REMEDI

Robust and Efficient Machine Translation in a Distributed Infrastructure

Bi-Annual Report: Month 6

Principal Investigator:

Dr. Christof Monz
Informatics Institute
University of Amsterdam
Science Park 904
1098 XH Amsterdam

Phone: +39 (0)20 525 8676
Fax: +39 (0)20 525 7940
E-mail: c.monz@uva.nl

Scientific Programmer:

Dr. Ivan Zapreev
Informatics Institute
University of Amsterdam
Science Park 904
1098 XH Amsterdam

E-mail: i.zapreev@uva.nl

Month	Deliverable	Progress
6	Implementation of language model data structure; implementation of multi-threading with large shared data structures; month 6 report	✓
12	Implementation of translation and reordering model data structures; implementation of decoder search and pruning strategies; month 12 report	
18	Implementation of search lattice data structure and feature passing; implementation of server front end; implementation of robustness preserving load balancing and restarting; complete software deliverable of decoder infrastructure; month 18 report	
24	Implementation of language model server in distributed environment; implementation of distributed translation model service; month 24 report	
30	Implementation of distributed reordering model services; complete software deliverable of distributed translation infrastructure; Implementation of k-best hypothesis extraction; month 30 report	
36	Implementation of Margin Infused Relaxed Algorithm for parameter tuning; implementation of distributed optimization infrastructure, including hyper-parameter estimation; final report; complete manual	

Contents

1 Introduction

For machine translation it is important to estimate and compare the fluency of different possible translation outputs for the same source (i.e., foreign) sentence. This is commonly achieved by using a language model, which measures the probability of a string (which is commonly a sentence). Since entire sentences are unlikely to occur more than once, this is often approximated by using sliding windows of words (n-grams) occurring in some training data.

An *n-gram* refers to a continuous sequence of n tokens. For instance, given the following sentence: "our neighbor , who moved in recently , came by ". If n = 3, then the possible n-grams of this sentence include: "`our neighbor ,`", "`neighbor , who`", "`, who moved`", ... , "`, came by`", and "`came by .`". Note that, punctuation marks such as comma and full stop are treated just like any 'real' word and that all words are lower cased.

Information about known n-grams together with their frequencies (probabilities) and back-off weights form a language model that is used in the process of statistical machine translation to obtain the probabilities of n-grams. The latter is used to evaluate the likelihood of the given translation variant. In software implementations, language models are often stored in data structures called Tries. The language models' implementation, supplied with this deliverable, was inspired by the following two papers:

1. "Faster and Smaller N-Gram Language Models" by Pauls and Klein (2011)

2. "Efficient In-memory Data Structures for n-grams Indexing" by Robenek et al. (2013)

The first paper discusses optimal Trie structures for storing the learned text corpus and the second indicates that using *unordered_map* of C++ delivers one of the best time and space performances, compared to other data structures, when using Trie implementations.

Since the language model implementation presented in this project deliverable is to be competitive with the other available tools, we have also looked at the two other major tools: SRILM (Stolcke et al., 2011) and KenLM (Heafield, 2011). Both of these tools are use for experimental comparison, and KenLM was also used for exploring the source code internals.

The rest of the report is organized as follows. Section 2 describes the structure of the deliverable. Section 3 provides all necessary details about the developed software such as: availability, supported platforms, directory structure, and code details. Further in Section 4, we present an experimental comparison of the implemented tries and the two aforementioned tools SRILM and KenLM. Section 5 concludes and indicates some future work directions.

2 Software deliverables

This deliverable consists of this report as well as software deliverables. Both are distributed as a downloadable archive with the following standard structure that will be re-used for the subsequent deliveries:

- **REMEDI/month-6/data/** - stores example models and query files
- **REMEDI/month-6/software/** - stores the software project
- **REMEDI/month-6/report/** - stores this document

Next, we provide some details on the structure of the software part of the delivery. The software components are located in `REMEDI/month-6/software/`, and is accompanied by a short markdown Gruber (2015) file README.md explaining how to build and run it. Similar, but extended, information is also provided in Section 3 of this document. The delivered software is a standard Netbeans 8.0.2 C++ project, and its' top-level structure is as follows:

doc/ - contains the project-related documents

inc/ - stores the C++ header files used in the implementation

src/ - stores the C++ source files used in the implementation

nbproject/ - stores the Netbeans project data, such as makefiles

doxygen/ - stores the Doxygen-generated code documentation

`LICENSE` - the code license (GPL 2.0)

`Makefile` - the Makefile used to build the project

`README.md` - the project information document

`Doxyfile` - the Doxygen configuration file

`make_centos.sh` - a script for building on Centos platform

`make_debug_centos.sh` - a script for the debug build on Centos platform

`make_profile_centos.sh` - a script for the profiling build on Centos platform

`make_release_centos.sh` - a script for the release build on Centos platform

3 Software details

In this section, we discuss the main aspects of the delivered software such as licensing, building, and running it as well as present several details about the code structure and mention several important implementation aspects. This is an open source free distributable software that is also available as an open github project storing the actual project version:

```
https://github.com/ivan-zapreev/Back-Off-Language-Model-SMT
```

The rest of this section is organized as follows: Section 3.1 stores the licensing information. In Section 3.2, we talk about the supported platforms. Building and running the project is described in Sections 3.3 and 3.4 respectively. Section 3.5 talks about the main elements of the code structure, including the implementation details of the supported trie variants.

3.1 License

The delivered program is a free software, allowed to be redistributed and modified under the terms of the GNU General Public License, as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

A copy of the GNU General Public License is distributed along with this program, it is also available at `http://www.gnu.org/licenses/`.

3.2 Platforms

Currently this project supports two major platforms: Linux and Mac Os X. It has been successfully build and tested on:

- **Centos 6.6 64-bit** - Complete functionality.
- **Ubuntu 15.04 64-bit** - Complete functionality.
- **Mac OS X Yosemite 10.10 64-bit** - Limited by inability to collect memory-usage statistics.

Software testing was mostly done on 64-bit systems. Testing on 32-bit systems was limited. Some trie types might not support 32 bit machines. In this case an error message shall be reported by the software at run time.

3.3 Building

The delivered software is a standard Netbeans 8.0.2 C++ project, and building it requires gcc version 4.9.1 and higher. Managing the project, including building and cleaning, can be done in a number of ways, some of which are listed below:

Building from Netbeans Building from Netbeans is done in the regular manner by running the Build in the IDE. The binary will be generated and placed into `./dist/[CONFIG]_[PLATFORM]/` folder, and the name of the executable is `back-off-language-model-smt`. Where `[CONFIG]` is the chosen available configuration: {Release, Debug, Profile}; and `[PLATFORM]` is the chosen available platforms: {MacOs, Linux, Centos}.

Building from console Building from Linux console is also possible. The simplest way to do that is just running:

```
% make all
```

from console. This will attempt to build the code for all available configurations on all supported platforms. In case this is not desired, but building from console is still needed, one can employ the following command pattern to build a specific software configuration for a specific platform:

```
% make -f nbproject/Makefile-[CONFIG]__[PLATFORM]_.mk \
        dist/[CONFIG]__[PLATFORM]_/back-off-language-model-smt
```

Building on Centos For the sake of simplicity and speed, building on Centos platform is facilitated by using the `make_*_centos.sh` build scripts. There are four of such scripts available:

1. `make_debug_centos.sh` - builds the debug configuration: ready to be used with **Valgrind** Armour-Brown et al. (2000) and **GDB** Free Software Foundation (2015).
2. `make_profile_centos.sh` - builds the profiling configuration: ready to be used for gathering performance information for **GProf** Free Software Foundation (1988): produce the `gmon.out` file.
3. `make_release_centos.sh` - builds the release configuration: the version build for performance and includes platform specific optimization's.
4. `make_centos.sh` - builds all the above configurations at once.

Cleaning the project from console In order to clean the project from the command line run:

```
% make clean
```

Changing the logging levels @ compile time The software is equipped with an extended range of logging levels: { ERROR, WARNING, USAGE, RESULT, INFO, INFO1, INFO2, INFO3, DEBUG, DEBUG1, DEBUG2, DEBUG3, DEBUG4 }. By default, and for the sake of performance efficiency, the maximum available logging level is set to be `INFO3`. However, it is possible to change the maximum available run-time logging level @ compile time. This is done by changing the value of the `LOGER_MAX_LEVEL` constant in `./inc/Configuration.hpp`. Further after the program is build, at run-time it is only possible to change the logging level in the range between `ERROR` and the value of `LOGER_MAX_LEVEL`, see Section 3.4 for more details.

3.4 Running

Out software is currently implemented as a stand-alone single threaded command-line application accepting the language model and the input queries as two input files. The programm is therefore is to be run from the terminal, and to get its detailed usage information please run:

```
.dist/[DONFIG]_[PLATFORM]/back-off-language-model-smt
```

The program's output contains all the needed version and usage information:

```
$ ./dist/Release__MacOs_/back-off-language-model-smt
USAGE:  ------------------------------------------------------------------
USAGE: |                 Back Off Language Model(s) for SMT     :)\___/(: |
USAGE: |                      Software version 1.0              {(@)v(@)} |
USAGE: |                        The Owl release.                {|~- -~|} |
USAGE: |             Copyright (C) Dr. Ivan S Zapreev, 2015     {/^'^'^\} |
USAGE: |  ==========================================================m-m==  |
USAGE: |        This software is distributed under GPL 2.0 license        |
USAGE: |          (GPL stands for GNU General Public License)             |
USAGE: |          The product comes with ABSOLUTELY NO WARRANTY.           |
USAGE: |   This is a free software, you are welcome to redistribute it.    |
USAGE: |                     Running in 64 bit mode!                       |
USAGE: |                 Build on: Sep 21 2015 17:26:44                    |
USAGE:  ------------------------------------------------------------------
ERROR: Incorrect number of arguments, expected >= 3, got 0
USAGE: Running:
USAGE:   back-off-language-model-smt <model_file> <test_file> <trie_type> [debug-level]
USAGE:       <model_file> - a text file containing the back-off language model.
USAGE:        This file is supposed to be in ARPA format, see:
USAGE:      http://www.speech.sri.com/projects/srilm/manpages/ngram-format.5.html
USAGE:        for more details. We also allow doe tags listed here:
USAGE:      https://msdn.microsoft.com/en-us/library/office/hh378460%28v=office.14%29.aspx
USAGE:       <test_file>  - a text file containing test data.
USAGE:        The test file consists of a number of N-grams,
USAGE:        where each line in the file consists of one N-gram.
USAGE:       <trie_type>  - the trie type, one of {c2dm, w2ch, c2wa, w2ca, c2dh, g2dm}
USAGE:      [debug-level] - the optional debug flag from { ERROR, WARN, USAGE, RESULT, INFO,
                                                           INFO1, INFO2, INFO3, DEBUG,
                                                           DEBUG1, DEBUG2, DEBUG3, DEBUG4 }
USAGE: Output:
USAGE:     The program reads in the test queries from the <test_file>.
USAGE:     Each of these lines is a N-grams of the following form, e.g:
USAGE:        word1 word2 word3 word4 word5
USAGE:     For each of such N-grams the probability information is
USAGE:     computed, based on the data from the <model_file>. For
USAGE:     example, for a N-gram such as:
USAGE:        mortgages had lured borrowers and
USAGE:     the program may give the following output:
USAGE:  log_10( Prob( word5 | word1 word2 word3 word4 ) ) = <log-probability>
```

This output clearly shows and explains all the program's input parameters and options as well as its outputs. Note that, the default program's `[debug-level]` is `RESULT`. In order to get more information about the program run one can use higher debug levels, such as `INFO3`. The higher debug levels are available by setting the `LOGER_MAX_LEVEL` constant value in `./inc/Configuration.hpp` and re-compiling the software, see Section 3.3.

3.5 Code

The software part of this REMEDI deliverable originated from the Automated-Translation-Tries GitHub project made by the scientific developer as a test exercise for machine translation:

https://github.com/ivan-zapreev/Automated-Translation-Tries

Since that time, the project has significantly changed in terms of functionality and code structure. One of the new key code features of the project is using C++ template mechanisms for the sake of high run-time performance. Templating the code means static—compile time—resolution of a number of choices whereas standard programming, including object oriented programming (OOP), relies on run-time resolution. For example, in OOP virtual method calls are resolved at run time, meaning implicit overhead for each virtual member function call, whereas if one takes the template version of inheritance, then this overhead is eliminated by the compile-time function call resolution. Unfortunately, using templates does not make the code simpler or more readable. This is mitigated by means of providing extended code documentation with the project.

The remainder of this section consists of two parts. The short description of the main software classes given in Section 3.5.1 and the more detailed explanation of the various implemented trie variants, given in Section 3.5.2.

3.5.1 Main classes

Among others, the project code contains the following main source files:

- `main.cpp` - contains the entry point of the program.
- `Executor.cpp` - contains some utility functions including the ones reading the ARPA file and query input and performing the queries on a filled in Trie instance.
- `ARPATrieBuilder.hpp/ARPATrieBuilder.cpp` - contains the class responsible for reading the ARPA file and building up the trie model using the ARPAGramBuilder.
- `TrieDriver.hpp` - is the driver for all the provided trie implementations - allows to build up and query the tries, see Section 3.5.2.
- `LayeredTrieDriver.hpp` - is a wrapper driver for all the layered trie implementations - allows to retrieve N-gram probabilities and back-off weights, see Section 3.5.2.
- `C2DHashMapTrie.hpp/C2DHashMapTrie.cpp` - contains the Context-to-Data mapping trie implementation based on std::unordered_map.
- `C2DMapArrayTrie.hpp/C2DMapArrayTrie.cpp` - contains the Context-to-Data mapping trie implementation based on std::unordered_map and ordered arrays.
- `C2WOrderedArrayTrie.hpp/C2WOrderedArrayTrie.cpp` - contains the Context-to-Word mapping trie implementation based on ordered arrays.
- `G2DHashMapTrie.hpp/G2DHashMapTrie.cpp` - contains the M-Gram-to-Data mapping trie implementation based on self-made hash maps.
- `W2CHybridMemoryTrie.hpp/W2CHybridMemoryTrie.cpp` - contains the Word-to-Context mapping trie implementation based on std::unordered_map and ordered arrays.

- `W2COrderedArrayTrie.hpp/W2COrderedArrayTrie.cpp` - contains the Word-to-Context mapping trie implementation based on ordered arrays.
- `Configuration.hpp` - contains all of the available compile-time configuration parameters for the word indexes, tries, and memory management entities.
- `Exceptions.hpp` - stores the implementations of the used exception classes.
- `HashingUtils.hpp` - stores the hashing utility functions.
- `ARPAGramBuilder.hpp/ARPAGramBuilder.cpp` - contains the class responsible for building n-grams from a line of text and storing it into Trie.
- `StatisticsMonitor.hpp/StatisticsMonitor.cpp` - contains a class responsible for gathering memory and CPU usage statistics
- `Logger.hpp/Logger.cpp` - contains a basic logging facility class

3.5.2 Trie classes

In this section we sketch the implementation details and concepts of the Trie types provided by our software. There are two clearly distinct Trie sorts that we have:

Layered These tries are memory efficiency oriented and are targeted at reusing the m-gram information stored in trie levels as much as possible. The peculiarity of this trie type is that, when stored, each m-gram is spread over the trie layers. Each m-gram $w_1w_2w_3w_4 \dots w_m$ of some level $m > 1$ is then defined by its end word w_m and the $(m-1)$ gram $w_1w_2w_3w_4 \dots w_{m-1}$ called context. In this way there is a significant re-use of memory needed by the language model as each m-gram is a context for multiple $(m+1)$-grams. However, this kind of trie is not very performance efficient as is requires searching through the trie layers from 1 to $m-1$ in order to obtain the context information (the $w_1w_2w_3w_4 \dots w_{m-1}$ prefix) needed to identify the given m-gram $w_1w_2w_3w_4 \dots w_m$.

Generic These tries are not layered, i.e., each layer m contains all the needed information about the available m-grams and can be worked with directly and independently from other trie layers. This kind of trie can be very performance efficient as it allows for direct access to the m-gram information, without searching for the context, but will require significantly more memory for storing each m-gram. Each m-gram will require storing not only probabilities and back-off weights but also all the words or word ids that it consists of. Considering the language model sizes, this is a very significant amount of data.

Let us now provide some details about each of the implemented trie types, also specifying its trie sort and the way to invoke its within the tool.

C2DHashMapTrie.hpp/C2DHashMapTrie.cpp - Layered This trie type is available with the `c2dm` value of the `<trie_type>` option of the tool's command line parameter. It is a cross-layer context-to-data mapping trie implemented using the C++11 standard std::unordered_map. In the approach realized in this trie each m-grams gets a unique id c_m^g based on its context id c_{m-1}^g and its end word id c_m^w. The value of c_m^g is mapped, via the std::unordered_map, to the m-gram's data: probability and back-off weight. To be more concrete, for an m-gram $w_1 w_2 w_3 w_4 \dots w_m$ with $m > 1$, the m-gram id c_m^g is computed in the following way:

$$\begin{aligned} \forall i \in 1 \dots m : c_i^w = word\_id(w_i), \\ c_m^g = context\_id(c_{m-1}^g, c_m^w) = szudzik(c_{m-1}^g, c_m^w) \end{aligned} \tag{1}$$

Here $szudzik$ is the Elegant Pairing Function, Szudzik (2006).

The idea behind this trie implementation is that one can always efficiently compute a unique m-gram id (as $context\_id(c_{m-1}^g, c_m^w)$) by just following a number of inexpensive computational steps. Once the m-gram id is computed, one can retrieve the data associated with this id from the hash map with constant average time complexity. This is similar to a hash-based approach, when the m-gram is hashed to a hash value and the latter is then used as a key in a hash map. This hashing approach however is flawed by that the used hashing functions are not ideal, i.e. can cause hash collision: produce the same hash value for two or more different m-grams. The latter leads to improper probability estimates and as a result worsens the translation quality by introducing yet another error into this approximate process. The approach we took is better in that each m-gram gets a truly unique identifier thus eliminating improper m-gram identification.

Note that, in place of the Elegant pairing function a simple Cantor pairing function could be used, but the latter experiences a faster value growth. Yet this is a general problem of all the enumerating functions as they do enumerate all the possible pairs. Therefore with the increasing number of distinct words in the dictionary or large values of m, we get fast growing id values that can exceed the capacity of the `uint64_t` C++ type values, causing overflows and thus id collisions in the Trie. Note that the number of possible 5-grams with, e.g. 676750 unique words is 676750^5, which is a very large number. Still in our experiments on 5-gram tries with 676750 id collisions have not been detected, most likely due to overflows still resulting in smaller but not-used id values. So, this approach is theoretically better than hashing, as it ensures truly unique m-gram identifiers. Its overflow issues are caused by the limits on the computer-supported data types for storing ids. Switching to a larger data type, such as $uint128\_t$, is always possible but that will increase the memory consumption.

C2DMapArrayTrie.hpp/C2DMapArrayTrie.cpp - Layered This trie type is available with the `c2dh` value of the `<trie_type>` option of the tool's command line parameter. It is a cross-layer context-to-data mapping trie implemented using the C++11 std::unordered_map and arrays. In this trie we also use the concept of m-gram or context id. Each m-gram is identified by the pair of values (c_{m-1}^g, c_m^w) but unlike the previous approach the context id of an m-gram is not arithmetically computed but is issued by an id counter during filling up the trie with the grams. In other words:

$$\begin{aligned} \forall i \in 1 \dots m : c_i^w = word\_id(w_i), \\ c_m^g = context\_id(c_{m-1}^g, c_m^w) = id\_counters[m-2]++ \end{aligned} \tag{2}$$

Here for the trie model of maximum level $k \geq 0$, $id\_counters$ is an array of $k-2$ elements[1] storing the last issued m-gram id in $id\_counters[m-2]$. On each trie level $m > 1$ an m-gram information is stored using an std::unordered_map that relates a pair (c^g_{m-1}, c^w_m) to the m-gram's data: the issued m-gram id, its probability, and the back-off weight. Note that, here two $uint32\_t$ data type values are used for storing context and words id key pair[2], which requires the same amount of space as using one $uint64\_t$ type m-gram id key in the previously trie.

Unlike the previously discussed trie implementation, the advantage of this approach is that the m-gram id is safe against overflows as its value is as big as there are m-grams for the given level m. Yet, searching for an m-gram in this trie is more time consuming, as the id can not be computed numerically and requires a sequential searching through the trie levels. Still, the memory consumption of this trie is also higher than that of the previous one due to, at least, the following reason: Along with the back-off and probability data of an m-gram, we need to store the issued m-gram id. This requires is another 4 bytes per m-gram for each $1 < m < k$ where k is the maximum m-gram level in the trie.

W2CHybridMemoryTrie.hpp/W2CHybridMemoryTrie.cpp - Layered This trie type is available with the `w2ch` value of the `<trie_type>` option of the tool's command line parameter. It is a cross-layer word-to-context mapping trie implemented using the C++11 std::unordered_map and ordered arrays. Similar to the previous one, in this trie we also use the concept of m-gram or context ids. However, unlike the previously discussed tries, here we use word-to-context mappings to build up the trie structure. For each layer m, except for the first and the last one, we store a mapping from the m-gram end-word id c^w_m to the map from the previous context ids c^g_{m-1} to the next ones c^g_m. In mathematical terms we store mappings $\left(c^w_m \rightarrow \left(c^g_{m-1} \rightarrow c^g_m\right)\right)$. Here $(c^w_m \rightarrow (\ldots))$ is stored in an array indexed by the word id c^w_m, and $\left(c^g_{m-1} \rightarrow c^g_m\right)$ is stored in an instance of the std::unordered_map. As in the `c2dh` trie, each context id is issued during filling up the trie via a context id counter. The m-gram data is then stored in an array indexed by the m-gram (next context) id for each level m. Note that the number of context ids is known and is equal to the number of m-grams for every given level m. This trie implementation is supposed to be somewhat faster than that of `c2dh`, since it allows to get a sub-set mappings of $\left(c^g_{m-1} \rightarrow c^g_m\right)$ in a very fast way via the word id indexed array $(c^w_m \rightarrow (\ldots))$. Yet the memory consumption of this trie is expected to be even higher than that of the previous one. This is due to using a word id indexed array of pointers to std::unordered_map instances, where also not all the cells will be used. In addition, each std::unordered_map instance gives its own memory overhead related to the internal implementation specifics.

W2COrderedArrayTrie.hpp/W2COrderedArrayTrie.cpp - Layered This trie type is available with the `w2ca` value of the `<trie_type>` option of the tool's command line parameter. It is a cross-layer word-to-context mapping trie implemented using the ordered arrays. The foundation of this trie type is the same as that of the previous `w2ch`. The only major difference is

[1] Storing the context ids for the first and last levels are not needed as: the 1-gram id is equal to the word id; the k-gram id is not necessary as there is no next level referencing it.

[2] Actually, we encode each tuple (c^g_{m-1}, c^w_m) as a single $uint64\_t$ hashmap key

that instead of storing the hash-maps for the $\left(c^g_{m-1} \rightarrow c^g_m\right)$ mappings we use ordered arrays. To be more specific, for all m-grams ending with the same word w_m, the end word id c^w_m is mapped to an array of previous context ids c^g_{m-1} via an ordered array. The number of context id mappings for the words with smaller word ids than c^w_m plus the relative index of the context c^g_{m-1} within the mapping array give the id for the m-gram ending with the word w_m and the previous context id c^g_{m-1}. The purpose of this trie, compared to the previous one, is to further optimize memory usage. This again at the expense of searching within the ordered array of context ids when querying. Note that searching is done using the binary search, as using interpolated search showed worse results due to unevenly distributed search key values.

It is expected, that this trie will be more memory efficient than `w2ch` but might have higher average querying times. The latter due to using binary search algorithms as opposed to hashing. Also, this trie will load slower as after adding each trie layer we need to re-order previous context ids for each end word w_m in this layer. The latter is needed for being able to apply binary search algorithms at querying stage, which are clearly faster than plain linear search.

C2WOrderedArrayTrie.hpp/C2WOrderedArrayTrie.cpp - Layered This trie type is available with the `c2wa` value of the `<trie_type>` option of the tool's command line parameter. It is a cross-layer context-to-word mapping trie implemented using the ordered arrays. This trie mimics the approach taken in `w2ca` but instead of mapping word id to previous context id it maps the previous context id to the end word id. The idea here is based on the observation that with the increase of m, for up to ≈ 50 Gb 5-gram models:

1. The ratio between the number of unique words and the number of m-grams in language models typically grows, and can reach values ≥ 250.

2. The ratio between the number of m and $m-1$ grams typically declines and even for large language models can reach values ≤ 1.3.

This means that mapping words to previous context ids results in relatively large arrays of previous context ids per word and this number increases with the increase of the model size. This results in longer search times for the m-gram ids. On the other hand, the number of the m-grams with each new layer is growing slower and slower and at some point shall start decreasing as sentences have limited average length. Therefore, the number of word ids per previous context id is decreasing and gets very low facilitating for faster searching.[3] Therefore, in this trie for a layer m we store mappings $\left(c^g_{m-1} \rightarrow (c^w_m \rightarrow c^g_m)\right)$, where $\left(c^g_{m-1} \rightarrow (\ldots)\right)$ is stored in an ordered array, of the number of $(m-1)$-grams, indexed by these gram's ids (the previous context ids). Each element of this array is a structure with two fields: `begin_idx` and `end_idx`. The latter refer to the array of the number of m-gram elements, let's call it `m_gram_data`. This array stores end word ids corresponding to the given previous context id. These word ids, per previous context id, are also ordered. In this way for each previous context id c^g_{m-1} we have a, possibly empty, range of end word ids stored between `m_gram_data[begin_idx]` and `m_gram_data[end_idx]`.

[3]This actually resembles the hash-table bucket approach when map elements are placed into buckets based on their hash values. If the number of buckets is large then the payload of each bucket is at most one value. This facilitates constant time average look up times.

The number of elements between `begin_idx` and `end_idx`, on average, should be decreasing with the increase of m. The m-gram id here is the absolute index of the word id c_m^w for previous context id c_{m-1}^g in the `m_gram_data` array.

This trie is expected to be faster when querying, when compared to `w2ca` but will require more memory due to storing `begin_idx` and `end_idx` index values per previous context id even if the range is empty. This is especially true for the higher level m-grams, as then the ratio between the number of previous and current level $(m-1)$-grams is decreasing, amplifying the overhead of storing the `begin_idx` and `end_idx` values.

G2DHashMapTrie.hpp/G2DHashMapTrie.cpp - Generic This trie type is available with the `g2dm` value of the `<trie_type>` option of the tool's command line parameter. It is a layer-independent m-gram-to-data mapping trie implemented based on self-made hash maps. This trie was made for performance and still requires memory-efficiency as well as performance tuning.

The idea behind this trie is that we do not want to store m-grams spread throughout the layers of the trie. We want to have layers fully independent from each other such that having an m-gram we can get its probability or a back-off weight by just one look-up in a hash table. This should have constant average time complexity, if the hash table is implemented well and the hashing function is proper. This approach also allows for higher levels of distributability[4].

Our approach is similar to the hash-based approach taken by KenLM. The latter works with hash values of the words and m-grams to save space and facilitate fast look-up. However, KenLM seems to treat word and m-gram hash values as unique identifiers, see Section 4.2. This is not trustable and introduces an unevaluated possibility of hash collisions leading to translation errors. See the discussion in the paragraph about the `c2dm` trie above. In our approach we resolve this uniqueness issue by following the standard hash-map approach: We allow more than one m-gram in each bucket by to issuing each m-gram an unique identifier.

The m-gram identifiers that we use are naturally based on the sequences of the individual m-gram word ids. Remember that, each word id is 4 bytes (`uint32_t`) which means that for a 5-gram such an identifier can be as long as 20 bytes, leading to a significant memory consumption. We attempt to fight this issue by the following two rules:

1. Give the most probable words the lowest ids.

2. Use only the word id meaning-full bytes.

In other words, we suggest to use a byte-packed[5] sequence of the m-gram's word ids as the m-gram id.

Clearly, when using byte packing we need to specify how many bytes of each word id we used to form the given m-gram id. This introduces the notion of the m-gram id type defined by the number of bytes we use from each of the word ids. This m-gram id type information must

[4]If the trie is to be distributed, and each level is independent of one another, then we can put them on separate server nodes and query them independent of each other.

[5]We also had a bit-packed version thereof but it showed significantly worse performance (30%) with minimal memory consumption improvements (1-2%). The bit backing is still available in the code and can be put to use at any time for further experiments.

also become part of the m-gram id. This raises the question how expensive it will be to store it. Let us consider a 5-gram case. Considering that each word id is stored as a 4 byte value, the number of possible m-gram id types is $4^5 = 1024$ which only requires 2 bytes to be stored. Then in our case, each 5-gram id should consists of a two byte id type value followed by the meaningful m-gram word id bytes. Note that issuing the lower word id to more frequently used word should ensure lower average m-gram id lengths. Therefore, the suggested two rules should allow to minimize the number of bytes needed to store an m-gram id. In our experiments, we observed an average of 11 byte long m-gram ids for $m = 5$ versus the maximum of 20 bytes.

Overall the querying performance of this trie is expected to be the best among the considered tries, but not that efficient memory-wise. Here, we intentionally did not follow the approach of KenLM that discards words and only stores their hashes as this introduces an unknown probability of error from hash collisions. To our knowledge KenLM also does the same for m-grams, but this is yet to be verified from its code. The indirect evidence is however there, the amount of memory used by KenLM is significantly lower than needed to store proper unique m-gram ids.

4 Empirical comparison

In this section we perform an empirical comparison of the developed software with two other well known tools, namely SRILM and KenLM, both of which provide language model implementations that can be queried.

This section is organized as follows: The test targets and test configuration is discussed in Section 4.1. Information on KenLM and SRILM is given in Section 4.2. Further, Section 4.3 presents the experimental results. Finally, Section 4.5 discusses some conclusions.

4.1 Test target and configuration

The main target of this experimental comparison is to evaluate memory consumption and query times of the implemented tries. For doing that we do not rely on the time and memory statistics reported by the tools but rather, for the sake of uniform and independent opinion, rely on the Linux standard `time` utility available in the `zsh` Linux shell. The latter provides system-measured statistics about the program run. We choose to measure:

- **MRSS** - the maximum resident memory usage of the program
- **CPU time** - the CPU time in seconds

We chose to measure maximum resident memory usage as this is what defines the amount of RAM needed to run the program. Also, the CPU times are the actual times that the program was executed on the CPU. Measuring CPU times allows for a fair comparison as excludes possible results influence by the other system processes.

It is important to note that system I/O such as reading from files or writing to console results in a significant slow down of a tool's performance. Therefore the experiments, as much as possible, were run during the night and the tool's outputs were disabled. In case of clear outliers experiments were re-run.

The experiments were performed on shared-use Linux blades running Centos 6.x Linux. We used several identical blades named: `smt7`, `smt9`, and `smt10`; detailed information on which can be found in Appendix A.

The experiments are set up to be run with different-size 5-gram language models given in the ARPA format with two types of inputs:

1. The single 5-gram query that defines the baseline
2. The file input with $100,000,000$ of 5-gram queries

The delta in execution CPU times between the baseline and the $100,000,000$ query files defines the pure query execution time of the tool. Note that, the query files were produced from the text corpus different from the one used to produce the considered language models. The MRSS values are reported in gigabytes (Gb) and the CPU times are measured in seconds. The plots provide MRSS and CPU times relative to the input model size in Gb. For more details on the language models and query files see Appendix B.

4.2 Considered Tools

This section provides brief information about the considered (external) tools as well as the tool developed within this project. This information is mostly obtained from the tools' official web pages and documentation. We also provide a few details on how the tools are run in our experiments.

4.2.1 Back Off Language Model(s) for SMT v 1.0 - the Owl release

This tool is the main focus of this deliverable. It is built in release configuration, with `LOGER_MAX_LEVEL` set `USAGE` to to avoid unneeded output, see Section 3.3 for more details. Further in our experiments, the tool is run with the following command-line options:

```
% back-off-language-model-smt model-file queries-file usage
```

It is important to note that we experiment not only with the available trie model types but two additional options: optimizing and non-optimizing word indexes and hash bitmap caching. Let us briefly discuss what they are all about.

Optimizing and non-optimizing word indexes Word index is an internal program dictionary used to issue each word a unique id. The non-optimizing word index is implemented as a simple std::unordered_map storing the mappings from a word to its id. The optimizing word index differs in that the unordered_map is substituted with our own array-based hash map implementation. This should allow for a faster word id look-up which is one of the most intensively used operations. In our optimizing word index we use 10 times more buckets than there are different words in the trie. This parameter value was experimentally chosen but can be easily changed, see `__OptimizingWordIndex::BUCKETS_FACTOR` constant in `./inc/Configuration.hpp`.

Hash bitmap caching The bitmap hashing is an optimization technique allowing to relax querying for all trie types. The idea is that each m-gram can be given a 32 bit hash value. Then, during creating and filling it up the trie layer by layer with m-grams, for each layer m, we create a bit set with $N = K * |m - grams|$ elements in it. Here K is some coefficient chosen to be 20 and $|m - grams|$ is the number of m-grams in layer m of the model. The computed m-gram hash is then subjected to the `mod` operation: $i = H \% N$, allowing to bring its value within the range of $0 \ldots N - 1$. Then the ith bit in the created set is enabled. When querying, each query m-gram, the same way, we compute its index i and check whether the corresponding bit in the bit set is enabled. If it is then there is a chance that the given m-gram is present in the trie, so we need search for it. If it is not, then the m-gram is definitely not in the trie, so we immediately switch to computing the back-off. The value of $K = 20$ was chosen by experiments[6] and is configurable via the `__BitmapHashCache::BUCKET_MULTIPLIER_FACTOR` constant in `./inc/Configuration.hpp`

[6] On a 20 Gb model it showed the best performance/memory usage ratio.

4.2.2 SRILM - The SRI Language Modeling Toolkit

SRILM is a toolkit for building and applying statistical language models (LMs), primarily for use in speech recognition, statistical tagging and segmentation, and machine translation. It has been under development in the SRI Speech Technology and Research Laboratory since 1995. The employed tool version is 1.7.0. The tool is run with the following command-line options:

```
% ngram -lm model-file -order 5 -ppl queries-file \
      -no-sos -no-eos -memuse -debug 0
```

No changes were done to the tool's source code.

4.2.3 KenLM

KenLM is a tool for estimating, filtering, and querying language models. The tool does not have clear version indication, so we used the tool's GitHub snapshot of the Git revision:

$$0f306088c3d8b3a668c934f605e21b693b959d4d$$

KenLM does not allow to switch off the probability reports from the command line. Therefore we had to modify the tool's code. In the `kenlm/lm/ngram_query.hh` file we commented out the output code lines as follows:

```
struct BasicPrint {
  void Word(StringPiece, WordIndex, const FullScoreReturn &) const {}
  void Line(uint64_t oov, float total) const {
    //std::cout << "Total: " << total << " OOV: " << oov << '\n';
  }
  void Summary(double, double, uint64_t, uint64_t) {}

};

struct FullPrint : public BasicPrint {
  void Word(StringPiece surface, WordIndex vocab,
            const FullScoreReturn &ret) const {
    //std::cout << surface << '=' << vocab << ' '
    //<< static_cast<unsigned int>(ret.ngram_length)
    //<< ' ' << ret.prob << '\t';
  }

  void Summary(double ppl_including_oov, double ppl_excluding_oov,
               uint64_t corpus_oov, uint64_t corpus_tokens) {
    std::cout <<
      "Perplexity including OOVs:\t" << ppl_including_oov << "\n"
      "Perplexity excluding OOVs:\t" << ppl_excluding_oov << "\n"
      "OOVs:\t" << corpus_oov << "\n"
      "Tokens:\t" << corpus_tokens << '\n'
      ;
  }
};
```

After this change, the tool was run with the following command-line options:

```
% query -n model-file < queries-file
```

Note that KenLM is a tool that highly relies on hashing and treats hash values of words as if there were unique word identifiers. As is stated in Section 2.1 of Heafield (2011): "Vocabulary lookup is a hash table mapping from word to vocabulary index. In all cases, the key is collapsed to its 64-bit hash.". The same applies to storing the m-grams. As indicated in the same section of the same article: "When keys are longer than 64 bits, we conserve space by replacing the keys with their 64-bit hashes." It is also stated there that: "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." The latter seems rather questionable as no further analysis of information over the baseline model and queries is provided. Strictly speaking using hash as a unique identifier introduces an unknown error which depends on the quality of the hashing function and the input queries as well as the language model itself. The risk of hash collision in KenLM is somewhat reduced by only putting m-grams of the same level m into one hash table, as described in Section 2.1 of Heafield (2011), but this still does not eliminate the problem.

4.3 Experiment results

Before we proceed with the experiments results, let us note that for each 5-gram query $w_1w_2w_3w_4w_5$ our tool computes a single conditional probability $Prob\,(w_5|w_1w_2w_3w_4)$. SRILM and KenLM however compute the sum of conditional probabilities:

$$\begin{aligned} log_{10}\,(Prob\,(w_1)) + log_{10}\,(Prob\,(w_2|w_1)) + log_{10}\,(Prob\,(w_3|w_1w_2)) \\ + log_{10}\,(Prob\,(w_4|w_1w_2w_3)) + log_{10}\,(Prob\,(w_5|w_1w_2w_3w_4)) \end{aligned} \tag{3}$$

In addition, they compute perplexities. This means that their do more that our tooling does. However, although the other tools seem to perform more work, this should not be seen as a 5-times computation difference with our tool. The higher the m-gram level the more time it needs to be computed using the layered model, for the generic non-layered models the difference is less vivid. Also note that computing the hash of an m-gram is linear in its length. So higher m typically means longer hash computation. Also, the lower the level of the m-gram is the less m-grams there are, resulting in faster look-up. These together imply that computing $Prob\,(w_5|w_1w_2w_3w_4)$ is more expensive than computing $Prob\,(w_2|w_1)$, let alone computing $Prob\,(w_1)$. The exact ratio is yet to be computed but the worst case scenario is indeed that they are computationally equal. In that case the CPU timing results of SRILM and KenLM are to be divided by 5. In the future we plan to change our tool to compute the same cumulative probabilities as SRILM and KenLM. Then the experiments are to be repeated for more comparable results.

Further, we will argue about performance results obtained in several experimental configurations. Doing that we will be referring to the aforementioned expectations about the trie performance implemented in our tool, see Section 3.5.2.

4.3.1 Bitmap caching - Off; Optimizing Word Index - Off;

The memory consumption of the run configuration is given in Figure 1. As expected `w2ch` and especially `g2dm` show a very high memory consumption, even higher than that of SRILM. Yet the two other tries, `c2wa` and especially `w2ca`, show an up two two times lower memory

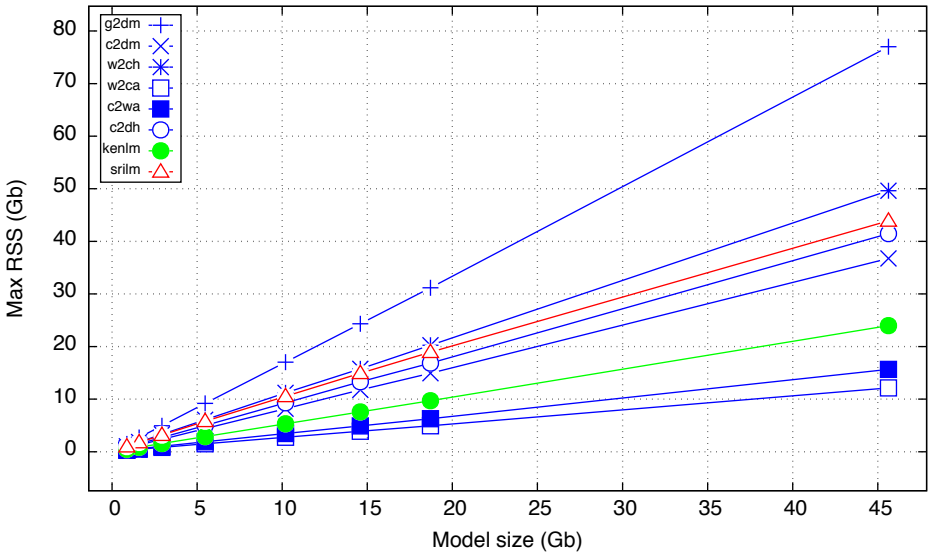

Figure 1: MRSS, smt7: Bitmap caching - Off; Optimizing Word Index - Off

consumption than that of KenLM. The latter is without loosing any information of the original language model, such as happens in KenLM due to hashing. The memory consumption of `c2dh` and `c2dm` is also superior to SRILM and is expectedly higher than that of `c2wa` and `w2ca`.

The CPU times are given in Figure 2. At the moment `c2dm` is the fastest trie, which is expected as the m-gram id is computed by a simple mathematical formula. Remember that, unfortunately, this is the trie that might suffer from the overflows in m-gram id computations. In this experiment, `w2ca` is the slowest trie, as it requires a significant amount of binary searching the context id mappings. Both of these results are expected. The other tries reside close to each other, although the `g2dm` would be expected to be faster. This requires further investigation as well as performance profiling. Presumably the reason for this is the fact that the hashing function is overused. Each word id is stored in an optimizing word index that uses a word hash to provide the bucket id. However, each m-gram is then also hashed in order to get its bucket id as well. Therefore hashing is done twice and this will be taken care of in the next versions. Note that the current `g2dm` implementation has not yet been profiled and re-worked yet. Noticeable enough, `c2dh` is faster than `c2wa` as it uses hash maps and does not require binary searching the data. Although `w2ch` also does not require binary searching the data, it is even slower that `c2dh` and `c2wa`. The reason is most likely to be its complex structure.

4.3.2 Bitmap caching - On; Optimizing Word Index - Off;

The memory consumption of the run configuration is given in Figure 3. As one can see, compared to the memory results in Figure 1 the memory consumption is increased only by a couple of percents. This is due to the fact that bitmaps require comparably less memory that the other data needed to be stored.

The performance impact of the Bitmap caching is given in Figure 4. As one can see, intro-

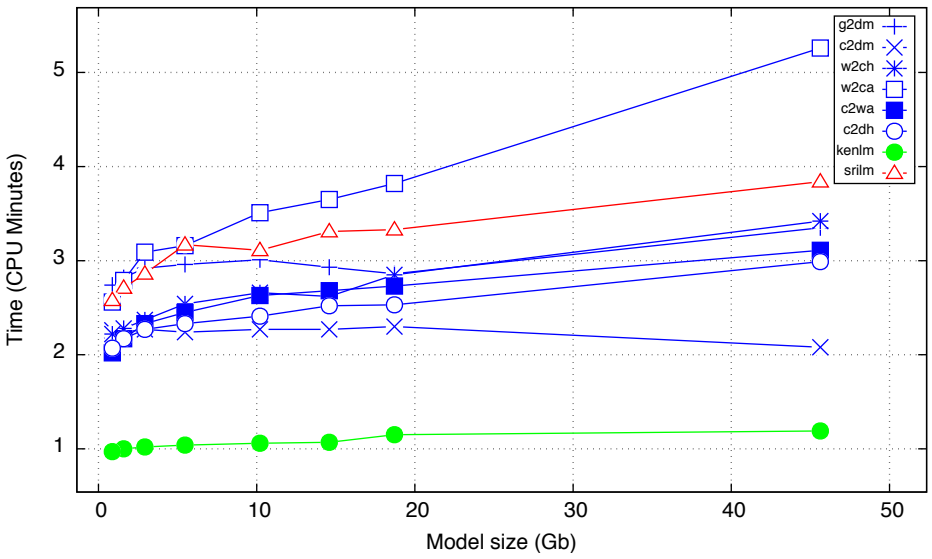

Figure 2: CPU usage, smt7: Bitmap caching - Off; Optimizing Word Index - Off

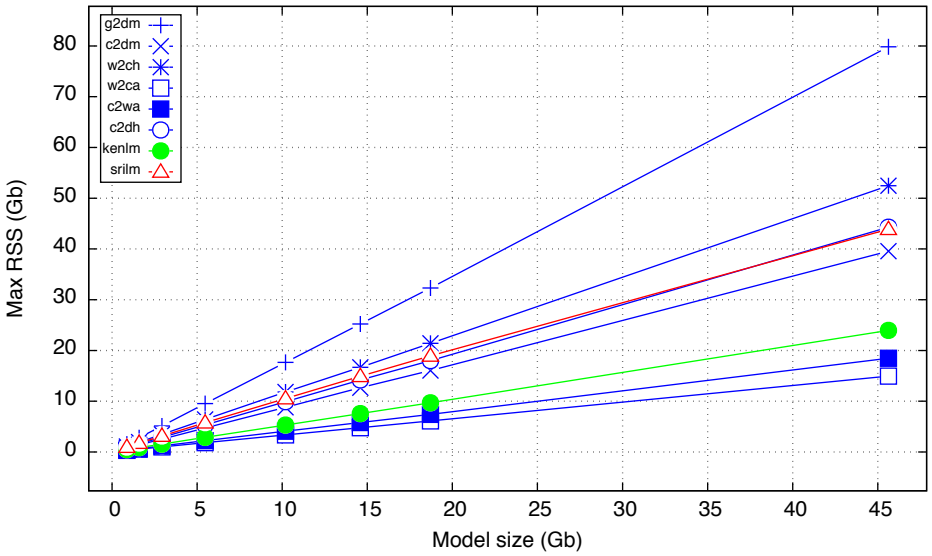

Figure 3: MRSS, smt7: Bitmap caching - On; Optimizing Word Index - Off

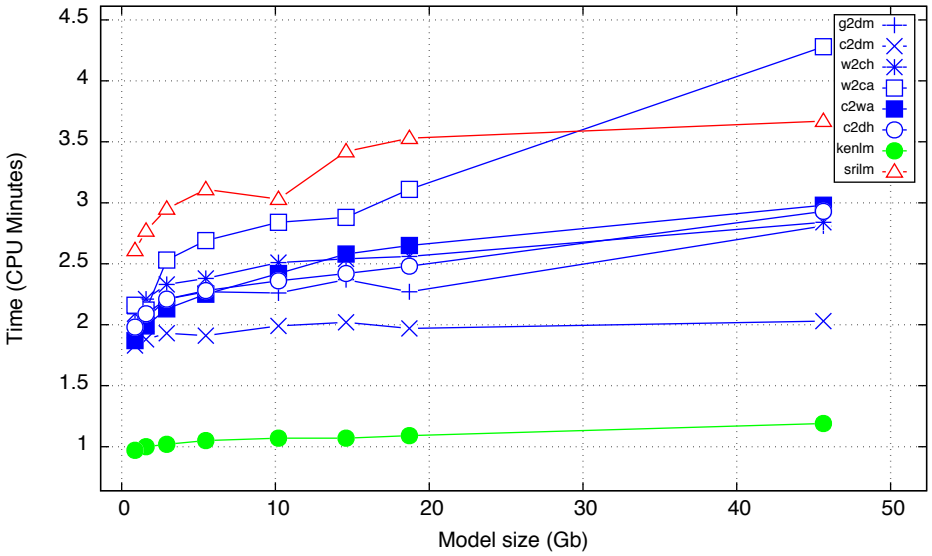

Figure 4: CPU usage, smt7: Bitmap caching - On; Optimizing Word Index - Off

ducing the bitmap hash caching yields a performance improvement from about 10 to 15 percent.

4.3.3 Bitmap caching - Off; Optimizing Word Index - On;

The memory consumption of the run configuration is given in Figure 5. When compared with Figure 1, we see that using optimizing for the word index did not really affect the memory consumption. This means that our hash map implementation of the optimized word index, in this application, is at least as memory efficient as the C++11 std::unordered_map.

The CPU times are reported in Figure 6. The performance improvements are rather uniform throughout the tries as all of them require the same number of word id look-ups. The improvements are within 10% when compared to Figure 2 but are indeed smaller than that of bitmap cache, see Figure 4. Perhaps an even more efficient hash-map implementation could improve the performance even further.

4.3.4 Bitmap caching - On; Optimizing Word Index - On;

The memory consumption of the run configuration is given in Figure 7. Since optimizing the word index had no impact on memory consumption, here we observe the same memory usage pattern as with having the bitmap caching on, see Figure 3.

Of course, since we combine both of the employed optimization techniques, each of which seems to work, we expect cumulative performance improvement. Consider Figure 8, where it clearly shows that enabling two optimization techniques brings us yet closer to the performance of KenLM. Especially when it is done for the hash map based implementations such as `c2dm` and `c2dh`. Interesting enough, `c2dh` showed the same performance as `g2dm` as if the cumulative impact was higher on the `g2dm` trie. The latter is reasonable as `g2dm` requires some rather

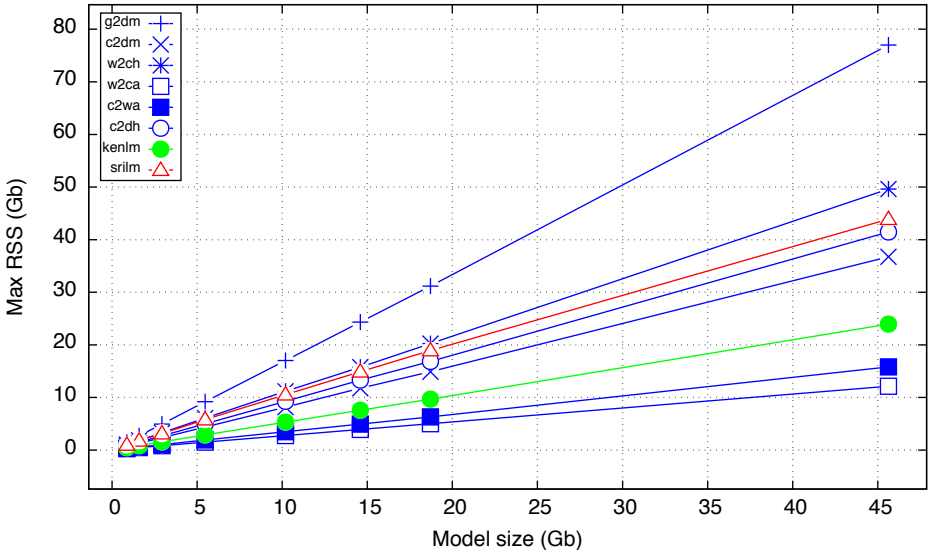

Figure 5: MRSS, smt9: Bitmap caching - Off; Optimizing Word Index - On

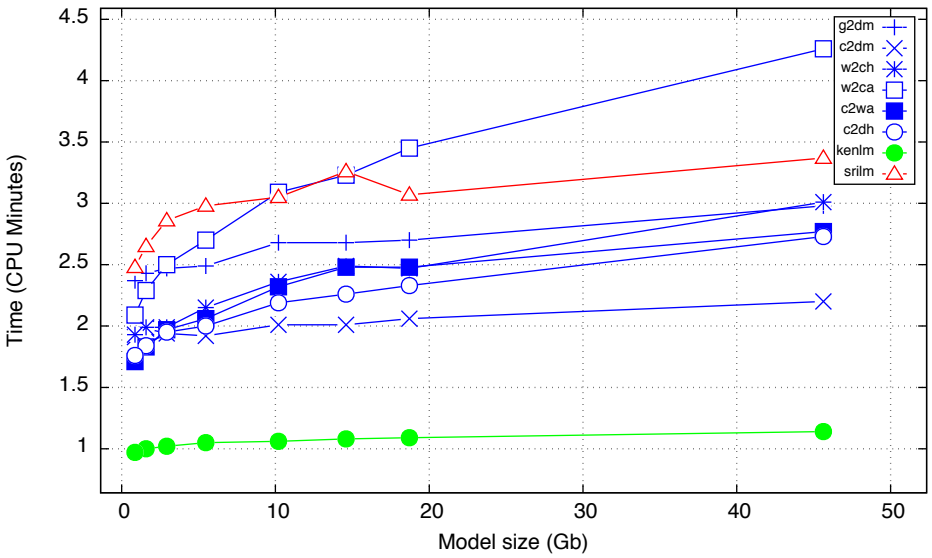

Figure 6: CPU usage, smt9: Bitmap caching - Off; Optimizing Word Index - On

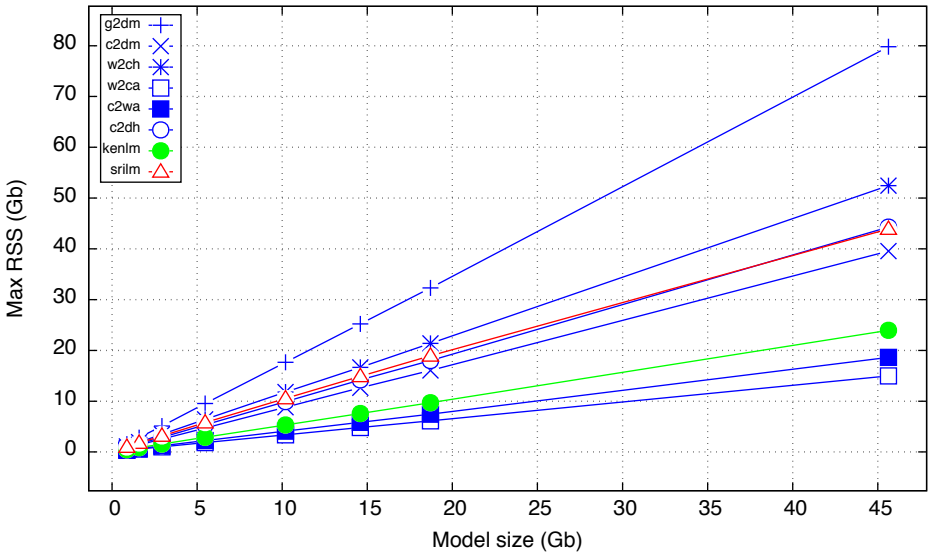

Figure 7: MRSS, smt10: Bitmap caching - On; Optimizing Word Index - On

intricate m-gram ids computations and search algorithms whereas `c2dh` is arithmetic and hash-map based.

4.4 Multi-threading

While the current set of experiments does not include explicit comparisons of multi-threading, great care has been taken to ensure thread safety of the various language model implementations. Therefore, we do not anticipate any problems regarding multi-threading. Nevertheless, this will be experimentally validated at the very beginning of the coming 6-month period.

4.5 Summary

The following conclusions can be drawn from the performed experiments:

- Bitmap hashing is beneficial for all sorts of tries, even for the `g2dm` trie which is also a hash map. It provides 10 to 15% of performance improvement and requires only about 3% extra memory.
- Optimizing word index is beneficial for all sorts of tries, as obtaining the word id is one of the most frequent operations when querying the model. It provides up to 10% performance improvement and requires no extra memory.
- Optimizing word index could be further improved by employing an even more efficient sort of hash map implementation. This has not been shown in the report but word id look-up takes about 10% of the application's run time.

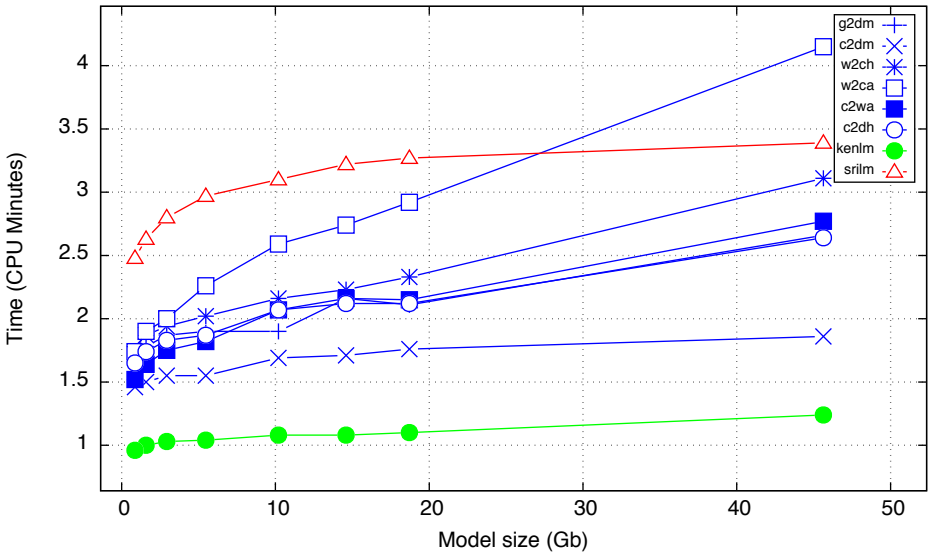

Figure 8: CPU usage, smt10: Bitmap caching - On; Optimizing Word Index - On

- Using layered tries results in very small MRSS values. For example, the `w2ca` trie needs only 30% of memory taken by the language model file. This is about half the memory needed by KenLM to store the same model. However, a smaller memory footprint comes at a price of lower performance: `w2ca` is about two times slower than the other considered tries. Also, growing the model size gives a logarithmic increase in query times.[7]

- As is proven by KenLM, using a hash-map based non-layered approach yields moderately memory efficient and very fast tries.[8] However, the reason for KenLM being fast and memory efficient is based on heavily relying on hash values. The latter are treated as unique identifiers of words and m-grams. This is not 100% reliable due to unpredictable hash collisions.[9]

Overall our tool's performance is comparable to that of SRILM and KenLM. Yet the performance comparison is to be improved by making our tool to compute the total probability sum instead of the conditional probability of the longest m-gram, see the beginning of Section 4.3.

[7]Due to using binary searches whereas hash-map based implementations, on average, have constant query times.
[8]The latter is not yet achieved by our `g2dm` implementation
[9]We are still eager to implement a similar sort trie for performance critical applications

5 Conclusions

In this report we have presented our work on the language model implementations. We have implemented a number of trie variants and compared them with with respect to peak residence memory consumption and CPU time consumption with two existing and well-known tools: KenLM and SRILM.

Our experiments show that using layered tries can result in up to 70% memory reduction, at the expense of twice lower performance efficiency. The latter can be effectively mitigated by optimization techniques that allowed us to get about 25% performance improvement. The querying performance can be significantly increased by using hash-map based, layer-independent tries which naturally require more memory.

The multi-threading aspects of our trie implementation have not been fully tested. At the same time, thread-safety was taking into account when developing our current data structures by avoiding shared memory write use at the language model querying stages within the tries.

Further, we give the following usage guidelines for the implemented tries:

- `w2ca` and `c2wa` tries are beneficial for the machines with limited RAM. If low memory usage is very critical then bitmap hash caching can also be disabled.
- `c2dm` trie provides the fastest performance with moderate memory consumption. This is recommended when high performance is needed but one should be aware of possible m-gram id collisions.[10]
- `c2dh` trie is preferable if performance, as well as moderate memory consumption, is needed. This is the second-fastest trie which, unlike `c2dm`, is fully reliable.
- `w2ch` trie did not show itself useful and `g2dm` is yet to be re-worked and improved for better performance and memory usage.

Overall we created a compatible trie model implementation that is still to be improved in the number of ways listed in Section 5.1. We have also investigated performance gains of KenLM and came to the conclusion that its approach of using hash values as unique identifiers, although efficient and fast, is rather risky. More advanced error analysis is needed to evaluate its applicability.

5.1 Future work

Below we provide a brief list of future work we see important and needed to bring the language model implementation to the desired quality level.

- `C2DHashMapTrie.hpp / C2DHashMapTrie.cpp` - the current implementation is potentially error prone to hash collisions in case of context id overflows. Overflows were not observed on the tries of up to 20 Gb but more thorough testing is needed and perhaps the collision detection must be always enabled for this trie.

[10] It is therefore preferable to use it on smaller models < 20 Gb.

- `Tries` - It is possible to introduce more templating into the tries, e.g. the gram-level-based templating. It should improve performance as many checks can be resolved at compile-time.
- `G2DHashMapTrie.hpp / G2DHashMapTrie.cpp` - This trie is potentially very performance efficient, but this has not been achieved yet. Also, its memory consumption is sub optimal at present. It needs a significant re-work in the way data is stored and queried.
- `Thread safety` - Tries are to be additionally reviewed and tested for using class data members during filling in the tries or querying. One can just make the entire trie interface synchronized but this is sub-optimal.
- `Testing` - Testing done with this code was limited. The trie code may still contains error. It is recommended to add unit and functional tests for this project.
- `Code` - The naming convention is not always ideally followed.
- `Functionality 1` - Switch to computing the cumulative probability of the m-gram instead of computing only one conditional probability.
- `Functionality 2` - Implement a word index and a Trie that use word and m-gram hashes as unique ids.

References

Cerion Armour-Brown, Christian Borntraeger, Jeremy Fitzhardinge, Tom Hughes, Petar Jovanovic, Dejan Jevtic, Florian Krohm, Carl Love, Maynard Johnson, Paul Mackerras, Dirk Mueller, Nicholas Nethercote, Julian Seward, Bart Van Assche, Robert Walsh, Philippe Waroquiers, and Josef Weidendorfer. Valgrind, 2000. URL `http://valgrind.org/`.

Inc. Free Software Foundation. Gprof: Gnu performance profiler, 1988. URL `http://www.gnu.org/software/binutils/`.

Inc. Free Software Foundation. Gdb: The gnu project debugger, 2015. URL `http://www.gnu.org/software/gdb/`.

John Gruber. Markdown: A plain text formatting syntax, 2015. URL `https://daringfireball.net/projects/markdown/`.

Kenneth Heafield. Kenlm: Faster and smaller language model queries. In *Proceedings of the Sixth Workshop on Statistical Machine Translation*, WMT '11, pages 187–197, Stroudsburg, PA, USA, 2011. Association for Computational Linguistics. ISBN 978-1-937284-12-1. URL `http://dl.acm.org/citation.cfm?id=2132960.2132986`.

Adam Pauls and Dan Klein. Faster and smaller n-gram language models. In Dekang Lin, Yuji Matsumoto, and Rada Mihalcea, editors, *The 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies, Proceedings of the Conference, 19-24 June, 2011, Portland, Oregon, USA*, pages 258–267. The Association for Computer Linguistics, 2011. ISBN 978-1-932432-87-9. URL `http://www.aclweb.org/anthology/P11-1027`.

Daniel Robenek, Jan Platos, and Václav Snásel. Efficient in-memory data structures for n-grams indexing. In Václav Snásel, Karel Richta, and Jaroslav Pokorný, editors, *Proceedings of the Dateso 2013 Annual International Workshop on DAtabases, TExts, Specifications and Objects, Pisek, Czech Republic, April 17, 2013*, volume 971 of *CEUR Workshop Proceedings*, pages 48–58. CEUR-WS.org, 2013. URL `http://ceur-ws.org/Vol-971/paper21.pdf`.

Andreas Stolcke, Jing Zheng, Wen Wang, and Victor Abrash. Srilm at sixteen: Update and outlook. In *Proc. IEEE Automatic Speech Recognition and Understanding Workshop*, Hawaii, December 2011. IEEE SPS. URL `http://research.microsoft.com/apps/pubs/default.aspx?id=165613`.

Matthew Szudzik. An elegant pairing function. In *NKS2006: Wolfram Science Conference*, Washington, DC, US, June 2006. URL `https://www.wolframscience.com/conference/2006/presentations/szudzik.html`.

Appendices

A Hardware configurations

Further, we give details about the configurations of the compute machines: `smt7`, `smt9`, and `smt10`, used for the experiments in this report. All of the machine specifications are identical. Therefore below, we only provide the machine details of `smt7`.

```
[~ smt7 ~]$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                40
On-line CPU(s) list:   0-39
Thread(s) per core:    2
Core(s) per socket:    10
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 62
Stepping:              4
CPU MHz:               1200.000
BogoMIPS:              4999.23
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              25600K
NUMA node0 CPU(s):     0-9,20-29
NUMA node1 CPU(s):     10-19,30-39
[~ smt7 ~]$ lsb_release -irc
Distributor ID: CentOS
Release: 6.7
Codename: Final
[~ smt7 ~]$ grep MemTotal /proc/meminfo
MemTotal:       264496688 kB
```

B Language models and query files info

The considered language models and their sizes (in bytes) are:

```
[~ smt10~]$ ls -al *.lm
-rw-r--r-- 1     937792965 Sep 21 15:55 e_10_641093.lm
```

```
-rw-r--r-- 1    1708763123 Sep 21 17:36 e_20_1282186.lm
-rw-r--r-- 1    3148711562 Sep 21 17:45 e_30_2564372.lm
-rw-r--r-- 1    5880154140 Sep 21 18:09 e_40_5128745.lm
-rw-r--r-- 1   10952178505 Sep 21 18:29 e_50_10257490.lm
-rw-r--r-- 1   15667577793 Sep 21 20:22 e_60_15386235.lm
-rw-r--r-- 1   20098725535 Sep 21 20:37 e_70_20514981.lm
-rw-r--r-- 1   48998103628 Sep 21 21:08 e_80_48998103628.lm
```

The considered query files and their sizes are:

```
[~ smt10 ~]$ ls -al q_5_gram_1*.txt
-rw-r--r-- 1   2697064872 Sep 21 15:47 q_5_gram_100.000.000.txt
-rw-r--r-- 1           35 Sep 21 15:57 q_5_gram_1.txt
[~ smt10 ~]$
```

B.1 e_10_641093.lm

```
[~ smt10 ~]$ head -n 15 e_10_641093.lm

\data\
ngram 1=105682
ngram 2=1737132
ngram 3=5121040
ngram 4=7659442
ngram 5=8741158
```

B.2 e_20_1282186.lm

```
[~ smt10 ~]$ head -n 8 e_20_1282186.lm

\data\
ngram 1=143867
ngram 2=2707890
ngram 3=8886067
ngram 4=14188078
ngram 5=16757214
```

B.3 e_30_2564372.lm

```
[~ smt10 ~]$ head -n 8 e_30_2564372.lm

\data\
ngram 1=199164
ngram 2=4202658
```

```
ngram 3=15300577
ngram 4=26097321
ngram 5=31952150
```

B.4 e_40_5128745.lm

```
[~ smt10 ~]$ head -n 8 e_40_5128745.lm

\data\
ngram 1=298070
ngram 2=6675818
ngram 3=26819467
ngram 4=48897704
ngram 5=62194729
```

B.5 e_50_10257490.lm

```
[~ smt10 ~]$ head -n 8 e_50_10257490.lm

\data\
ngram 1=439499
ngram 2=10447874
ngram 3=46336705
ngram 4=90709359
ngram 5=120411272
```

B.6 e_60_15386235.lm

```
[~ smt10 ~]$ head -n 8 e_60_15386235.lm

\data\
ngram 1=568105
ngram 2=13574606
ngram 3=63474074
ngram 4=129430409
ngram 5=176283104
```

B.7 e_70_20514981.lm

```
[~ smt10 ~]$ head -n 8 e_70_20514981.lm

\data\
```

```
ngram 1=676750
ngram 2=16221298
ngram 3=78807519
ngram 4=165569280
ngram 5=229897626
```

B.8 e_80_48998103628.lm

```
[~ smt10 ~]$ head -n 8 e_80_48998103628.lm

\data\
ngram 1=2210728
ngram 2=67285057
ngram 3=183285165
ngram 4=396600722
ngram 5=563533665
```