Pipeline-tolerant decoder

Jozef Mokry

Master of Science by Research
School of Informatics
University of Edinburgh

2016

Abstract

This work presents a modified version of the cube pruning decoding algorithm used for machine translation. Our *pipeline-tolerant* decoding algorithm allows multiple translation hypotheses to be scored at the same time. We demonstrate the benefits of scoring multiple hypotheses simultaneously using a pipeline for language model scoring. Our pipeline builds on top of previous work on memory prefetching and improves the speed of computing language model scores. We implemented our pipeline-tolerant decoder and the language model pipeline inside Moses and our benchmarks showed improvements in translation speed.

Acknowledgements

This work was supported in part by the EPSRC Centre for Doctoral Training in Data Science, funded by the UK Engineering and Physical Sciences Research Council (grant EP/L016427/1) and the University of Edinburgh.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Jozef Mokry*)

Contents

Chapter 1

Introduction

Machine translation (MT) is the field of building systems that can automatically translate from one language to another. Low latency translations are often needed in areas including high-frequency trading or for on-demand translation systems, such as Google Translate or the digital single market. However, machine translation systems are often very slow because at their core they run a time-consuming search [Koehn et al., 2003, Bahdanau et al., 2014, Yamada and Knight, 2001].

The high-level goal of this project was to modify an existing MT system to make this search faster. Faster searching algorithm can either simply save time or allow a more extensive search in the same amount of time. One possible way of making machine translation systems faster is by scoring multiple translation hypotheses simultaneously. This can be achieved with a *pipeline* into which hypotheses are inserted for scoring. Once the pipeline finishes computing the score, it notifies the system. The advantage of employing a pipeline is that it can decide to score multiple hypotheses at the same time. For example, scoring with a distributed language model involves sending network requests. If the pipeline is scoring multiple hypotheses at once then it can decide to batch several network requests together, and thus save latency. Similarly for neural language models [Bengio et al., 2003, Mikolov et al., 2010], to score one hypothesis a forward-propagation through a neural network is necessary, but a pipeline can decide to minibatch (merge) multiple forward-propagations of different hypotheses into a single forward-propagation. Green et al. [2014] claim that MT systems spend 50% time of the search time doing language model scoring. Also Kocberber et al. [2015] showed that *prefetching* can speed up hash-table querying by a factor of 4.3x. Considering these two results, we decided to showcase the power of employing a pipeline on scoring with a local hash-table based language model.

The search algorithm in machine translation systems is commonly called a decoder because it ‘decrypts’ from the language of the source sentence to the target language. Decoders iteratively explore translation hypotheses which partially translate the source sentence, score them and the high-scoring ones are expanded to translate more words of the source sentence. A hypothesis score consists of at least the translation model and language model probability. Decoders usually have many parameters such as how many hypotheses can be expanded in every iteration. These parameters allow machine translation systems to trade between speed and translation quality. The more extensive the search is, the better the translation quality[1].

This work started by studying the large code base of KenLM [Heafield, 2011], a popular toolkit for language model scoring. KenLM uses a custom linear probing hash table to store language model scores, but our experiment confirmed the efficiency of prefetching and showed speed-up by a factor of 3.2x for random key querying with 6 simultaneous prefetches in-flight. We also noticed that the speed-up from prefetching decreases with increasing number of threads, but with prefetching queries were still faster by a factor of 1.5x even with 16 threads. Next we implemented a pipeline and integrated it into KenLM. Our benchmarks showed that computing the language model score of a long piece of text (*perplexity*) can go faster by a factor of 1.35x in a single threaded setting. Finally, we integrated our pipeline into Moses and we observed speed-up by a factor which varied between 1.01x and 1.05x depending on the number of threads.

The structure of this thesis is as follows: Chapter 2 discuss relevant background for this work including details on hash tables, prefetching, language models and cube pruning decoding. We also describe KenLM and Moses, open-source toolkits for language modeling and machine translation. Chapter 3 consists of 3 subsections describing our work on prefetching hash table lookups, speeding up language model queries and integrating our language model pipeline into Moses. Each subsection is followed by description of the corresponding benchmark and results. We conclude our work and propose ideas for further investigation in Chapter 4.

[1]It is possible that a more extensive search finds hypotheses with higher score but the actual translation quality is lower. This problem is known as *fortuitous search error* and is common in neural machine translation [Bahdanau et al., 2014].

Chapter 2

Background

This thesis builds on previous work on prefetching in order to make statistical machine translation systems run faster. Section 2.1 describes the details of hash tables because they are the underlying data structure of language models used in machine translation. Section 2.2 discusses the details of prefetching, an important method for speeding up code with low spatial locality, such as hash table lookups. Section 2.3 talks about how the probability of text is modeled and why it is useful for machine translation. Details of KenLM [Heafield, 2011], a popular open-source toolkit for language modeling this project was working with and which influenced the design of our language model automaton, are also given. A description and pseudo-code for cube pruning decoding are in Section 2.4. This project extended the cube pruning algorithm inside Moses, an open-source statistical machine translation system, which is described in Section 2.4.1.

2.1 Hash tables

Hash tables are a common data structure used for storing mappings from keys to values. It consists of an array $\mathcal{A}$ for storing key-value pairs and a hashing function $h : key \to \{0, \ldots, |\mathcal{A}| - 1\}$ where $|\mathcal{A}|$ is the length of the array $\mathcal{A}$. Ideally, a key-value pair is stored at slot $h(key)$.

Since the space of possible keys is usually larger than $|\mathcal{A}|$, collisions between hashes of different keys are unavoidable. There are various strategies for where a value should be inserted if the slot chosen by $h(key)$ is already taken. One family of strategies is called *open addressing* [Peterson, 1957]. An open addressing strategy visits slots in a deterministic sequence until it finds an empty slot or concludes that the table is full. Upon lookup of the key, the same sequence is followed until the value is found in the table. If the sequence visits an empty slot before it finds the value, it concludes that the value is not present in the table.

One particular open addressing strategy is called *linear probing*. The linear probing strategy searches through the array in the sequence: $h(key), h(key)+\delta, h(key)+2\delta, \ldots$ wrapping around the end of the array appropriately. It is common to have $\delta = 1$. The main advantage of this strategy is that it provides temporal locality since the search sequence scans the memory sequentially. The disadvantage of linear probing is that it can become very slow if the hashing function does not distribute keys evenly. With an uneven hashing function, many keys are mapped to the same slot and the search sequence becomes longer, thus making the operations slower. This problem is commonly referred to as *clustering* and hash functions should be designed to avoid it.

An alternative to open addressing is *chaining*. Chaining strategies store in each array slot a pointer to a data structure (commonly a linked-list or a balanced tree) which stores all key-value pairs that were mapped to that slot. The advantages of chaining strategies are faster deletions and lower memory overhead for large entries. Although deletions are simpler with chaining strategies, a language model does not use the delete operation. On the other hand, chaining strategies make worse use of spatial locality, since at least two random accesses are always needed – one into array $\mathcal{A}$, one into data structure storing the key-value pairs and possibly more random accesses are needed to find the right pair inside the data structure.

The runtime cost of hash table operations depends on the *load factor* which is the fraction of slots that are taken. As more key-value pairs are inserted into the hash table, the load factor grows from 0 to 1 and the runtime cost of each operation slows down from $O(1)$ to $O(N)$. It is therefore common to resize the array and re-insert all the values in order to keep the load factor small. This is not necessary in language modeling since the number of entries is known before the hash table is created.

2.2 Prefetching

Prefetching is an important method for speeding up code with low spatial locality, such as hash table lookups. A prefetch instruction signals to the CPU that a particular memory location will be required in the near future and that the CPU should load it into cache. Prefetching provides performance gains because other code executes while a memory location is being fetched into cache. Once the execution needs the memory, it can access the data faster (assuming the data has not been evicted from cache).

Today's processors can reorder instructions themselves and do other work during a memory fetch. However, Chen et al. [2007] show that databases' hash-join operation performance improves by a factor of 2.0–2.9× with prefetching. Chen exploits independence between subsequent database tuples and compares performance gains from *group prefetching* [Chen et al., 2007] and *software-pipelined prefetching* [Allan et al., 1995], which are both prefetching schemes trying to increase the interval between a prefetch and a corresponding memory location visit.

Kocberber et al. [2015] introduce *Asynchronous memory chaining* (AMAC) which they show is a prefetching technique that is superior to both group prefetching and software pipelining. Those techniques need to execute expensive cleanup code when irregularities occur. An example of such irregularity is early termination of pointer chasing chain in a hash table. AMAC avoids these problems by keeping the full state of each in-flight memory access separate from other accesses. This for example allows to immediately start a new pointer chasing chain in hash tables or binary trees when one chain finishes.

2.3 Language models

A language model is simply a probability distribution over any sequence of words in a language. More formally, given a sequence of words $w_1, w_2, \ldots, w_n$ (denoted w_1^n from now on), a language model computes the probability $\Pr\left(w_1^n \mid \text{<s>}\right)$ where <s> is the beginning of sentence marker. By the product rule of probability, this can be expressed as

$$\Pr\left(w_1^n \mid \text{<s>}\right) = \Pr\left(w_1 \mid \text{<s>}\right) \prod_{i=2}^{n} \Pr\left(w_i \mid \text{<s>} w_1^{i-1}\right) \tag{2.1}$$

which is simply a product of probabilities of words given all their previous words. The most common language model, called the *n-gram* model, makes an independence assumption that probability of a word depends only on the previous $n-1$ words, i.e.

$$\Pr\left(w_i \mid w_0^{i-1}\right) = \Pr\left(w_i \mid w_{\max(0,\, n-i+1)}^{i-1}\right) \tag{2.2}$$

where w_0 is <s> to simplify the notation. With this assumption[1], a language model only needs to know the probability of at most (vocabulary size)^n different word sequences of length n, called *n-grams*.

In practice, we want a language model to be a probability distribution which reflects well how likely a word sequence is produced. This means that common phrases should have a higher probability than rare phrases, especially ungrammatical ones. For example, the model should give higher probability to *'the countries are'* and *'I am going home'* then to *'the country are'* and *'I am going house'*. Thus, a good language model helps machine translation systems to produce a more grammatical and fluent output.

Language models use a long piece of text (*training data*) to calculate the n-gram statistics using which they can compute probabilities of word sequences. This process is called *estimation* and a common estimation technique is the *Kneser-Ney smoothing* [Kneser and Ney, 1995].

A problem that language models need to be able to solve is how to handle n-grams that were not observed in the training data. Katz [1987] proposes a non-linear *back-off* procedure that uses scores $\rho(\cdot)$ and back-offs $b(\cdot)$ to recursively compute the probability of an n-gram as

$$\Pr\left(w_n \mid w_1^{n-1}\right) = \begin{cases} \rho\left(w_1^n\right) & \text{if } \rho\left(w_1^n\right) \text{ known} \\ b\left(w_1^{n-1}\right)\Pr\left(w_n \mid w_2^{n-1}\right) & \text{otherwise} \end{cases} \tag{2.3}$$

A simple interpretation of Equation 2.3 is that probability of an n-gram is the score of the longest matching m-gram with the appropriate back-offs charged as a penalty for not matching the full n-gram.

In this work, the scores $\rho(\cdot)$ and back-offs $b(\cdot)$ were always given as input when necessary and how these statistics are estimated was not our focus. However, the details of how these statistics are used was important for implementing the language model automaton in Section 3.2. For example, the probability of the 4-gram 'the red quick fox' would be

$$\Pr(\text{fox} \mid \text{the red quick}) = b(\text{the red quick})\, b(\text{red quick})\, \rho(\text{quick fox})$$

[1]A different type of language models, called neural language models, do not make the independence assumption

assuming that the n-grams 'the red quick fox' and 'red quick fox' were not seen in the training data, but 'quick fox' was. If the back-off $b(\text{the red quick})$ did not appear in the model either, then the back-off is 1.

2.3.1 KenLM

KenLM [Heafield, 2011] is an open-source language modeling toolkit that was extended in this project by adding a language model pipeline. It is therefore useful to describe the details of KenLM that motivated the design decisions taken in this project.

KenLM defines a language model *state* object that speeds up the process of calculating language model probability. A *state* object consists of *context words*, *back-offs* and *length*. The purpose of these is best explained with an example.

Suppose we are interested in $\Pr(\text{day} \mid \text{today is a sunny})$. Then the language model would take the word *day* and a state whose context words are *today*, *is*, *a* and *sunny*. The back-offs would be $b(\text{today is a sunny})$, $b(\text{is a sunny})$, $b(\text{a sunny})$ and $b(\text{sunny})$. The length is the number of context words which is the same as the number of back-offs. In this case the length is 4. KenLM computes the required probability by searching in pessimistic order for $\rho(\text{day}), \rho(\text{sunny day}), \rho(\text{a sunny day}), \rho(\text{is a sunny day})$ and $\rho(\text{today is a sunny day})$. Suppose the longest matching n-gram found was *a sunny day*. Then, following Katz's back-off model KenLM calculates

$$\Pr(\text{day} \mid \text{today is a sunny}) = b(\text{today is a sunny})\, b(\text{is a sunny})\, \rho(\text{a sunny day})$$

finding the necessary back-offs in the input state. Apart from calculating the probability, KenLM also produces a new state that can be used later as input state if we want to query for the probability $\Pr(\text{X} \mid \text{today is a sunny day})$ for any word X. In this case, the context words would be *a sunny day*. The words *today* and *is* would not be in the context words since if $\rho(\text{is a sunny day})$ was not found then $\rho(\text{is a sunny day X})$ will never be found for any word X. Finally, the back-offs in the new state $b(\text{a sunny day})$, $b(\text{sunny day})$, $b(day)$ are found when KenLM does the pessimistic search for scores $\rho(\text{day})$, $\rho(\text{sunny day})$, $\rho(\text{a sunny day})$ because it keeps the $\rho(\cdot)$ and $b(\cdot)$ for an n-gram together.

KenLM lets the user choose between two data structures for storing scores $\rho(\cdot)$ and back-offs $b(\cdot)$. One is a linear probing hash table optimized for lookup time and the other is a bit packed trie optimized for memory usage. This project extends the hash table variant with a prefetch pipeline.

2.4 Cube pruning decoding

In machine translation, decoding is the process of searching for the highest scoring translation of a given source sentence. In phrase based machine translation, it is not possible to search through all translations, score them and return the best one in acceptable running time[2]. One way to search for good translations is cube pruning decoding [Chiang, 2007]. It is an approximate search algorithm that does not guarantee finding the translation with the highest score, but runs in polynomial time and has parameters that let us trade between the running time and extent of the search. The more extensive the search is, the more translations are considered but the longer the search runs. Cube pruning decoding is a popular modification of stack based decoding [Germann et al., 2001].

Before explaining how cube pruning works, we need to define a few terms.

source span	contiguous range of words in the source sentence
target phrases	possible translations of a given source span
hypothesis	a translation of some number of words from the source sentence
compatibility	a source span is compatible with a hypothesis if the hypothesis has not translated any words in that source span
rectangle	for a given source span a rectangle contains a list of hypotheses that are compatible with that span, a fixed list of target phrases for that span (both lists are in decreasing score order) and one priority queue (described below)

Given a source sentence of length N, cube pruning decoder creates $N+1$ *stacks*. Indexing from zero, the i^{th} stack will be a container for the hypotheses that translate i words of the source sentence. Consequently, the last stack stores complete translations of the source sentence and the 0^{th} stack contains only the default empty hypothesis. Cube pruning iteratively fills these stacks with hypotheses, and at the end takes the highest scoring hypothesis from the last stack to be the final translation.

[2]Actually, an exact polynomial time algorithm does not exist unless P = NP [Udupa and Maji, 2006]

The pseudo-code for cube pruning is shown in Algorithm 2.1. The main for loop (line 1) runs over each stack. At the end of each iteration the stack at index *stackId* contains scored hypotheses translating *stackId* words of the source sentence. The loop starts by creating a map that lazily creates a rectangle for each span. Next we iterate over previous stacks. A hypothesis from the stack at index *prevStackId* can be extended to a hypothesis for the current stack only if there is a source span of length $(stackId - prevStackId)$ compatible with it. In such a case, it is added to the rectangle corresponding to this source span. Note that a single hypothesis can be assigned to multiple rectangles. Also, if we know that no target phrase can translate a source span of length greater than L, then we only need to process previous L stacks.

Once hypotheses are assigned to rectangles, rectangles sort their lists of hypotheses in decreasing score order (line 10). Moreover, each rectangle takes the highest scoring hypothesis and the highest scoring target phrase (both at index 0 in the hypothesis and target phrase list respectively) and creates a new hypothesis by appending the target phrase to the hypothesis. This hypothesis is then scored and pushed in the rectangle's priority queue which keeps the highest scoring hypothesis at the top.

Finally, each rectangle that contains at least one hypothesis and one target phrase is inserted into a top level priority queue on line 12. This priority queue will order rectangles by the score of the hypothesis that is on top in the priority queue inside each rectangle. Effectively we have a priority queue of priority queues. The last for loop (line 13) in each iteration pops the best rectangle, this rectangle will pop the best hypothesis from its queue and push new ones into it (see Algorithm 2.2). If the rectangle's queue is non-empty then the rectangle returns into the top level priority queue. Once *popLimit* hypotheses were added to current stack (or when the top level priority queue becomes empty), the algorithm proceeds by filling the next stack.

The PopBestAndPushNeighbours method in Algorithm 2.2 requires further explanation. First, when SortHyposAndPush first is invoked on a rectangle, it creates a new hypothesis from the best hypothesis and best target phrase from its lists of hypotheses and target phrases. This new hypothesis is then pushed into an empty priority queue of the rectangle. When PopBestAndPushNeighbours is invoked on a rectangle, the rectangle pops the best hypothesis from its queue. The *prevHypoId* and *tpId* are the indices into the lists of hypotheses and target phrases, pointing to the hypothesis and target phrase that created the popped hypothesis. The method then tries to create two new hypotheses (see Figure 2.1). One is created by incrementing the *prevHypoId* index

Algorithm 2.1 Cube pruning decoding

```
1:  for $stackId$ from 1 to $N$ do
2:      $rectangles \leftarrow$ MapFromSpanToRectangle()
3:      for $prevStackId$ from 0 to $stackId - 1$ do
4:          $spanLen \leftarrow stackId - prevStackId$
5:          for $hypo$ in $stacks[prevStackId]$.GetHypotheses() do
6:              for every source span $span$ of length $spanLen$ do
7:                  if $span$ compatible with $hypo$ then
8:                      $rectangles[span]$.AddHypothesis($hypo$)
9:      for $rect$ in $rectangles$ do
10:         $rect$.SortHyposAndPushFirst()
11:     $queue \leftarrow$ PriorityQueue()
12:     $queue$.InsertEachNonEmptyRectangle($rectangles$)
13:     for $pop$ from 1 to $popLimit$ do
14:         $bestRect \leftarrow queue$.PopBest()
15:         $bestHypo \leftarrow bestRect$.PopBestAndPushNeighbours()
16:         $stacks[stackId]$.AddHypothesis($bestHypo$)
17:         if not $bestRect.queue$.Empty() then
18:             $queue$.Insert($bestRect$)
```

Algorithm 2.2 PopBestAndPushNeighbours

```
1:  $hypo \leftarrow queue$.PopBest()
2:  $prevHypoId \leftarrow hypo$.GetPreviousHypoId()
3:  $tpId \leftarrow hypo$.GetTargetPhraseId()
4:  if $prevHypoId + 1 < hypos.length$ and not Seen($prevHypoId + 1, tpId$) then
5:      SetSeen($prevHypoId + 1, tpId$)
6:      $newHypo \leftarrow$CreateAndScore($hypos[prevHypoId + 1], tPhrases[tpId]$)
7:      $queue$.Insert($prevHypoId + 1, tpId, newHypo$)
8:  if $tpId + 1 < tPhrases.length$ and not Seen($prevHypoId, tpId + 1$) then
9:      SetSeen($prevHypoId, tpId + 1$)
10:     $newHypo \leftarrow$CreateAndScore($hypos[prevHypoId], tPhrases[tpId + 1]$)
11:     $queue$.Insert($prevHypoId, tpId + 1, newHypo$)
12: return $hypo$
```

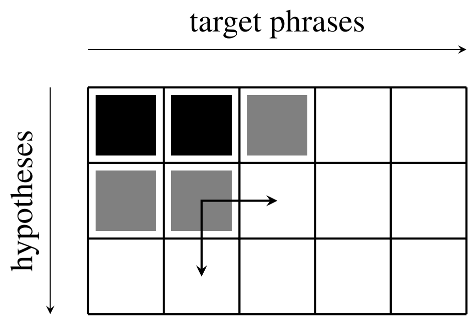

Figure 2.1: New hypothesis created from second hypothesis and second target phrase is popped and pushes its neighbours into the rectangle's queue. Black square means a hypothesis was already popped, gray means hypothesis is in the queue.

and the other by incrementing the *tpId* index. The purpose of the Seen method is to forbid recreating a hypothesis that was already created.

A hypothesis is scored when it is created on lines 6 and 10 in Algorithm 2.2. This includes the language model scoring. Brants et al. [2007] trained a large distributed language model trained on up to 2 trillion tokens. As the language model is distributed across multiple machines, each language model query incurs a network latency cost. Brants et al. reduce the latency per LM query by batching 1,000 to 10,000 LM queries into single network requests. They implement a decoder that tentatively extends all current hypotheses and re-prunes them after their scores are returned.

2.4.1 Moses

Moses [Koehn et al., 2007b] is an open-source statistical machine translation system that started as a project here at the University of Edinburgh. It supports both phrase-based [Koehn et al., 2003] and tree-based [Yamada and Knight, 2001] machine translation. For phrase based translation, Moses offers the standard stack decoding algorithm and its modification, cube pruning decoding. This work focused on extending the phrase-based cube pruning decoding system inside Moses described in Section 2.4.

Chapter 3

Completed work

3.1 Hash-table prefetching

As mentioned in Section 2.3.1, KenLM uses a custom linear probing hash table because it optimizes for lookup time and it knows all the keys in advance. This is a non-standard choice of hash table architecture and we decided to test whether the prefetching results from Section 2.2 still hold. Moreover, the test also gave us an upper bound on how much faster decoding with pipelined language model can be.

Our benchmark compares the time it takes to run a batch of hash table lookups with and without prefetching. Before presenting our results, we first describe what we mean by running a batch of hash table lookups with prefetching and why prefetching should provide a speed-up.

Consider a simple program in algorithm 3.1(a) which reads memory locations $m_{1\ldots n}$ and runs $before_i$ before reading each m_i and $after_i$ after reading each m_i. This code is a generalization of a program that runs a batch of hash table queries: $before_i$ is any computation required including hashing that must be done before each query, memory m_i is the location pointed to by the computed hash and $after_i$ is any computation required after each query. Assuming that all memory accesses in algorithm 3.1(a) for $m_{1\ldots n}$ are cache misses, each memory read incurs a time penalty $\mathcal{M}$ for reading from main memory. Overall, a penalty of $n\mathcal{M}$ is incurred.

However, if no computation $before_i$ depends on values computed by any computation $after_j$ $(j < i)$ then it is possible to reduce the penalty with reordering and prefetching

```
1: for $i \in 1,...,n$ do
2:     Run code $before_i$
3:     Read memory $m_i$
4:     Run code $after_i$
```

(a) Simple querying

```
1: for $i \in 1,...,k$ do
2:     Run code $before_i$
3:     Do prefetch $p_i$ for memory $m_i$
4: for $i \in 1,...,n-k$ do
5:     Read memory $m_i$
6:     Run code $after_i$
7:     Run code $before_{i+k}$
8:     Do prefetch $p_{i+k}$ for memory $m_i$
9: for $i \in 1,...,k$ do
10:    Read memory $m_{i+n-k}$
11:    Run code $after_{i+n-k}$
```

(b) Querying with prefetching

Alg. 3.1: Code snippet (a) runs a simple batch of queries. Code snippet (b) performs the same operations but in different order to allow prefetching.

as shown in algorithm 3.1(b) and in Figure 3.1. Algorithm 3.1(b) shows that we first prefetch the memory locations $m_{1...k}$ (first **for** loop). Only after the first k prefetches, we start reading the memory locations and we still prefetch the memory location read k iterations later (second **for** loop). In Section 3.1.1 we will refer to k as queue size.

Assuming k is small enough to avoid cache conflicts, all memory reads should hit cache instead of main memory. The total penalty changes from $n\mathcal{M}$ to $(n\mathcal{P} + n\mathcal{X})$ where $\mathcal{P}$ is the penalty incurred for 1 prefetch instruction and $\mathcal{X}$ for 1 cache read.

3.1.1 Implementation details

Our benchmark used a simple circular buffer that kept the state of each in-flight lookup. We will refer to it as *prefetch queue*. When a new key is added to the queue, the table looks up the old key stored in the current buffer slot and only prefetches the memory pointed to by the hash of the new key as described in Algorithm 3.2. Using a prefetch queue of size k achieves the same execution reordering as shown in Algorithm 3.1.

The prefetch queue was already implemented in KenLM in order to test prefetching gains in single threaded environment. Our work extended the benchmark, by making

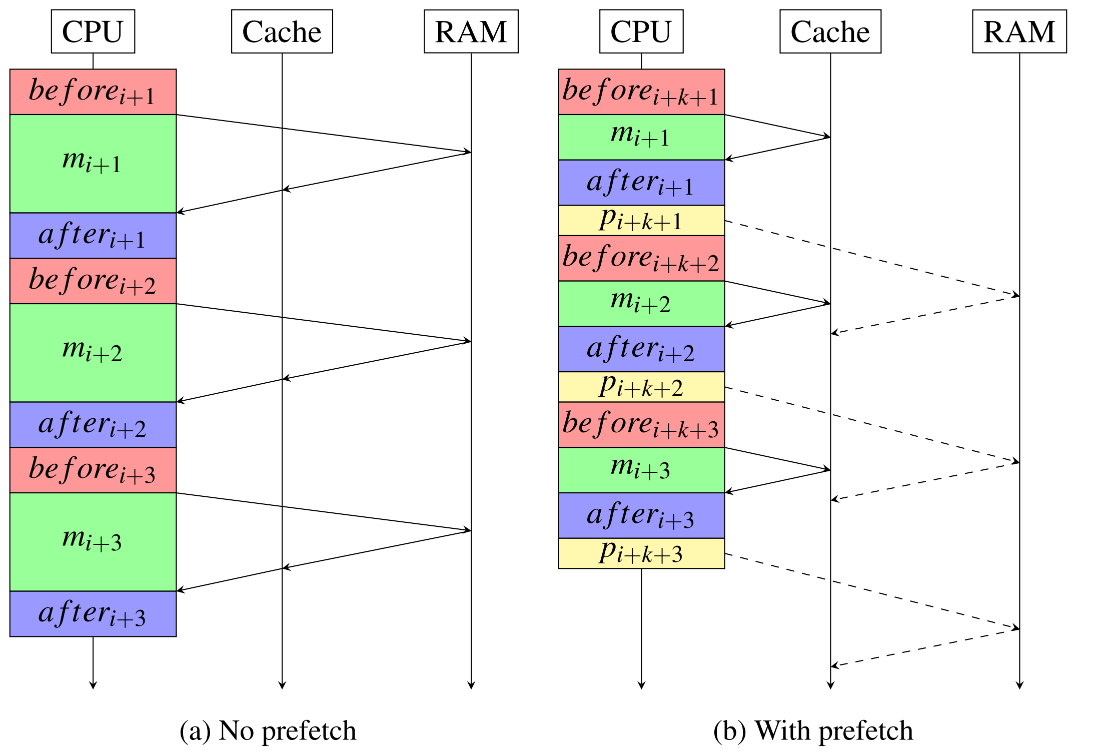

Figure 3.1: Sequence diagrams for algorithms 3.1(a) and 3.1(b). Sequence diagram (b) shows only the second **for** loop of algorithm 3.1(b) but we can see how the same number of memory reads m and computations $before$ and $after$ can be done in less time with prefetches p.

it multi-threaded. This required getting familiar with the existing code and creating a thread worker that could read a batch lookup request from a producer-consumer queue. Every worker had its own prefetch queue but they all queried the same table.

Algorithm 3.2 Prefetch queue

```
1: procedure ADD(newKey)
2:     Lookup(table, Curr().key)
3:     Curr().key = newKey
4:     Prefetch(hash(newKey))
5:     AdvanceCurr()
```

3.1.2 Experiment setup & Results

Our benchmark consisted of filling the hash table with random key entries and then querying the table directly and through a prefetch queue. We experimented with tables of different sizes (from 2 to 11.4B entries), different prefetch queue sizes (from 2 to 16)

# threads	speed-up	queue size	# threads	speed-up	queue size
1	3.21	6	9	1.82	6
2	2.17	8	10	1.83	8
3	2.01	8	11	1.70	6
4	2.06	4	12	1.78	8
5	2.16	9	13	1.60	7
6	2.54	8	14	2.48	5
7	2.06	7	15	1.64	4
8	1.91	7	16	1.54	6

Table 3.2: Speedups for different number of threads for 137GB hash-table. For every thread setting, we show the best speed-up and the size of the prefetch queue that achieved it.

and different number of threads (from 1 to 16). During each experiment the table was queried 2M times for random keys with 4k queries used as burn period to make sure memory from previous experiment is not cached. The load factor of the hash table was always $\frac{2}{3}$. Each experiment was repeated 3 times and only the fastest time was kept. That way, there is less chance that random fluctuation will impact the results. All benchmarks in this work were run on a server with 32 logical cores (Xeon E5-2680) and 297GB of memory.

The final results are presented in Table 3.2 and Figure 3.2. Table 3.2 shows for every thread setting the largest speedup achieved by querying with prefetching compared to without prefetching. We see that in a single threaded setting, a prefetch queue can speed up hash table lookups by a factor of 3.21x. This result confirms that the custom hash table implementation in KenLM can too benefit from prefetching gains.

The results also show that with increasing number of threads querying the hash table, the gains from prefetching decrease. This is because prefetch registers and memory bandwidth is shared across threads. However, even with 16 threads running lookups at the same time, we got a 1.5x speedup. Moreover, in machine translation threads are in different states and do not always read from memory at the same time. Finally, from Figure 3.2 we see that prefetching decreases look up time as the size of the table increases. We notice that there is practically no gain from prefetching if the size of the table was less or of the order of 10MB. This is unsurprising since the cache size of the machine used was 20MB.

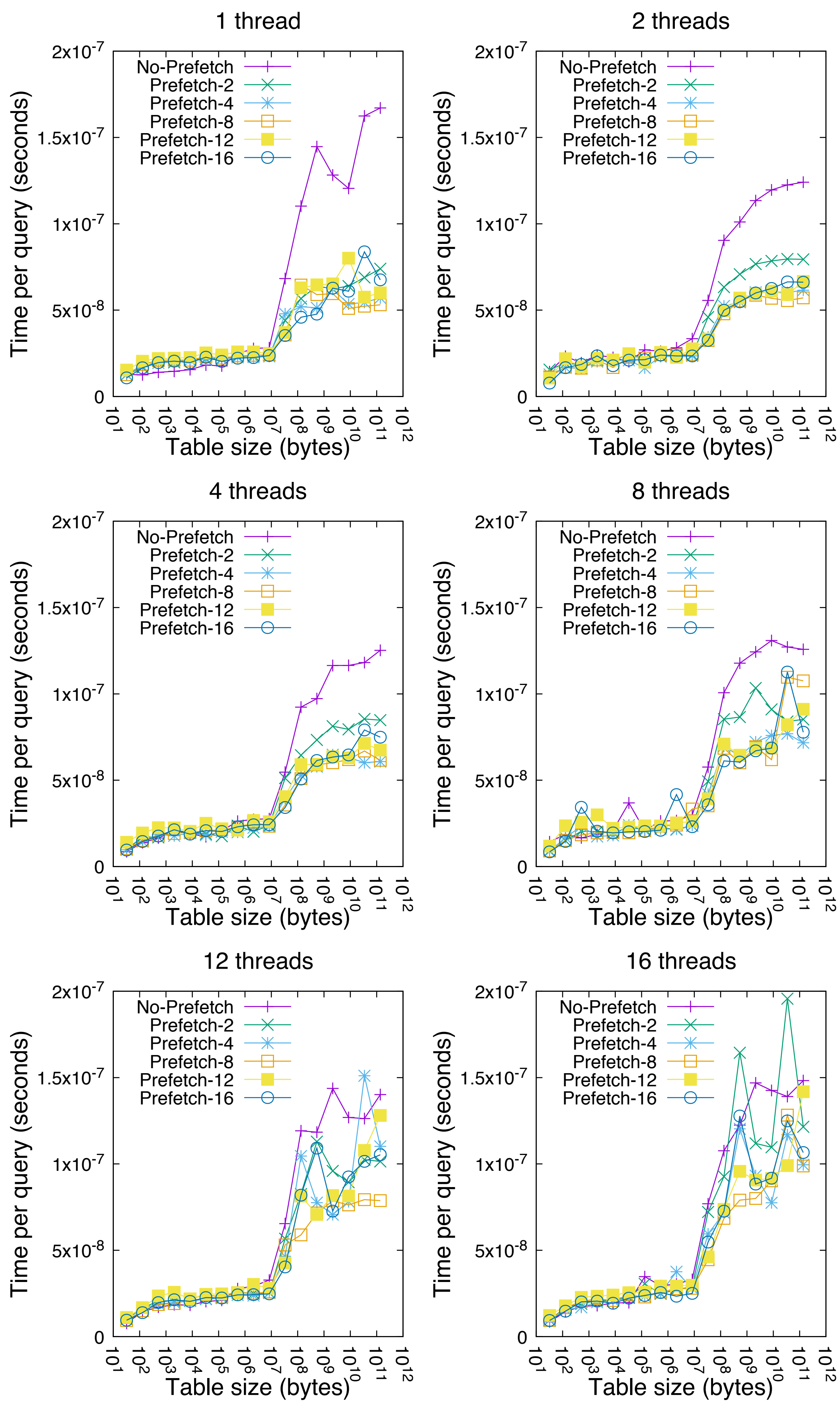

Figure 3.2: Time per look up for different number of in-flight prefetches

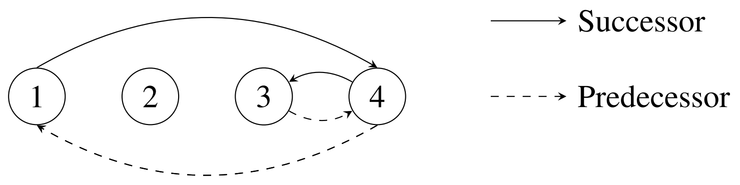

Figure 3.3: Pipeline with 4 automata: Automaton 3 does not have a successor because it handles the last query of a word sequence. Automaton 2 does not have a predecessor nor a successor because it handles the first query of a word sequence.

3.2 Language model automaton

Once prefetching gains were confirmed on KenLM custom hash table, the next step was to bring those gains to language model querying. Simple insertion of prefetch statements into KenLM query code would not give us any speed-up because memory is read almost immediately after the memory location is computed. A prefetch instruction would not have enough time to bring the memory into cache. To increase the time between a prefetch and the read, we decided to implement an *automaton queue*.

Our automaton queue is a circular buffer of automata, in which each automaton handles a separate query and has one in-flight memory access. This follows AMAC's separation of full state of each in-flight memory access from other accesses. When a new query arrives, the automaton queue is circularly traversed invoking STEP on each automaton. STEP performs a lookup for current n-gram and possibly prefetches the next $(n+1)$-gram. The traversal stops when STEP returns that the current automaton completed its query and this automaton will be assigned the new query.

As described in Section 2.3.1, in KenLM a language model query originally starts with a context state and a new word. However, with our automaton queue running multiple queries "simultaneously", the context state for a query might not be available at its start since the query responsible for computing that state is still running in the automaton queue. However, when computing the language model probability of a word sequence we submit the corresponding language model queries into the automaton queue in order. Hence, the query responsible for computing the context state for another query will be added to the queue just before the other query. We therefore solved the problem of missing context state by telling each automaton its predecessor, from which it can receive context state information. An automaton queue of 4 automata is illustrated in Figure 3.3.

The flow chart in Figure 3.4 shows how an automaton behaves. When an automaton is assigned a query to compute, say $\Pr\left(w_4 \mid w_1^3\right)$, it starts by prefetching the location of the unigram score $\rho\left(w_4\right)$. The unigram score is not looked up until STEP is invoked on the automaton. At that moment, automaton enters a loop in which each iteration is a n-gram lookup and $(n+1)$-gram prefetch. One invocation of STEP corresponds to one iteration of this loop. The loop ends when we know for sure that the $(n+1)$-gram will not be found. This can happen for 3 reasons:

1. The current n-gram was not found
2. There are no more context words
3. The current n-gram is flagged as independent left, i.e. no $(n+1)$-gram has it as suffix

When an automaton breaks out of the lookup-prefetch loop, it has to apply the back-offs. However, the back-offs are computed as part of the previous query. This is where the automaton's predecessor (which was assigned the previous query) becomes useful. If the predecessor has already completed its query, then it has handed the back-offs to this automaton. In such case, the current automaton simply applies the necessary back-offs. However, if predecessor is not finished with its own query, then it has not yet computed the necessary back-offs. In this case the automaton gives its state to its predecessor, so that once the predecessor completes, it can apply the back-offs.

The role of predecessor is not only to compute the back-offs for its successor, but to also tell its successor the number of context words. This is done by incrementing the successor's notion of this number after every successful lookup.

3.2.1 Experiment setup & Results

To benchmark the gains from prefetching for language modeling we prepared 6 KenLM language models of different sizes. The smallest model was trained on ~20K sentences (11MB) and the largest one on ~300M sentences (137GB). Next we prepared one test document with 1.33M sentences that were not used for training of the models. For each model we translated the test document words into vocabulary IDs in advance to avoid including the time for converting words to IDs in our results. Finally, our benchmark measured the time it takes to compute the perplexity of the test document.

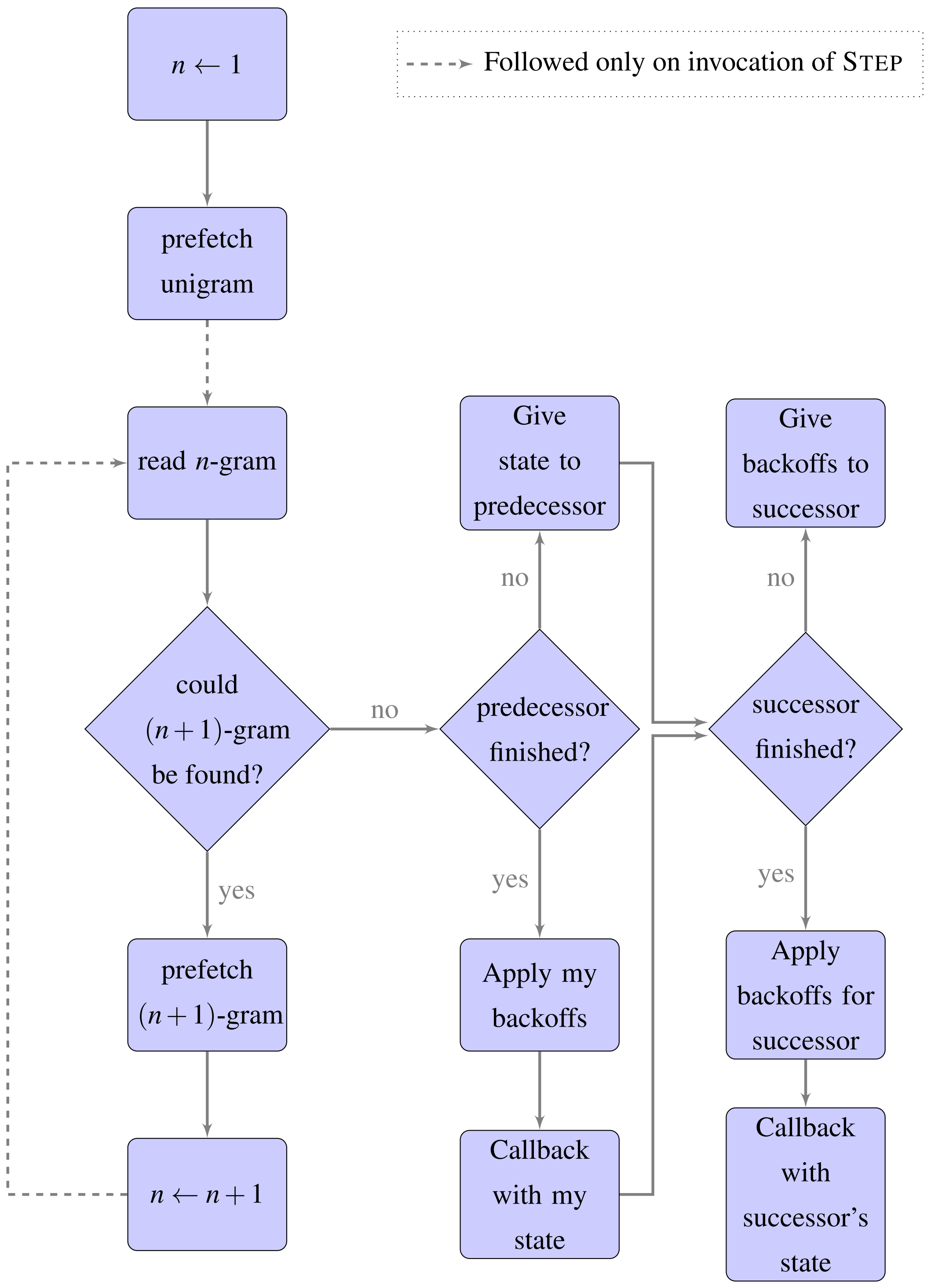

Figure 3.4: Automaton flow chart

Model size (MB)	speed-up	queue size
11	0.80	1
93	0.97	16
913	1.17	14
8300	1.06	12
73000	1.26	12
137000	1.35	11

Table 3.3: Table of best speed-up achieved for different language model sizes. Queue size is the size of the automaton queue that achieved the speed-up.

The experiment was run on the same machine as the one used for benchmarking hash table lookups in Section 3.1.2. Again, each experiment was repeated 3 times and the minimum time was used. Before each benchmark, the model file and the test document were printed to make sure they are loaded in memory. Huge pages were used for larger models.

We experimented with automaton queues of different sizes, from 1 to 16, which equals the number of in-flight memory accesses. All our experiments run in single threaded setting. In Table 3.3 we show the best speed-up achieved for each model size. Again we see that with increasing model size we were able to achieve greater speed-up from prefetching. Moreover, Figure 3.5 shows that increasing the number of prefetches above 8 does not change the running time. Unfortunately, the speed-ups were significantly smaller than the ones achieved for pure hash table lookups in Table 3.2. We believe that the reason for this is a more complicated control flow and thus memory accesses formed a smaller portion of the execution time in this experiment.

3.3 Pipeline-tolerant decoder in Moses

Once we had a working automaton queue, the final step was to integrate it into Moses cube pruning decoding algorithm. This cannot be done straightforwardly because when a hypothesis is scored, it is immediately pushed into a priority queue of a rectangle (see Algorithm 2.2). However, what makes the automaton queue faster, is that it has multiple in-flight memory accesses. Hence, we needed to modify the decoder such that it allows multiple hypotheses being scored at the same time.

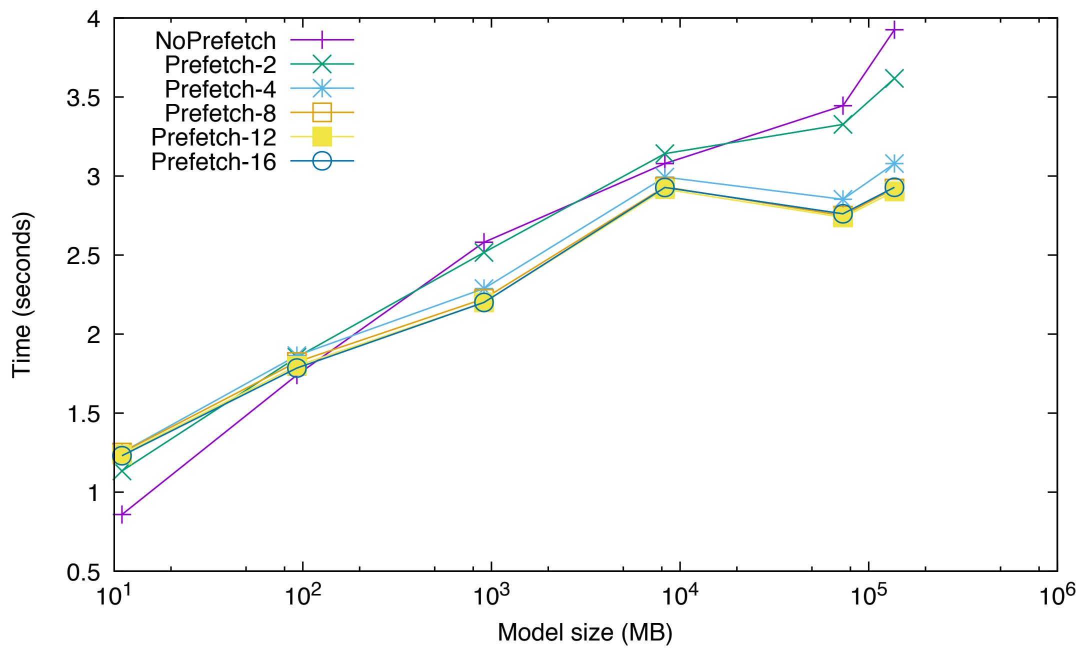

Figure 3.5: Time to compute perplexity of test document with different number of prefetches in-flight and language models of different sizes

We decided to follow the simplest approach and change Moses such that when a hypothesis should be scored, the language model feature functions are skipped. This of course makes the scores of hypotheses temporarily inaccurate. This inaccurate score was then used for ordering hypotheses inside a rectangle's priority queue. Once a hypothesis was popped from its rectangle's priority queue, instead of being added into the current stack (line 16 in Algorithm 2.1) it was added to the automaton queue. Inside the automaton queue, the language model score of the hypothesis was computed and through a callback the overall score of the hypothesis was corrected and the hypothesis was added to the stack. At the end, just before we started filling a new stack, we drained the automaton queue, forcing all automata to finish.

We noticed that although our implementation uses inaccurate scores when popping hypothesis from priority queues, in our experiments in Section 3.3.1, the BLEU score [Papineni et al., 2002] (which is a measure of translation quality) practically stayed the same (changed only from 0.3696 to 0.3692).

In decoding, the language model feature functions are not only responsible for computing the language model score, but also the language model state that can be used as context state in the next scoring session. However, to compute the language model score for a hypothesis, multiple queries might be generated and only the context state from the automaton handling the last query is important. The automaton handling the last query might not be the last one to finish and so it might not be the one invoking the callback. We solved this problem by giving automata handling queries for the same hypothesis a pointer to the same output array slot (same *session*). The automaton handling the last query saves the context state in the session. When an automaton completes, it decrements the counter inside its session and if the counter is zero then all queries of the hypothesis are completed and the automaton invokes callback with the context state.

3.3.1 Experiment setup & Results

To benchmark our modified decoder we used an existing machine translation system translating from Romanian to English. This system is practically the same as the system submitted by UEDIN-NMT team for WMT 2016 news-task [Sennrich et al., 2016, Bojar et al., 2016]. The only difference is that we did not use the neural features in their system because our test machine did not have GPUs. We used the same machine as we did in the previous experiments and again we repeated each experiment 3 times and kept only the shortest time to avoid random fluctuations affecting the result. The benchmark measured the time it takes to translate 999 sentences in the development set of the news task. In our timings we did not include the time it takes to setup and destroy all feature functions at the beginning and end of translation.

We experimented with number of threads and the size of the automaton queue and we present our results in Table 3.4. Unfortunately, our implementation was faster than the baseline version only by 3-6 seconds which corresponds to speedups of 1.01-1.05x. The speed-ups are lower than expected. It is possible that language model scoring might take less than 50% of decoding time. The translation system we used for benchmarking has 9 feature functions altogether, out of which only 2 are KenLM language model feature functions. Also, adding sessions to the automaton queue and changing Moses decoding could also decrease the achieved speed-ups. Figure 3.6 shows the CPU time for decoding with different number of threads and automaton queue sizes. Although the baseline implementation is only slightly slower, our version is consistently faster across different queue sizes.

# threads	original time	time with automaton queue	speed-up	queue size
1	635.32	629.14	1.01x	16
2	334.76	329.94	1.01x	10
4	180.82	175.59	1.03x	8
6	131.68	127.14	1.04x	4
8	107.27	103.40	1.04x	6
16	67.59	64.38	1.05x	4
24	62.10	59.37	1.05x	10

Table 3.4: Decoding real time (in seconds) for different number of threads setting. Time with automaton queue is the best achieved time across queue size setting.

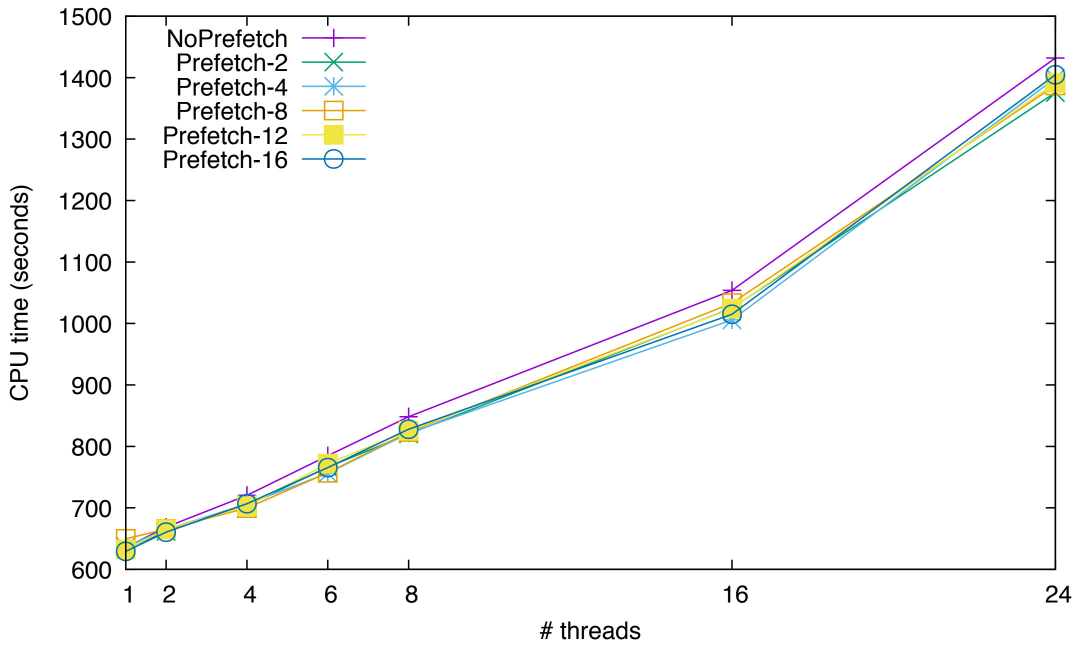

Figure 3.6: CPU time for decoding with different number of threads and automaton queue sizes

Chapter 4

Conclusions & Future work

Our goal in this work was to combine previous work on prefetching and decoding to make machine translation systems run faster. We started by improving the lookup performance of the custom hash table implementation in KenLM. We were able to achieve 1.5-3.2x speed-ups depending on the number of threads we used. Next we moved from simple hash table lookups, to a more challenging task of speeding up language model scoring. There we only succeeded to obtain a modest speed-up of 1.35x. We attributed this decrease to more complex code path with smaller portion of time spent in memory accesses. Finally, we integrated our automaton queue into Moses and benchmarked performance improvements on a Romanian to English system. Our implementation showed 1.01-1.05x speed-ups over standard Moses. Although, the overall speed-up was lower than expected, such speed-ups can be very important in fields such as high-frequency trading.

As future work, we propose investigating other possible ways of integrating of pipelines into decoders. In our work, hypotheses entered the pipeline after they were popped from the priority queue. Another option is to insert the hypotheses into the pipeline before they are pushed into the priority queue. This would avoid usage of inaccurate scores inside the priority queue and could lead to better results. Another important direction of study in this area is to improve the data structures used for language modeling. Currently, hash tables have very low spatial locality and this is why data needs to be prefetched in software. It would be interesting to see if a better data structure can be used, one that would have better spatial locality of memory accesses and would thus remove the need for prefetching.

Bibliography

Vicki H Allan, Reese B Jones, Randall M Lee, and Stephen J Allan. Software pipelining. *ACM Computing Surveys (CSUR)*, 27(3):367–432, 1995.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *journal of machine learning research*, 3(Feb):1137–1155, 2003.

Ondrej Bojar, Rajen Chatterjee, Christian Federmann, Yvette Graham, Barry Haddow, Matthias Huck, Antonio Jimeno Yepes, Philipp Koehn, Varvara Logacheva, Christof Monz, et al. Findings of the 2016 conference on machine translation (wmt16). 2016.

Thorsten Brants, Ashok C. Popat, Peng Xu, Franz J. Och, and Jeffrey Dean. Large language models in machine translation. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Language Learning*, pages 858–867, June 2007. URL `www.aclweb.org/anthology/D07-1090.pdf`.

Shimin Chen, Anastassia Ailamaki, Phillip B Gibbons, and Todd C Mowry. Improving hash join performance through prefetching. *ACM Transactions on Database Systems (TODS)*, 32(3):17, 2007.

David Chiang. Hierarchical phrase-based translation. *computational linguistics*, 33 (2):201–228, 2007.

Ulrich Germann, Michael Jahr, Kevin Knight, Daniel Marcu, and Kenji Yamada. Fast decoding and optimal decoding for machine translation. In *Proceedings of the 39th Annual Meeting on Association for Computational Linguistics*, pages 228–235. Association for Computational Linguistics, 2001.

Spence Green, Daniel Cer, and Christopher Manning. Phrasal: A toolkit for new directions in statistical machine translation. In *Proceedings of the Ninth Workshop on Statistical Machine Translation*, pages 114–121, Baltimore, Maryland, USA, June 2014. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/W/W14/W14-3311`.

Kenneth Heafield. KenLM: faster and smaller language model queries. In *Proceedings of the EMNLP 2011 Sixth Workshop on Statistical Machine Translation*, pages 187–197, Edinburgh, Scotland, United Kingdom, July 2011. URL `http://kheafield.com/professional/avenue/kenlm.pdf`.

Slava M Katz. Estimation of probabilities from sparse data for the language model component of a speech recognizer. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 35(3):400–401, 1987.

Reinhard Kneser and Hermann Ney. Improved backing-off for m-gram language modeling. In *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, volume 1, pages 181–184. IEEE, 1995.

Onur Kocberber, Babak Falsafi, and Boris Grot. Asynchronous memory access chaining. *Proceedings of the VLDB Endowment*, 9(4):252–263, 2015.

Philipp Koehn, Franz Josef Och, and Daniel Marcu. Statistical phrase-based translation. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1*, pages 48–54. Association for Computational Linguistics, 2003.

Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, et al. Moses: Open source toolkit for statistical machine translation. In *Proceedings of the 45th annual meeting of the ACL on interactive poster and demonstration sessions*, pages 177–180. Association for Computational Linguistics, 2007a.

Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, et al. Moses: Open source toolkit for statistical machine translation. In *Proceedings of the 45th annual meeting of the ACL on interactive poster and demonstration sessions*, pages 177–180. Association for Computational Linguistics, 2007b.

Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Interspeech*, volume 2, page 3, 2010.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318. Association for Computational Linguistics, 2002.

W Wesley Peterson. Addressing for random-access storage. *IBM journal of Research and Development*, 1(2):130–146, 1957.

Rico Sennrich, Barry Haddow, and Alexandra Birch. Edinburgh neural machine translation systems for wmt 16. *arXiv preprint arXiv:1606.02891*, 2016.

Raghavendra Udupa and Hemanta Kumar Maji. Computational complexity of statistical machine translation. In *EACL*, 2006.

Kenji Yamada and Kevin Knight. A syntax-based statistical translation model. In *Proceedings of the 39th Annual Meeting on Association for Computational Linguistics*, pages 523–530. Association for Computational Linguistics, 2001.