# Compression Speed Enhancements to LZO for Multi-Core Systems

Jason Kane, *Student Member, IEEE*, Qing Yang, *Fellow, IEEE*
Department of Electrical, Computer and Biomedical Engineering
University of Rhode Island
Kingston, RI, 02881, USA
E-mail: {jkane,qyang}@ele.uri.edu

*Abstract*— **This paper examines several promising throughput enhancements to the Lempel-Ziv-Oberhumer (LZO) 1x-1-15 data compression algorithm. Of many algorithm variants present in the current library version, 2.06, LZO 1x-1-15 is considered to be the fastest, geared toward speed rather than compression ratio. We present several algorithm modifications tailored to modern multi-core architectures in this paper that are intended to increase compression speed while minimizing any loss in compression ratio. On average, the experimental results show that on a modern quad core system, a 3.9x speedup in compression time is achieved over the baseline algorithm with no loss to compression ratio. Allowing for a 25% loss in compression ratio, up to a 5.4x speedup in compression time was observed.**

***Keywords- Lossless Data Compression; Lempel-Ziv-Oberhumer (LZO); Real-Time Systems; Parallel Computing; Parallel Algorithms.***

## I. INTRODUCTION

Real-time systems are more and more often being required to process an increasing amount of data. Interface throughput and storage bottlenecks may be reached in such systems because of the amount of data involved. Data compression can be used to help alleviate such problems by reducing the amount of data injected into a pipeline. For the purpose of this paper we will consider targeting a theoretical real-time system, requiring a lossless compression system to pass data between two interfaces. Compression may be required in such a situation due to bandwidth limitations or space constraints on the destination interface. We will assume a constant input stream of data is available to the compression device and that the final compressed output is able to be passed to the secondary interface with zero delay. The target lossless compression algorithm for this system will be Lempel-Ziv-Oberhumer (LZO) variant 1x-1-15.

The LZO compression library is a collection of lossless dictionary based data compression algorithms that favor speed over compression ratio. The LZO library was first released in 1996 and has received periodic updates since. The library has experienced widespread use, being implemented in a variety of technologies, including NASA's Mars Rovers [1] and Oracle Corporation's B-tree Linux file system. A comparison of the LZO 1x-1 algorithm speed performance against common compression formats such as GZIP, can be found in [2]. Of the available LZO algorithms, 1x-1-15 is considered by the author, Oberhumer, to have the fastest compression speeds [1], exceeding that of LZO 1x-1 at the cost of compression ratio. This paper investigates possible enhancements to the 1x-1-15 algorithm to improve data compression speeds for use in real-time systems.

Utilizing the special architectural features of modern multi-core processors, we will examine the effects of optimizing LZO in the following ways: parallelizing block compression, using Intel SSE (Streaming SIMD Extensions) vector instructions to perform data copy operations, modifying the search algorithm, enforcing cache-aligned reads, and calculating CRC-32 checksums via hardware. All five enhancements have been implemented on the LZO open source code. Performance evaluation and comparison have been carried out using real world data sets. Experimental results have shown significant performance improvements in terms of compression time. For the same compression ratio, over a factor 3 speedup was observed. If trading off a slightly lower compression ratio is allowed and all enhancements are combined, over a factor 5 speedup was observed.

The remainder of this paper is organized as follows. First, an analysis of the existing LZO 1x-1-15 algorithm is conducted, revealing the overall structure and identifying unique characteristics. Next the proposed enhancements to the algorithm are discussed in detail. After this, experimental data is given to compare against baseline performance. Finally, conclusions regarding the obtained results are presented.

## II. ANALYSIS OF LZO 1X-1-15

LZO 1x-1-15 is a variation of the Lempel-Ziv 1977 (LZ77) compression algorithm, which is described in [3]. LZ77 achieves data compression via a sliding window mechanism: bytes from a look-ahead buffer are shifted one by one into a search buffer. When matches are found between the look-ahead buffer and locations in the search buffer, tokens are output on the compression stream rather than literals, resulting in compression. Major differences in Oberhumer's LZO variation include: the optimization of operations through the use of integer computer hardware, a quick hash lookup table for match data, and better optimized output tokens.

The LZO 1x-1-15 algorithm is structured to take advantage of the fact that most computers are optimized for performing integer operations. Instruction latency/throughput tables illustrating this on Intel ATOM architecture CPUs can be found

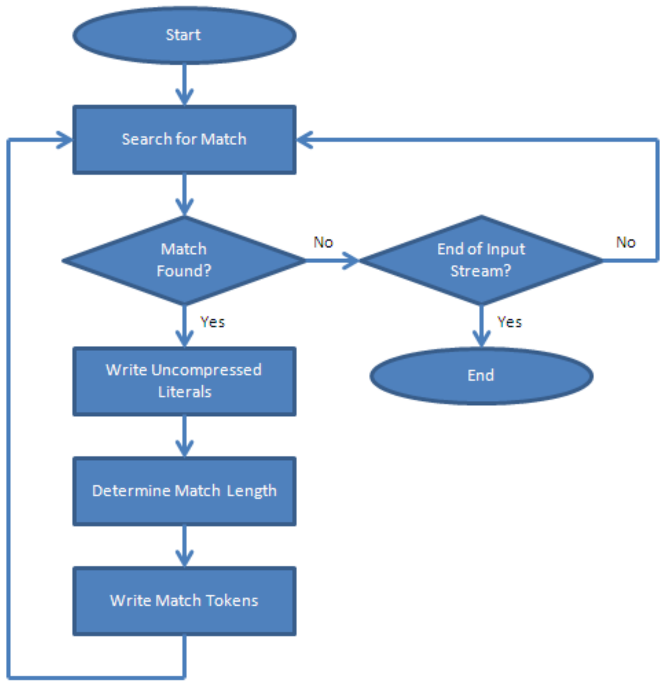

Figure 1. LZO 1x-1-15 Algorithm Flow

in [4]. Throughout the algorithm, no time-consuming floating point operations are used. The most complex instruction performed is an integer-multiply operation, which takes roughly five or less CPU clock cycles on the latest Intel architecture processors [5]. When pipelining and out-of-order execution operations are taken into account, the delay introduced by this multiply instruction is made even less significant.

Fetches to and from cache are also optimized. The algorithm is relatively small when compiled into binary format, likely fitting within a modern level 1 instruction cache. Compiling for i386 target hardware was found to result in roughly 600 instructions, which occupied a total size of roughly 1.84kB. This low instruction count is confirmed by that of the slightly larger, more complex cousin algorithm LZO 1x-1, given in [6]. Since the algorithm operates on data in a block manner, successive iterations do not require the frequent re-fetching of instructions from main memory. Data cache misses are also minimized, as a maximum of 48kB of data at a time is compressed; regardless of the user-defined input block size. On a modern data cache of size 16kB or larger, this will result in few level 1 cache misses. A 48kB sub-block of input data to be compressed should easily fit within level 2 cache, if not most of level 1 cache. Intel processors in particular predict data fetch patterns and automatically pre-fetch sequential data from a detected input stream, resulting in further performance gain during the block compression algorithm [4][7].

As seen in Fig. 1, the 1x-1-15 algorithm itself can be divided up into four major sections of code:

1. Search for a match
2. Write unmatched literal data
3. Determine match length
4. Write match tokens

When searching for match data, the algorithm examines 32-bits of data from the input stream and computes a hash value into a small 8192 (8k) entry dictionary of 16-bit pointers to recently found data. Initially all entries in this dictionary are initialized to point to the same first 32-bit data element in the input stream, which can result in misses for the first several iterations of the search loop, until the dictionary becomes sufficiently populated. The hash value into the dictionary is completed quickly by first using an integer operation to multiply the current 32-bit unsigned data with the fixed value 0x1824429D. Pseudo-randomness, to reduce the frequency of hash table conflicts, is guaranteed based on the fact that the number being multiplied is a fairly large 32-bit prime number. Next, integer shift operations perform a quick division, resulting in the final 13-bit hash index to be used for simple indexing into the dictionary. If a match is found, or if the end of the input stream is reached, the algorithm jumps to copying any previously unmatched literal data to the output stream. Otherwise, the pointer to the next potential data match is obtained from the input stream according to (1); where *ip* is the pointer to the current search location in the input stream and *ii* is a pointer to the location immediately following the last detected match in the input stream, or the beginning of the current 48kB sub-block of data in the instance that a match has yet to be found.

$$ip \mathrel{+}= 1 + ((ip - ii) >> 5); \qquad (1)$$

In (1), first the difference between *ip* and *ii* is taken and the intermediate result is then divided by 32. Finally *ip* is incremented by this value plus one. A difference between *ip* and *ii* indicates the number of missed matches that have occurred. Larger differences have a greater effect on the equation. Fig. 2 illustrates the effect of misses on the input data stream. As consecutive misses become more frequent, *ip*, jumps exponentially until either a match is found or the end of the current 48kB sub-block is reached. The search algorithm appears to make the assumption that data matches are related

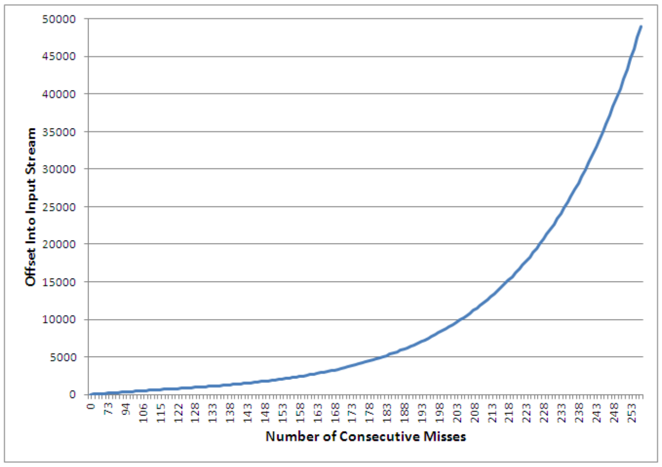

Figure 2. LZO 1x-1-15 Input Stream Traversal (assuming no matches found)

by spatial locality. When a match is found, the search algorithm assumes nearby matches will also occur, only incrementing the search pointer, *ip*, by one byte. As match detections repeatedly fail, *ip* is increasingly incremented in an exponential fashion.

Consider the following simplified example: a 100-byte data set with 4-byte long matches located at byte offsets 30,34,38, and 84. Misses will occur at offsets 0 through 29 and the search pointer *ip* will be incremented by one byte offset each time. Keep in mind that every time a miss occurs, the dictionary is still populated. Then three consecutive matches will occur, with *ip* being updated to the next unmatched location for successive search iterations. From offsets 42 through 73 *ip* is incremented by one byte. From offsets 74 through 82, the difference between the last match, *ii*, and the search pointer has grown beyond a distance of 32, and as such a byte offset of 2 is added to *ip* upon each detected miss. At byte offset 84 a match is found once again. As the match is only 4 bytes in length, the search will resume at byte offset 88 where a miss will occur. Since a match was detected, *ii* was updated and the difference between *ip* and *ii* is once again less than 32. This results in *ip* being incremented by one byte once again rather than two. Miss detections and single offset increments will continue to the end of the dataset.

An interesting aspect of the code used by the algorithm to perform the searches is that it involves only integer operations and no comparison instructions. The effect of this is that the compiled code will produce no conditional branch instructions to calculate the exponential behavior. Consequently, no part of the CPU pipeline will need to be reserved for branch prediction to determine how far to advance into the input data stream on the next iteration. This results in overall faster code.

The second major portion of the LZO 1x-1-15 algorithm performs the copying of unmatched literal data to the output stream. First, the number of bytes of uncompressed literal data is written in an encoded manner to the output stream. This is followed by the literal data. The algorithm assumes a 32-bit data bus, and, when possible, copies data to the output stream in groups of 32-bits in an attempt to optimize any writes performed. After copying the literal data, if the end of the data stream was reached, a special block marker is written to the output stream. Otherwise, the match length is calculated.

The algorithm calculates match length by performing 32-bit XOR operations on sequential sets of data pointed to by the input stream and the dictionary. If a 32-bit match comparison is successful, the result of the XOR operation will be zero. When the comparison fails for the first time, 8-bit byte comparisons are performed to determine if any leftover partial-word singular bytes in the stream matched. Thus, similar to the literal copy operation, the algorithm assumes that the computer's hardware will be optimally used if matches are primarily performed on a 32-bit integer basis.

Following the match length calculation, the algorithm encodes the following tokens on the output stream: a marker denoting the type of match that occurred, the offset to the location of the data that matched the current set, and the length of the match. To minimize the number of bytes used to store this information and improve the compression ratio, five different encodings exist. Each of the five encodings offers a unique variation of offset/length encoding. After outputting the match information, the algorithm returns to searching for the next 32-bit set of matching data.

## III. Algorithm Enhancements

The five enhancements that follow in this section have been applied to the existing LZO 1x-1-15 algorithm to determine the impact on compression performance.

### A. *Parallelization of Block Compression*

The original LZO algorithm is serial in nature. Data is compressed on a block by block basis, but there is no explicit support for multiple CPU hardware cores. By implementing a divide and conquer approach to individually compress and reassemble blocks of data in the input stream, a performance gain should be attained. This performance gain should be directly proportional to the amount of CPU cores utilized.

To investigate this improvement, a thread-based variation of the 1x-1-15 algorithm has been constructed as follows. As seen in Fig. 3, three main types of threads are created: control, compression, and reconstruction. Communication between the three types of threads is accomplished through the use of semaphore-protected shared memory. The main thread is created to control the flow of input data, manage memory utilization, and initiate compression threads. N user-defined threads are created to perform the actual compression on input data blocks when commanded. The compression threads operate independently from one another, each with their own set of temporary resources. The software and operating system ensure the compression threads are load-distributed to all available processor cores in the system. The main control thread will initiate a compression thread when the following conditions are determined: available input data exists, temporary output compression buffers exist, and an idle compression thread exists. Finally, a reconstruction thread is

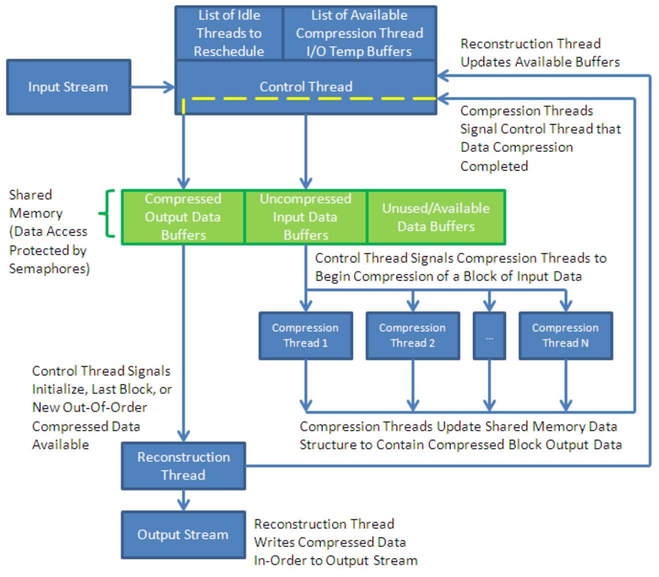
List of Idle Threads to Reschedule
List of Available Compression Thread I/O Temp Buffers
Reconstruction Thread Updates Available Buffers
Input Stream
Control Thread
Compression Threads Signal Control Thread that Data Compression Completed
Shared Memory (Data Access Protected by Semaphores)
Compressed Output Data Buffers
Uncompressed Input Data Buffers
Unused/Available Data Buffers
Control Thread Signals Compression Threads to Begin Compression of a Block of Input Data
Compression Thread 1
Compression Thread 2
...
Compression Thread N
Control Thread Signals Initialize, Last Block, or New Out-Of-Order Compressed Data Available
Compression Threads Update Shared Memory Data Structure to Contain Compressed Block Output Data
Reconstruction Thread
Output Stream
Reconstruction Thread Writes Compressed Data In-Order to Output Stream

Figure 3. Multi-Threaded Implementation

created to accept the output LZO-compressed block data from the individual compression threads. It is the job of the reconstruction thread to ensure that the resultant output stream is created in-order. Since data is being compressed in parallel on separate CPUs, there is no guarantee as to when a particular block of data will finish compression, i.e., Input Block 1 could be compressed first, followed by Block 3, followed by Block 2. Thus, without the reconstruction thread, there would be no mechanism to correctly reconstruct the compressed output data in-order. Unless special headers were added to identify the out-of-order data, existing LZO decompression routines would be incompatible with the produced output format.

### B. *Optimize Copying of Literal Data*

In the current LZO algorithm, uncompressed literal data is copied to the output stream on a 32-bit long word basis. The copying of this data is one of the more time intensive operations. Instead of copying on a 32-bit integer basis, data can be copied faster using available vector instructions. On current generation Intel processors, this allows for up to 128 bits of data to be copied at a time. This should potentially quadruple performance when this portion of the code is executed. To investigate this optimization, Intel SSE (Streaming SIMD (Single Instruction Multiple Data) Extensions) vectorized memory copy related source code from Agner Fog's freely available asmlib library [8] was utilized.

### C. *Search for Matches Every 32-Bits*

LZO 1x-1-15 shifts data from a look-ahead buffer into the search buffer by one byte, similar to LZ77. This algorithm variant has been created to advance the input stream by 32-bits (one integer) each time a match detection fails. In this manner, the input stream will be traversed faster, although with the extra side effect that the dictionary will be less frequently populated. A compression ratio loss is expected, the magnitude of which will need to be determined experimentally.

### D. *Force Cache-Line-Aligned Reads*

Most current generation Intel processors suffer a penalty when reading data that exists between cache line boundaries [7]. Figure 4 illustrates this issue. The search portion of the 1x-1-15 algorithm has the potential of suffering the most from this observation. In the worst case, successive new matches could continually be found on cache line boundaries, incurring a performance penalty each time. By modifying the search equation and match length detection to operate entirely on a 32-bit basis, all accesses to memory can be ensured to be cache-line-aligned.

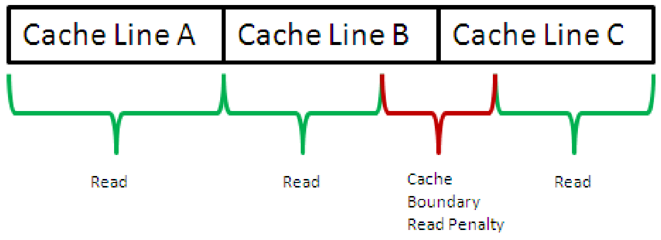

Figure 4. Cache-Line Boundary Read Penalty

Assuming the cache line boundaries in a system lie on an address location that is divisible by 4 bytes, and knowing that up to 4 bytes of data can be read from memory at a time, the following replacement search equation was constructed to ensure that a boundary would never be crossed:

$$ip \mathrel{+}= 4 + (((ip - ii) >> 5) \ \&\ 0xFFFFFFFC); \qquad (2)$$

Four bytes are added to the current input pointer instead of one byte to ensure that the input data is always advanced by 32 bits. The exponential portion of the equation is bit-wise ANDed to ensure that the result of that mathematical operation is 32-bit aligned as well.

Match length calculation was also required to be modified to ensure that reads from the input stream remain on cache line boundaries. Since the existing algorithm allowed for match lengths to be determined on a byte basis, this variation was altered to only perform matches on a 32-bit basis, guaranteeing inter-cache line boundary locations will not be read the next time the search equation is executed.

This modification is in essence an extension of the previously described 32-bit search variation. In addition to searching every 32-bits for a match, the search pointer is verified to lie on a 4-byte boundary and the lengths of matching datasets are terminated early to ensure they result in multiples of 32-bits. Similar to the 32-bit search algorithm, a compression ratio loss is expected. The compression ratio loss will likely be greater due to the fact that matches will be shorter. A speed enhancement may result due to the cache line read penalty being avoided.

### E. *Utilize Hardware CRC-32 Instruction*

The existing LZO library incorporates a CRC-32 calculation routine derived from the freely available zlib library [9]. This variation implements a tabular method using the 0x4C11DB7 polynomial. This method requires a considerable amount of processor utilization to generate checksums. The particular approach taken for this paper was to observe the performance of the relatively new Intel SSE 4.2 assembly instruction "CRC32".

It should be noted that the improved CRC-32 library function is not explicitly utilized by the LZO 1x-1-15 algorithm, or any other of the provided LZO library algorithms for that matter. The function is provided by Oberhumer to the end-user so that CRC's may be calculated on a need basis as determined by the writer of the final produced compression executable. In this manner the end-user is able to adjust the balance between speed and error-checking capability. The author of the LZO library has written and maintained the executable "lzop". The current version of lzop, 1.03, incorporates a small subset of the LZO library for its compression, including the 1x-1, 1x-1-15, and 1x-999 algorithms. This executable calls the CRC-32 library function twice for every block of input data compressed; once to determine the CRC for the original uncompressed data block and once to determine the CRC for the compressed output block. By default, the application sets the user-defined input block size to 256kB.

Table I. Dataset Information

| **Compression Item** | **File Type / Description** | **File Size (GB)** |
|---|---|---|
| images_en.nq | Text (Dump of Wikipedia Thumbnail Links) | 1.2 |
| e-coli | HDF RAW Binary (E-Coli C227 Strain Information) | 30.0 |
| gb-1gram-(n).csv | Text, where n= 0 to 9. (From Google Books 1-gram corpus) | 0.95 |
| ERR007772_(n).fastq | FASTQ Text-Based, where n= 1,2. (Mouse Genome Sequence Data) | 5.6 |
| xtf_Files.tar | XTIFF Images (NY Hudson River Side-scan SONAR images) | 1.5 |
| enwiki.xml | Text (Wikipedia dump of articles, templates, media/file descriptions, and primary meta pages) | 37.9 |
| ERR007772_1.fastq.gz | Compressed GZIP (Compressed ERR007772_1.fastq) | 2.4 |
| enwiki.xml.bz2 | Compressed BZIP2 (Compressed enwiki.xml) | 8.6 |

## IV. Performance Analysis

To determine compression performance, first a dataset of sample files to be compressed was constructed. The particular set was chosen to demonstrate performance over a variety of file types that could represent potential data streams in a real-time system. A brief description of the files used can be seen in Table I.

Text files such as the Wikipedia backups and the Google Books 1-gram corpus were chosen to demonstrate performance on highly redundant data. Since LZO is a dictionary-based method, it was expected that the compression time and ratio should yield fairly good results for these files. The files already containing various degrees of compressed data (GZIP and BZIP2) were chosen to show compression performance when file expansion has a high potential of occurrence. An uncompressed tarball compilation of extended TIFF image data was picked to demonstrate image compression characteristics of the algorithm. A rather large uncompressed tarball compilation of HDF biological E-coli data was chosen to show performance against binary data. Finally, genome data in FASTQ format was chosen, as it is considered the standard for storing the output of high throughput sequencing instruments. Thus, the FASTQ data should provide a somewhat realistic real-time data stream.

As stated in Section I, performance analysis was evaluated in regards to a theoretical real-time compression system consisting of two interfaces: a constant input stream and an output stream that can be written to with zero delay. A software command-line executable was created to simulate such a system and facilitate batch compression of the files identified in Table I. The executable loads input data and compresses it in memory to reduce the impact of slow file I/O subsystems. Time spent reading data from the hard disk is not counted against compression time. This serves two purposes; first, the algorithms can be more fairly evaluated independent of hardware overhead limitations, and second, the input data stream is simulated as a continuous stream, as desired by the real-time system simulation. Parameters were passed to the executable dictating which algorithm to utilize and how many iterations of compression to perform on a per file basis. After testing a particular algorithm's performance on a file from the dataset, the program outputs the average throughput, the average compression time, the average compression time per input data block, the compression ratio, and the final compressed file size.

### *A. Test Setup*

A Dell Optiplex 790 computer running Microsoft Windows 7 Professional 64-bit was utilized to determine the impact of the algorithm modifications. The system was configured with

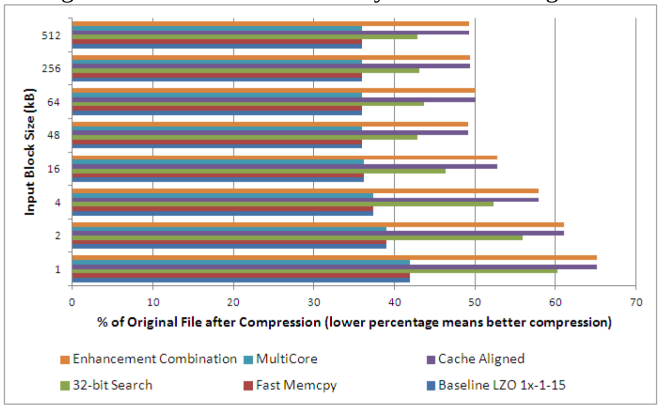

Figure 5. Compression Ratio at Various Input Block Sizes

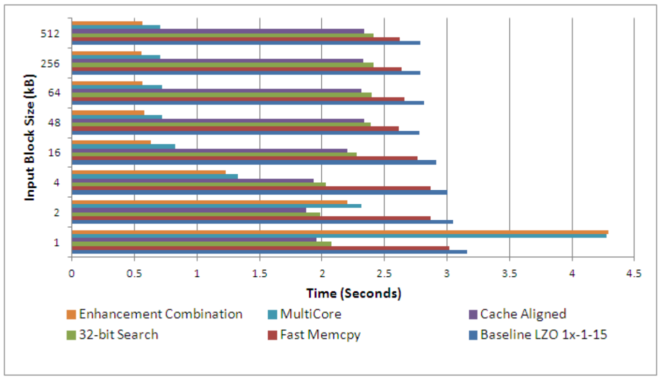

Figure 6. Compression Time at Various Input Block Sizes

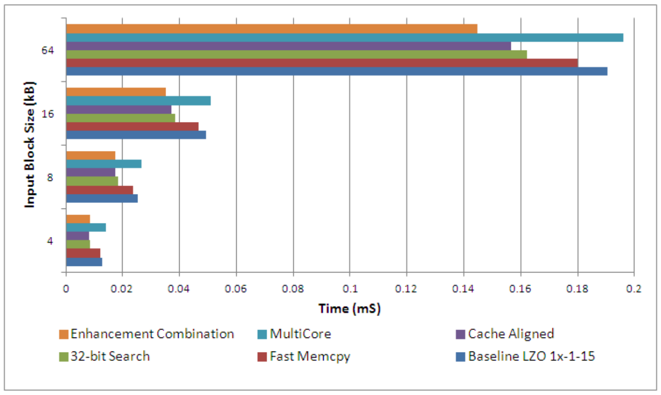

Figure 7. Compression Time per Block at 4k, 8k, 16k, 32k Input Block Sizes

an Intel 3.4GHz i7-2600 Sandy Bridge Quad Core Processor (4 physical cores, 8 logical cores), 8 GBytes of Dual Channel 1333MHz DDR3 RAM, and a 500GB 7200RPM hard disk drive.

Prior to the start of testing, an optimal parameter was determined for the input block size. The first Google Book 1-gram dataset was run with several input block sizes, as seen in Fig. 5 through Fig 7. Compression ratio and compression timing performance noticeably increased from about 1kB to 16kB. From 48kB onward, performance seemed to level off and remain relatively constant. Compression time per block was found to be on the order of milliseconds. It should be noted that the time to compress one block of data is virtually the same for the baseline and multi-threaded implementations, as the multi-threaded compression routine is identical. The multi-threaded implementation gains its speed due to the fact that it is processing multiple compression blocks simultaneously. For testing purposes, an input block size of 256kB was chosen, as it is the default used by Oberhumer's lzop executable and demonstrated no obvious disadvantage in the preliminary tests.

For those algorithms utilizing multiple CPU cores, in addition to the input block size parameter, the optimum number of compression threads was determined empirically prior to testing. Some results of this testing can be seen in Fig. 8. Performance with one thread in the multi-core version of LZO was found to be slightly worse than the baseline. This is most likely due to the overhead involved in coordinating the multi-core algorithm. Performance for both multi-threaded implementations increased up to a maximum value of four compression threads. It was originally expected that performance would increase up to eight maximum threads as the system under test has eight logical cores. Further testing showed that due to the specific software implementation, the CPU cores never reached maximum utilization of all eight cores simultaneously. In order to accommodate large file sizes, only 64 MBytes at a time were read to physical memory from disk and compressed in a loop. The stall time imparted from performing the disk reads was accounted for and subtracted from the total compression time. However, the operating system, due to the frequency of the stalls, appeared to determine that delegating work to the logical cores was not necessary, as the physical cores remained slightly

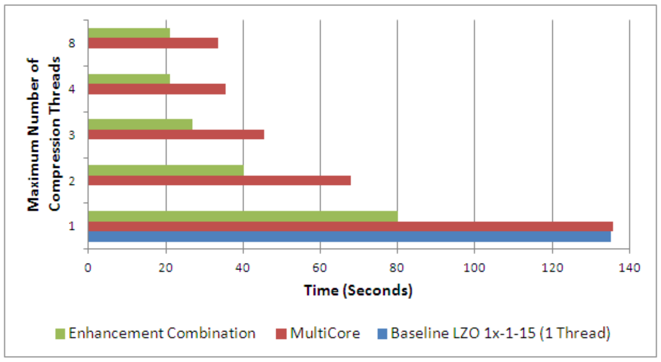

Figure 8. Average Compression Time for Various Maximum Thread Settings (Control and Reconstruction Threads Not Counted in Totals)

underutilized. A modified version of the compression benchmark program was created to iterate on the first 512 MB of file data in memory. This software was able to exercise all eight cores, showing the expected performance boost from one to eight threads.

For the purposes of this testing it was decided to continue utilizing the original version of software with the maximum thread parameter set to 4 or greater. The memory-only software variation would result in inaccurate compression ratio measurements, as larger file streams would not be able to fit within available memory space.

## *B. Benchmark Test Results*

Testing was conducted in the following manner. First the performance of the existing LZO 1x-1-15 algorithm was determined. Then each of the four new performance enhancements to 1x-1-15 was tested separately. A version of the compression algorithm that combined all of the enhancements was tested and the results recorded. Finally, the CRC-32 library modification was tested against the baseline library function using the unmodified LZO 1x-1-15 algorithm with a 256kB input block size. The CRC-32 function calls were constructed similar to those used in lzop, calling the function twice per each processed input data block.

Results of testing the LZO 1x-1-15 variants on the dataset can be found in Fig. 9 through Fig. 11. In terms of compression ratio, in all cases, the baseline, optimal memory copy, and multi-core implementations yielded the best results. This was expected, as the three of these algorithms did not deviate from the original algorithm in a manner that would affect file traversal behavior. The remaining three algorithms employed modifications to the search portion of the algorithm to enhance compression speed. Compared to the baseline algorithm, the 32-bit search, cache aligned read, and combination implementations experienced losses in compression ratio. The 32-bit search algorithm resulted in files up to 1.3 times larger than the baseline, whereas the cache aligned read and combination variants created files up to 2.3 times in size.

In terms of compression time, the baseline algorithm proved the slowest for most files. In all cases, the optimized vector memory copy algorithm outperformed the baseline speed-wise by roughly 5 percent. The multi-core and combination algorithms consistently completed file compression within 25 and 20 percent respectively of the time taken by the baseline algorithm. On average, for the files tested, the speedup gained was 3.9x for multi-core and 5.4x for the combination algorithm. The 32-bit search algorithm resulted in a 10 percent speed increase for text files and a 25 to 50 percent increase for binary/image files. The cache-line-aligned read algorithm, with one exception, proved to be slightly faster than the 32-bit search modification.

The one exception to this trend occurred with the file "image_en.nq". This is a text file composed of Wikipedia thumbnail hyperlinks. A brief examination of the file shows that the contents appears to be highly redundant, as it consists of many very similar hypertext Uniform Resource Locator (URL) ASCII text strings, all beginning with "<http://". The

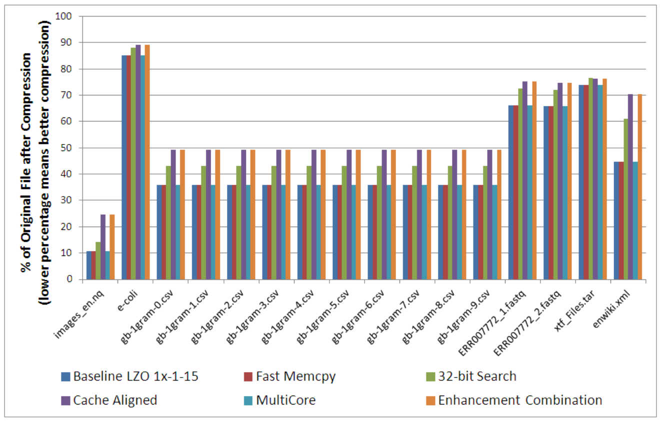

Figure 9. Compression Ratio

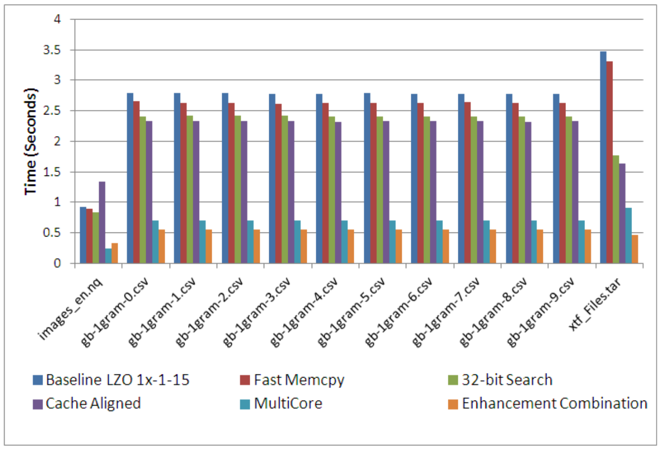

Figure 10. Compression Time (small streams)

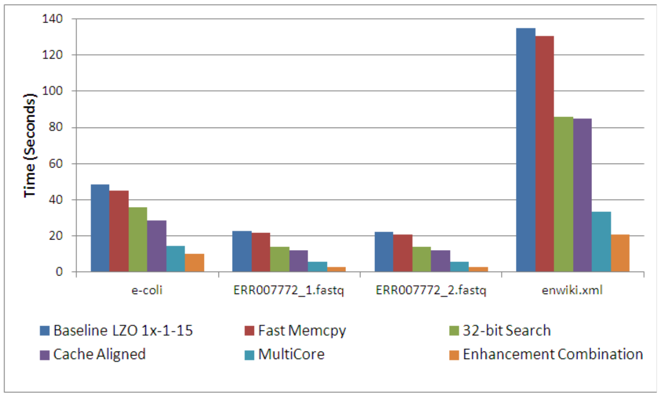

Figure 11. Compression Time (large streams)

baseline algorithm confirms that this is the most redundant of the files tested, as it achieved the greatest compression on this file in Fig. 9. It was expected that the compression ratio would be the worst for the cache-line-aligned version, as this version not only searches every 32-bits for a match and ignores matches less than 32-bits in size, but also moves the search pointer forward if it is not currently on a 32-bit boundary. The extent of the speed degradation, however, was not anticipated.

Table II. Cache-Line-Aligned vs. 32-Bit Search Performance
(Higher Ratios Indicate Worse Performance for Cache-Line-Aligned Alg.)

| **Code Portion** | **% Total Compression Time** | **%Total Compression Time** | **Ratio of Loop Iterations** | **Ratio of Compression Time** |
|---|---|---|---|---|
| | **32-Bit Search Alg.** | **Cache – Line Alg.** | **(Cache-Line Alg. : 32-Bit Search Alg.)** | |
| Search | 27.0% | 31.9% | 1.35 | 1.65 |
| Literal Copy | 14.8% | 16.3% | 2.15 | 1.55 |
| Match Length | 41.7% | 32.3% | 0.95 | 1.08 |
| Token | 16.5% | 19.5% | 1.70 | 1.66 |
| Overall | 100% | 100% | 1.28 | 1.40 |

To determine why the cache-line-aligned modification may experience issues with highly redundant data, a special version of the compression software was created to gather more in-depth timing information. This version of software recorded execution time spent in the four main sections of the compression code, described in Section II. Compression of the file “image_en.nq” was re-run using the 32-bit search and cache-line-aligned algorithms. The results can be found in Table II.

It was found that the cache-line-aligned algorithm overall executed roughly 1.28 times the number of loop iterations, resulting in a total compression time of 1.40 times longer than that of the 32-bit search variation. Supplemental testing on an Intel Core2 Duo system (not shown), which according to [7] should benefit more from the aligned reads than the i7 processor used for benchmarking, showed that the algorithm still performed 1.30 times slower than the 32-bit search variation. Examining Table II, fewer match length determination loops were executed, implying that fewer dictionary matches were found. This makes sense, as during dictionary searches the 32-bit alignment of the input pointer may skip up to three consecutive bytes, leading to a more sparsely populated dictionary. Those matches found would also have been shorter in length due to the imposed 32-bit boundary restriction. This led to more execution time in the rather expensive search and uncompressed literal copy code segments. The ability of the search function to exponentially skip through a 48kB sub-block is also hindered by the high level of data redundancy: the algorithm periodically finds short matches, resulting in an inability to skip forward to the end of a sub-block, as it would with data of lower redundancy. These combined factors appear to have attributed to the poor speed performance for highly redundant data.

Compression timing performance for highly non-redundant data can be seen in Fig. 12. Once again, the multi-threaded algorithms experienced considerable speed boosts compared with the others. Results indicated that file expansion only achieved an excess growth of 0.4 percent of the original file size for all tested algorithm variants.

CRC-32 testing was conducted on one of the Google Book 1-gram data files. Testing showed that on average an extra 1.47 msec per block was added to compression time when using the tabular method, and an extra 0.23 msec per block was added when using the hardware method. As expected, the

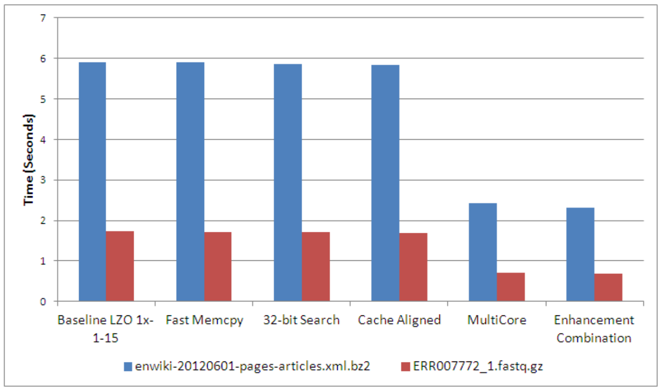

Figure 12. Average Compression Time for Non-Redundant Data

hardware implementation demonstrates a considerable performance increase of about six times that of the baseline CRC-32 function.

## V. Conclusions

This paper examined several methods in which the compression speed of LZO 1x-1-15 can be improved. Slight adjustments to the code were made to increase parallelization, improve the copying of literal data, speed up input stream traversal, optimize cache accesses, and perform hardware CRC-32 calculations. The multi-core and optimized memory copy algorithms both demonstrated their ability to speed compression without affecting compression ratio. Combining these two mutually exclusive operations should likely achieve a more robust implementation of LZO 1x-1-15. Hardware CRC-32 calculations were seen to add verification to the output stream with minimal impact on total compression time. Of the algorithm variants tested, the multi-core algorithm displayed the greatest speed enhancement without degradation to compression ratio - an improvement of 3.9x over that of the baseline. In instances where compression ratio is not of primary importance, the combined optimization algorithm demonstrated the ability to increase compression speed performance by 5.4x.

In general, the results suggest that LZO and other token-based block compression algorithms similar to it can benefit from recent CPU hardware optimizations. Increased processor bus widths allow for larger block memory copies when storing uncompressed literal data. The introduction of multiple cores allow for multiple blocks to be simultaneously and independently compressed. These optimizations should port to embedded multi-core architectures containing data buses greater than 32-bits.

Future work may be explored by investigating potential optimizations from the Intel AVX (Advanced Vector Extensions) and AVX2 vector instruction sets [10]. These instruction sets extend upon SSE, allowing for SIMD instructions to run on 256-bit data types. Further vector optimizations may be possible in hash calculation, literal copying, and match length determination. Preliminary prototypes of the software under test showed that utilizing existing x86 SSE 128-bit vector operations for these purposes often resulted in performance loss due to operand alignment related setup costs. Operands typically become unaligned due to the byte-oriented searching nature of the LZO algorithm. Performing enough vector calculations in parallel may outweigh the setup disadvantage. It should be noted that these costs are entirely architecture dependent. Migrating to another processor family may prove to have better or worse performance when executing similar instructions.

Another area of future work may be to implement an adaptive version of the algorithm similar to that described in [11]. Such a system would attempt to maintain a minimum compression ratio, switching back and forth between the different algorithms for speed gains as need allows.

## Acknowledgment

This research was sponsored in part by the Department of the Navy (Naval Undersea Warfare Center, Newport, Rhode Island), and NSF CCF-1017177. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. The authors are very grateful to the anonymous reviewers for providing detailed comments that have improved the quality of the paper greatly.

## References

[1] Oberhumer, M.F.X.J., 2011. LZO real-time data compression library. [Online]. Available: http://www.oberhumer.com/opensource/lzo/. [20 June 2012]

[2] L. Yang, R.P. Dick, Haris Lekatsas, and Srimat Chakradhar, "Online memory compression for embedded systems", ACM Transactions on Embedded Computer Systems. vol 9, issue 3, no. 27, February 2010.

[3] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. IEEE Transactions on Information Theory, vol. 23, no. 3, pp. 337-343, 1977.

[4] Intel Corporation. (2012, April). "Intel® 64 and IA-32 Architectures Optimization Reference Manual" [online]. Available: http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf [20 June 2012]

[5] Agner Fog. (8 June 2011). "Instruction tables. Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD, and VIA Cpus" [online]. Available: http://www.agner.org/optimize/instruction_tables.pdf [20 June 2012].

[6] C.M. Sadler and M. Maronosi, "Data Compression Algorithms for energy-constrained devices in delay tolerant networks", Proc. Of IEEE SenSys '06, pp. 265-278, Nov 2006.

[7] Agner Fog. (8 June 2011). "The microarchitecture of Intel, AMD and VIA CPUs" [online]. Available: http://www.agner.org/optimize/microarchitecture.pdf [20 June 2012]

[8] Agner Fog. (21 August 2011). "Instructions for asmlib. A multi-platform library of highly optimized functions for C and C++" [online]. Available: http://www.agner.org/optimize/asmlib-instructions.pdf [20 June 2012]

[9] Mark Adler, 2012. zlib Home Site. [Online]. Available: http://www.zlib.net. [20 June 2012]

[10] Intel Corporation. (2011, April). "Intel® 64 and IA-32 Software Developer's Manuals" [online]. Available: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html [20 June 2012]

[11] Chandra Krintz and Sezgin Sucu, "Adaptive On-the-Fly Compression", Transactions on Parallel and Distributed Systems, January 2006, Vol. 17 No. 1