Intel® ISA-L: Semi-Dynamic Compression Algorithms

By Thai Le (Intel), Steven B. (Intel), Added December 29, 2016

Download Code Sample

Download PDF

Introduction

DEFLATE compression algorithms traditionally use either a dynamic or static compression table. Those who want the best compression results use a dynamic table at the cost of more processing time, while the algorithms focused on throughput will use static tables. The Intel® Intelligent Storage Acceleration Library (Intel® ISA-L) semi-dynamic compression comes close to getting the best of both worlds. In addition, Intel® ISA-L offers a version of the decompression (inflate) algorithm which substantially improves the decompression performance.

Testing shows the usage of semi-dynamic compression and decompression is only slightly slower than using a static table and almost as space-efficient as algorithms that use dynamic tables. This article's goal is to help you incorporate Intel ISA-L's semi-dynamic compression and optimized decompression algorithms into your storage application. It describes prerequisites for using Intel ISA-L, and includes a downloadable code sample, with full build instructions. The code sample is a compression and decompression tool that can be used to compare the ration and performance of Intel ISA-L's semi-dynamic compression algorithm on a public data set with the standard DEFLATE implementation, zlib*.

Hardware and Software Configuration

CPU and Chipset	**Intel® Xeon® processor E5-2699 v4, 2.2 GHz** Number of cores per chip: 22 (only used single core) Number of sockets: 2 Chipset: Intel® C610 series chipset, QS (B-1 step) System bus: 9.6 GT/s Intel® QuickPath Interconnect Intel® Hyper-Threading Technology off Intel SpeedStep® technology enabled

	Intel® Turbo Boost Technology disabled
Platform	**Platform: Intel® Server System R2000WT product family (code-named Wildcat Pass)** BIOS: GRRFSDP1.86B.0271.R00.1510301446 ME:V03.01.03.0018.0 BMC:1.33.8932 DIMM slots: 24 Power supply: 1x1100W
Memory	Memory size: 256 GB (16X16 GB) DDR4 2133P Brand/model: Micron – MTA36ASF2G72PZ2GATESIG
Storage	Brand and model: 1 TB Western Digital (WD1002FAEX) Intel® SSD Data Center P3700 Series (SSDPEDMD400G4)
Operating System	Ubuntu* 16.04 LTS (Xenial Xerus) Linux* kernel 4.4.0-21-generic

Note: Depending on the platform capability, Intel ISA-L can run on various Intel® processor families. Improvements are obtained by speeding up the computations through the use of the following instruction sets:

- Intel® Advanced Encryption Standard New Instruction (Intel® AES-NI)
- Intel® Streaming SIMD Extensions (Intel® SSE)
- Intel® Advanced Vector Extensions (Intel® AVX)
- Intel® Advanced Vector Extensions 2 (Intel® AVX2)

Why Use Intel® Intelligent Storage Library (Intel® ISA-L)?

Intel ISA-L has the ability to compress and decompress faster than zlib* with only a small sacrifice in the compression ratio. This capability is well suited for high throughput storage applications. This article includes a sample application that simulates a compression and decompression scenario where the output will show the efficiency. Click on the button at the top of this article to download.

Prerequisites

Intel ISA-L supports Linux and Microsoft Windows*. A full list of prerequisite packages can be found here.

Building the sample application (for Linux):

1. Install the dependencies:

- a c++14 compliant c++ compiler
- cmake >= 3.1
- git
- autogen
- autoconf
- automake
- yasm and/or nasm
- libtool
- boost's "Filesystem" library and headers
- boost's "Program Options" library and headers
- boost's "String Algo" headers

```
>sudo apt-get update
>sudo apt-get install gcc g++ make cmake git zlib1g-dev autogen autoconf
automake yasm nasm libtool libboost-all-dev
```

2. You also need the latest versions of isa-l and zlib. The get_libs.bash script can be used to get them. The script will download the two libraries from their official GitHub* repositories, build them, and then install them in `./libs/usr` directory.

```
>`bash ./libs/get_libs.bash`
```

3. Build from the `ex1` directory:

- `mkdir <build-dir>`
- `cd <build-dir>`
- `cmake -DCMAKE_BUILD_TYPE=Release $OLDPWD`

- `make`

Getting Started with the Sample Application

The sample application contains the following files:

```
plse@ebi2s22c02:/home/ISA-L-examples/ex1/src$ ls
benchmark_info.cpp  bm_isal.cpp          file_wrapper.cpp  reporter.h
benchmark_info.h    bm_isal.h            file_wrapper.h    reporter_utils.cpp
benchmarks.cpp      bm_isal_semidyn.cpp  main.cpp          reporter_utils.h
benchmarks.h        bm_isal_semidyn.h    options.cpp
bm.cpp              bm_zlib.cpp          options.h
bm.h                bm_zlib.h            reporter.cpp
plse@ebi2s22c02:/home/ISA-L-examples/ex1/src$
```

This example goes through the following steps at a high-level work flow and focuses on the “main.cpp”, “bm_isal.cpp”, and “bm_isal_semidyn.cpp" files:

Setup

1. In the “main.cpp” file, the program parses the command line and displays the options that are going to be performed.

```
int main(int argc, char* argv[])
{
     options options = options::parse(argc, argv);
```

Parsing the option of the command line

2. In the options.cpp file, the program parses the command line arguments using `options::parse()`.

Create the benchmarks object

3. In the “main.cpp” file, the program will benchmark each raw file using a compression-level inside the benchmarks::add_benchmark() function. Since the benchmarks do not run concurrently, there is only one file “pointer” created.

```
benchmarks benchmarks;

// adding the benchmark for each files and libary/level combination
 for (const auto& path : options.files)
 {
```

```
    auto compression   = benchmark_info::Method::Compression;
    auto decompression = benchmark_info::Method::Decompression;
    auto isal_static   = benchmark_info::Library::ISAL_STATIC;
    auto isal_semidyn  = benchmark_info::Library::ISAL_SEMIDYN;
    auto zlib          = benchmark_info::Library::ZLIB;


    benchmarks.add_benchmark({compression, isal_static, 0, path});
    benchmarks.add_benchmark({decompression, isal_static, 0, path});


    if (options.isal_semidyn_stateful)
    {
        benchmarks.add_benchmark({compression, isal_semidyn, 0, path});
        benchmarks.add_benchmark({decompression, isal_semidyn, 0, path});
    }
    if (options.isal_semidyn_stateless)
    {
        benchmarks.add_benchmark({compression, isal_semidyn, 1, path});
        benchmarks.add_benchmark({decompression, isal_semidyn, 1, path});
    }


    for (auto level : options.zlib_levels)
    {
        if (level >= 1 && level <= 9)
        {
            benchmarks.add_benchmark({compression, zlib, level, path});
            benchmarks.add_benchmark({decompression, zlib, level, path});
        }
        else
        {
            std::cout << "[Warning] zlib compression level " << level
<< "will be ignored\n";
        }
    }
 }
```

Intel® ISA-L compression and decompression

4. In the “bm_isal.cpp” file, the program performs the static compression and decompression on the raw file using a single thread. The key functions to note are isal_deflate and isal_inflate. Both

functions accept a stream as an argument, and this data structure holds the data about the input buffer, the length in bytes of the input buffer, and the output buffer and the size of the output buffer. end_of_stream indicates whether it will be last iteration.

```
std::string bm_isal::version()
{
    return std::to_string(ISAL_MAJOR_VERSION) + "." +
std::to_string(ISAL_MINOR_VERSION) + "." +
           std::to_string(ISAL_PATCH_VERSION);
}

bm::raw_duration bm_isal::iter_deflate(file_wrapper* in_file, file_wrapper*
out_file, int /*level*/)
{
    raw_duration duration{};

    struct isal_zstream stream;

    uint8_t input_buffer[BUF_SIZE];
    uint8_t output_buffer[BUF_SIZE];

    isal_deflate_init(&stream);
    stream.end_of_stream = 0;
    stream.flush         = NO_FLUSH;

    do
    {
        stream.avail_in      = static_cast<uint32_t>(in_file-
>read(input_buffer, BUF_SIZE));
        stream.end_of_stream = static_cast<uint32_t>(in_file->eof());
        stream.next_in       = input_buffer;
        do
        {
            stream.avail_out = BUF_SIZE;
            stream.next_out  = output_buffer;

            auto begin = std::chrono::steady_clock::now();
            isal_deflate(&stream);
            auto end = std::chrono::steady_clock::now();
            duration += (end - begin);

            out_file->write(output_buffer, BUF_SIZE - stream.avail_out);
```

```
        } while (stream.avail_out == 0);
    } while (stream.internal_state.state != ZSTATE_END);

    return duration;
}

bm::raw_duration bm_isal::iter_inflate(file_wrapper* in_file, file_wrapper*
out_file)
{
    raw_duration duration{};

    int         ret;
    int         eof;
    struct inflate_state stream;

    uint8_t input_buffer[BUF_SIZE];
    uint8_t output_buffer[BUF_SIZE];

    isal_inflate_init(&stream);

    stream.avail_in = 0;
    stream.next_in  = nullptr;

    do
    {
        stream.avail_in = static_cast<uint32_t>(in_file->read(input_buffer,
BUF_SIZE));
        eof             = in_file->eof();
        stream.next_in  = input_buffer;
        do
        {
            stream.avail_out = BUF_SIZE;
            stream.next_out  = output_buffer;

            auto begin = std::chrono::steady_clock::now();
            ret        = isal_inflate(&stream);
            auto end   = std::chrono::steady_clock::now();
            duration += (end - begin);

            out_file->write(output_buffer, BUF_SIZE - stream.avail_out);
```

```
        } while (stream.avail_out == 0);
    } while (ret != ISAL_END_INPUT && eof == 0);

    return duration;
}
```

5. In the “bm_isal_semidyn.cpp” file, the program performs the dynamic compression and decompression on the raw file using multiple threads.

```
Std::string bm_isal_semidyn::version()
 {
     return std::to_string(ISAL_MAJOR_VERSION) + “.” +
std::to_string(ISAL_MINOR_VERSION) + “.” +
            std::to_string(ISAL_PATCH_VERSION);
 }

 bm::raw_duration
 bm_isal_semidyn::iter_deflate(file_wrapper* in_file, file_wrapper*
out_file, int config)
 {
     raw_duration duration{};

     bool stateful = (config == 0);

     struct isal_zstream        stream;
     struct isal_huff_histogram histogram;
     struct isal_hufftables     hufftable;

     long   in_file_size = in_file->size();
     uint8_t* input_buffer = new (std::nothrow) uint8_t[in_file_size];

     if (input_buffer == nullptr)
         return raw_duration{0};

     long   out_buffer_size = std::max((int)(in_file_size * 1.30), 4 * 1024);
     uint8_t* output_buffer   = new (std::nothrow) uint8_t[out_buffer_size];

     if (output_buffer == nullptr)
```

```
        return raw_duration{0};

    stream.avail_in = static_cast<uint32_t>(in_file->read(input_buffer,
in_file_size));
    if (stream.avail_in != in_file_size)
        return raw_duration{0};

    int segment_size = SEGMENT_SIZE;
    int sample_size  = SAMPLE_SIZE;
    int hist_size    = sample_size > segment_size ? segment_size :
sample_size;

    if (stateful)
        isal_deflate_init(&stream);
    else
        isal_deflate_stateless_init(&stream);

    stream.end_of_stream = 0;
    stream.flush         = stateful ? SYNC_FLUSH : FULL_FLUSH;
    stream.next_in       = input_buffer;
    stream.next_out      = output_buffer;
    if (stateful)
        stream.avail_out = out_buffer_size;
    int remaining        = in_file_size;
    int chunk_size       = segment_size;

    while (remaining > 0)
    {
        auto step = std::chrono::steady_clock::now();
        memset(&histogram, 0, sizeof(struct isal_huff_histogram));
        duration += std::chrono::steady_clock::now() - step;

        if (remaining < segment_size * 2)
        {
            chunk_size           = remaining;
            stream.end_of_stream = 1;
        }

        step         = std::chrono::steady_clock::now();
```

```
        int hist_rem = (hist_size > chunk_size) ? chunk_size : hist_size;
        isal_update_histogram(stream.next_in, hist_rem, &histogram);
        if (hist_rem == chunk_size)
            isal_create_hufftables_subset(&hufftable, &histogram);
        else
            isal_create_hufftables(&hufftable, &histogram);
        duration += std::chrono::steady_clock::now() - step;

        stream.avail_in = chunk_size;
        if (!stateful)
            stream.avail_out = chunk_size + 8 * (1 + (chunk_size >> 16

        stream.hufftables = &hufftable;
        remaining -= chunk_size;
        step = std::chrono::steady_clock::now();
        if (stateful)
            isal_deflate(&stream);
        else
            isal_deflate_stateless(&stream);
        duration += std::chrono::steady_clock::now() - step;

        if (stateful)
        {
            if (stream.internal_state.state != ZSTATE_NEW_HDR)
                break;
        }
        else
        {
            if (stream.avail_in != 0)
                break;
        }
    }

    if (stream.avail_in != 0)
        return raw_duration{0};

     out_file->write(output_buffer, stream.total_out);

    delete[] input_buffer;
    delete[] output_buffer;
```

```
    return duration;
}

bm::raw_duration bm_isal_semidyn::iter_inflate(file_wrapper* in_file,
file_wrapper* out_file)
{
    raw_duration duration{};

    int         ret;
    int         eof;
    struct inflate_state stream;

    uint8_t input_buffer[INFLATE_BUF_SIZE];
    uint8_t output_buffer[INFLATE_BUF_SIZE];

    isal_inflate_init(&stream);

    stream.avail_in = 0;
    stream.next_in  = nullptr;

    do
    {
        stream.avail_in = static_cast<uint32_t>(in_file->read(input_buffer,
INFLATE_BUF_SIZE));
        eof             = in_file->eof();
        stream.next_in  = input_buffer;
        do
        {
            stream.avail_out = INFLATE_BUF_SIZE;
            stream.next_out  = output_buffer;



            auto begin = std::chrono::steady_clock::now();
            ret        = isal_inflate(&stream);
            auto end   = std::chrono::steady_clock::now();
            duration += (end - begin);

            out_file->write(output_buffer, INFLATE_BUF_SIZE -
stream.avail_out);
```

```
        } while (stream.avail_out == 0);
    } while (ret != ISAL_END_INPUT && eof == 0);

    return duration;
}
```

6. When all compression and decompression tasks are complete, the program displays the results on the screen. All temporary files are deleted using `benchmarks.run()`.

Execute the sample application

In this example, the program will run through the compression and decompression functions of the Intel ISA-L and zlib. For Intel ISA-L functions, the results will show both static and semi-dynamic compression and decompression.

Run

From the ex1 directory:

```
cd <build-bir>/ex1

./ex1 --help
```

Usage

```
Usage: ./ex1 [--help] [--folder <path>]... [--file <path>]... :
  --help                     display this message
  --file path                use the file at 'path'
  --folder path              use all the files in 'path'
  --zlib-levels n,...        coma-separated list of compression level [1-9]
  --semidyn-config flag,...  coma-separated list of flags for the semi-dynamic
                             compression ('stateful', 'stateless') [stateful]
```

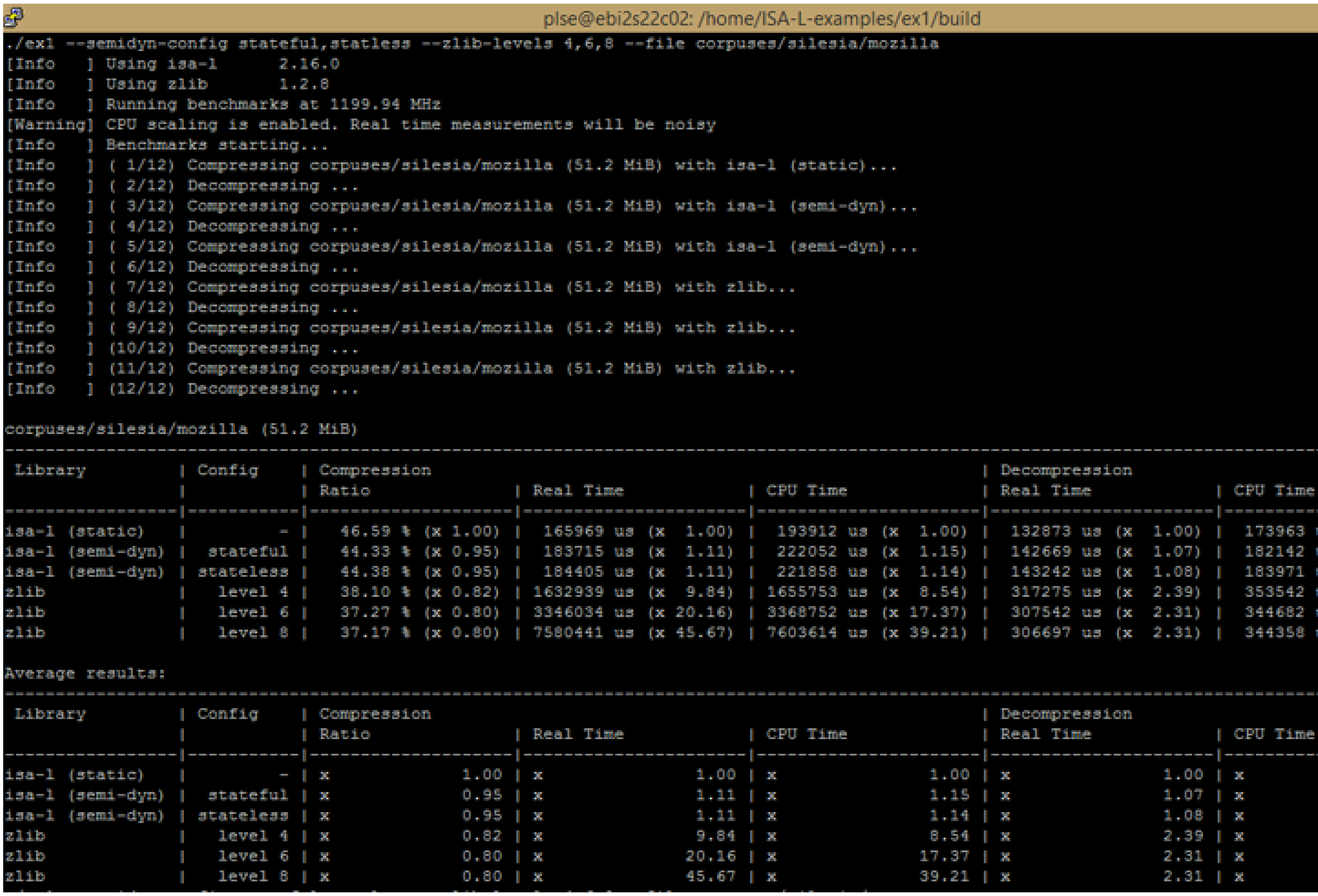
- `--file and --folder can be used multiple times to add more files to the benchmark`
- `--folder will look for files recursively`
- `the default --zlib-level is 6`

Test corpuses are public data files designed to test the compression and decompression algorithms, which are available online (for example, Calgary and Silesia corpuses). The --folder option can be used to easily benchmark them: `./ex1 --folder /path/to/corpus/folder`.

Running the example

Here is an example of how to run the application:

```
./ex1 -zlib-levels 4,6,8 -file corpuses/silesia/mozilla
```



```
plse@ebi2s22c02: /home/ISA-L-examples/ex1/build
./ex1 --semidyn-config stateful,statless --zlib-levels 4,6,8 --file corpuses/silesia/mozilla
[Info   ] Using isa-l      2.16.0
[Info   ] Using zlib       1.2.8
[Info   ] Running benchmarks at 1199.94 MHz
[Warning] CPU scaling is enabled. Real time measurements will be noisy
[Info   ] Benchmarks starting...
[Info   ] ( 1/12) Compressing corpuses/silesia/mozilla (51.2 MiB) with isa-l (static)...
[Info   ] ( 2/12) Decompressing ...
[Info   ] ( 3/12) Compressing corpuses/silesia/mozilla (51.2 MiB) with isa-l (semi-dyn)...
[Info   ] ( 4/12) Decompressing ...
[Info   ] ( 5/12) Compressing corpuses/silesia/mozilla (51.2 MiB) with isa-l (semi-dyn)...
[Info   ] ( 6/12) Decompressing ...
[Info   ] ( 7/12) Compressing corpuses/silesia/mozilla (51.2 MiB) with zlib...
[Info   ] ( 8/12) Decompressing ...
[Info   ] ( 9/12) Compressing corpuses/silesia/mozilla (51.2 MiB) with zlib...
[Info   ] (10/12) Decompressing ...
[Info   ] (11/12) Compressing corpuses/silesia/mozilla (51.2 MiB) with zlib...
[Info   ] (12/12) Decompressing ...

corpuses/silesia/mozilla (51.2 MiB)
---------------------------------------------------------------------------------------------------------------------------
 Library          | Config     | Compression                                                       | Decompression
                  |            | Ratio               | Real Time             | CPU Time              | Real Time             | CPU Time
------------------|------------|---------------------|-----------------------|-----------------------|-----------------------|---------
isa-l (static)    |          - |   46.59 % (x 1.00)  |  165969 us  (x  1.00) |  193912 us  (x  1.00) |  132873 us  (x  1.00) |  173963
isa-l (semi-dyn)  |   stateful |   44.33 % (x 0.95)  |  183715 us  (x  1.11) |  222052 us  (x  1.15) |  142669 us  (x  1.07) |  182142
isa-l (semi-dyn)  |  stateless |   44.38 % (x 0.95)  |  184405 us  (x  1.11) |  221858 us  (x  1.14) |  143242 us  (x  1.08) |  183971
zlib              |    level 4 |   38.10 % (x 0.82)  | 1632939 us  (x  9.84) | 1655753 us  (x  8.54) |  317275 us  (x  2.39) |  353542
zlib              |    level 6 |   37.27 % (x 0.80)  | 3346034 us  (x 20.16) | 3368752 us  (x 17.37) |  307542 us  (x  2.31) |  344682
zlib              |    level 8 |   37.17 % (x 0.80)  | 7580441 us  (x 45.67) | 7603614 us  (x 39.21) |  306697 us  (x  2.31) |  344358

Average results:
---------------------------------------------------------------------------------------------------------------------------
 Library          | Config     | Compression                                                       | Decompression
                  |            | Ratio               | Real Time             | CPU Time              | Real Time             | CPU Time
------------------|------------|---------------------|-----------------------|-----------------------|-----------------------|---------
isa-l (static)    |          - | x              1.00 | x                1.00 | x                1.00 | x                1.00 | x
isa-l (semi-dyn)  |   stateful | x              0.95 | x                1.11 | x                1.15 | x                1.07 | x
isa-l (semi-dyn)  |  stateless | x              0.95 | x                1.11 | x                1.14 | x                1.08 | x
zlib              |    level 4 | x              0.82 | x                9.84 | x                8.54 | x                2.39 | x
zlib              |    level 6 | x              0.80 | x               20.16 | x               17.37 | x                2.31 | x
zlib              |    level 8 | x              0.80 | x               45.67 | x               39.21 | x                2.31 | x
```

Program output displays a column for the compression library, either ‘isa-l’ or ‘zlib’. The table shows the compression ratio (compressed file/raw file), and the system and processor time that it takes to perform the operation. For decompression, it just measures the elapsed time for the decompression operation. All the data was produced on the same system. Both results for Intel ISA-L results of static and semi-dynamic compression and decompression are displayed in the table.

Notes: 2x Intel® Xeon® processor E5-2699v4 (HT off), Intel® Speed Step enabled, Intel® Turbo Boost Technology disabled, 16x16GB DDR4 2133 MT/s, 1 DIMM per channel, Ubuntu 16.04 LTS, Linux kernel 4.4.0-21-generic, 1 TB Western Digital* (WD1002FAEX), 1 Intel® SSD P3700 Series (SSDPEDMD400G4), 22x per CPU socket. Performance measured by the written sample application in this article.*

Conclusion

This tutorial and its sample application demonstrates one method through which you can incorporate the Intel ISA-L static and semi-dynamic compression and decompression features into your storage application. The sample application's output data shows there is a balancing act between processing time (CPU time) and disk space. It can assist you in determining which compression and decompression algorithm best suits your requirements, then help you to quickly adapt your application to take advantage of Intel® Architecture with the Intel ISA-L.

Other Useful Links

- Accelerating your Storage Algorithms using Intelligent Storage Acceleration Library (ISA-L) video
- Accelerating Data Deduplication with ISA-L blog post

Authors

Thai Le is a software engineer who focuses on cloud computing and performance computing analysis at Intel.

Steven Briscoe is an application engineer focusing on cloud computing within the Software Services Group at Intel Corporation (UK).

Notices

System configurations, SSD configurations and performance tests conducted are discussed in detail within the body of this paper. For more information go to http://www.intel.com/content/www/us/en/benchmarks/intel-product-performance.html.

intel-isa-l-semi-dynamic-compression-algorithms.pdf (1000.22 KB)**Download Now**