-iOS-forensics. Accessed October 19, 2013. 10

[36] JEDEC. JEDEC Announces Publication of LPDDR2 Standard for Low Power Memory Devices. Web Site, 2009. URL http://www.jedec.org/news/pressreleases/jedec-announces-publication-lpddr2-standard-low-power-memory-devices. Accessed November 5, 2013. 8

[37] JEDEC. JEDEC Announces Publication of LPDDR3 Standard for Low Power Memory Devices. Web Site, 2012. URL http://www.jedec.org/news/pressreleases/jedec-announces-publication-lpddr3-standard-low-power-memory-devices. Accessed November 5, 2013. 8

[38] JEDEC Standard JESD209. Low Power Double Data Rate (LPDDR) SDRAM Specification. *JEDEC Solid State Technology Association*, 2007. 8

[39] Alan H. Karp and Horace P. Flatt. Measuring parallel processor performance. *Commun. ACM*, 33(5):539–543, May 1990. ISSN 0001-0782. doi: 10.1145/78607.78614. URL http://doi.acm.org/10.1145/78607.78614. 18

[40] Tom Krazit. ARMed for the living room. Web Site, 2006. URL http://news.cnet.com/ARMed-for-the-living-room/2100-1006_3-6056729.html. Accessed November 2, 2013. 6

[41] Anand Lal Shimpi. ipad 4 gpu performance analyzed: Powervr sgx 554mp4 under the hood, 2012. URL http://www.anandtech.com/show/6426/ipad-4-gpu-performance-analyzed-powervr-sgx-554mp4-under-the-hood. Accessed December 16, 2013. 9

[42] Anand Lal Shimpi. The iPad Air Review. Web Site, 2013. URL http://www.anandtech.com/show/7460/apple-ipad-air-review/2. Accessed December 10, 2013. 63

[43] Anand Lal Shimpi. The iphone 5s review, 2013. URL http://anandtech.com/show/7335/the-iphone-5s-review/. Accessed December 16, 2013. 9

[44] Anand Lal Shimpi and Vivek Gowri. The apple ipad review (2012), 2012. URL http://www.anandtech.com/show/5688/apple-ipad-2012-review/. Accessed December 16, 2013.

[45] Anand Lal Shimpi and Vivek Gowri. The iphone 5 review, 2012. URL http://www.anandtech.com/show/6330/the-iphone-5-review. Accessed December 16, 2013. 9

[46] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008. ISBN 0521865719, 9780521865715. 23, 25, 27, 30

[47] Chanik Park, Jaeyu Seo, Dongyoung Seo, Shinhan Kim, and Bumsoo Kim. Cost-efficient memory architecture design of nand flash memory embedded systems. In *Computer Design, 2003. Proceedings. 21st International Conference on*, pages 474–480. IEEE, 2003. 63

[48] Benjamin Schlegel, Rainer Gemulla, and Wolfgang Lehner. Fast integer compression using simd instructions. In *Proceedings of the Sixth International Workshop on Data Management on New Hardware*, DaMoN '10, pages 34–40, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0189-3. doi: 10.1145/1869389.1869394. URL http://doi.acm.org/10.1145/1869389.1869394. 30, 31

[49] Falk Scholer, Hugh E. Williams, John Yiannis, and Justin Zobel. Compression of inverted indexes for fast query evaluation. In *Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '02, pages 222–229, New York, NY, USA, 2002. ACM. ISBN 1-58113-561-0. doi: 10.1145/564376.564416. URL http://doi.acm.org/10.1145/564376.564416. 25, 30

[50] David Seal. *ARM architecture reference manual*. Pearson Education, 2000. 6

[51] Anand Lal Shimpi and Brian Klug. iPhone 5 Memory Size and Speed Revealed: 1 GB LPDDR2-1066. Web Site, 2012. URL http://www.anandtech.com/show/6297/iphone-5-memory-size-and-speed-revealed-1gb-lpddr21066. Accessed November 5, 2013. 8

[52] Shoshanna Solomon and Jonathan Ferziger. Apple Said to Acquire Israel's Anobit Technologies for About $390 Million. Web Site, 2012. URL http://www.bloomberg.com/news/2012-01-11/apple-is-said-to-acquire-israeli-component-maker-anobit-for-390-million.html. Accessed October 19, 2013. 10

[53] JEDEC Standard. Low Power Double Data Rate 2 (LPDDR2). *JESD209-2E, April*, 2011. 8

[54] JEDEC Standard. Low Power Double Data Rate 3 SDRAM (LPDDR3). *JESD209-3, May*, 2012. 8

[55] Alexander A Stepanov, Anil R Gangolli, Daniel E Rose, Ryan J Ernst, and Paramjit S Oberoi. Simd-based decoding of posting lists. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 317–326. ACM, 2011. 48, 49, 104

[56] Morrison Jim James Dick Fontaine Ray Tanner, Jason and Phil Gamache. Inside the iphone 5s | chipworks blog, 2013. URL http://www.chipworks.com/en/technical-competitive-analysis/resources/blog/inside-the-iphone-5s/. Accessed December 16, 2013. 9

[57] Gabriel Torres. Inside Pentium 4 Architecture. Web Site, 2005. URL http://www.hardwaresecrets.com/article/Inside-Pentium-4-Architecture/235/4. Accessed November 8, 2013. 7

[58] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes (2Nd Ed.): Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999. ISBN 1-55860-570-3. 30

[59] George Kingsley Zipf. Human behavior and the principle of least effort. 1949. 38

Appendix A

Bundled Scripts and Applications

Together with this thesis one should find the following scripts and applications:

- Within *PostingList* folder: Pythons script to generate an inverted index with postings lists according to Zipf's law.
- Within *SSDPerformanceMapping* folder: iOS application to measure the sequential reading and random access reading.
- Within *PostingListApp* folder: iOS application to measure encoding and decoding performance of implemented coding schemes.
- Within *PostingListCompression* folder: iOS framework used by PostingListApp to perform encoding and decoding.

The three iOS applications have their respective Xcode projects included as well.

Appendix B

Executing SSDPerformanceMapping

"SSDPerformanceMapping" is the iOS application used to measure the sequential reading and random access reading performance of iOS devices. Execution of SSD-PerformanceMapping must happen through Apple's Xcode IDE. It is not available in either Apple's App store or via a third-party developer tool such as TestFlight. The applications Xcode project file is attached to the thesis.

B.1 Preparations

Due to the size of the file used during benchmarking in thesis, this is not bundled with the delivery. As such, one must generate a file before executing and bundle it together with the application before execution. There are no requirements to the data file, other than it being identified as `data.random`. The file is bundled with the application by dragging and dropping it inside the opened Xcode project.

Appendix C

Executing PostingListApp

“PostingListApp” is the iOS application used to configure and measure the encoding and decoding performance of the three implemented coding schemes. As with SSDPerformanceMapping, it must be executed through Xcode.

C.1 Preparations

With the postings list used during benchmarking measuring over 40 MiB, it has not been delivered with the thesis. Therefore, a postings list must be generated beforehand and bundled with the application. This can be done with the Python script attached to this thesis in the "PostingList" folder as such:

```
python generate.py generate <parameters>
```

Available parameters are:

- `-a<number>`: The value of the exponent characterizing the distribution (defaults to 2).
- `<number>`: The total number of postings to distribute among terms (defaults to 10 000 000).
- `<number>`: The total number of documents to simulate present in the index (defaults to 100 000 000).
- `-v`: Verbose output.

The postings list will be generated in the same folder as the script is executed from. This file can then be bundled with the benchmarking application by dragging and dropping it into the opened Xcode project.

C.1.1 Generating Encoded Data

Generating encoded data for each coding scheme to benchmark decoding is tedious. Preparations are best executed via the following steps:

1. Start the application through Xcode.
2. Select the coding scheme.
3. Press encode.
4. Copy the files produced by encoding to a folder on the computer and rename them accordingly:
    - Variable-byte coding:
        - `vbyte-data.out`
        - `vbyte-mapping.out`
    - Group Varint coding:
        - `gvi-data.out`
        - `gvi-mapping.out`
    - Elias γ coding:
        - `elias-data.out`
        - `elias-mapping.out`
5. Drag and drop each file into the opened Xcode project.

C.2 Caveats During Execution

PostingListApp is in its alpha stage of development and contains several issues one should be aware of:

- Encoding or decoding several consecutive time without restarting the application does not work.

- Encoding or decoding over several iterations is currently not properly supported.
- Encoding or decoding more than one term is not thoroughly tested and the behaviour of the application under a multiterm benchmark is undefined.