Faster and Smaller N-Gram Language Models

Adam Pauls   Dan Klein
Computer Science Division
University of California, Berkeley
{adpauls,klein}@cs.berkeley.edu

Abstract

N-gram language models are a major resource bottleneck in machine translation. In this paper, we present several language model implementations that are both highly compact and fast to query. Our fastest implementation is as fast as the widely used SRILM while requiring only 25% of the storage. Our most compact representation can store all 4 billion n-grams and associated counts for the Google n-gram corpus in 23 bits per n-gram, the most compact lossless representation to date, and even more compact than recent *lossy* compression techniques. We also discuss techniques for improving query speed during decoding, including a simple but novel language model caching technique that improves the query speed of our language models (and SRILM) by up to 300%.

1   Introduction

For modern statistical machine translation systems, language models must be both fast and compact. The largest language models (LMs) can contain as many as several hundred *billion* n-grams (Brants et al., 2007), so storage is a challenge. At the same time, decoding a single sentence can trigger hundreds of thousands of queries to the language model, so speed is also critical. As always, trade-offs exist between time, space, and accuracy, with many recent papers considering small-but-approximate noisy LMs (Chazelle et al., 2004; Guthrie and Hepple, 2010) or small-but-slow compressed LMs (Germann et al., 2009).

In this paper, we present several *lossless* methods for compactly but efficiently storing large LMs in memory. As in much previous work (Whittaker

and Raj, 2001; Hsu and Glass, 2008), our methods are conceptually based on tabular trie encodings wherein each n-gram key is stored as the concatenation of one word (here, the last) and an offset encoding the remaining words (here, the context). After presenting a bit-conscious basic system that typifies such approaches, we improve on it in several ways. First, we show how the last word of each entry can be implicitly encoded, almost entirely eliminating its storage requirements. Second, we show that the deltas between adjacent entries can be efficiently encoded with simple variable-length encodings. Third, we investigate block-based schemes that minimize the amount of compressed-stream scanning during lookup.

To speed up our language models, we present two approaches. The first is a front-end cache. Caching itself is certainly not new to language modeling, but because well-tuned LMs are essentially lookup tables to begin with, naive cache designs only speed up slower systems. We present a direct-addressing cache with a fast key identity check that speeds up our systems (or existing fast systems like the widely-used, speed-focused SRILM) by up to 300%.

Our second speed-up comes from a more fundamental change to the language modeling interface. Where classic LMs take word tuples and produce counts or probabilities, we propose an LM that takes a word-and-context encoding (so the context need not be re-looked up) and returns both the probability and also the context encoding for the *suffix* of the original query. This setup substantially accelerates the scrolling queries issued by decoders, and also exploits language model state equivalence (Li and Khudanpur, 2008).

Overall, we are able to store the 4 billion n-grams of the Google Web1T (Brants and Franz, 2006) cor-

258

pus, with associated counts, in 10 GB of memory, which is smaller than state-of-the-art *lossy* language model implementations (Guthrie and Hepple, 2010), and significantly smaller than the best published lossless implementation (Germann et al., 2009). We are also able to simultaneously outperform SRILM in both total size and speed. Our LM toolkit, which is implemented in Java and compatible with the standard ARPA file formats, is available on the web.[1]

2 Preliminaries

Our goal in this paper is to provide data structures that map n-gram keys to values, i.e. probabilities or counts. Maps are fundamental data structures and generic implementations of mapping data structures are readily available. However, because of the sheer number of keys and values needed for n-gram language modeling, generic implementations do not work efficiently "out of the box." In this section, we will review existing techniques for encoding the keys and values of an n-gram language model, taking care to account for every bit of memory required by each implementation.

To provide absolute numbers for the storage requirements of different implementations, we will use the Google Web1T corpus as a benchmark. This corpus, which is on the large end of corpora typically employed in language modeling, is a collection of nearly 4 billion n-grams extracted from over a trillion tokens of English text, and has a vocabulary of about 13.5 million words.

2.1 Encoding Values

In the Web1T corpus, the most frequent n-gram occurs about 95 billion times. Storing this count explicitly would require 37 bits, but, as noted by Guthrie and Hepple (2010), the corpus contains only about 770 000 *unique* counts, so we can enumerate all counts using only 20 bits, and separately store an array called the *value rank array* which converts the rank encoding of a count back to its raw count. The additional array is small, requiring only about 3MB, but we save 17 bits per n-gram, reducing value storage from around 16GB to about 9GB for Web1T.

We can rank encode probabilities and back-offs in the same way, allowing us to be agnostic to whether

we encode counts, probabilities and/or back-off weights in our model. In general, the number of bits per value required to encode all value ranks for a given language model will vary – we will refer to this variable as v.

2.2 Trie-Based Language Models

The data structure of choice for the majority of modern language model implementations is a *trie* (Fredkin, 1960). Tries or variants thereof are implemented in many LM tool kits, including SRILM (Stolcke, 2002), IRSTLM (Federico and Cettolo, 2007), CMU SLM (Whittaker and Raj, 2001), and MIT LM (Hsu and Glass, 2008). Tries represent collections of n-grams using a tree. Each node in the tree encodes a word, and paths in the tree correspond to n-grams in the collection. Tries ensure that each n-gram prefix is represented only once, and are very efficient when n-grams share common prefixes. Values can also be stored in a trie by placing them in the appropriate nodes.

Conceptually, trie nodes can be implemented as records that contain two entries: one for the word in the node, and one for either a pointer to the parent of the node or a list of pointers to children. At a low level, however, naive implementations of tries can waste significant amounts of space. For example, the implementation used in SRILM represents a trie node as a C `struct` containing a 32-bit integer representing the word, a 64-bit memory[2] pointer to the list of children, and a 32-bit floating point number representing the value stored at a node. The total storage for a node alone is 16 bytes, with additional overhead required to store the list of children. In total, the most compact implementation in SRILM uses 33 bytes per n-gram of storage, which would require around 116 GB of memory to store Web1T.

While it is simple to implement a trie node in this (already wasteful) way in programming languages that offer low-level access to memory allocation like C/C++, the situation is even worse in higher level programming languages. In Java, for example, C-style `structs` are not available, and records are most naturally implemented as objects that carry an additional 64 bits of overhead.

---

[1] http://code.google.com/p/berkeleylm/

[2] While 32-bit architectures are still in use today, their limited address space is insufficient for modern language models and we will assume all machines use a 64-bit architecture.

Despite its relatively large storage requirements, the implementation employed by SRILM is still widely in use today, largely because of its speed – to our knowledge, SRILM is the fastest freely available language model implementation. We will show that we can achieve access speeds comparable to SRILM but using only 25% of the storage.

2.3 Implicit Tries

A more compact implementation of a trie is described in Whittaker and Raj (2001). In their implementation, nodes in a trie are represented implicitly as entries in an array. Each entry encodes a word with enough bits to index all words in the language model (24 bits for Web1T), a quantized value, and a 32-bit[3] offset that encodes the contiguous block of the array containing the children of the node. Note that 32 bits is sufficient to index all n-grams in Web1T; for larger corpora, we can always increase the size of the offset.

Effectively, this representation replaces system-level memory pointers with offsets that act as logical pointers that can reference other entries in the array, rather than arbitrary bytes in RAM. This representation saves space because offsets require fewer bits than memory pointers, but more importantly, it permits straightforward implementation in any higher-level language that provides access to arrays of integers.[4]

2.4 Encoding n-grams

Hsu and Glass (2008) describe a variant of the implicit tries of Whittaker and Raj (2001) in which each node in the trie stores the prefix (i.e. parent). This representation has the property that we can refer to each n-gram $\mathbf{w}_1^n$ by its last word w_n and the offset $c(\mathbf{w}_1^{n-1})$ of its prefix $\mathbf{w}_1^{n-1}$, often called the *context*. At a low-level, we can efficiently encode this pair $(w_n, c(\mathbf{w}_1^{n-1}))$ as a single 64-bit integer, where the first 24 bits refer to w_n and the last 40 bits

encode $c(\mathbf{w}_1^{n-1})$. We will refer to this encoding as a *context encoding*.

Note that typically, n-grams are encoded in tries in the reverse direction (first-rest instead of last-rest), which enables a more efficient computation of back-offs. In our implementations, we found that the speed improvement from switching to a first-rest encoding and implementing more efficient queries was modest. However, as we will see in Section 4.2, the last-rest encoding allows us to exploit the scrolling nature of queries issued by decoders, which results in speedups that far outweigh those achieved by reversing the trie.

3 Language Model Implementations

In the previous section, we reviewed well-known techniques in language model implementation. In this section, we combine these techniques to build simple data structures in ways that are to our knowledge novel, producing language models with state-of-the-art memory requirements and speed. We will also show that our data structures can be very effectively compressed by implicitly encoding the word w_n, and further compressed by applying a variable-length encoding on context deltas.

3.1 Sorted Array

A standard way to implement a map is to store an array of key/value pairs, sorted according to the key. Lookup is carried out by performing binary search on a key. For an n-gram language model, we can apply this implementation with a slight modification: we need n sorted arrays, one for each n-gram order. We construct keys $(w_n, c(\mathbf{w}_1^{n-1}))$ using the context encoding described in the previous section, where the context offsets c refer to entries in the sorted array of $(n-1)$-grams. This data structure is shown graphically in Figure 1.

Because our keys are sorted according to their context-encoded representation, we cannot straightforwardly answer queries about an n-gram $\mathbf{w}$ without first determining its context encoding. We can do this efficiently by building up the encoding incrementally: we start with the context offset of the unigram w_1, which is simply its integer representation, and use that to form the context encoding of the bigram $\mathbf{w}_1^2 = (w_2, c(w_1))$. We can find the offset of
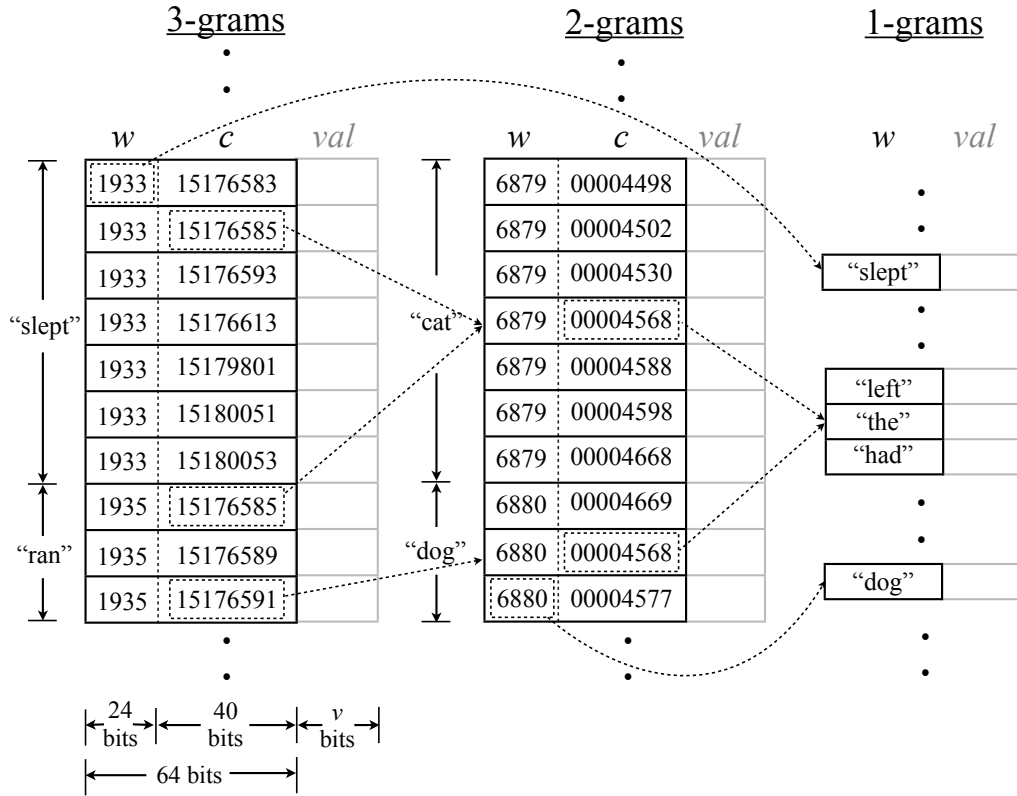
---

[3]The implementation described in the paper represents each 32-bit integer compactly using only 16 bits, but this representation is quite inefficient, because determining the full 32-bit offset requires a binary search in a look up table.

[4]Typically, programming languages only provide support for arrays of *bytes*, not bits, but it is of course possible to simulate arrays with arbitrary numbers of bits using byte arrays and bit manipulation.

Figure 1: Our SORTED implementation of a trie. The dotted paths correspond to "the cat slept", "the cat ran", and "the dog ran". Each node in the trie is an entry in an array with 3 parts: w represents the word at the node; val represents the (rank encoded) value; and c is an offset in the array of $n-1$ grams that represents the parent (prefix) of a node. Words are represented as offsets in the unigram array.

the bigram using binary search, and form the context encoding of the trigram, and so on. Note, however, that if our queries arrive in context-encoded form, queries are faster since they involve only one binary search in the appropriate array. We will return to this later in Section 4.2

This implementation, SORTED, uses 64 bits for the integer-encoded keys and v bits for the values. Lookup is linear in the length of the key and logarithmic in the number of n-grams. For Web1T ($v = 20$), the total storage is 10.5 bytes/n-gram or about 37GB.

3.2 Hash Table

Hash tables are another standard way to implement associative arrays. To enable the use of our context encoding, we require an implementation in which we can refer to entries in the hash table via array offsets. For this reason, we use an *open address* hash map that uses linear probing for collision resolution.

As in the sorted array implementation, in order to insert an n-gram $\mathbf{w}_1^n$ into the hash table, we must form its context encoding incrementally from the offset of w_1. However, unlike the sorted array implementation, at query time, we only need to be able to check equality between the query key $\mathbf{w}_1^n = (w_n, c(\mathbf{w}_1^{n-1}))$ and a key $\mathbf{w}_1^{'n} = (w'_n, c(\mathbf{w}_1^{'n-1}))$ in the table. Equality can easily be checked by first checking if $w_n = w'_n$, then recursively checking equality between $\mathbf{w}_1^{n-1}$ and $\mathbf{w}_1^{'n-1}$, though again, equality is even faster if the query is already context-encoded.

This HASH data structure also uses 64 bits for integer-encoded keys and v bits for values. However, to avoid excessive hash collisions, we also allocate additional empty space according to a user-defined parameter that trades off speed and time – we used about 40% extra space in our experiments. For Web1T, the total storage for this implementation is 15 bytes/n-gram or about 53 GB total.

Look up in a hash map is linear in the length of an n-gram and constant with respect to the number

261

of n-grams. Unlike the sorted array implementation, the hash table implementation also permits efficient insertion and deletion, making it suitable for stream-based language models (Levenberg and Osborne, 2009).

3.3 Implicitly Encoding w_n

The context encoding we have used thus far still wastes space. This is perhaps most evident in the sorted array representation (see Figure 1): all n-grams ending with a particular word w_i are stored contiguously. We can exploit this redundancy by storing only the context offsets in the main array, using as many bits as needed to encode all context offsets (32 bits for Web1T). In auxiliary arrays, one for each n-gram order, we store the beginning and end of the range of the trie array in which all (w_i, c) keys are stored for each w_i. These auxiliary arrays are negligibly small – we only need to store $2n$ offsets for each word.

The same trick can be applied in the hash table implementation. We allocate contiguous blocks of the main array for n-grams which all share the same last word w_i, and distribute keys within those ranges using the hashing function.

This representation reduces memory usage for keys from 64 bits to 32 bits, reducing overall storage for Web1T to 6.5 bytes/n-gram for the sorted implementation and 9.1 bytes for the hashed implementation, or about 23GB and 32GB in total. It also increases query speed in the sorted array case, since to find (w_i, c), we only need to search the range of the array over which w_i applies. Because this implicit encoding reduces memory usage without a performance cost, we will assume its use for the rest of this paper.

3.4 A Compressed Implementation

3.4.1 Variable-Length Coding

The distribution of value ranks in language modeling is Zipfian, with far more n-grams having low counts than high counts. If we ensure that the value rank array sorts raw values by descending order of frequency, then we expect that small ranks will occur much more frequently than large ones, which we can exploit with a variable-length encoding.

To compress n-grams, we can exploit the context encoding of our keys. In Figure 2, we show a portion
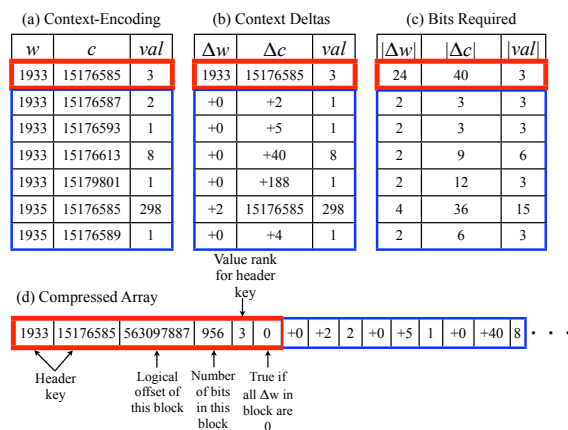

Figure 2: Compression using variable-length encoding. (a) A snippet of an (uncompressed) context-encoded array. (b) The context and word deltas. (c) The number of bits required to encode the context and word deltas as well as the value ranks. Word deltas use variable-length block coding with $k = 1$, while context deltas and value ranks use $k = 2$. (d) A snippet of the compressed encoding array. The header is outlined in bold.

of the key array used in our sorted array implementation. While we have already exploited the fact that the 24 word bits repeat in the previous section, we note here that consecutive context offsets tend to be quite close together. We found that for 5-grams, the median difference between consecutive offsets was about 50, and 90% of offset deltas were smaller than 10000. By using a variable-length encoding to represent these deltas, we should require far fewer than 32 bits to encode context offsets.

We used a very simple variable-length coding to encode offset deltas, word deltas, and value ranks. Our encoding, which is referred to as "variable-length block coding" in Boldi and Vigna (2005), works as follows: we pick a (configurable) radix $r = 2^k$. To encode a number m, we determine the number of digits d required to express m in base r. We write d in unary, i.e. $d - 1$ zeroes followed by a one. We then write the d digits of m in base r, each of which requires k bits. For example, using $k = 2$, we would encode the decimal number 7 as 010111. We can choose k separately for deltas and value indices, and also tune these parameters to a given language model.

We found this encoding outperformed other standard prefix codes, including Golomb codes (Golomb, 1966; Church et al., 2007)

and Elias γ and δ codes. We also experimented with the ζ codes of Boldi and Vigna (2005), which modify variable-length block codes so that they are optimal for certain power law distributions. We found that ζ codes performed no better than variable-length block codes and were slightly more complex. Finally, we found that Huffman codes outperformed our encoding slightly, but came at a much higher computational cost.

3.4.2 Block Compression

We could in principle compress the entire array of key/value pairs with the encoding described above, but this would render binary search in the array impossible: we cannot jump to the mid-point of the array since in order to determine what key lies at a particular point in the compressed bit stream, we would need to know the entire history of offset deltas.

Instead, we employ block compression, a technique also used by Harb et al. (2009) for smaller language models. In particular, we compress the key/value array in blocks of 128 bytes. At the beginning of the block, we write out a header consisting of: an explicit 64-bit key that begins the block; a 32-bit integer representing the offset of the header key in the uncompressed array;[5] the number of bits of compressed data in the block; and the variable-length encoding of the value rank of the header key. The remainder of the block is filled with as many compressed key/value pairs as possible. Once the block is full, we start a new block. See Figure 2 for a depiction.

When we encode an offset delta, we store the delta of the word portion of the key separately from the delta of the context offset. When an entire block shares the same word portion of the key, we set a single bit in the header that indicates that we do not encode any word deltas.

To find a key in this compressed array, we first perform binary search over the header blocks (which are predictably located every 128 bytes), followed by a linear search within a compressed block.

Using $k = 6$ for encoding offset deltas and $k = 5$ for encoding value ranks, this COMPRESSED implementation stores Web1T in less than 3 bytes per n-gram, or about 10.2GB in total. This is about

---

[5]We need this because n-grams refer to their contexts using array offsets.

6GB less than the storage required by Germann et al. (2009), which is the best published lossless compression to date.

4 Speeding up Decoding

In the previous section, we provided compact and efficient implementations of associative arrays that allow us to query a value for an arbitrary n-gram. However, decoders do not issue language model requests at random. In this section, we show that language model requests issued by a standard decoder exhibit two patterns we can exploit: they are highly *repetitive*, and also exhibit a *scrolling* effect.

4.1 Exploiting Repetitive Queries

In a simple experiment, we recorded all of the language model queries issued by the Joshua decoder (Li et al., 2009) on a 100 sentence test set. Of the 31 million queries, only about 1 million were unique. Therefore, we expect that keeping the results of language model queries in a cache should be effective at reducing overall language model latency.

To this end, we added a very simple cache to our language model. Our cache uses an array of key/value pairs with size fixed to $2^b - 1$ for some integer b (we used 24). We use a b-bit hash function to compute the address in an array where we will always place a given n-gram and its fully computed language model score. Querying the cache is straightforward: we check the address of a key given by its b-bit hash. If the key located in the cache array matches the query key, then we return the value stored in the cache. Otherwise, we fetch the language model probability from the language model and place the new key and value in the cache, evicting the old key in the process. This scheme is often called a *direct-mapped* cache because each key has exactly one possible address.

Caching n-grams in this way reduces overall latency for two reasons: first, lookup in the cache is extremely fast, requiring only a single evaluation of the hash function, one memory lookup to find the cache key, and one equality check on the key. In contrast, even our fastest (HASH) implementation may have to perform multiple memory lookups and equality checks in order to resolve collisions. Second, when calculating the probability for an n-gram
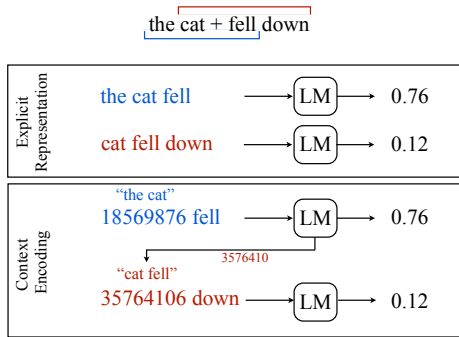
Figure 3: Queries issued when scoring trigrams that are created when a state with LM context "the cat" combines with "fell down". In the standard explicit representation of an n-gram as list of words, queries are issued atomically to the language model. When using a context-encoding, a query from the n-gram "the cat fell" returns the context offset of "cat fell", which speeds up the query of "cat fell down".

not in the language model, language models with back-off schemes must in general perform multiple queries to fetch the necessary back-off information. Our cache retains the full result of these calculations and thus saves additional computation.

Federico and Cettolo (2007) also employ a cache in their language model implementation, though based on traditional hash table cache with linear probing. Unlike our cache, which is of fixed size, their cache must be cleared after decoding a sentence. We would not expect a large performance increase from such a cache for our faster models since our HASH implementation is *already* a hash table with linear probing. We found in our experiments that a cache using linear probing provided marginal performance increases of about 40%, largely because of cached back-off computation, while our simpler cache increases performance by about 300% even over our HASH LM implementation. More timing results are presented in Section 5.

4.2 Exploiting Scrolling Queries

Decoders with integrated language models (Och and Ney, 2004; Chiang, 2005) score partial translation hypotheses in an incremental way. Each partial hypothesis maintains a language model context consisting of at most $n - 1$ target-side words. When we combine two language model contexts, we create several new n-grams of length of n, each of which generate a query to the language model. These new

Order	#n-grams
1gm	4,366,395
2gm	61,865,588
3gm	123,158,761
4gm	217,869,981
5gm	269,614,330
Total	676,875,055

Order	#n-grams
1gm	13,588,391
2gm	314,843,401
3gm	977,069,902
4gm	1,313,818,354
5gm	1,176,470,663
Total	3,795,790,711

Table 1: Sizes of the two language models used in our experiments.

n-grams exhibit a *scrolling* effect, shown in Figure 3: the $n - 1$ suffix words of one n-gram form the $n - 1$ prefix words of the next.

As discussed in Section 3, our LM implementations can answer queries about context-encoded n-grams faster than explicitly encoded n-grams. With this in mind, we augment the values stored in our language model so that for a key $(w_n, c(\mathbf{w}_1^{n-1}))$, we store the offset of the *suffix* $c(\mathbf{w}_2^n)$ as well as the normal counts/probabilities. Then, rather than represent the LM context in the decoder as an explicit list of words, we can simply store context offsets. When we query the language model, we get back both a language model score and context offset $c(\mathbf{\hat{w}_1^{n-1}})$, where $\mathbf{\hat{w}_1^{n-1}}$ is the the longest suffix of $\mathbf{w}_1^{n-1}$ contained in the language model. We can then quickly form the context encoding of the next query by simply concatenating the new word with the offset $c(\mathbf{\hat{w}}_1^{n-1})$ returned from the previous query.

In addition to speeding up language model queries, this approach also automatically supports an equivalence of LM states (Li and Khudanpur, 2008): in standard back-off schemes, whenever we compute the probability for an n-gram $(w_n, c(\mathbf{w}_1^{n-1}))$ when $\mathbf{w}_1^{n-1}$ is not in the language model, the result will be the same as the result of the query $(w_n, c(\mathbf{\hat{w}}_1^{n-1})$. It is therefore only necessary to store as much of the context as the language model contains instead of all $n - 1$ words in the context. If a decoder maintains LM states using the context offsets returned by our language model, then the decoder will automatically exploit this equivalence and the size of the search space will be reduced. This same effect is exploited explicitly by some decoders (Li and Khudanpur, 2008).

LM Type	bytes/ key	bytes/ value	bytes/ n-gram	Total Size
SRILM-H	–	–	42.2	26.6G
SRILM-S	–	–	33.5	21.1G
HASH	5.6	6.0	11.6	7.5G
SORTED	4.0	4.5	8.5	5.5G
TPT	–	–	7.5**	4.7G**
COMPRESSED	**2.1**	**3.8**	**5.9**	**3.7G**

Table 2: Memory usages of several language model implementations on the WMT2010 language model. A ** indicates that the storage in bytes per n-gram is reported for a different language model of comparable size, and the total size is thus a rough projection.

5 Experiments

5.1 Data

To test our LM implementations, we performed experiments with two different language models. Our first language model, WMT2010, was a 5-gram Kneser-Ney language model which stores probability/back-off pairs as values. We trained this language model on the English side of all French-English corpora provided[6] for use in the WMT 2010 workshop, about 2 billion tokens in total. This data was tokenized using the `tokenizer.perl` script provided with the data. We trained the language model using SRILM. We also extracted a count-based language model, WEB1T, from the Web1T corpus (Brants and Franz, 2006). Since this data is provided as a collection of 1- to 5-grams and associated counts, we used this data without further pre-processing. The make up of these language models is shown in Table 1.

5.2 Compression Experiments

We tested our three implementations (HASH, SORTED, and COMPRESSED) on the WMT2010 language model. For this language model, there are about 80 million unique probability/back-off pairs, so $v \approx 36$. Note that here v includes both the cost per key of storing the value rank as well as the (amortized) cost of storing two 32 bit floating point numbers (probability and back-off) for each unique value. The results are shown in Table 2.

LM Type	bytes/ key	bytes/ value	bytes/ n-gram	Total Size
Gzip	–	–	7.0	24.7G
T-MPHR†	–	–	3.0	10.5G
COMPRESSED	1.3	1.6	2.9	10.2G

Table 3: Memory usages of several language model implementations on the WEB1T. A † indicates lossy compression.

We compare against three baselines. The first two, SRILM-H and SRILM-S, refer to the hash table- and sorted array-based trie implementations provided by SRILM. The third baseline is the Tightly-Packed Trie (TPT) implementation of Germann et al. (2009). Because this implementation is not freely available, we use their published memory usage in bytes per n-gram on a language model of similar size and project total usage.

The memory usage of all of our models is considerably smaller than SRILM – our HASH implementation is about 25% the size of SRILM-H, and our SORTED implementation is about 25% the size of SRILM-S. Our COMPRESSED implementation is also smaller than the state-of-the-art compressed TPT implementation.

In Table 3, we show the results of our COMPRESSED implementation on WEB1T and against two baselines. The first is compression of the ASCII text count files using `gzip`, and the second is the Tiered Minimal Perfect Hash (T-MPHR) of Guthrie and Hepple (2010). The latter is a *lossy* compression technique based on Bloomier filters (Chazelle et al., 2004) and additional variable-length encoding that achieves the best published compression of WEB1T to date. Our COMPRESSED implementation is even smaller than T-MPHR, despite using a *lossless* compression technique. Note that since T-MPHR uses a lossy encoding, it is possible to reduce the storage requirements arbitrarily at the cost of additional errors in the model. We quote here the storage required when keys[7] are encoded using 12-bit hash codes, which gives a false positive rate of about $2^{-12} = 0.02\%$.

---

[6] `www.statmt.org/wmt10/translation-task.html`

[7] Guthrie and Hepple (2010) also report additional savings by quantizing values, though we could perform the same quantization in our storage scheme.

LM Type	No Cache	Cache	Size
COMPRESSED	9264±73ns	565±7ns	3.7G
SORTED	1405±50ns	243±4ns	5.5G
HASH	495±10ns	179±6ns	7.5G
SRILM-H	428±5ns	159±4ns	26.6G
HASH+SCROLL	323±5ns	139±6ns	10.5G

Table 4: Raw query speeds of various language model implementations. Times were averaged over 3 runs on the same machine. For HASH+SCROLL, all queries were issued to the decoder in context-encoded form, which speeds up queries that exhibit scrolling behaviour. Note that memory usage is higher than for HASH because we store suffix offsets along with the values for an n-gram.

LM Type	No Cache	Cache	Size
COMPRESSED	9880±82s	1547±7s	3.7G
SRILM-H	1120±26s	938±11s	26.6G
HASH	1146±8s	943±16s	7.5G

Table 5: Full decoding times for various language model implementations. Our HASH LM is as fast as SRILM while using 25% of the memory. Our caching also reduces total decoding time by about 20% for our fastest models and speeds up COMPRESSED by a factor of 6. Times were averaged over 3 runs on the same machine.

5.3 Timing Experiments

We first measured pure query speed by logging all LM queries issued by a decoder and measuring the time required to query those n-grams in isolation. We used the the Joshua decoder[8] with the WMT2010 model to generate queries for the first 100 sentences of the French 2008 News test set. This produced about 30 million queries. We measured the time[9] required to perform each query in order with and without our direct-mapped caching, not including any time spent on file I/O.

The results are shown in Table 4. As expected, HASH is the fastest of our implementations, and comparable[10] in speed to SRILM-H, but using sig-

---

[8]We used a grammar trained on all French-English data provided for WMT 2010 using the `make` scripts provided at `http://sourceforge.net/projects/joshua/files/joshua/1.3/wmt2010-experiment.tgz/download`

[9]All experiments were performed on an Amazon EC2 High-Memory Quadruple Extra Large instance, with an Intel Xeon X5550 CPU running at 2.67GHz and 8 MB of cache.

[10]Because we implemented our LMs in Java, we issued queries to SRILM via Java Native Interface (JNI) calls, which introduces a performance overhead. When called natively, we found that SRILM was about 200 ns/query faster. Unfortu-

nificantly less space. SORTED is slower but of course more memory efficient, and COMPRESSED is the slowest but also the most compact representation. In HASH+SCROLL, we issued queries to the language model using the context encoding, which speeds up queries substantially. Finally, we note that our direct-mapped cache is very effective. The query speed of all models is boosted substantially. In particular, our COMPRESSED implementation with caching is nearly as fast as SRILM-H without caching, and even the already fast HASH implementation is 300% faster in raw query speed with caching enabled.

We also measured the effect of LM performance on overall decoder performance. We modified Joshua to optionally use our LM implementations during decoding, and measured the time required to decode all 2051 sentences of the 2008 News test set. The results are shown in Table 5. Without caching, SRILM-H and HASH were comparable in speed, while COMPRESSED introduces a performance penalty. With caching enabled, overall decoder speed is improved for both HASH and SRILM-H, while the COMPRESSED implementation is only about 50% slower that the others.

6 Conclusion

We have presented several language model implementations which are state-of-the-art in both size and speed. Our experiments have demonstrated improvements in query speed over SRILM and compression rates against state-of-the-art lossy compression. We have also described a simple caching technique which leads to performance increases in overall decoding time.

Acknowledgements

---

nately, it is not completely fair to compare our LMs against either of these numbers: although the JNI overhead slows down SRILM, implementing our LMs in Java instead of C++ slows down our LMs. In the tables, we quote times which include the JNI overhead, since this reflects the true cost to a decoder written in Java (e.g. Joshua).

References

Paolo Boldi and Sebastiano Vigna. 2005. Codes for the world wide web. *Internet Mathematics*, 2.

Thorsten Brants and Alex Franz. 2006. Google web1t 5-gram corpus, version 1. In *Linguistic Data Consortium, Philadelphia, Catalog Number LDC2006T13*.

Thorsten Brants, Ashok C. Popat, Peng Xu, Franz J. Och, and Jeffrey Dean. 2007. Large language models in machine translation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*.

Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. 2004. The Bloomier filter: an efficient data structure for static support lookup tables. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*.

David Chiang. 2005. A hierarchical phrase-based model for statistical machine translation. In *The Annual Conference of the Association for Computational Linguistics*.

Kenneth Church, Ted Hart, and Jianfeng Gao. 2007. Compressing trigram language models with golomb coding. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*.

Marcello Federico and Mauro Cettolo. 2007. Efficient handling of n-gram language models for statistical machine translation. In *Proceedings of the Second Workshop on Statistical Machine Translation*.

Edward Fredkin. 1960. Trie memory. *Communications of the ACM*, 3:490–499, September.

Ulrich Germann, Eric Joanis, and Samuel Larkin. 2009. Tightly packed tries: how to fit large models into memory, and make them load fast, too. In *Proceedings of the Workshop on Software Engineering, Testing, and Quality Assurance for Natural Language Processing*.

S. W. Golomb. 1966. Run-length encodings. *IEEE Transactions on Information Theory*, 12.

David Guthrie and Mark Hepple. 2010. Storing the web in memory: space efficient language models with constant time retrieval. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*.

Boulos Harb, Ciprian Chelba, Jeffrey Dean, and Sanjay Ghemawat. 2009. Back-off language model compression. In *Proceedings of Interspeech*.

Bo-June Hsu and James Glass. 2008. Iterative language model estimation: Efficient data structure and algorithms. In *Proceedings of Interspeech*.

Abby Levenberg and Miles Osborne. 2009. Stream-based randomised language models for smt. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*.

Zhifei Li and Sanjeev Khudanpur. 2008. A scalable decoder for parsing-based machine translation with equivalent language model state maintenance. In *Proceedings of the Second Workshop on Syntax and Structure in Statistical Translation*.

Zhifei Li, Chris Callison-Burch, Chris Dyer, Juri Ganitkevitch, Sanjeev Khudanpur, Lane Schwartz, Wren N. G. Thornton, Jonathan Weese, and Omar F. Zaidan. 2009. Joshua: an open source toolkit for parsing-based machine translation. In *Proceedings of the Fourth Workshop on Statistical Machine Translation*.

Franz Josef Och and Hermann Ney. 2004. The alignment template approach to statistical machine translation. *Computationl Linguistics*, 30:417–449, December.

Andreas Stolcke. 2002. SRILM: An extensible language modeling toolkit. In *Proceedings of Interspeech*.

E. W. D. Whittaker and B. Raj. 2001. Quantization-based language model compression. In *Proceedings of Eurospeech*.