open compress bench(mark)

Michael Strassberger
saremox@linux.com
3strassb@informatik.uni-hamburg.de

May 4, 2017

Contents

1. Introduction / Task

This report originates from the course "Parallelrechnerevaluation" (PRE) of Universität Hamburg (UHH). Supervisor was Dr. Michael Kuhn of Scientific Computing Research Group of UHH. The Project goal of PRE was to design and implement an evaluation tool for compression algorithms. It also needed to have an easy way for data extraction. The primary goal is therefore to create an instrument that assists scientific teams to choose the best suitable compression algorithm for their workload to archive:

1. better performance
2. more storage capacity
3. lower costs

The theory for this project originates from the Paper "Data Compression for Climate Data" released in "Supercomputing frontiers and innovations" Journal by Michael Kuhn, Julian Kunkel and Thomas Ludwig [KKL16].

The project is called ”open compress benchmark”. It emphasize the open-source character of the benchmark itself and its used algorithms. The Code of this project is released and maintained on github. `https://github.com/Saremox/ocbench`. Its licensed under the terms of GNU General Public License, version 2 `http://www.gnu.org/licenses/gpl-2.0.html`

2. Requirements

Since this tool's purpose is helping to evaluate the compressibility and throughput of several compression algorithms, it should be easy to use and easy to obtain the results. Therefore the requirements focus on being maintenance free and robust. To accomplish this, the model focuses heavily on the Unix philosophy, of single small libraries which do one job and doing it well.

3. Implementation

The Unified modeling language UML consist of various diagram standards. It is an ideal method to get an overview of the whole project. The modeling effort paid out in an early stage of the development. It made the testing of the single components of the system less complex. The methods used in this project orientates on human-centered methods used in the human-computer-interaction course at Universität Hamburg.

squash [1] is a ”Compression abstraction library and utilities” and is used in this project to support 31 plugins with various compression algorithms. It uniforms them to a simple to use interface.

3.1. Model & Design

3.1.1. Data Model

First off a basic data structure was modeled, to get an overview of what is necessary for a database layout. The model takes into account the various options different algorithms can have through an n:n relation in the model.

This model has the flexibility of unlimited option value pairs for each compression algorithm. However, this flexibility also adds complexity in the detection if a compression-option-value combination already exists. The complexity is discussed later in the implementation section.

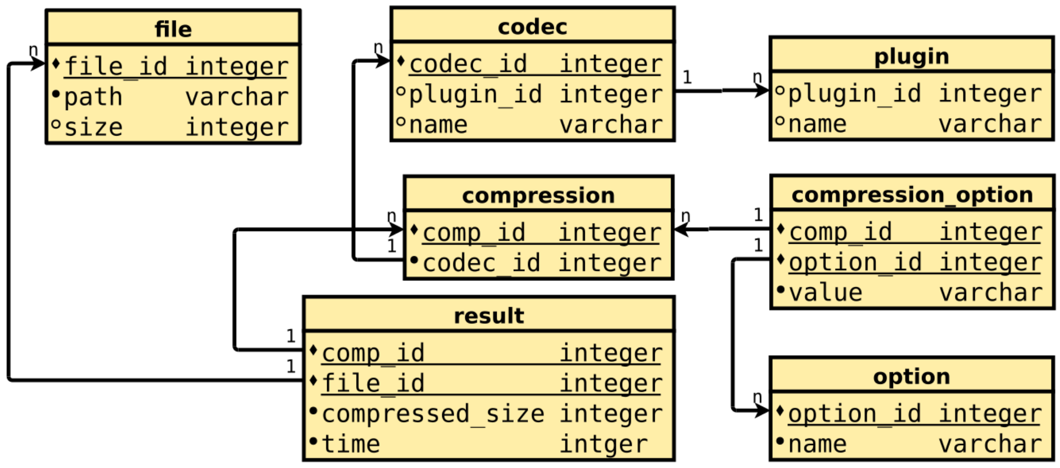

Figure 1: Database Layout

Figure 1 shows the different entities and their relations. Redundant information in the database scheme is reduced with the 3rd normal form.

3.1.2. Robustness

To archive the wanted robustness of the application, I have worked out with my supervisor to use an operation system similar approach. We decided to encapsulate the compression algorithms in their own process. Each individual process then gets monitored. With this method, we were able to restart crashed instances of algorithms on the fly without user

[1] `https://github.com/quixdb/squash`

interference. This feature is mandatory to run this program in a cluster environment. This robustness introduces a higher coordination effort for the different processes and threads.

3.1.3. Communication Scheme

The main components of the system are main thread, a scheduler, several watchdogs and their worker processes.

The scheduler is coordinating the shared memory buffers and job allocation to the different worker processes. After scheduling the jobs, it notifies the corresponding watchdog that the shared memory buffer is ready for reading. It also tells what compression algorithm the worker process will use to benchmark the provided file.

The watchdog communicates over Inter-Process-Communication [IPC] with the worker process and observes if the worker process gets signaled. If a process gets signaled, it gets automatically restarted and thus retries the failed job. An ideal solution would be, that the watchdog only takes a defined number of tries-per-algorithm and then skip the algorithm to overcome the risk of infinite loops.

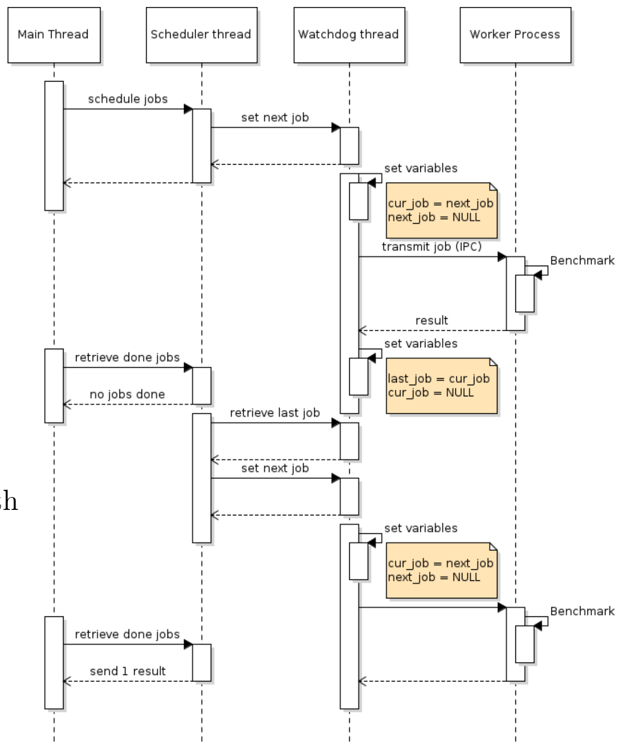

Figure 2: Sequence diagram for IPC communication scheme

An example iteration of the previous steps is given in figure 2. The sequence diagram demonstrates the whole process of scheduling jobs, transmitting it to the worker process and retrieving the results.

3.2. Coding

3.2.1. Shared Memory

To eliminate the bottlenecks of storage and/or network all files which will be benchmarked get loaded into a shared memory block that is accessible by all workers. The Linux kernel offers multiple implementations for shared memory.

Memfd is a new method for fast shared memory. Linux 3.17 introduces this new feature. It does not require a tmpfs mount as well as an shm mount [2]. As the man page of memfd implies there is no present libc binding[3]. Fortunately, it is easy to use Linux syscalls to create a memfd. Listing 1 shows a wrapper function that does this task.

```c
static inline long memfd_create(char* name, int flags)
{
  long fd = syscall(__NR_memfd_create,name,flags);
  check(fd > 0, "failed to create %s memfd file descriptor with 0x%x flags",
        name, flags);
  return fd;
error:
  return -1;
}
```

Listing 1: memfd-wrapper.h

As a fallback, for systems which do not support memfd, the memfd component uses *shm_ open* or *tmpfile* to imitate the behavior of memfd.

```c
  #if defined(HAVE_LINUX_MEMFD_H) && !defined(WITHOUT_MEMFD)
    ctx->fd = memfd_create(path,MFD_ALLOW_SEALING);
  #elif defined(HAVE_SHM_OPEN) && !defined(WITHOUT_SHM)
    ctx->fd = shm_open(path,O_CREAT | O_RDWR, S_IRUSR | S_IWUSR);
  #else
    ctx->fd = fileno(tmpfile());
  #endif
```

Listing 2: ocmemfd.c fallback mechanic for systems without memfd support

3.2.2. SQLite Database

The second component of the project is the implementation of the data model discussed in Section 3.1.1. One of the key features of the project is the easy evaluation of data generated by this tool. this easy evaluation is achieved through using SQLite as the database backend for the data-model. The structured-query-language offers various possibilities to process the generated information.

To enable queries like "Give me all files compressed with codec c and options i,k,j", all results in our database which use the same codec and options need to have a reference to the specific compression id. To maintain this integrity of our data, we need a method to check if a specific combination of codec and option already exists in our data-set. Since such an entry can have *n* options assigned, we need to dynamically generate a SQL query

[2]https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=9183df25fe7b194563db3fec6dc3202a5855839c

[3]http://man7.org/linux/man-pages/man2/memfd_create.2.html

that then is executed by the underlying SQL engine to check for an existing compression id.

The basic stub for searching a codec-option combination is shown in Listing 3

```
char * fmtquerry =    "SELECT comp_id, count(comp_id) as foundops"
                      "   FROM compression_option_codec_view"
                      "   WHERE codec_id = %d"
                      "   AND (%s)"
                      "   GROUP BY comp_id"
                      "   ORDER BY foundops DESC;";
```

Listing 3: ocdata.c Query to find existing compressions

The placeholder *%s* at line 4 in Listing 3 gets replaced with a dynamic generated key-value query of all *compression_options* for this particular compression for which we want to find if there is an existing compression id for it. Listing 4 displays the key-value format string. Now we only needs to concatenate each key-value pair of the list to one expression which then replaces the placeholder *%s*.

```
char * fmtAnd     =    "(OP_value = \"%s\" AND OP_name = \"%s\")";
```

Listing 4: ocdata.c key-value expression

This constructed query is then executed in *ocdata_get_comp_id(ocdataContext* ctx, ocdataCompresion* compression)* function call. This function gets called every time a new result comes from a worker. This could be avoided if the result would be cached for each combination. The performance gain by this optimization is compared to its programming effort not relevant enough for this particular use case. The program spends its most time in the compression algorithms. It may be covered in future releases of this software.

3.2.3. Interprocess communication

The third component of open compress benchmark is the IPC communication. It provides pthread like control over child processes. The main function primitive is

```
ocschedProcessContext * ocsched_fork_process(ocschedFunction
    work_function, char* childname, void* data)
```

Listing 5: ocsched.c forking primitive

The function takes care of the opening and closing of communication pipes between the primary process and its children. The main benefit of this approach is the ability to change the forking behavior of the children. One might change the implementation to a UNIX socket communication. The process creation function also allocates a POSIX message queue which is currently not used as of this writing.

The component also provides primitives for sending and receiving messages. To efficiently communicate with formatted strings used in printf, it also has a printf syntax compatible function shown in Listing 6

```
ocsched_printf(ocschedProcessContext * ctx,char * fmtstr,...)}
```

Listing 6: ocsched.c printf call for IPC

To maintain testability this component only contains process forking and communication functions. This also enables to change used backend of the Operating system or even port it to platforms like Windows, Mac OS or BSD.

3.2.4. Benchmark Process

The benchmark component combines all three previous components. The IPC module is used for the separation of main process and benchmark processes as stated in section 3.1.3. The main process hosts the scheduler thread and the watchdog threads. Each worker process has its own watchdog thread to monitor its state and restarts, if an error occurs.

Algorithm 1: compression

```
Data: bytestream
Result: runtime
while runtime < 1 second do
    start timer
    compress(bytestream)
    stop timer
    add needed time to runtime
end
divide runtime by compression cyles
```

Algorithm 2: decompression

```
Data: bytestream
Result: runtime
while runtime < 1 second do
    start timer
    decompress(bytestream)
    stop timer
    add needed time to runtime
end
divide runtime by decompression cycles
```

The benchmark measures the time spent in the individual algorithm implementation supplied by squash. To accommodate small fluctuations for smaller files the compression/decompression is repeated until 1 second of total runtime is reached. The process is sketched in Algorithm 1 and 2

The scheduler takes care of starting watchdogs according to worker count given by parameter. It then receives a file set and a list of codecs to test. These file codec pairs get transformed into jobs. Before scheduling a job to a worker it checks if the file of the job is loaded into memory, if not it feeds the file into a shared memory context provided by ocmemfd. After the file is loaded into its buffer, the scheduler begins to feed each worker with a job until there is no more job available for the loaded file. The buffer then gets released and continues with the next file in queue.

The watchdog takes care of the communication to and from his associated worker process. Experimental compression algorithms can sometimes segfault on some files, to accommodate this the watchdog constantly monitors the state of its process and, if needed, restarts the worker and retry the file. This process is then repeated up to 3 times. After 3 tries the watchdog gives up this algorithm and requests a new job from the scheduler.

3.3. Usage and Examples

For demonstration purpose the squash benchmark testset of files [4] was used to show how this project can be used to choose an appropriate algorithm. To start the benchmarking process the program needs:

- A directory, in which all files will be benchmarked
- A location of the database file, in which the results get stored
- A set of algorithms that will be tested
- A worker count

The worker count influences the quality and the runtime of the benchmark. To get fast insight if a file set is compressible the worker count should be set to around the amount of real cores in the system. To get the most precise results setting the worker count to 1 is more feasible.

Given the test set of files of the squash benchmark we want to have a fast insight by what factor storage capacity could be increased by compression at filesystem level. To accomplish this goal we start the benchmarking tool with the first line shown in Listing 7.

```
./ocbench -c "all" -n 4 -d ./squash-benchmark-files -D results-4
   cores.sqlite
./evaluateResults results-4cores.sqlite
```

Listing 7: Benchmark of the squash test set files

In the second line of Listing 7 the evaluation script that comes with ocbench gets executed and generates comma-separated values files for the benchmark results of the sqlite database and also some basic gnuplot graphs, an example is shown in Figure 3

The generated graphs only display algorithms in the squash suite which are capable of at least 10 MB/s write throughput, since having a good compression ratio with low data rate is not practical for compression at filesystem level.

The algorithm name structure in Figure 3 is *(plugin-name)-(codec-name)*. Some of the listed algorithms are headless versions. As an example: lz4-lz4 contains header information in the compression stream lz4-lz4-raw does not have them[5]. These information

[4] `https://github.com/quixdb/squash-benchmark`
[5] `https://github.com/quixdb/squash/blob/master/plugins/lz4/lz4.md`

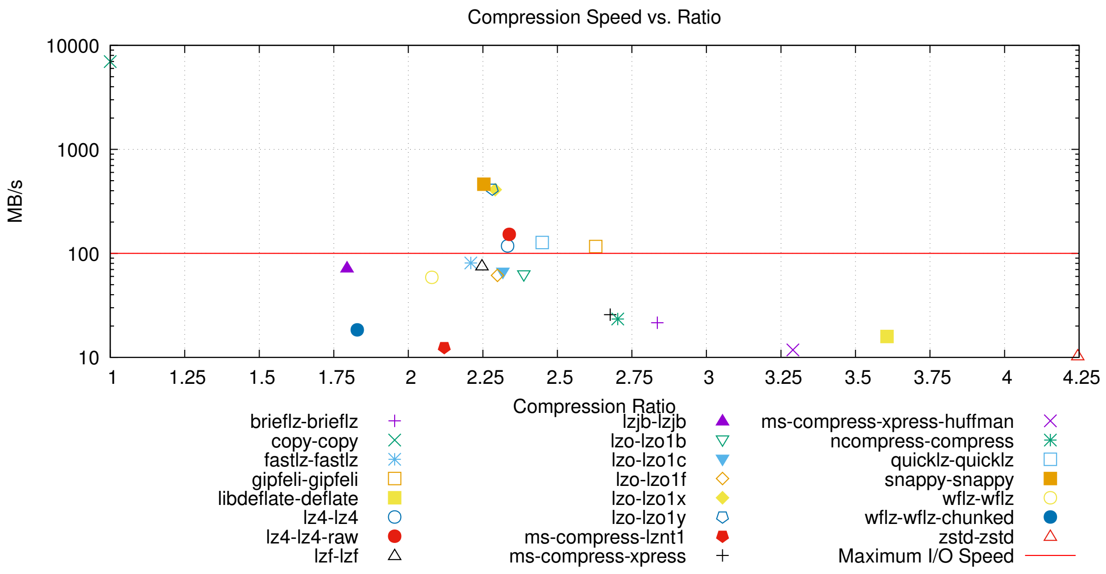

Figure 3: Compression speed / Compression Ratio comparison

are documented in the squash api[6]. Note: some information in this API applies only for the 0.7 branch of squash. squash 0.8 adds various new codec variations (with/without header).

Snappy[7] is a compression library that aims high throughput by acceptable compression ratios. Snappy is maintained by google and is used in their ”BigTable” and ”MapReduce” applications.

Gipfeli[8] is also a library that aims for high throughput rates and is also from Google. It claims to be about 5 times faster than zlib. The results in Figure 3 emphasize this claim since zlib did not even get 10MB/s compression speed to get listed.

Quicklz[9] claims to be the fastest compression library. It supports streaming of very small chunks (200 to 300 bytes). In this benchmark run for the set of files their claim being the fastest is not confirmed, but may apply for other files.

At the current development state of this project only the default configuration of each compression algorithm is used. It is planed to add these features in a later version of open compress benchmark.

If we look at Figure 3 the algorithms gipfeli and quicklz perform very good (above our 100 MB/s I/O limit). Gipfeli has a ratio of 2.62 and 116 MB/s average speed. Quicklz has a ratio of 2.44 and 127 MB/s average speed.

[6]`https://quixdb.github.io/squash/api/c/index.html`
[7]`https://google.github.io/snappy/`
[8]`https://github.com/google/gipfeli`
[9]`http://www.quicklz.com/`

Algorithm	Ratio	Min speed	Avg speed	Max speed
gipfeli	2.62	48.36 MB/s	116 MB/s	885.24 MB/s
quicklz	2.44	65.67 MB/s	127 MB/s	455.93 MB/s
snappy	2.25	52.94 MB/s	462.21 MB/s	7689.62 MB/s

Table 1: Compression statistic of 3 algorithms (1 Core)

Another interesting metric is also minimum and maximum speed of each algorithm which is shown in Table 1. By looking at the minimum speed we encounter that quicklz offer an 25 % higher worst case performance than gipfeli by only a reduction of 7.5 % of its compression ratio. If a higher average and maximum performance is wanted snappy seems to be a good choice given its good compression ratio of 2.25.

Algorithm	Ratio	Min speed	Avg speed	Max speed
gipfeli	2.62	580.31 MB/s	1392 MB/s	10622 MB/s
quicklz	2.44	788.04 MB/s	1524 MB/s	5471.16 MB/s
snappy	2.25	635.28 MB/s	5546.51 MB/s	92275.44 MB/s

Table 2: Compression statistic of 3 algorithms (12 Cores)

One should also take into account that at filesystem level the cores used for compression can and should be higher than this single core benchmark shows. If we take as an example a 12 Core / 24 Thread machine we can have 12 concurrent streams (if the compression algorithm does not support multithreading) which results in the overall throughput in Table 2. With quicklz we can than already utilize two SATA-3 SSD's to their full link speed.

4. Conclusion & Future work

In this current state of development the tool already serves most of its requirements. As shown in Section 3.3 it is very useful to get an insight if for a given fileset filesystem compression could achieve an increasement of capacity and throughput.

The program is missing some key features to be considered production ready and usable for all compression benchmark purposes. It currently lacks the ability to split a file into chunks, so it is not able to process large files that would fill the entire memory. The evaluation tools will also need some more work to better support the finding of a suitable algorithm.

References

[KKL16] Michael Kuhn, Julian Kunkel, and Thomas Ludwig. "Data Compression for Climate Data". In: *Supercomputing frontiers and innovations* 3.1 (2016). ISSN: 2313-8734. URL: `http://superfri.org/superfri/article/view/101`.

Appendices

A. Used Technology (Hard & Software)

A.1. Hardware

1. Dell 755 Headless PC at UHH VSYS Research Group
    - Intel(R) Core(TM)2 Quad CPU Q6600 @ 2.4GHz
    - 8 GiB DDR2 Ram
2. Dell 980 Office Pc at UHH VSIS department
    - Intel(R) Core(TM) i5 CPU 750
    - 8 GiB DDR3 Ram
    - Nvidia Quadro NVS 295

A.2. Software

1. Gnu Compiler Toolchain
2. cmake
3. latex + bibtex
4. squash Compression Layer
5. Sqlite3
6. Atom source code editor
7. Dia diagram editor