Compression, SIMD, and Postings Lists

Andrew Trotman
Department of Computer Science
University of Otago
Dunedin, New Zealand
andrew@cs.otago.ac.nz

ABSTRACT

The three generations of postings list compression strategies (Variable Byte Encoding, Word Aligned Codes, and SIMD Codecs) are examined in order to test whether or not each truly represented a generational change – they do. Some weaknesses of the current SIMD-based schemes are identified and a new scheme, QMX, is introduced to address both space and decoding inefficiencies. Improvements are examined on multiple architectures and it is shown that different SSE implementations (Intel and AMD) perform differently.

Categories and Subject Descriptors

H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing - *Indexing methods*

General Terms

Algorithms, Performance.

Keywords

Compression, Procrastination.

1. INTRODUCTION

Modern search engines usually store their postings list in memory and compressed. The variety of compression scheme (or *codecs*) has changed periodically and we are now entering a third generation.

In the first generation, typified by Elias [5], Golomb [6] and Variable Byte Encoding, indexes were stored on disk. It was generally accepted that a disk seek and read was slow by comparison to the decoding time, regardless of the cost of decoding. Indeed, disk was so slow that the main objective of index compression research was to create an encoding scheme targeting the smallest possible postings lists. Moffat & Stuiver [8] present the Binary Interpolative Coding, an elegant codec, for exactly this purpose.

The first generational change happened when Scholer *et al.* [9] observed that Variable Byte Encoding was faster *in situ* in the search engine than Elias or Golomb codes. Trotman [12] analyzed the performance of a hard disk drive and the CPU when loading and decoding postings lists and also showed that, of the schemes he tested, Variable Byte Encoding was the most efficient. The conclusion, at that time, was that bit-wise schemes such as Elias and Golomb were slower than byte-wise schemes because they required more CPU instructions to decode.

In the second generation, typified by Simple-9 [1], Simple-16 [15], Simple-8b [2], PForDelta [16], and VSEncoding [10], the objective was to pack as many integers as possible into a machine word. In the case of Simple-9, that machine word is a 32-bit integer; in the case of Simple-8b it is a 64-bit integer.

Another generational change is happening right now. Authors such as Stepanov *et al.* [11] have proposed using SIMD instructions for decoding and have shown substantial improvements in decoding time by doing so. Others have improved on the result of Stepanov *et al.* including Lemire *et al.* [7].

Just as the shift from bit-wise schemes to byte-wise and word-wise schemes resulted in improvement, so too has the use of SIMD-word schemes, and for the same reasons. More work can be done in a single CPU instruction.

Recently Catena *et al.* [3] re-examined the result of Scholer *et al.* in light of hardware, operating system, compiler, language, and search engine algorithmic improvements over the last 10 years. They show that, of the schemes they tested, the PForDelta family of schemes resulted in the highest throughput. Such an experiment should be re-conducted periodically as new codecs are introduced – however that is not the topic of this contribution.

There are two important aspects to consider with any codec to be used in a search engine. The first is the compression ratio (i.e. index size), which henceforth shall be referred to as *effectiveness*, the second is the decoding time, which henceforth shall be referred to as *efficiency*. The first should be invariant to implementation, but the second certainly is not. This leads to the first research question of this investigation:

Are all implementations essentially equal in decoding efficiency?

To examine this question several different implementations of several different codecs were downloaded and compared to each other. In result, there are huge discrepancies in implementation efficiency. Which leads to the second question:

How do the most efficient implementations of different codecs compare to each other?

This questions is fundamentally different from the question usually asked when a new codec is introduced, that question being: is new codec *a* more efficient or effective than previous codec *b,* when implemented by the author of codec *a*? This is not to suggest deliberate bias on the part of the author of codec *a*, but rather that the intricacies of codec *b* are best understood by the originator of codec *b*, and therefore their implementation should be used (wherever possible), or at the very least any new implementation should be shown to be an improvement on it.

In result of this second question, it is unsurprising to find that the decoding efficiency and space effectiveness of the implementations tested does show a generational leap between the first and second generation (when compared to Variable Byte Encoding). There is evidence to suggest that the third generation made a contribution to decoding efficiency, but not space effectiveness. Leading to the third research question:

Can effectiveness or efficiency improvements be made on current SIMD codecs?

ADCS '14, November 27 - 28 2014, Melbourne, VIC, Australia
Copyright 2014 ACM 978-1-4503-3000-8/14/11…$15.00
http://dx.doi.org/10.1145/2682862.2682870

A new codec, QMX, is introduced. This codec combines word-alignment and SIMD, but is novel in that it also combines run-length encoding. Experiments show that this scheme is highly space effective and can be decoded extremely quickly.

CPU architectures are not universal. Even within the x86-64 family different aspects of the CPU are optimized for different purposes. This leads to the final research question:

Are the reported results generally applicable or platform specific?

The answer to this question is left to Section 5.

2. PRELIMINARIES

Experiments were conducted on one core of an otherwise idle 64-core 2.3GHz AMD Opteron 6276 based computer with power management disabled. This CPU has 48KB Level-1 cache, 1MB Level-2 cache and a shared 16MB Level-3 cache. Ubuntu Linux 12.04 kernel version 2.6.32 and g++ version 4.8.0 with the –O3 and –msse4 flags was used. This choice is examined in Section 5.

The TREC GOV2 document collection of 25,205,179 web pages crawled from the .gov domain in 2004 was used for tests because it is commonly used for experiments of this nature.

Each postings list was first loaded from disk, then for each codec it was encoded and the space needed for it was recorded. The time to decode was measured 5 times using the `CPUID RDTSC` instruction combination and the smallest value stored. This approach minimizes the effect of external events (such as interrupts) on the timing – however it also emphasizes the effect of the cache. The reported lengths and times are the mean recorded score over all lists of the same length.

The postings lists were created using the ATIRE search engine [13]. By default, ATIRE stores postings lists as term frequency ordered d-gaps. The postings list for a term, *t*, is normally thought of as a list of pairs <*d*, *tf*>, where *d* is the document id and *tf* is the term frequency (more accurately, occurrence count) of term *t* in document *d*. ATIRE sorts these lists on decreasing *tf*, then increasing *d*. Doing so makes it possible to store lists in the form <tf_1:$d_{1,1}$, $d_{1,2}$,…,$d_{1,n1}$, …, tf_m:$d_{m,1}$, $d_{m,2}$, …, $d_{m,nm}$>. Since the sequence $d_{k,1}$, $d_{k,2}$, …, $d_{k,n}$ is strictly monotonic, it is d-gap (also known as delta or difference) encoded, that is, consecutive differences are stored. For example, for the (<*d*, *tf*>) postings <1, 1>, <3, 1>, <5, 2>, <9, 2>; ATIRE will store <1: 1, 2><2: 5, 4>. ATIRE caps *tf* values at 255 and stores them uncompressed in a single byte. Consequently, experiments are conducted on the sequences of d-gap encoded document ids, not the term frequencies.

This contribution contains many color coded graphs. It is assumed the reader is reading on a color enabled medium (i.e. a screen). References to source code are footnoted on first occurrence.

3. BASELINES

In this section three different classes of codec are compared in an effort to identify a suitable baseline under which any new codec can be compared. Those classes are Variable Byte Encoding from the first generation, Word Aligned Codes from the second generation, and SIMD Codecs from the third. A baseline performance is established by comparing several different implementations.

3.1 Variable Byte Encoding

Variable Byte Encoding has proven popular for many years. At the end of the first generation Scholer *et al.* [9] as well as Trotman [12] suggested using this scheme, and it continues to be used as a baseline. However, not all implementations are equally effective, nor do they decode at the same rate.

3.1.1 Variable Byte Encoding: Algorithms

Although Variable Byte Encoding is usually described as an algorithm, it is more accurately a family of algorithms in which an integer is encoded in a variable number of whole bytes. This is problematic if used as a baseline for a scientific investigation because it is unclear what the comparison is against. Indeed, many prior investigators use their own byte-based compression scheme and called it Variable Byte Encoding, the decoding performance of these algorithms is, as expected, variable.

$$1905_{10} = 771_{16} = 11101110001_2 \rightarrow \boxed{\boxed{0}\,1\,1\,1\,0\,0\,0\,1}\ \boxed{\boxed{1}\,0\,0\,0\,1\,1\,1\,0}$$

Figure 1: Variable Byte Encoding of decimal 1905 (771 hex, 11101110001 binary) using the encoding of Silbestri *et al.*

Silvestri *et al.* [10][1] implement an algorithm which takes a 32-bit integer and breaks it into 7-bit chunks. For an integer that can be stored in 7 bits, it adds a leading (i.e. high) 1-bit and stores one byte. For an integer that requires 8-14 bits, it stores the low 7 bits with a leading 0 and then the high 7 bits with a leading 1. In other words, it stores the integer little endian with a termination flag stored as the top bit of each byte. Figure 1 depicts the encoding of decimal 1905_{10}. In binary (11101110001_2) it takes 11 bits, those bits require 2 bytes, the low 7 bits are stored in the first byte and the remaining 4 are stored in the second byte with leading 0s; the high bit of the first byte is set to indication continuation (0) and the high bit of the second byte is set to indicate termination (1) giving the sequence 01110001_2, 10001110_2 (hexadecimal 71_{16} $8E_{16}$). Silvestri *et al.* use this implementation to demonstrate the superiority of VSEncoding word-aligned codes (see Section 3.2)

It is the decoding algorithm that is more important for search engine efficiency. Silvestri *et al.* implement a decoder that first primes a bit-manipulation library with a long bit string. When decoding an integer they first ask this library for the next 8 consecutive bits, and store the low 7 bits of this byte for the resultant integer, then, and in a loop, they check the high bit of this byte and if non-zero ask for the next 8-bits, shift them the correct number of places (counted in integers) and OR it with the sum so far.

For Trotman's experiment discussed in Section 1 [12][2] he implements a similar algorithm, but stores integers big endian. His decompression algorithm manipulates bytes directly thus eliminating the calls to a general-purpose bit-manipulation library. By storing big endian he also eliminates the need to store how far to shift the current byte. He always shifts the current accumulated total to the left by 7 bits and ORs the next byte with the high bit turned off. Interestingly, he also has a short circuit for single byte integers.

Williams & Zobel [14][3] also break an integer into 7-bit chunks and store them big endian. The remaining bit in each byte (the low bit in this case) is used as a continuation bit, with a 0 indicating the end of the integer. During decompression this bit is turned off using a right shift, and the bytes are reconstructed into an integer using an ADD. It is presumed herein (perhaps incorrectly) that this is the implementation used by Scholer *et al.*

Zhang *et al.* [15] also store big endian and with a continuation bit stored in the low bit[4]. Different from Williams & Zobel, Zhang *et al.* they unroll the loop over the input byte stream when decom-

[1] http://hpc.isti.cnr.it/~integerencoding/

[2] http://atire.org/

[3] http://www.seg.rmit.edu.au/projects.html

[4] http://code.google.com/p/poly-ir-toolkit/

pressing. This implementation is distributed with a popular implementation of PForDelta.

Anh & Moffat [1] use a termination bit, have special handling for the single byte case, and store their integers big endian. However, they assume the sequence contains no zeros and so subtract one on encoding and add one on decoding. Their implementation is distributed with their code[5] for Simple-4b and Simple-8b.

The experiments of Catena *et al*. [3][6], discussed in Section 1, use the Hadoop implementation, which can also store negative numbers. Positive integers less than 128 are stored in one byte with a leading (high bit) 0. All other positive integers are broken into two parts. First the length of the integer (in bytes) is stored in a leading byte, and then the integer itself is stored big endian, using the minimum number of bytes possible. Decompression requires computation of the number of bytes used to store the integer. If this is one then the integer has been stored directly and decoding is over. Elsewise an accumulator is shifted left by 8 bits and the next byte is ORed with it – a process that is repeated until the integer is complete.

Unfortunately, the Hadoop implementation is in Java. For the experiments herein it was line-by-line translated into C++ with routines being marked `inline`. Results on this code are not the original code base and must be taken with caution.

Dean [4] suggests that Google use a Variable Byte Encoding scheme they call Group Varint. They compute the number of 8-bit bytes need to store an integer, being 1, 2, 3, or 4. Storing this number takes 2 bits. Storing 4 such numbers takes 8 bits. So rather than encoding each integer separately they encode 4 consecutive integers. First they store a byte representing the 4 lots of lengths, then they store each integer in the minimum number of bytes necessary. Decompression is of particular note. They take the leading length byte, `switch` on it, and from that know how to unpack the next 4 integers. This unpacking is extremely fast: a 1-byte integer can be unpacked by dereferencing a byte pointer and casting to a 32-bit integer; for a 2-byte integer the first byte is shifted left 8 places and the next byte is ORed with it; similarly for 3 and 4 byte integers. No loop is required for a tuple of 4 integers, and the compiler implements the `switch` using a branch table. Unexpectedly, comparison to Group Varint is not common in the literature. Burgess[2] provides the implementation used in the experiments.

It should be clear from the discussion in this section that Variable Byte Encoding is a family of algorithms, not a single algorithm. It should also be clear that decoding performance should vary from implementation to implementation.

3.1.2 Variable Byte Encoding: Performance

The previous section discussed Variable Byte Encoding with particular reference to different implementations seen in the literature. This section discusses the effectiveness (in bits per integer) and the decoding throughput (in clock-cycles) of these implementations.

Figure 2 shows on a log log scale the decoding efficiency of the various implementations tested. The horizontal axis is the number of integers being decoded while the vertical axis is the number of clock cycles necessary to decode that number of integers (smaller is better). The figure demonstrates the variability of decoding performance across implementations. The fastest implementation tested was the Group Varint implementation by Burgess.

[5] http://ww2.cs.mu.oz.au/~alistair/coders-64bit/

[6] http://hadoop.apache.org/

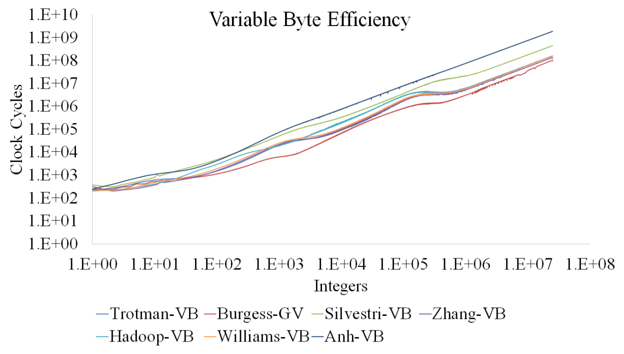

Figure 2: Decoding efficiency of Variable Byte Encoding Implementations. Of those tested, the Burgess implementation of Group Varint was the most efficient.

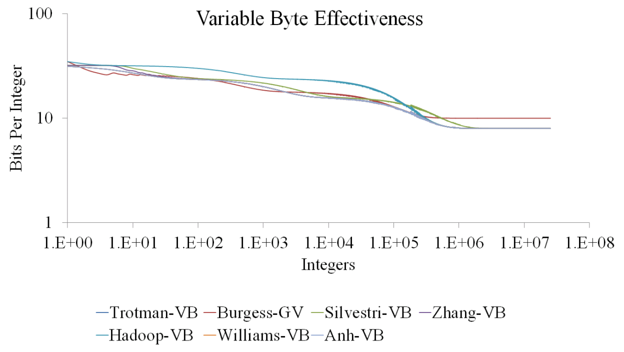

Figure 3: Effectiveness in bits per integer. Variable Byte Encoding reaches an effectiveness of 8 bits per integer while Group Varint reaches an effectiveness of 10 bits per integer.

Figure 3 shows on a log log scale the number of bits per integer necessary for storing one integer as the length of the postings list grows (and consequently d-gaps decrease). Once the d-gaps become small enough, each can be stored in one 8-bit byte with the termination (or continuation) bit set – the effectiveness tends to 8 bits per integer. The exception is Group Varint which stores 4 small integers in 8-bits each plus a 2-bit selector each (totaling 10 bits per integer).

In conclusion, Group Varint is less effective but more efficient than the others – the extra memory accesses do not inhibit throughput. The others are more effective but less efficient than Group Varint. In the next section the more space effective Word Aligned Codes are examined in a similar way.

3.2 Word Aligned Codes

Word Aligned Codecs were originally introduced by Anh & Moffat [1], but subsequently others have reported improvements in effectiveness and efficiency.

3.2.1 Word Aligned Codes: Algorithms

Word Aligned Codes, also known as The Simple Family, are a family of algorithms and not a single algorithm. Anh & Moffat [1] initially introduced Simple-9, Relative-10, and Carryover-12, however it is Simple-9 that sparked interest because it is more efficient to decode. To this Zhang *et al*. [15] added Simple-16, Anh & Moffat [2] added Simple-4b and Simple-8b. Zukowski *et al*. [16] introduced the PForDelta algorithm, and Silvestri & Venturini [10] contributed VSEncoding.

1 0 1 1	1 1 1 0 1 0 0	1 1 0 0 1 1 1	0 1 0 0 1 0 1	0 1 1 0 0 1 1

Figure 4: Simple-9 encodes a string of integers in two parts, on the left a selector indicating the length of each integer and on the right some number of integers of that length; all packed into 32-bits.

In Simple-9 a 32-bit word is divided into two parts (called *snips*), a 4-bit selector and a 28-bit payload. The payload is divided into some number of integers all of equal length. The 4-bit selector describes the number of integers stored in the payload, and consequently their bit-width. Figure 4 provides an example: the selector (1011_2) identifies a payload carrying four 7-bit integers (1110100_2 1100111_2, 0100101_2, and 0110011_2). There are 9 possible selector values for a 32-bit integer, hence the name Simple-9. Decoding is achieved by `switch`-ing the selector and then (in an un-wound loop) extracting each integer. Anh & Moffat do not provide an implementation but Silvestri *et al.*[1] do, as do Trotman & Subramanya[2].

Zhang *et al*. [15] extend Simple-9 by using the remaining selector values to describe asymmetric combinations, such as seven 2-bit integers followed by fourteen 1-bit integers; and reassigning codes that wasted bits. Their approach is known as Simple-16. Implementations by Silvestri *et al.*[1] and also by Burgess & Trotman[2] are available.

Anh & Moffat [2], use the spare selector values differently. They assume long runs of 1s are common and so the remaining selectors are used to store various run lengths of 1s. Anh & Moffat provide code[5] for their 32-bit version, Simple-4b, and 64-bit version, Simple-8b.

Zukowski *et al*. [16] realize that when using fixed bit-width blocks, a single outlier can lead to substantial deterioration in compression effectiveness. Consequently they identify outliers in the block and store those in a patch table rather than in-line with the sequence of numbers. Decoding involves patching up the sequence if outliers are present. They call their algorithm PForDelta, but do not provide an implementation. The implementation tested is that of Zhang *et al.*[4] using blocks of 128 integers (the default).

Silvestri & Venturini [10] observe that The Simple Family of compressors are greedy when assigning integers to payloads; they pack as many integers as possible into the current codeword before moving on to the next codeword. Their VSEncoding approach partitions integers into buckets of a fixed bit-width, then additionally stores the width of each bucket and the number of integers in each bucket. They describe two implementations. Their source code[1] has others including VSEncodingSimple v2, which they identify as the fastest, however it uses non-standard language features (specifically: they take the address of a label) and it would require a substantial re-write to make the code portable (to, for example, Visual Studio on Windows). The fastest portable variant they provide is VSEncodingBlocks.

3.2.2 Word Aligned Codes: Performance

Unlike Variable Byte Encoding, implementations of Word Aligned Codes tend to have different names and are often available directly from the author. However in the case of Simple-9 and Simple-16, two implementations have been identified.

Figure 5 shows that the decoding efficiency of the various Word Aligned Codes algorithms and implementations. The most efficient is the PForDelta algorithm as implemented by Zhang *et al.*

Figure 6 shows the effectiveness in bits per integer as the postings lists increase in length. When the lists are short, the 32-bit word codecs are most effective. Unlike Variable Byte Encoding, the effectiveness does not plateau at 8 bits per integer. Also unlike Variable Byte Encoding, there is no clear winner or loser when postings lists are long.

Inverted files are dominated by short postings lists, but the longer lists can have an overwhelming effect on total index size. Figure 7 shows, for all the implementations examined herein, the sum of the lengths of all postings lists once encoded. For this dataset, the smallest index occurs when Simple-8b is used. PForDelta, although efficient at decoding long lists, is ineffective for the large number of short lists. As with Variable Byte Encoding, the most effective and the most efficient algorithm are different.

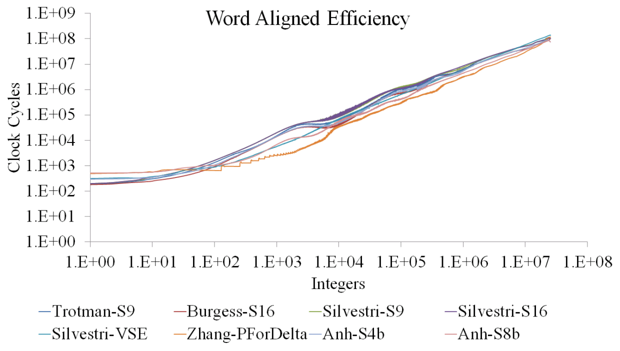

Figure 5: Decoding efficiency, in clock cycles, of the Word Aligned Codes algorithms. The most efficient decoder tested was that of Zhang *et al.* for the PForDelta variant.

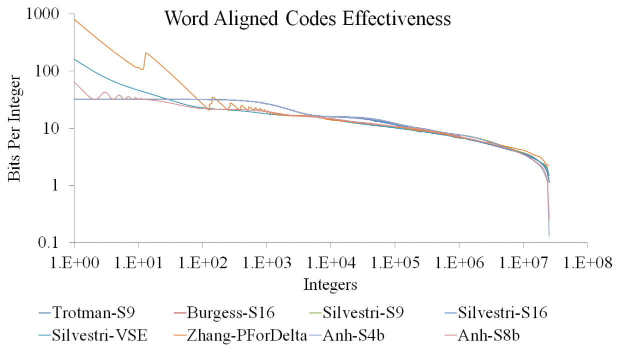

Figure 6: Effectiveness in bits per integer. There is no clear winner for long lists, but for short lists, the 32-bit versions of the Simple family are most effective.

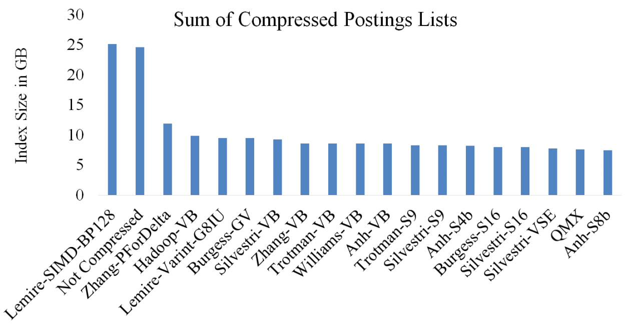

Figure 7: Sum of compressed lengths sorted largest to smallest. Simple-8b is smaller than QMX but is slower.

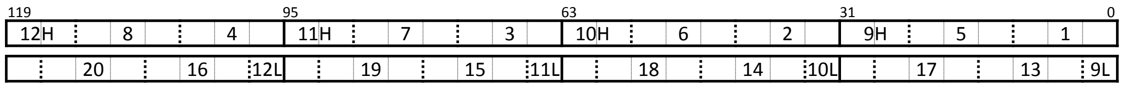

Figure 8: Twenty integers striped across two 128-bit words. The first 8 integers are stored in the first word, integers 9-12 are stored with the high 8 bits of each integer in the first word and the low 4 bits stored in the second 128-bit word. Integers 13-20 are stored in the final word. Vertical lines are nybble boundaries.

3.3 SIMD Codecs

The third generation of codecs use the SIMD instructions present on modern processors. Research has normally been conducted on Intel processors using the SSE instructions.

3.3.1 SIMD Codecs: Algorithms

Stepanov *et al.* [11] introduce varint-G8IU in which as many integers as possible are encoded into 8 consecutive bytes preceded by a one-byte descriptor that explains how many bytes each integer takes; it follows the form of Group Varint. Decoding is performed using the `PSHUFB` shuffle instruction, the descriptor being a key on how to shuffle. Stepanov *et al.* do not concern themselves with word-aligning their accesses, stating that "We depend on the ability of the CPU to perform unaligned reads and writes efficiently". Lemire & Boytsov provide an implementation[7].

Lemire & Boytsov [7] provide a comprehensive review of current codecs and implement several using SIMD instructions. Their fastest at decompression is SIMD-BP128. This codec uniformly Binary Packs blocks of 128 consecutive integers into the smallest number of 128-bit SIMD words possible. Much like Simple-9, each integer is stored using the same number of bits. Much like Group Varint, the selector is stored before a sequence. Specifically, a 16-byte selector is stored before 16 encoded blocks of 128 integers each. Storing in this way makes it possible to decode using only 128-bit SIMD word aligned reads and writes. Lemire & Boytsov provide an implementation[7].

3.3.2 SIMD Codecs: Performance

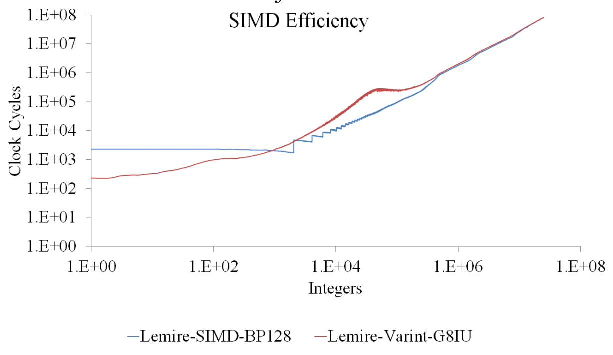

Figure 9: Decoding efficiency, in clock cycles. SIMD-BP128 is more efficient than varint-G8IU when lists are long, but the reverse is true when the lists are short

Figure 9 shows the decoding efficiency of the two SIMD algorithms. On the data used, SIMD-BM128 is more efficient than varint-G8IU when the lists are long, but the reverse is true when the lists are short. It should be noted that the two algorithms were implemented by the same author which might affect the efficiency – that said, it is reasonable to expect the word-aligned reading and writing to be more efficient that non-aligned reads and writes[8].

Figure 10 shows the effectiveness of the two codecs. SIMD-BP128, which encodes in 16 lots of 128 integers, is ineffective when postings lists are short, but effective when they are long. Varint-G8IU, however, plateaus at 9 bits per integer whereas SIMD-BP128 does not.

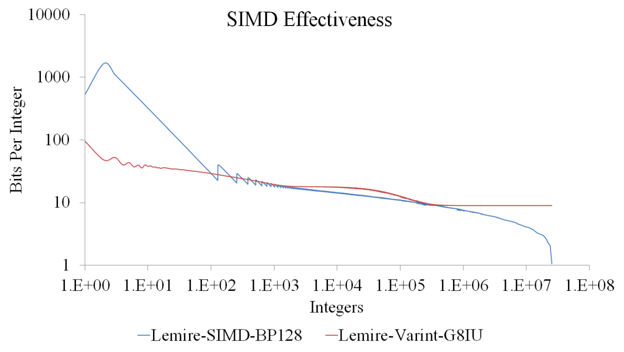

Figure 10: Effectiveness in bits per integer of the SIMD codecs. Neither algorithm is universally better than the other.

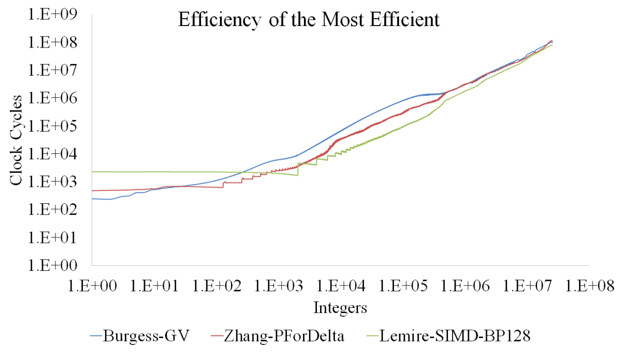

Figure 11: The most efficient decoder for long lists is the Lemire implementation of SIMD-BP128.

3.4 Comparing Baselines

This section has examined the three generations of codecs for compressing inverted files – in each case the most effective and the most efficient implementations have been identified. This section examines how those perform relative to each other.

In terms of effectiveness, Word Aligned Codes are the most effective – this can be seen by comparing Figure 3, Figure 6, and Figure 10. However, for a search engine throughput using an in-memory index, the decoding efficiency is more important.

Figure 11 compares the most efficient from each generation (Group Varint, PForDelta, and SIMD-BP128). There is no universally best codec, but once lists get long SIMD-BP128 is more efficient than the others. Indeed, this figure shows quite clearly

[7] https://github.com/lemire/FastPFor

[8] Preliminary experiments not reported here showed that a block copy using SSE instructions was faster when the aligned instructions were used than when the unaligned equivalents were used.

the generational change from Variable Byte Encoded to Word Aligned Codes to SIMD Codec and how each successive generation is more efficient on long lists.

4. NEW CODEC: QMX

The previous section demonstrated that the third generation of codecs has shown an improvement in decoding efficiency, even if they are not more effective. This section builds on the prior work and introduces a new codec called QMX.

It is assumed that integers being encoded and decoded are 2^{32} or smaller. This assumption is (currently) reasonable as it is unlikely that a search engine will host more than this number of documents in a single shard. Compression is lossless.

The first principle is to target short as well as long postings lists. Section 3.3 shows that neither of the two tested SIMD codecs is effective at both short and long lists.

The second principle is to use aligned reads and writes whenever possible. Preliminary experiments suggest that doing so will have a substantial effect on efficiency. The consequence of alignment is that a single 128-bit integer can be retrieved from main memory in a single read, and that a tuple of 4 integers (each 32 bits) can be written in a single write.

The implementation targets SSE4, an architecture that has been available in Intel CPUs since 2008 and AMD CPUs since 2011.

4.1 Overview of QMX

QMX is a codec for encoding positive integer sequences, typified by d-gap encoded postings lists seen in a search engine. Like Simple-9, a variable number of integers are fixed-width binary packed into payloads (128-bit SIMD words), and a selector is stored describing how they are packed. Like SIMD-BP128, the selector is stored separate from the payload. Unlike either, and novel to QMX, selectors are run-length encoded. So there are three parts: the payload (or Quantities), the run length (or Multipliers), and the selector (or eXtractor), hence the name.

4.2 QMX Payloads (Quantities)

In Simple-9 Anh & Moffat uniformly pack as many integers as possible into a 32-bit word. In SIMD-BP128, 128 consecutive integers are packed into as few 128-bit words as possible. The approach of Anh & Moffat is chosen for QMX: as many integers as possible are uniformly packed into a single 128-bit word. In this case uniformly means each and every integer in a 128-bit word is stored using the same number of bits.

Table 1: Packing of integers into 128-bit words

Bits	32	16	10	8	6	5	4	3	2	1
Integers	4	8	12	16	20	24	32	40	64	128
Waste	0	0	8	0	8	8	0	8	0	0
Writes	1	2	3	4	5	6	8	10	16	32

Table 1 shows all the possible ways that a whole number of tuples of 4 integers (each a single write) can be packed into a single 128-bit word (a single read). The first row lists the number of bits per integer. The second row lists the number of integers that can be packed using that number of bits. The third row lists the number of wasted bits. The final row gives the number of machine word (tuple) writes necessary to store those integers once decoded into 32-bit integers. For example, 24 integers of 5 bits each can be stored in a 128-bit word, once decoded into 32-bit integers they can be written back to main memory in 6 writes; doing so leaves 8 spare bits in the 128-bit word because $5 \times 24 = 120$.

In the Simple-9 method of Anh and Moffat, there are several spare selectors once all possible encodings are accounted for. Different authors used them in different ways. For example, in Simple-4b they are used to identify sequences of 1s, and in Simple-16 they are used for asymmetric combinations.

Table 2: Packing of integers into 256-bit words

Bits	21	12	9	7
Integers	12	20	28	36
Waste	4	16	4	4
Writes	3	5	7	9

When packing whole tuples of 4 integers into 128 bits, only 10 combinations are possible. However, when also including the number of ways that tuples of 4 integers can be stored in 256 bits (two consecutive words), there are several more. Table 2 shows these additional ways. The first row shows the number of bits, the second row shows the number of integers, the third rows shows the number of bits left over, and the final row lists the number of writes necessary to write these to memory. For example, 36 integers of 7 bits each can be packed into 252 bits with 4 bits wasted.

Combining the two tables gives 14 combinations. Following the technique seen in Simple-4b and Simple-8b, a 0-bit combination is added to these 14 ways giving 15 ways in total. This final packing is for 0-bit integers and identifies a sequence of 256 values, each implicitly 0. As the values are implicit no payload is stored.

There are many different ways that a number of integers might be packed into a 128-bit word. In Simple-9 they are packed consecutively. In SIMD-BP128 they are allocated across the four 32-bit words in a round-robin fashion with any spare bits spilling over to the next 128-bit word. Figure 8 shows this encoding.

In QMX integers are stored in three different ways. For the combinations seen in Table 2, the integers are stored round-robin with spill over. These can be decoded into tuples using a series of `SHIFT`, `AND`, and `OR` operations. In the case of combinations seen in Table 1 (except 8, 16, or 32-bits per integer), they are stored in a run-robin fashion with no spill over. These can be decoded using a series of `SHIFT` and `AND` operations. Both these combinations are similar to that seen in SIMD-BP128.

In the remaining cases (8, 16, or 32 bits per integer) the integers are always stored consecutively (similar to Simple-9). They are decoded using `PMOVZXBD`, `SHUFPS`, and `MOVHLPS` instructions.

This final encoding is chosen for effectiveness reasons. The lengths of postings lists in an inverted file typically follow a power law distribution. To avoid having to store long encodings for short sequences (i.e. a 128-bit word for a single 32-bit integer), any sequence of integers shorter than 16 is stored in one of these three ways, if doing so means a partial 128-bit word can be stored without loss. For example, striping the sequence 7, 9, 4, 6 would require a whole 128-bit word, but if stored byte packed consecutively would require only 32 bits. Unlike either SIMD-BP128 or Simple-9, short lists are stored truncated – this can only happen for entire (short) postings lists or at the end of a postings list; it cannot happen in the middle of a list.

4.3 QMX Selectors (eXtractors)

In Simple-9 the selector that identifies how many integers are stored in the 32-bit word is stored in 4 of the 32 bits in the word, leaving only 28 bits per word for coding. In SIMD-BP128 the selector is stored in a byte, 16 such bytes are collected together and stored before 16 payloads (in a similar way to Group Varint).

The approach taken for QMX is to store a selector in a byte. If the selectors were stored in-line with the data then the first payload would no longer be word aligned – this does not matter for streamed data where the data transfer is likely to take many times longer than decoding. Inverted indexes are, however, often loaded into main memory on search engine startup and served from there.

To ensure the reads remain aligned, the implementation stores the selectors in a sequence after the payload data.

In total 15 selectors are used: the combinations in Table 1 and Table 2, as well as 0 (for 256 0-bit numbers, without a payload).

4.4 QMX run lengths (Multipliers)

If selectors are stored in a byte there are 4 remaining unused bits in each selector. Unlike any of the algorithms discussed in Section 3, and novel to QMX, these bits are used to encode a run-length.

There are two parts that could be run-length encoded: the payload or the selector. It is unlikely (but possible) that the payload will repeat many times. As there are only 16 possible selectors, these are more likely to repeat. It is these that are run-length encoded. As a 0-length run is not possible, `runlength - 1` is stored, allowing for the same selector to represent up-to 16 consecutive payloads encoded in the same way.

4.5 QMX Implementation

In the implementation tested in the experiments, the payload data is laid out first. This is followed by the run-length encoded selectors. The selector is stored in the high nybble of a byte and the run length is encoded in 2s complement in the low nybble. In this way the decoding routine can be coded as a `switch` statement with fall through. For example, the (base 16) selector 00_{16} refers to 64 × 256 0-bit integers, the selector 01_{16} identifies 63 × 256 0-bit integers, and so on to $0F_{16}$ identifying 1 × 256 0-bit integers; and so the routine that decodes integers is a repeat of the same code for the cases of 00_{16}, 01_{16}, … $0F_{16}$ with a single `break` after `case` $0F_{16}$, and not the prior cases. That is, for this example, the inner loop (256 integers) is unrolled one time each for each integer, and the outer loop (run length times) is unrolled due to the `case` fall-through. This approach is somewhat similar to that seen in Group Varint Encoding, but novel in that fall-though is also utilized.

Sixteen selectors ($F0_{16}$, $F1_{16}$, …, FF_{16}) are unused, their use is left for future work. They could be used, for example, to identify exception cases much as PForDelta stores exceptions separately.

Silvestri *et al*., in their VSEncoding codec spend some time identifying the best packing of consecutive integers into words. The packing technique clearly affects effectiveness and efficiency – improving one, the other, or both. A left-greedy packing is used in the implementation. That is, working from left to right (first to last integer in the list), the best way to store the leftmost *q*-integers using the same number of bits is determined before moving to the right by *q*. Finding an optimal packer is left for future work.

Finally, there must be some way to identify the point separating the payload and selectors. This is done by storing, on the end of the encoded sequence, a pointer to the start of the selectors. In the implementation this is a Variable Byte Encoded integer encoding the length of the selector list plus the length of the encoded length (encoded backwards from the end of the string).

In this way, a single integer will be stored in at worst a 4-byte payload, a 1-byte run-length encoded selector, and a 1-byte pointer to the selector. The best case is a 1-byte payload, a 1-byte run-length encoded selector, and a 1-byte pointer to the selector.

As 0s cannot occur in a strictly monotonic sequence, the implementation uses the 0-bit selector for long runs of 1s rather than 0s.

4.6 QMX Performance

Figure 12 presents the efficiency of QMX when compared to the most efficient codecs seen in Section 3. For short postings lists QMX is the most efficient. For medium-length lists QMX is surpassed by SIMD-BP128, but for long lists QMX again outperforms the others.

Figure 13 shows the effectiveness of the codecs. Although Group Varint is the most effective with short postings lists, it is also the most ineffective with long ones. QMX is effective at almost all lengths of postings lists – taking slightly under 48 bits per integer for short lists and tending to slightly above 0 bits per integer when long runs of 1s are seen.

The selector utilization (excluding run-length) for the GOV2 collection is shown in Figure 14. The most used selector is for 8 bits per integer. There is a small hump at 12 and 16 bits per integer, most likely because there is no selector for 11, 13, 14, or 15 and so integers that might otherwise be stored in those number of bits must be stored in a larger number of bits. There is a slump at 1 bit per integer because, in the implementation, the 0-bit selector is used for storing runs of 1s, as 0s cannot occur. That is, the 1-bit selector can only be used to store runs of exactly 128 1s.

As QMX is effective at compressing both long and short postings lists it is reasonable to expect the overall index size will be substantially smaller when compressed using QMX than the other codecs. Figure 7 shows, for all those tested, the sum of compressed lengths for all postings lists – the part of the index variable in size due to compression. Not compressed is included for comparison. QMX is the second most effective (7.59GB) beaten only by Simple-8b (7.45GB) by 1.8%. However, sum of lengths does not account for alignment, necessary for SIMD decoding.

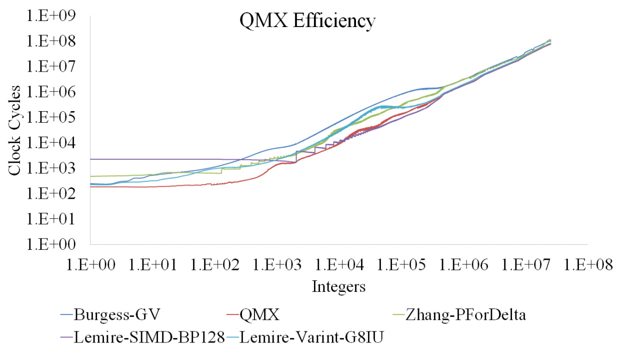

Figure 12: The efficiency of QMX is comparable, often better than others for short, medium, and long postings lists.

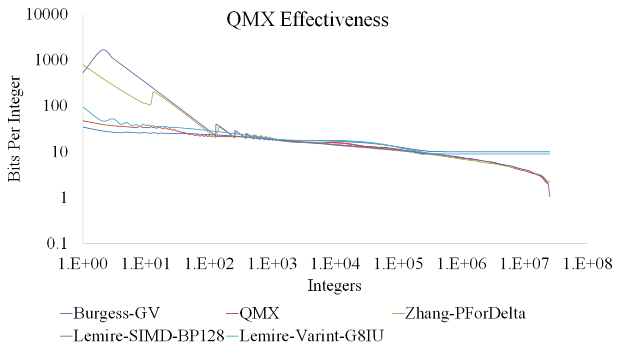

Figure 13: QMX is efficient at short and long sequences.

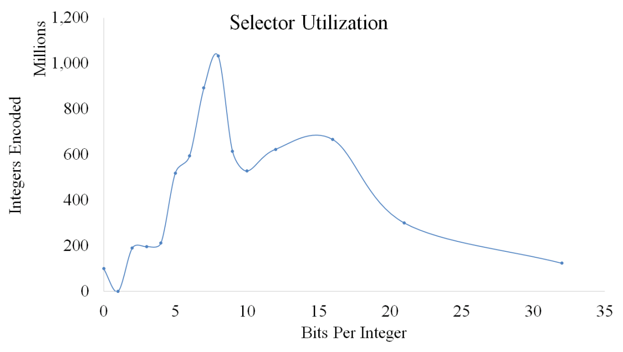

Figure 14: Selector utilization.

5. DISCUSSION

The QMX codec has shown to be effective at compressing term frequency ordered d-gaps, and efficient at decoding them, but the ATIRE search engine authors suggest BM25-impacted indexes are more efficient than term frequency ordered indexes. Such indexes store pre-computed BM25 values (quantized into 1-byte) rather than term frequencies in the postings lists. The experiments were re-conducted on such an index, the results being, in essence, the same as those reported for term frequency ordered indexes – and for space reasons they are not included here.

The experiment was also conducted on a three additional platforms: OS X 10.9.2 with Apple LLVM version 5.0 (clang-500.2.79) on a 3.2GHz Intel i5-4570 CPU (c. 2013); RedHat Linux 6.4 kernel version 2.6.32 with g++ 4.4.7 on a 2.4GHz Intel Xeon E5-2609 (c. 2012); and Windows-7 with cl 16.00.40219.01 on a 2.5 GHz Intel Xeon E5420 (c. 2007). AMD Opteron 6276 is circa 2011. It is reasonable to expect the same result irrespective of architecture.

Figure 15 presents (on a linear scale) the efficiency of the three algorithms, with linear trend lines added. Parallel trend lines represent implementations of equal efficiency per integer but with different start-up costs. Divergent trend lines represent algorithms with different costs per integer. Smaller is better.

On Opteron 6276, the i5, and the much slower Xeon E5420, the order of most to least efficient: QMX, SIMD-BP128, and then Varint-G8IU. On Xeon E5-2609 it is: Varint-G8IU, QMX, and then SIMD-BP128. That is, the behavior is inconsistent across the platforms. But in all cases, QMX proves to be efficient. Further investigation is required to measure and understand this performance difference – it is unclear whether it is a compiler, operating system, or hardware effect.

6. CONCLUSIONS

This investigation compared the three generations of compression strategies for inverted files. In answer to the first research question: "*are all implementations essentially equal in decoding efficiency*?", it shows that no, implementations vary considerably in efficiency. In answer to the question "*How do the most efficient implementations of different codecs compare to each other?*", it shows that the second generation is, indeed, more effective and efficient than the first generation; and that the third generation is more efficient and not less effective than the second.

The third generation (SIMD Codecs) were shown to be space ineffective (especially for short lists) and consequently a new codec, QMX, was introduced. In answer to the third research question "*Can effectiveness or efficiency improvements be made on current SIMD codecs?*", QMX was show to be more space effective and more efficient than SIMD-BP128 and Varint-G8IU.

However, differences in computer architecture, operating systems, and compilers lead to the fourth research question "*Are the reported results generally applicable or platform specific?*", to which the answer is platform specific. QMX was shown to be more efficient that the other SIMD codecs on most, but not all, platforms examined.

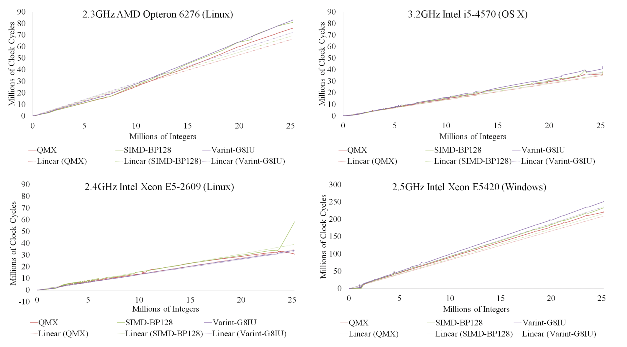

Figure 15: Efficiency on four different architectures suggests the results of Section 4 are platform dependent.

REFERENCES

[1] Anh, V.N., A. Moffat, *Inverted Index Compression using Word-Aligned Binary Codes*. Inf. Ret., 2005. 8(1):151-166.

[2] Anh, V.N., A. Moffat, *Index compression using 64-bit words*. Softw. Pract. Exper., 2010. 40(2):131-147.

[3] Catena, M., C. Macdonald, I. Ounis, *On Inverted Index Compression for Search Engine Efficiency*, in *ECIR 2014*, pp. 359-371.

[4] Dean, J., *Challenges in Building Large-scale Information Retrieval Systems: Invited Talk*, in *WSDM 2009*.

[5] Elias, P., *Universal Codeword Sets and the Representation of the Integers*. IEEE Trans. Inf. Theory, 1975. 21(2):194-203.

[6] Golomb, S.W., *Run-length Encodings*. IEEE Trans. Inf. Theory, 1966. 12(3):399-401.

[7] Lemire, D., L. Boytsov, *Decoding Billions of Integers per Second through Vectorization*. Software: Prac. Exper.

[8] Moffat, A., L. Stuiver, *Binary Interpolative Coding for Effective Index Compression*. Inf. Ret., 2000. 3(1):25-47.

[9] Scholer, F., H.E. Williams, J. Yiannis, J. Zobel. *Compression of Inverted Indexes for Fast Query Evaluation*. in *SIGIR 2002*, pp. 222-229

[10] Silvestri, F., R. Venturini, *VSEncoding: Efficient Coding and Fast Decoding of Integer Lists via Dynamic Programming*, in *CIKM 2010*, pp. 1219-1228.

[11] Stepanov, A.A., A.R. Gangolli, D.E. Rose, R.J. Ernst, P.S. Oberoi, *SIMD-based Decoding of Posting Lists*, in *CIKM 2011*, pp. 317-326.

[12] Trotman, A., *Compressing Inverted Files*. Inf Ret., 2003. 6(1):5-19.

[13] Trotman, A., X.-F. Jia, M. Crane, *Towards an Efficient and Effective Search Engine*, in *SIGIR 2012 Workshop on Open Source Information Retrieval*. 2012. pp. 40-47.

[14] Williams, H.E., J. Zobel, *Compressing Integers for Fast File Access*. Computer Journal, 1999. 42(3):193-201.

[15] Zhang, J., X. Long, T. Suel, *Performance of Compressed Inverted List Caching in Search Engines*, in *WWW 2008*, pp. 387-396.

[16] Zukowski, M., S. Heman, N. Nes, P. Boncz, *Super-Scalar RAM-CPU Cache Compression*, in *ICDE 2006*.