KenLM: Faster and Smaller Language Model Queries

Kenneth Heafield
Carnegie Mellon University
5000 Forbes Ave
Pittsburgh, PA 15213 USA
heafield@cs.cmu.edu

Abstract

We present KenLM, a library that implements two data structures for efficient language model queries, reducing both time and memory costs. The PROBING data structure uses linear probing hash tables and is designed for speed. Compared with the widely-used SRILM, our PROBING model is 2.4 times as fast while using 57% of the memory. The TRIE data structure is a trie with bit-level packing, sorted records, interpolation search, and optional quantization aimed at lower memory consumption. TRIE simultaneously uses less memory than the smallest lossless baseline and less CPU than the fastest baseline. Our code is open-source[1], thread-safe, and integrated into the Moses, cdec, and Joshua translation systems. This paper describes the several performance techniques used and presents benchmarks against alternative implementations.

1 Introduction

Language models are widely applied in natural language processing, and applications such as machine translation make very frequent queries. This paper presents methods to query N-gram language models, minimizing time and space costs. Queries take the form $p(w_n|w_1^{n-1})$ where w_1^n is an n-gram. Backoff-smoothed models estimate this probability based on the observed entry with longest matching history w_f^n, returning

$$p(w_n|w_1^{n-1}) = p(w_n|w_f^{n-1}) \prod_{i=1}^{f-1} b(w_i^{n-1}). \quad (1)$$

where the probability $p(w_n|w_f^{n-1})$ and backoff penalties $b(w_i^{n-1})$ are given by an already-estimated model. The problem is to store these two values for a large and sparse set of n-grams in a way that makes queries efficient.

Many packages perform language model queries. Throughout this paper we compare with several packages:

SRILM 1.5.12 (Stolcke, 2002) is a popular toolkit based on tries used in several decoders.

IRSTLM 5.60.02 (Federico et al., 2008) is a sorted trie implementation designed for lower memory consumption.

MITLM 0.4 (Hsu and Glass, 2008) is mostly designed for accurate model estimation, but can also compute perplexity.

RandLM 0.2 (Talbot and Osborne, 2007) stores large-scale models in less memory using randomized data structures.

BerkeleyLM revision 152 (Pauls and Klein, 2011) implements tries based on hash tables and sorted arrays in Java with lossy quantization.

Sheffield Guthrie and Hepple (2010) explore several randomized compression techniques, but did not release code.

TPT Germann et al. (2009) describe tries with better locality properties, but did not release code.

These packages are further described in Section 3. We substantially outperform all of them on query

[1] http://kheafield.com/code/kenlm

speed and offer lower memory consumption than lossless alternatives. Performance improvements transfer to the Moses (Koehn et al., 2007), cdec (Dyer et al., 2010), and Joshua (Li et al., 2009) translation systems where our code has been integrated. Our open-source (LGPL) implementation is also available for download as a standalone package with minimal (POSIX and g++) dependencies.

2 Data Structures

We implement two data structures: PROBING, designed for speed, and TRIE, optimized for memory. The set of n-grams appearing in a model is sparse, and we want to efficiently find their associated probabilities and backoff penalties. An important subproblem of language model storage is therefore sparse mapping: storing values for sparse keys using little memory then retrieving values given keys using little time. We use two common techniques, hash tables and sorted arrays, describing each before the model that uses the technique.

2.1 Hash Tables and PROBING

Hash tables are a common sparse mapping technique used by SRILM's default and BerkeleyLM's hashed variant. Keys to the table are hashed, using for example Austin Appleby's MurmurHash[2], to integers evenly distributed over a large range. This range is collapsed to a number of buckets, typically by taking the hash modulo the number of buckets. Entries landing in the same bucket are said to collide.

Several methods exist to handle collisions; we use linear probing because it has less memory overhead when entries are small. Linear probing places at most one entry in each bucket. When a collision occurs, linear probing places the entry to be inserted in the next (higher index) empty bucket, wrapping around as necessary. Therefore, a populated probing hash table consists of an array of buckets that contain either one entry or are empty. Non-empty buckets contain an entry belonging to them or to a preceding bucket where a conflict occurred. Searching a probing hash table consists of hashing the key, indexing the corresponding bucket, and scanning buckets until a matching key is found or an empty bucket is encountered, in which case the key does not exist in the table.

Linear probing hash tables must have more buckets than entries, or else an empty bucket will never be found. The ratio of buckets to entries is controlled by space multiplier $m > 1$. As the name implies, space is $O(m)$ and linear in the number of entries. The fraction of buckets that are empty is $\frac{m-1}{m}$, so average lookup time is $O\left(\frac{m}{m-1}\right)$ and, crucially, constant in the number of entries.

When keys are longer than 64 bits, we conserve space by replacing the keys with their 64-bit hashes. With a good hash function, collisions of the full 64-bit hash are exceedingly rare: one in 266 billion queries for our baseline model will falsely find a key not present. Collisions between two keys in the table can be identified at model building time. Further, the special hash 0 suffices to flag empty buckets.

The PROBING data structure is a rather straightforward application of these hash tables to store N-gram language models. Unigram lookup is dense so we use an array of probability and backoff values. For $2 \leq n \leq N$, we use a hash table mapping from the n-gram to the probability and backoff[3]. Vocabulary lookup is a hash table mapping from word to vocabulary index. In all cases, the key is collapsed to its 64-bit hash. Given counts c_1^n where e.g. c_1 is the vocabulary size, total memory consumption, in bits, is

$$(96m + 64)c_1 + 128m \sum_{n=2}^{N-1} c_n + 96mc_N.$$

Our PROBING data structure places all n-grams of the same order into a single giant hash table. This differs from other implementations (Stolcke, 2002; Pauls and Klein, 2011) that use hash tables as nodes in a trie, as explained in the next section. Our implementation permits jumping to any n-gram of any length with a single lookup; this appears to be unique among language model implementations.

2.2 Sorted Arrays and TRIE

Sorted arrays store key-value pairs in an array sorted by key, incurring no space overhead. SRILM's compact variant, IRSTLM, MITLM, and BerkeleyLM's

[2] `http://sites.google.com/site/murmurhash/`

[3] N-grams do not have backoff so none is stored.

sorted variant are all based on this technique. Given a sorted array A, these other packages use binary search to find keys in $O(\log |A|)$ time. We reduce this to $O(\log \log |A|)$ time by evenly distributing keys over their range then using interpolation search[4] (Perl et al., 1978). Interpolation search formalizes the notion that one opens a dictionary near the end to find the word "zebra." Initially, the algorithm knows the array begins at $b \leftarrow 0$ and ends at $e \leftarrow |A| - 1$. Given a key k, it estimates the position

$$pivot \leftarrow \frac{k - A[b]}{A[e] - A[b]}(e - b).$$

If the estimate is exact ($A[pivot] = k$), then the algorithm terminates succesfully. If $e < b$ then the key is not found. Otherwise, the scope of the search problem shrinks recursively: if $A[pivot] < k$ then this becomes the new lower bound: $l \leftarrow pivot$; if $A[pivot] > k$ then $u \leftarrow pivot$. Interpolation search is therefore a form of binary search with better estimates informed by the uniform key distribution.

If the key distribution's range is also known (i.e. vocabulary identifiers range from 0 to the number of words), then interpolation search can use this information instead of reading $A[0]$ and $A[|A| - 1]$ to estimate pivots; this optimization alone led to a 24% speed improvement. The improvement is due to the cost of bit-level reads and avoiding reads that may fall in different virtual memory pages.

Vocabulary lookup is a sorted array of 64-bit word hashes. The index in this array is the vocabulary identifier. This has the effect of randomly permuting vocabulary identifiers, meeting the requirements of interpolation search when vocabulary identifiers are used as keys.

While sorted arrays could be used to implement the same data structure as PROBING, effectively making $m = 1$, we abandoned this implementation because it is slower and larger than a trie implementation. The trie data structure is commonly used for language modeling. Our TRIE implements the popular reverse trie, in which the last word of an n-gram is looked up first, as do SRILM, IRSTLM's inverted variant, and BerkeleyLM except for the scrolling variant. Figure 1 shows an example. Nodes in the trie are based on arrays sorted by vocabulary identifier.

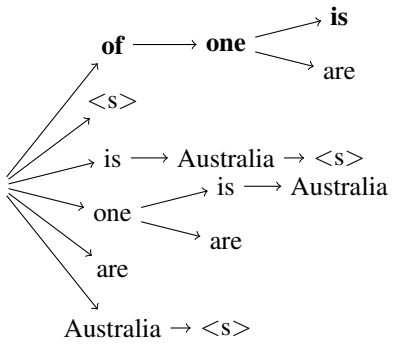

Figure 1: Lookup of "is one of" in a reverse trie. Children of each node are sorted by vocabulary identifier so order is consistent but not alphabetical: "is" always appears before "are". Nodes are stored in column-major order. For example, nodes corresponding to these n-grams appear in this order: "are one", "<s> Australia", "is one of", "are one of", "<s> Australia is", and "Australia is one".

We maintain a separate array for each length n containing all n-gram entries sorted in suffix order. Therefore, for n-gram w_1^n, all leftward extensions w_0^n are an adjacent block in the $n + 1$-gram array. The record for w_1^n stores the offset at which its extensions begin. Reading the following record's offset indicates where the block ends. This technique was introduced by Clarkson and Rosenfeld (1997) and is also implemented by IRSTLM and BerkeleyLM's compressed option. SRILM inefficiently stores 64-bit pointers.

Unigram records store probability, backoff, and an index in the bigram table. Entries for $2 \leq n < N$ store a vocabulary identifier, probability, backoff, and an index into the $n+1$-gram table. The highest-order N-gram array omits backoff and the index, since these are not applicable. Values in the trie are minimally sized at the bit level, improving memory consumption over trie implementations in SRILM, IRSTLM, and BerkeleyLM. Given n-gram counts $\{c_n\}_{n=1}^N$, we use $\lceil \log_2 c_1 \rceil$ bits per vocabulary identifier and $\lceil \log_2 c_n \rceil$ per index into the table of n-grams.

When SRILM estimates a model, it sometimes removes n-grams but not $n + 1$-grams that extend it to the left. In a model we built with default settings, 1.2% of $n + 1$-grams were missing their n-

[4]Not to be confused with interpolating probabilities, which is outside the scope of this paper.

gram suffix. This causes a problem for reverse trie implementations, including SRILM itself, because it leaves $n+1$-grams without an n-gram node pointing to them. We resolve this problem by inserting an entry with probability set to an otherwise-invalid value ($-\infty$). Queries detect the invalid probability, using the node only if it leads to a longer match. By contrast, BerkeleyLM's hash and compressed variants will return incorrect results based on an $n-1$-gram.

2.2.1 Quantization

Floating point values may be stored in the trie exactly, using 31 bits for non-positive log probability and 32 bits for backoff[5]. To conserve memory at the expense of accuracy, values may be quantized using q bits per probability and r bits per backoff[6]. We allow any number of bits from 2 to 25, unlike IRSTLM (8 bits) and BerkeleyLM ($17-20$ bits). To quantize, we use the binning method (Federico and Bertoldi, 2006) that sorts values, divides into equally sized bins, and averages within each bin. The cost of storing these averages, in bits, is

$$[32(N-1)2^q + 32(N-2)2^r$$

Because there are comparatively few unigrams, we elected to store them byte-aligned and unquantized, making every query faster. Unigrams also have 64-bit overhead for vocabulary lookup. Using c_n to denote the number of n-grams, total memory consumption of TRIE, in bits, is

$$(32+32+64+64)c_1 + \sum_{n=2}^{N-1}(\lceil \log_2 c_1 \rceil + q + r + \lceil \log_2 c_{n+1} \rceil)c_n + (\lceil \log_2 c_1 \rceil + q)c_N$$

plus quantization tables, if used. The size of TRIE is particularly sensitive to $\lceil \log_2 c_1 \rceil$, so vocabulary filtering is quite effective at reducing model size.

3 Related Work

SRILM (Stolcke, 2002) is widely used within academia. It is generally considered to be fast (Pauls and Klein, 2011), with a default implementation based on hash tables within each trie node. Each trie node is individually allocated and full 64-bit pointers are used to find them, wasting memory. The compact variant uses sorted arrays instead of hash tables within each node, saving some memory, but still stores full 64-bit pointers. With some minor API changes, namely returning the length of the n-gram matched, it could also be faster—though this would be at the expense of an optimization we explain in Section 4.1. The PROBING model was designed to improve upon SRILM by using linear probing hash tables (though not arranged in a trie), allocating memory all at once (eliminating the need for full pointers), and being easy to compile.

IRSTLM (Federico et al., 2008) is an open-source toolkit for building and querying language models. The developers aimed to reduce memory consumption at the expense of time. Their default variant implements a forward trie, in which words are looked up in their natural left-to-right order. However, their inverted variant implements a reverse trie using less CPU and the same amount of memory[7]. Each trie node contains a sorted array of entries and they use binary search. Compared with SRILM, IRSTLM adds several features: lower memory consumption, a binary file format with memory mapping, caching to increase speed, and quantization. Our TRIE implementation is designed to improve upon IRSTLM using a reverse trie with improved search, bit level packing, and stateful queries. IRSTLM's quantized variant is the inspiration for our quantized variant. Unfortunately, we were unable to correctly run the IRSTLM quantized variant. The developers suggested some changes, such as building the model from scratch with IRSTLM, but these did not resolve the problem.

Our code has been publicly available and intergrated into Moses since October 2010. Later, BerkeleyLM (Pauls and Klein, 2011) described ideas similar to ours. Most similar is *scrolling queries*, wherein left-to-right queries that add one word at a time are optimized. Both implementations employ a state object, opaque to the application, that carries information from one query to the next; we

[5] Backoff "penalties" are occasionally positive in log space.

[6] One probability is reserved to mark entries that SRILM pruned. Two backoffs are reserved for Section 4.1. That leaves $2^q - 1$ probabilities and $2^r - 2$ non-zero backoffs.

[7] Forward tries are faster to build with IRSTLM and can efficiently return a list of rightward extensions, but this is not used by the decoders we consider.

discuss both further in Section 4.2. State is implemented in their scrolling variant, which is a trie annotated with forward and backward pointers. The hash variant is a reverse trie with hash tables, a more memory-efficient version of SRILM's default. While the paper mentioned a sorted variant, code was never released. The compressed variant uses block compression and is rather slow as a result. A direct-mapped cache makes BerkeleyLM faster on repeated queries, but their fastest (scrolling) cached version is still slower than uncached PROBING, even on cache-friendly queries. For all variants, we found that BerkeleyLM always rounds the floating-point mantissa to 12 bits then stores indices to unique rounded floats. The 1-bit sign is almost always negative and the 8-bit exponent is not fully used on the range of values, so in practice this corresponds to quantization ranging from 17 to 20 total bits.

Lossy compressed models RandLM (Talbot and Osborne, 2007) and Sheffield (Guthrie and Hepple, 2010) offer better memory consumption at the expense of CPU and accuracy. These enable much larger models in memory, compensating for lost accuracy. Typical data structures are generalized Bloom filters that guarantee a customizable probability of returning the correct answer. Minimal perfect hashing is used to find the index at which a quantized probability and possibly backoff are stored. These models generally outperform our memory consumption but are much slower, even when cached.

4 Optimizations

In addition to the optimizations specific to each data-structure described in Section 2, we implement several general optimizations for language modeling.

4.1 Minimizing State

Applications such as machine translation use language model probability as a feature to assist in choosing between hypotheses. Dynamic programming efficiently scores many hypotheses by exploiting the fact that an N-gram language model conditions on at most $N-1$ preceding words. We call these $N-1$ words *state*. When two partial hypotheses have equal state (including that of other features), they can be recombined and thereafter efficiently handled as a single packed hypothesis. If there are too many distinct states, the decoder prunes low-scoring partial hypotheses, possibly leading to a search error. Therefore, we want state to encode the minimum amount of information necessary to properly compute language model scores, so that the decoder will be faster and make fewer search errors.

We offer a state function $s(w_1^n) = w_m^n$ where substring w_m^n is guaranteed to extend (to the right) in the same way that w_1^n does for purposes of language modeling. The state function is integrated into the query process so that, in lieu of the query $p(w_n|w_1^{n-1})$, the application issues query $p(w_n|s(w_1^{n-1}))$ which also returns $s(w_1^n)$. The returned state $s(w_1^n)$ may then be used in a follow-on query $p(w_{n+1}|s(w_1^n))$ that extends the previous query by one word. These make left-to-right query patterns convenient, as the application need only provide a state and the word to append, then use the returned state to append another word, etc. We have modified Moses (Koehn et al., 2007) to keep our state with hypotheses; to conserve memory, phrases do not keep state. Syntactic decoders, such as cdec (Dyer et al., 2010), build state from null context then store it in the hypergraph node for later extension.

Language models that contain w_1^k must also contain prefixes w_1^i for $1 \leq i \leq k$. Therefore, when the model is queried for $p(w_n|w_1^{n-1})$ but the longest matching suffix is w_f^n, it may return state $s(w_1^n) = w_f^n$ since no longer context will be found. IRSTLM and BerkeleyLM use this state function (and a limit of $N-1$ words), but it is more strict than necessary, so decoders using these packages will miss some recombination opportunities.

State will ultimately be used as context in a subsequent query. If the context w_f^n will never extend to the right (i.e. $w_f^n v$ is not present in the model for all words v) then no subsequent query will match the full context. If the log backoff of w_f^n is also zero (it may not be in filtered models), then w_f should be omitted from the state. This logic applies recursively: if w_{f+1}^n similarly does not extend and has zero log backoff, it too should be omitted, terminating with a possibly empty context. We indicate whether a context with zero log backoff will extend using the sign bit: $+0.0$ for contexts that extend and -0.0 for contexts that do not extend. RandLM and SRILM also remove context that will not extend, but

SRILM performs a second lookup in its trie whereas our approach has minimal additional cost.

4.2 Storing Backoff in State

Section 4.1 explained that state s is stored by applications with partial hypotheses to determine when they can be recombined. In this section, we extend state to optimize left-to-right queries. All language model queries issued by machine translation decoders follow a left-to-right pattern, starting with either the begin of sentence token or null context for mid-sentence fragments. Storing state therefore becomes a time-space tradeoff; for example, we store state with partial hypotheses in Moses but not with each phrase.

To optimize left-to-right queries, we extend state to store backoff information:

$$s(w_1^{n-1}) = \left(w_m^{n-1}, \left\{b(w_i^{n-1})\right\}_{i=m}^{n-1}\right)$$

where m is the minimal context from Section 4.1 and b is the backoff penalty. Because b is a function, no additional hypothesis splitting happens.

As noted in Section 1, our code finds the longest matching entry w_f^n for query $p(w_n|s(w_1^{n-1}))$ then computes

$$p(w_n|w_1^{n-1}) = p(w_n|w_f^{n-1}) \prod_{i=1}^{f-1} b(w_i^{n-1}).$$

The probability $p(w_n|w_f^{n-1})$ is stored with w_f^n and the backoffs are immediately accessible in the provided state $s(w_1^{n-1})$.

When our code walks the data structure to find w_f^n, it visits $w_n^n, w_{n-1}^n, \ldots, w_f^n$. Each visited entry w_i^n stores backoff $b(w_i^n)$. These are written to the state $s(w_1^n)$ and returned so that they can be used for the following query.

Saving state allows our code to walk the data structure exactly once per query. Other packages walk their respective data structures once to find w_f^n and again to find $\left\{b(w_i^{n-1})\right\}_{i=1}^{f-1}$ if necessary. In both cases, SRILM walks its trie an additional time to minimize context as mentioned in Section 4.1.

BerkeleyLM uses states to optimistically search for longer n-gram matches first and must perform twice as many random accesses to retrieve backoff information. Further, it needs extra pointers in the trie, increasing model size by 40%. This makes memory usage comparable to our PROBING model. The PROBING model can perform optimistic searches by jumping to any n-gram without needing state and without any additional memory. However, this optimistic search would not visit the entries necessary to store backoff information in the outgoing state. Though we do not directly compare state implementations, performance metrics in Table 1 indicate our overall method is faster.

4.3 Threading

Only IRSTLM does not support threading. In our case multi-threading is trivial because our data structures are read-only and uncached. Memory mapping also allows the same model to be shared across processes on the same machine.

4.4 Memory Mapping

Along with IRSTLM and TPT, our binary format is memory mapped, meaning the file and in-memory representation are the same. This is especially effective at reducing load time, since raw bytes are read directly to memory—or, as happens with repeatedly used models, are already in the disk cache.

Lazy mapping reduces memory requirements by loading pages from disk only as necessary. However, lazy mapping is generally slow because queries against uncached pages must wait for the disk. This is especially bad with PROBING because it is based on hashing and performs random lookups, but it is not intended to be used in low-memory scenarios. TRIE uses less memory and has better locality. However, TRIE partitions storage by n-gram length, so walking the trie reads N disjoint pages. TPT has theoretically better locality because it stores n-grams near their suffixes, thereby placing reads for a single query in the same or adjacent pages.

We do not experiment with models larger than physical memory in this paper because TPT is unreleased, factors such as disk speed are hard to replicate, and in such situations we recommend switching to a more compact representation, such as RandLM. In all of our experiments, the binary file (whether mapped or, in the case of most other packages, interpreted) is loaded into the disk cache in advance so that lazy mapping will never fault to disk. This is similar to using the Linux `MAP_POPULATE`

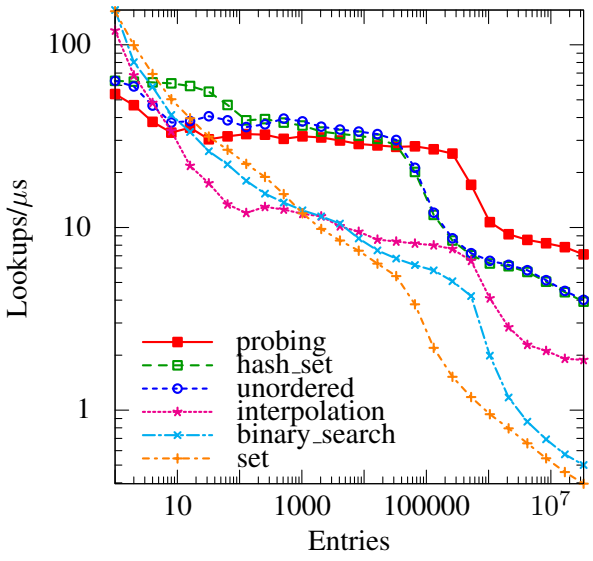

Figure 2: Speed in lookups per microsecond by data structure and number of 64-bit entries. Performance dips as each data structure outgrows the processor's 12 MB L2 cache. Among hash tables, indicated by shapes, probing is initially slower but converges to 43% faster than unordered or hash_set. Interpolation search has a more expensive pivot function but does less reads and iterations, so it is initially slower than binary_search and set, but becomes faster above 4096 entries.

flag that is our default loading mechanism.

5 Benchmarks

This section measures performance on shared tasks in order of increasing complexity: sparse lookups, evaluating perplexity of a large file, and translation with Moses. Our test machine has two Intel Xeon E5410 processors totaling eight cores, 32 GB RAM, and four Seagate Barracuda disks in software RAID 0 running Linux 2.6.18.

5.1 Sparse Lookup

Sparse lookup is a key subproblem of language model queries. We compare three hash tables: our probing implementation, GCC's hash_set, and Boost's[8] unordered. For sorted lookup, we compare interpolation search, standard C++ binary_search, and standard C++ set based on red-black trees. The data structure was populated with 64-bit integers sampled uniformly without replacement. For queries, we uniformly sampled 10 million hits and 10 million misses. The same numbers were used for each data structure. Time includes all queries but excludes random number generation and data structure population. Figure 2 shows timing results.

For the PROBING implementation, hash table sizes are in the millions, so the most relevant values are on the right size of the graph, where linear probing wins. It also uses less memory, with 8 bytes of overhead per entry (we store 16-byte entries with $m = 1.5$); linked list implementations hash_set and unordered require at least 8 bytes per entry for pointers. Further, the probing hash table does only one random lookup per query, explaining why it is faster on large data.

Interpolation search has a more expensive pivot but performs less pivoting and reads, so it is slow on small data and faster on large data. This suggests a strategy: run interpolation search until the range narrows to 4096 or fewer entries, then switch to binary_search. However, reads in the TRIE data structure are more expensive due to bit-level packing, so we found that it is faster to use interpolation search the entire time. Memory usage is the same as with binary_search and lower than with set.

5.2 Perplexity

For the perplexity and translation tasks, we used SRILM to build a 5-gram English language model on 834 million tokens from Europarl v6 (Koehn, 2005) and the 2011 Workshop on Machine Translation News Crawl corpus with duplicate lines removed. The model was built with open vocabulary, modified Kneser-Ney smoothing, and default pruning settings that remove singletons of order 3 and higher. Unlike Germann et al. (2009), we chose a model size so that all benchmarks fit comfortably in main memory. Benchmarks use the package's binary format; our code is also the fastest at building a binary file. As noted in Section 4.4, disk cache state is controlled by reading the entire binary file before each test begins. For RandLM, we used the settings in the documentation: 8 bits per value and false positive probability $\frac{1}{256}$.

We evaluate the time and memory consumption of each data structure by computing perplexity on 4 billion tokens from the English Gigaword corpus (Parker et al., 2009). Tokens were converted to vocabulary identifiers in advance and state was carried

[8] http://boost.org

from each query to the next. Table 1 shows results of the benchmark. Compared to decoding, this task is cache-unfriendly in that repeated queries happen only as they naturally occur in text. Therefore, performance is more closely tied to the underlying data structure than to the cache. In fact, we found that enabling IRSTLM's cache made it slightly slower, so results in Table 1 use IRSTLM without caching. Moses sets the cache size parameter to 50 so we did as well; the resulting cache size is 2.82 GB.

The results in Table 1 show PROBING is 81% faster than TRIE, which is in turn 31% faster than the fastest baseline. Memory usage in PROBING is high, though SRILM is even larger, so where memory is of concern we recommend using TRIE, if it fits in memory. For even larger models, we recommend RandLM; the memory consumption of the cache is not expected to grow with model size, and it has been reported to scale well. Another option is the closed-source data structures from Sheffield (Guthrie and Hepple, 2010). Though we are not able to calculate their memory usage on our model, results reported in their paper suggest lower memory consumption than TRIE on large-scale models, at the expense of CPU time.

5.3 Translation

This task measures how well each package performs in machine translation. We run the baseline Moses system for the French-English track of the 2011 Workshop on Machine Translation,[9] translating the 3003-sentence test set. Based on revision 4041, we modified Moses to print process statistics before terminating. Process statistics are already collected by the kernel (and printing them has no meaningful impact on performance). SRILM's compact variant has an incredibly expensive destructor, dwarfing the time it takes to perform translation, and so we also modified Moses to avoiding the destructor by calling `_exit` instead of returning normally. Since our destructor is an efficient call to `munmap`, bypassing the destructor favors only other packages. The binary language model from Section 5.2 and text phrase table were forced into disk cache before each run. Time starts when Moses is launched and therefore includes model loading time. These con-

[9] `http://statmt.org/wmt11/baseline.html`

Package	Variant	Queries/ms	RAM (GB)
Ken	PROBING	1818	5.28
	TRIE	1139	2.72
	TRIE 8 bits[a]	1127	1.59
SRI	Default	750	9.19
	Compact	238	7.27
IRST[b]	Invert	426	2.91
	Default	368	2.91
MIT	Default	410	7.72+1.34[c]
Rand	Backoff 8 bits[a]	56	1.30+2.82[c]
Berkeley	Hash+Scroll[a]	913	5.28+2.32[d]
	Hash[a]	767	3.71+1.72[d]
	Compressed[a]	126	1.73+0.71[d]
Estimates for unreleased packages			
Sheffield	C-MPHR[a]	607[e]	
TPT	Default	357[f]	

Table 1: Single-threaded speed and memory use on the perplexity task. The PROBING model is fastest by a substantial margin but generally uses more memory. TRIE is faster than competing packages and uses less memory than non-lossy competitors. The timing basis for Queries/ms includes kernel and user time but excludes loading time; we also subtracted time to run a program that just reads the query file. Peak virtual memory is reported; final resident memory is similar except for BerkeleyLM. We tried both aggressive reading and lazy memory mapping where applicable, but results were much the same.

[a] Uses lossy compression.

[b] The 8-bit quantized variant returned incorrect probabilities as explained in Section 3. It did 402 queries/ms using 1.80 GB.

[c] Memory use increased during scoring due to batch processing (MIT) or caching (Rand). The first value reports use immediately after loading while the second reports the increase during scoring.

[d] BerkeleyLM is written in Java which requires memory be specified in advance. Timing is based on plentiful memory. Then we ran binary search to determine the least amount of memory with which it would run. The first value reports resident size after loading; the second is the gap between post-loading resident memory and peak virtual memory. The developer explained that the loading process requires extra memory that it then frees.

[e] Based on the ratio to SRI's speed reported in Guthrie and Hepple (2010) under different conditions. Memory usage is likely much lower than ours.

[f] The original paper (Germann et al., 2009) provided only 2s of query timing and compared with SRI when it exceeded available RAM. The authors provided us with a ratio between TPT and SRI under different conditions.

Package	Variant	Time (m) CPU	Wall	RAM (GB) Res	Virt
Ken	PROBING-L	72.3	72.4	7.83	7.92
	PROBING-P	73.6	74.7	7.83	7.92
	TRIE-L	80.4	80.6	4.95	5.24
	TRIE-P	80.1	80.1	4.95	5.24
	TRIE-L 8[a]	79.5	79.5	3.97	4.10
	TRIE-P 8[a]	79.9	79.9	3.97	4.10
SRI	Default	85.9	86.1	11.90	11.94
	Compact	155.5	155.7	9.98	10.02
IRST	Cache-Invert-L	106.4	106.5	5.36	5.84
	Cache-Invert-R	106.7	106.9	5.73	5.84
	Invert-L	117.2	117.3	5.27	5.67
	Invert-R	117.7	118.0	5.64	5.67
	Default-L	126.3	126.4	5.26	5.67
	Default-R	127.1	127.3	5.64	5.67
Rand	Backoff[a]	277.9	278.0	4.05	4.18
	Backoff[b]	247.6	247.8	4.06	4.18

Table 2: Single-threaded time and memory consumption of Moses translating 3003 sentences. Where applicable, models were loaded with lazy memory mapping (-L), prefaulting (-P), and normal reading (-R); results differ by at most than 0.6 minute.

[a]Lossy compression with the same weights.
[b]Lossy compression with retuned weights.

Package	Variant	Time (m) CPU	Wall	RAM (GB) Res	Virt
Ken	PROBING-L	130.4	20.2	7.91	8.53
	PROBING-P	132.6	21.7	7.91	8.41
	TRIE-L	132.1	20.6	5.03	5.85
	TRIE-P	132.2	20.5	5.02	5.84
	TRIE-L 8[a]	137.1	21.2	4.03	4.60
	TRIE-P 8[a]	134.6	20.8	4.03	4.72
SRI	Default	153.2	26.0	11.97	12.56
	Compact	243.3	36.9	10.05	10.55
Rand	Backoff[a]	346.8	49.4	5.41	6.78
	Backoff[b]	308.7	44.4	5.26	6.81

Table 3: Multi-threaded time and memory consumption of Moses translating 3003 sentences on eight cores. Our code supports lazy memory mapping (-L) and prefaulting (-P) with `MAP_POPULATE`, the default. IRST is not threadsafe. Time for Moses itself to load, including loading the language model and phrase table, is included. Along with locking and background kernel operations such as prefaulting, this explains why wall time is not one-eighth that of the single-threaded case.

[a]Lossy compression with the same weights.
[b]Lossy compression with retuned weights.

ditions make the value appropriate for estimating repeated run times, such as in parameter tuning. Table 2 shows single-threaded results, mostly for comparison to IRSTLM, and Table 3 shows multi-threaded results.

Part of the gap between resident and virtual memory is due to the time at which data was collected. Statistics are printed before Moses exits and after parts of the decoder have been destroyed. Moses keeps language models and many other resources in static variables, so these are still resident in memory. Further, we report current resident memory and peak virtual memory because these are the most applicable statistics provided by the kernel.

Overall, language modeling significantly impacts decoder performance. In line with perplexity results from Table 1, the PROBING model is the fastest followed by TRIE, and subsequently other packages. We incur some additional memory cost due to storing state in each hypothesis, though this is minimal compared with the size of the model itself. The TRIE model continues to use the least memory of the non-lossy options. For RandLM and IRSTLM, the effect of caching can be seen on speed and memory usage. This is most severe with RandLM in the multi-threaded case, where each thread keeps a separate cache, exceeding the original model size. As noted for the perplexity task, we do not expect cache to grow substantially with model size, so RandLM remains a low-memory option. Caching for IRSTLM is smaller at 0.09 GB resident memory, though it supports only a single thread. The BerkeleyLM direct-mapped cache is in principle faster than caches implemented by RandLM and by IRSTLM, so we may write a C++ equivalent implementation as future work.

5.4 Comparison with RandLM

RandLM's stupid backoff variant stores counts instead of probabilities and backoffs. It also does not prune, so comparing to our pruned model would be unfair. Using RandLM and the documented settings (8-bit values and $\frac{1}{256}$ false-positive probability), we built a stupid backoff model on the same data as in Section 5.2. We used this data to build an unpruned ARPA file with IRSTLM's

Pack	Variant	Time (m)	RAM (GB) Res	Virt	BLEU
Ken	TRIE	82.9	12.16	14.39	27.24
	TRIE 8 bits	82.7	8.41	9.41	27.22
	TRIE 4 bits	83.2	7.74	8.55	27.09
Rand	Stupid 8 bits	218.7	5.07	5.18	25.54
	Backoff 8 bits	337.4	7.17	7.28	25.45

Table 4: CPU time, memory usage, and uncased BLEU (Papineni et al., 2002) score for single-threaded Moses translating the same test set. We ran each lossy model twice: once with specially-tuned weights and once with weights tuned using an exact model. The difference in BLEU was minor and we report the better result.

`improved-kneser-ney` option and the default three pieces. Table 4 shows the results. We elected run Moses single-threaded to minimize the impact of RandLM's cache on memory use. RandLM is the clear winner in RAM utilization, but is also slower and lower quality. However, the point of RandLM is to scale to even larger data, compensating for this loss in quality.

6 Future Work

There any many techniques for improving language model speed and reducing memory consumption. For speed, we plan to implement the direct-mapped cache from BerkeleyLM. Much could be done to further reduce memory consumption. Raj and Whittaker (2003) show that integers in a trie implementation can be compressed substantially. Quantization can be improved by jointly encoding probability and backoff. For even larger models, storing counts (Talbot and Osborne, 2007; Pauls and Klein, 2011; Guthrie and Hepple, 2010) is a possibility. Beyond optimizing the memory size of TRIE, there are alternative data structures such as those in Guthrie and Hepple (2010). Finally, other packages implement language model estimation while we are currently dependent on them to generate an ARPA file.

While we have minimized forward-looking state in Section 4.1, machine translation systems could also benefit by minimizing backward-looking state. For example, syntactic decoders (Koehn et al., 2007; Dyer et al., 2010; Li et al., 2009) perform dynamic programming parametrized by both backward- and forward-looking state. If they knew that the first four words in a hypergraph node would never extend to the left and form a 5-gram, then three or even fewer words could be kept in the backward state. This information is readily available in TRIE where adjacent records with equal pointers indicate no further extension of context is possible. Exposing this information to the decoder will lead to better hypothesis recombination. Generalizing state minimization, the model could also provide explicit bounds on probability for both backward and forward extension. This would result in better rest cost estimation and better pruning.[10] In general, tighter, but well factored, integration between the decoder and language model should produce a significant speed improvement.

7 Conclusion

We have described two data structures for language modeling that achieve substantial reductions in time and memory cost. The PROBING model is 2.4 times as fast as the fastest alternative, SRILM, and uses less memory too. The TRIE model uses less memory than the smallest lossless alternative and is still faster than SRILM. These performance gains transfer to improved system runtime performance; though we focused on Moses, our code is the best lossless option with cdec and Joshua. We attain these results using several optimizations: hashing, custom lookup tables, bit-level packing, and state for left-to-right query patterns. The code is open-source, has minimal dependencies, and offers both C++ and Java interfaces for integration.

Acknowledgments

Alon Lavie advised on this work. Hieu Hoang named the code "KenLM" and assisted with Moses along with Barry Haddow. Adam Pauls provided a pre-release comparison to BerkeleyLM and an initial Java interface. Nicola Bertoldi and Marcello Federico assisted with IRSTLM. Chris Dyer integrated the code into cdec. Juri Ganitkevitch answered questions about Joshua. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. 0750271 and by the DARPA GALE program.

[10] One issue is efficient retrieval of bounds, though these could be quantized, rounded in the safe direction, and stored with each record.

References

Philip Clarkson and Ronald Rosenfeld. 1997. Statistical language modeling using the CMU-Cambridge toolkit. In *Proceedings of Eurospeech*.

Chris Dyer, Adam Lopez, Juri Ganitkevitch, Johnathan Weese, Ferhan Ture, Phil Blunsom, Hendra Setiawan, Vladimir Eidelman, and Philip Resnik. 2010. cdec: A decoder, alignment, and learning framework for finite-state and context-free translation models. In *Proceedings of the ACL 2010 System Demonstrations*, pages 7–12.

Marcello Federico and Nicola Bertoldi. 2006. How many bits are needed to store probabilities for phrase-based translation? In *Proceedings of the Workshop on Statistical Machine Translation*, pages 94–101, New York City, June.

Marcello Federico, Nicola Bertoldi, and Mauro Cettolo. 2008. IRSTLM: an open source toolkit for handling large scale language models. In *Proceedings of Interspeech*, Brisbane, Australia.

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 NAACL HLT Workshop on Software Engineering, Testing, and Quality Assurance for Natural Language Processing*, pages 31–39, Boulder, Colorado.

David Guthrie and Mark Hepple. 2010. Storing the web in memory: Space efficient language models with constant time retrieval. In *Proceedings of EMNLP 2010*, Los Angeles, CA.

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

Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, Chris Dyer, Ondrej Bojar, Alexandra Constantin, and Evan Herbst. 2007. Moses: Open source toolkit for statistical machine translation. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, Prague, Czech Republic, June.

Philipp Koehn. 2005. Europarl: A parallel corpus for statistical machine translation. In *Proceedings of MT Summit*.

Zhifei Li, Chris Callison-Burch, Chris Dyer, Sanjeev Khudanpur, Lane Schwartz, Wren Thornton, Jonathan Weese, and Omar Zaidan. 2009. Joshua: An open source toolkit for parsing-based machine translation. In *Proceedings of the Fourth Workshop on Statistical Machine Translation*, pages 135–139, Athens, Greece, March. Association for Computational Linguistics.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: A method for automatic evalution of machine translation. In *Proceedings 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, PA, July.

Robert Parker, David Graff, Junbo Kong, Ke Chen, and Kazuaki Maeda. 2009. English gigaword fourth edition. LDC2009T13.

Adam Pauls and Dan Klein. 2011. Faster and smaller n-gram language models. In *Proceedings of ACL*, Portland, Oregon.

Yehoshua Perl, Alon Itai, and Haim Avni. 1978. Interpolation search—a log log N search. *Commun. ACM*, 21:550–553, July.

Bhiksha Raj and Ed Whittaker. 2003. Lossless compression of language model structure and word identifiers. In *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 388–391.

Andreas Stolcke. 2002. SRILM - an extensible language modeling toolkit. In *Proceedings of the Seventh International Conference on Spoken Language Processing*, pages 901–904.

David Talbot and Miles Osborne. 2007. Randomised language modelling for statistical machine translation. In *Proceedings of ACL*, pages 512–519, Prague, Czech Republic.