Using the OpenFEC.org Performance Evaluation Tools

Vincent Roca (INRIA), Jonathan Detchart (INRIA), Mathieu Cunche (INRIA)
Valentin Savin (CEA-LETI), Jérôme Lacan (ISAE)
http://openfec.org

December 16, 2014

Contents

version: `$Id: howto_performance_evaluation.tex 109 2014-04-08 09:10:12Z roca $`

1 Introduction

The performance evaluation tools are a set a Perl scripts meant to assess the code and codec performances, using many different metrics, in an automatic way.

1.1 Principles

These scripts use the `eperftool` program to simulate a transmission between a sender and a receiver over a lossy channel. Therefore, we are doing an actual AL-FEC encoding (using a real encoder), and an actual AL-FEC decoding (using a real decoder). Only the transmissions are simulated. You, as the user, control many parameters, like the transmission type (in which order should the source and repair packets for the various blocks be transmitted), the loss probability, and all the code specific parameters (e.g. the object size, code-rate, symbol size).

A first script, `run_tests.pl`, runs several iterations of the `eperftool` program with the desired parameters. All test results are written into log files. These log files are then analyzed in real-time and the results are inserted into a database that can be either a **MySQL database** or an **SQLite file**.

A second script, `generate_graph.pl`, is used to create the curves using various kinds of metrics (e.g. you can have several kinds of curves without having to re-run tests). This script connects to the database to execute `select` SQL requests and generates `gnuplot` files (`.dem` for the `gnuplot` commands and `.dat` for the raw data). In no case does the script modify the database itself, so you even run it during tests, to generate preliminary curves.

Both scripts need a parameter file (e.g. `params.txt`, but you can rename it as you want). This ASCII file contains all the necessary parameters to run tests and to generate the curves (even if in the latest case, only the database connection string is required).

1.2 Requirements and limits

- The set of tools provided require you use a **Linux or Mac OSX operating systems**. This does not mean it won't work on different operating systems, just that we did not test and cannot guaranty anything.
- Performance analysis tools require you use either a MySQL or SQLite tools. Make sure one of them is available on your system, otherwise install it. See section 2.1 for more information.
- LDPC matrices plot facilities have specific requirements. See section 5 for more information.

2 Configuring the "params.txt" file

> → *In short: edit the "params.txt" file to define the simulation parameters.*

The parameters file (called by default `params.txt`) contains the following sections:

- *Tools:* paths to the various tools;
- *Files:* paths to the various files that may be needed during simulations;
- *Tests:* tests to perform;
- *Code/codec configuration:* codes to use and how to use them;
- *Transmission and loss configuration:* kind of channel to use;
- *Database configuration:* SQL related parameters;

See the provided file for explanations on the syntax. The present document only contains additional information not present in the `params.txt` file.

2.1 The "Database configuration" section

The "Database configuration" part defines parameters for using database:

Name	Description
erase_database	Allow or not the script to erase the database before running tests
database	Kind of database, i.e. either MySQL database or SQLite file

The **erase_database** parameter must be a boolean (true or false). If this parameter is set to true, the database is erased before running tests. Setting this parameter to false allows to generate manually intermediate curves between several executions of the `run_tests.pl` script, e.g. to refine test areas that are worth the pain, instead of executing once, with a lot of iterations, the `run_tests.pl` script.

The **database** parameter contains the information needed to connect to the database. Two models are supported:

- the MySQL server model:
  Syntax: `database server <database_name> <host> <port> <user> <password>`
  This model requires a MYSQL driver, with a Perl DBD::MySQL module.
  - `database_name` is the name of an existing database. If the database doesn't exist, the script displays an error;
  - `host` is the address of the database server;
  - `port` is the port for TCP connexion (default: 3306);
  - `user` and `password` are the login and the password of a user allowed to execute SQL request on the database (e.g. insert);

  Example: to connect to the `perf_stats` database, located on server 192.168.1.1/3306, for user "toto" and password "otot", use:
  `database server perf_stats 192.168.1.1 3306 toto otot`

- the SQLite file model:
  Syntax: `database file <name>`
  This model requires an SQLite driver.

  If not already installed on your system, you can either use a package management system (`yum` or similar), or install everything manually. To that purpose go to the DBD SQLite cpan page, at URL:
  http://search.cpan.org/ adamk/DBD-SQLite-1.29/
  download the package, and follow the installation instructions given at URL:
  http://www.cpan.org/modules/INSTALL.html.

  The `name` parameter is the name of the database file. If this file doesn't exist, the script will create it automatically.

For an introduction to the use of MySQL, you can have a look at:
http://dev.mysql.com/doc/refman/5.0/en/tutorial.html

3 Running tests

→ In short: once the "params.txt" file is ready, launch simulations and take a few cofees or some vacation, depending on the tests carried out ;-).

Tests are run with the `run_tests.pl` script. It requires a single parameter, the `params.txt` file whose syntax is described in section 2 (note that any file name can be used, `params.txt` is just the default name).

During execution, the script loops over all the parameter values. If some parameters have been set but are not required, they are ignored. For instance, if the Reed-Solomon and LDPC-staircase codes are both considered, the `ldpc_N1` parameter is considered for LDPC tests but silently ignored for Reed-Solomon tests.

In order to speed-up the tests, several processes can be run in parallel, one per simulation thread. More precisely, if the `using_threads` parameter is set to `true` in the `params.txt` file, the number of threads is set equal to the number of CPU cores [1]. Each simulation thread has in charge a subset of the desired number of iterations (as specified by the `iteration` parameter in the `params.txt` file). Then, for each iteration, the `eperftool` program is launched (as a process), which means that several `eperftool` processes can run at the same time.

Each thread writes the `eperftool` results in a separate temporary file. Then, every `nb_tests_for_partial_results` iterations, the content of a temporary file is copied into the general log file and at the same time, analyzed and the simulation results sent to the SQL database. The thread temporary file is then reset.

This approach is particularly useful in case of very long tests (e.g. that last several hours/days), in order to save partial results and get quickly an idea of the results (and possibly change parameters if a mistake is found).

In order to further speed-up the tests, you can also use several simulation hosts, each of them being in charge of a subset of the tests. Using a MySQL central database, the results are automatically aggregated in the SQL database. Note that several `params.txt` files have to be used in this case, one per subset of the tests, and it is your responsibility to do this split.

To summarize:

Input	a single `params.txt` file (for tests on a single simulation machine), or several `params.txt` files (for tests on several machines, using the MySQL mode)
During tests	on a simulation machine, one temporary log file per thread, plus a general log file. The general log file and the SQL database are updated periodically.
Output	on a simulation machine, the general log file, plus the SQL database (at the server or at the simulation machine with SQLite).

After running tests, you can analyze the results with another dedicated script and generate curves automatically, as explained below.

4 Analyzing the Results

> → *In short: once tests are completed, analyze the results and generate different kinds of curves automatically.*

Curve types: Different kinds of curves can be generated by the post-simulation analyzes tools:

1. decoding throughput as a function of the channel loss percentage;
2. decoding failure probability as a function of the channel loss percentage;
3. decoding failure probability as a function of the number of received symbols;
4. min/average/max inefficiency ratio as a function of the code rate;
5. min/average/max inefficiency ratio as a function of the object size;

[1] This number is determined by analyzing the /opt/cpuinfo on Linux systems, or by analyzing the output of the sysctl hw command on Mac OSX systems

6. number of `XOR` operations as a function of the object size (only in Debug mode);

7. number of `XOR` operations as a function of the channel loss percentage (only in Debug mode);

Analysis tools: Result analysis is performed by means of `generate_curves.pl` Perl script. For instance:
`./generate_curves.pl params.txt -curve=2`
is used to generate curves that plot the decoding failure probability as a function of the number of received symbols.

This script connects to the database to execute `select` SQL requests in order to extract the appropriate measurements. It then generates `gnuplot` files, namely a `.dem` containing the `gnuplot` commands, and a `.dat` file containing the raw data. In order to view the graphs, use:
`gnuplot <filename>.dem`

Note that the analysis process as a whole never modifies the SQL database (it only performs the appropriate SQL `select` requests).

Note that the `.dem` can be edited, in order to fix details, remove curves in case there are several curves, some of them not being of interest to you, change titles, etc. See the `gnuplot` documentation for additional information.

About the inefficiency ratio metric: The `inefficiency ratio` is defined as the number of symbols (source or repair) needed for decoding to complete (in a given test iteration) divided by the number of source symbols (A.K.A. code dimension). MDS codes (e.g. Reed-Solomon) have an inefficiency ratio equal to 1, iff there is a single block, non-MDS codes (e.g. all LDPC variants) have an inefficiency ratio greater or equal to 1 (the smaller this ratio, the better). This ratio is either provided as is (e.g. "1.006"), or as the decoding `overhead`, where $overhead = inefficiency\_ratio - 1$, expressed in percentage (e.g. 0.6% in the previous example).

4.1 The decoding throughput as a function of the channel loss percentage

This curve shows the decoding throughput as a function of the channel loss percentage. This curve allows you to analyze codec decoding speed and to refine the decoding algorithms or their implementation accordingly.

Warning: be very careful when you carry out this kind of experiment, since the results can potentially largely vary. In particular, make sure you are using the Release executable, that the simulation machine is idle, and that `using_threads` is set to `false` (to avoid problems).

4.2 Decoding failure probability as a function of the channel loss percentage

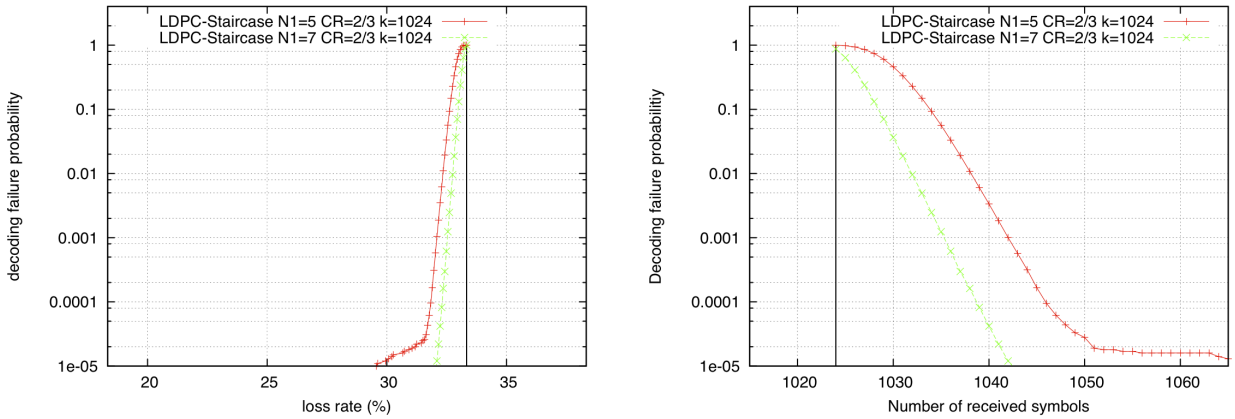
This curve represents the decoding failure probability as a function of the channel loss percentage (see 1(a)). This curve requires to enable the `using_ml` parameter in `params.txt` for the `run_tests.pl` script to run tests accordingly. Typically, the tests required to produce this kind of curve are extremely long (several hours or days). Also, in order to have enough precision, if you want a curve with failure probabilities as low as 10^{-b}, you have to specify at least 10^{2+b} iterations in the `params.txt` file.

4.3 Decoding failure probability as a function of the number of received symbols

This curve represents the decoding failure probability as a function of the number of received symbols rather than channel loss probability (the two curves show similar behaviors in a different way) (see 1(b)). Like the previous curve, you have to enable the the `using_ml` parameter in `params.txt` for the`run_tests.pl` script to run tests accordingly.

4.4 Inefficiency ratio as a function of the code rate

In this curve, the object size is fixed.



(a) As a function of the loss probability (%)

(b) As a function of the number of received symbols

Figure 1: Example of decoding failure probability curve (LDPC-Staircase, $k = 1.024$ symbols, code rate 2/3, $N1 = 5$ or 7).

Note that if in theory the `inefficiency ratio` of MDS codes is equal to 1, this is no longer true if there are several blocks and with certain transmission types. For instance, using Reed-Solomon over an object that is composed of 5000 source symbols (which leads to the creation of 30 blocks of size 166 or 167 source symbols each), and using a random permutation of all symbols before transmitting, the resulting inefficiency ratio is around 8%: `eperftool -codec=1 -tot_src=5000 -tot_rep=2500 -tx_type=0` In that case, LDPC-staircase codes largely outperform Reed-Solomon codes...

4.5 Inefficiency ratio as a function of the object size

In this curve, the code rate is fixed.

4.6 Number of XOR operations as a function of the object size

(Valid only in Debug mode)

This curve represents the number of operations (e.g. `XOR` on symbols) required to decode as a function of the object size. You have to use the `Debug` mode for the openfec library, because in `Release` mode, statistics on operations are not enabled. This curve gives a complementary view of the code and codec decoding speed, by focusing on its internal complexity rather than speed.

4.7 Number of XOR operations as a function of the channel loss percentage

(Valid only in Debug mode)

This is the same kind of curve as the previous one, but the object size is fixed.

5 Plotting LDPC matrices

→ *In short: OpenFEC includes the possibility to plot LDPC parity check matrices.*

An LDPC parity check matrix can be plot for analysis purposes. To do so:

- If you are interested by plotting LDPC-staircase parity check matrix, then edit file: `src/lib_stable/ldpc_staircase/of_codec_profile.h`.
  Define: `#define IL_SUPPORT`
  Do the same for all LDPC codecs you are interested in (codecs are independent).

- Edit file `src/lib_stable/CMakeLists.txt`.
  Un-comment line: `#target_link_libraries(openfec pthread IL)`
  (and of course comment the similar line that omits IL) to make sure the (Dev)IL library be used during link edition.

- Make sure DevIL library is installed on your machine. On a Linux machine, you can look for `libIL.so` to check it (e.g. use command `locate libIL` and look at the output). If not, you can either use a package management system (`yum` or similar), or install everything manually. To that purpose go to URL:
  http://openil.sourceforge.net
  and install the library as indicated.

- Compile the library and tools in `DEBUG` mode, using:
  ```
  cd build
  cmake .. -DDEBUG:STRING=ON
  make
  ```

- Launch eperftool, using one of the LDPC codes. A `.bmp` file is created in the local directory, containing an image of the parity check matrix. Open it with an appropriate tool (e.g. `okular` or `eog` on a Linux machine). You can also launch automatically the visualisation tool, from OpenFEC, using the `system("eog IL_file_image.bmp");` call (for instance).

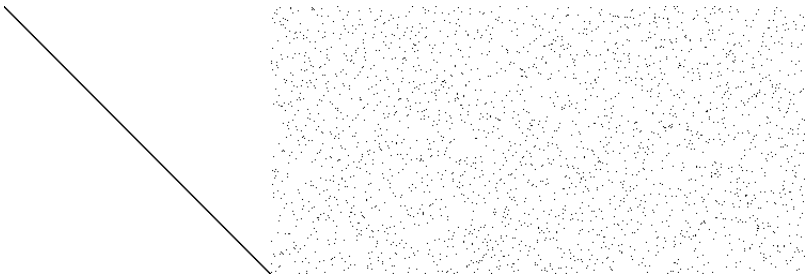


Figure 2: Example of matrix (LDPC-staircase H matrix, with k=500, n=750) shown in reverse order (H_2, the double diagonal sub-matrix, appears before H_1).

NB 1: For internal reasons, LDPC parity check matrices, $H = H_1|H_2$ are often stored in reverse order and H_2 (double diagonal sub-matrix in case of LDPC-Staircase) appears before H_1.

NB 2: You can of course use image manipulation tools (e.g. `gimp`) if you prefer a negative view of the matrix (by default "1"s appear as white pixels, over a black background, the example of figure 2 is for instance inverted).