Implementation and Testing of GBDI Memory Compression Algorithm

Promit Panja
Dept. of Electrical and Computer Engineering
Virginia Tech
Blacksburg, United States
ppanja@vt.edu

TingHung Chiu
Dept. of Electrical and Computer Engineering
Virginia Tech
Blacksburg, United States
kennychiu0818@vt.edu

***Abstract*—This project aims to implement and test a lossless memory compression technique called GBDI (Global Base-Delta-Immediate) using C. GBDI compresses data by only storing the difference (deltas) between the global base value and the actual values in the memory block and is an extension of the BDI memory compression algorithm. GBDI enables high compression ratios and low decompression latencies, which can improve memory bandwidth and performance. The project involves implementing the GBDI compressor and decompressor, evaluating their performance on C++ and Java benchmarks and comparing them to the results the authors have shown.**

***Index Terms*—GBDI, BDI, cache-compression**

I. Introduction

Due to exponential growth in processor performance there has been an exponential growth in the demand for memory bandwidth. One of the solution for meeting this high bandwidth demand is by using high-bandwidth memory (HBM) but they are very expensive. To mitigate the latency and bandwidth limitations of accessing main memory, modern microprocessors contain multi-level on-chip cache hierarchies [1]. Large on-chip caches help with meeting bandwidth demands but similar to HBMs implementing large caches is expensive and also require large die area.

Memory compression is a technique used to increase the effective size of a computer's physical memory by compressing the data stored in memory. Memory compression techniques abound to maximize the compression ratio (CR) while maintaining a low latency [2]. Memory compression can be implemented on both *hardware* and *software*. *Hardware* based use dedicated hardware on the memory controller to compress and decompress. Similarly *software* based use software algorithms to compress and decompress.

Cache compression is a promising technique to increase on-chip cache capacity and to decrease on-chip and off-chip bandwidth usage [1]. Cache compression is implemented using dedicated hardware in the memory controller. However, directly applying well-known compression algorithms (usually implemented in software) leads to high hard- ware complexity and unacceptable decompression/compression latencies, which in turn can negatively affect performance [1]. Compression algorithms like BDI and GBDI aim at being a simple yet efficient compression technique that can effectively compress common in-cache data patterns, and has minimal effect on cache access latency.

Through this project we try to implement GBDI memory compression algorithm made for on-chip caches on *software* using the C programming language and test its performance on a varied set of benchmarks of C++ and Java workloads and compare this performance with the results obtained by the authors of the GBDI paper.

II. Background and Motivation

Applying data compression to an on-chip cache can potentially allow the cache to store more cache lines in compressed form, as a result, a compressed cache has the potential to provide the benefits of a larger cache at the area and the power of a smaller cache [1].

Our motivation behind this project was to implement and test a modern cache compression algorithm. We decided to go with Global Base-Delta-Immediate (GBDI) compression algorithm which is based on Base-Delta-Immediate (BDI) compression algorithm as it is a relatively new algorithm and to best of our knowledge there has not been much work to test its performance apart from the work done by the authors themselves.

Before we move on to the implementation of the GBDI algorithm we need to understand the working of both BDI and GBDI.

A. Base-Delta-Immediate (BΔI)

Base-Delta-Immediate (BΔI) compression is a practical technique for compressing data in on-chip caches. The key idea is that, for many cache lines, the values within the cache line have a low dynamic range – i.e., the differences between values stored within the cache line are small [1]. As a result the entire cache can be stored using a base value and an array of differences whose combined size is much smaller than the original cache line [1].

A compressed cache line can be decompressed by using the B+Δ decompression algorithm. To carry out decompression we need to take the base value $B*$ and an array of deltas and generate the corresponding set of values $S = (\mathbf{v_1}, \mathbf{v_2}, ..., \mathbf{v_n})$.

From the above decompressor design we can see that the decomrpression can be carried out in parallel. When

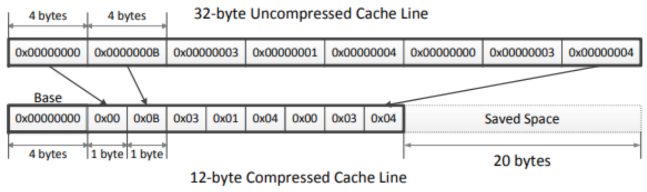

Fig. 1. BDI Compressor Design [1].

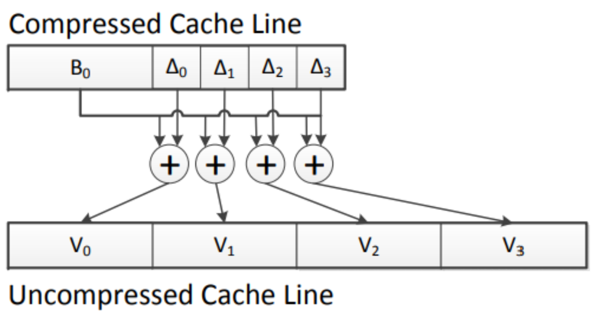

Fig. 2. BDI Decompressor Design [1].

implemented in hardware the values in the cache line can be computed in parallel using a SIMD-style vector adder. Consequently, the entire cache line can be decompressed in the amount of time it takes to do an integer vector addition, using a set of simple adders [1].

B. Global Base-Delta-Immediate

Global Base-Delta-Immediate (GBDI) is a generalization of Bse-Delta- Immediate. A. Angerd et al. made a surprising insight that the two hypothesis on which BDI is based on: intra-block values are numerically similar and deltas can be encoded compactly, are are not true, in general, especially not for floating-point values [2].

GBDI uses global bases, shared by all input data, instead of a single intra-block base value as in BDI. It compresses and decompresses individual blocks on-the-fly at run-time. Global bases are established through a data analysis phase, by software algorithms, in the background [2]. GBDI aims at selecting a number of global bases, across all targeted data, that minimizes deltas when each value is encoded with a pointer to the closest global base and a delta with respect to that base [2].

GBDI follows BDI for the most part apart from the data analysis step which is used to establish the global bases. The authors mention any kind of clustering algorithm can be used to find the global bases across memory blocks like kmeans which minimizes the distance between the base and cluster values. But, from their testing they have shown that kmeans does not maximize compression and they propose *histogram binning* that focuses on the clusters with most frequent values and selects bases to maximize compression.

The *Data Analysis* phase includes the following two steps:

1) Establishing Global Base Values: The authors propose histogram binning (HB), which empirically yields a good CR at a lower computational cost (time complexity and wall clock time). In HB, we construct a histogram of all values, with a number of equal-ranged bins, N. Among the bins, we identify the $B(B < N)$ bins that contain most values. Within each of the B bins, we locate the single value with the highest occurrence, and choose that as a base resulting in B base values.

2) Establishing Maximum Deltas: We set the maximum delta for each global base by considering all values that are closest to that base. The maximum delta is the number of bits needed to establish the distance to the furthest of these values. To improve compression, the authors establish an upper bound on the maximum delta. If the maximum delta is greater than the upper bound, it is lowered to be equal to the upper bound.

$$16 - \log_2(B) \tag{1}$$

Where B is the number of global bases. This allows each individual value to be compressed to

$$deltasize + ptrsize = (16 - \log_2(B)) + \log_2(B) = 16bits \tag{2}$$

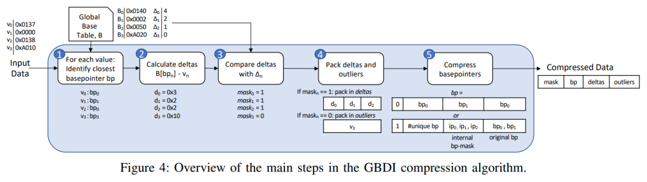
v_0 0x0137
v_1 0x0000
v_2 0x0138
v_3 0xA010
Input Data
Global Base Table, B
B_0 0x0140 Δ_0 4
B_1 0x0002 Δ_1 2
B_2 0x0050 Δ_2 1
B_3 0xA020 Δ_3 0
1
For each value: Identify closest basepointer bp
$v_0 : bp_0$
$v_1 : bp_1$
$v_2 : bp_0$
$v_3 : bp_3$
2
Calculate deltas $B[bp_n] - v_n$
d_0 = 0x3
d_1 = 0x2
d_2 = 0x2
d_3 = 0x10
3
Compare deltas with Δ_n
$mask_0 = 1$
$mask_1 = 1$
$mask_2 = 1$
$mask_3 = 0$
4
Pack deltas and outliers
If $mask_n$ == 1: pack in *deltas*
d_0 d_1 d_2
If $mask_n$ == 0: pack in *outliers*
v_3
5
Compress basepointers
bp =
0 bp_0 bp_1 bp_0
or
1 #unique bp ip_0, ip_1, ip_2 bp_0, bp_1
internal bp-mask
original bp
Compressed Data
mask bp deltas outliers
Figure 4: Overview of the main steps in the GBDI compression algorithm.

Fig. 3. GBDI Compressor Design [2].

One of the optimizations the authors have used is encoding the base pointers using Huffman coding, as the size of the base pointer increases with number of bases. During block compression, a string of Huffman codes is stored rather than the base pointers. The compression is only applied when beneficial, and a single additional bit in the block metadata signifies whether the pointers are encoded with Huffman or not. While this encoding adds latency, it can also significantly reduce the traffic and therefore improve performance [2].

III. Implementation

We implemented the GBDI compression algorithm in software using the C programming language. The implementation of the algorithm is comprised of three functions— **establish_global_base_set_hb()**, **gbdi_compress()**, **gbdi_decompress()**, and **main()**.

- **establish_global_base_set_hb()**: This function calculates and initializes the global base set used in GBDI compression. It takes input data, data count, a global base set pointer, and the maximum number of bases as input. It computes the most common deltas and adds them to the global base set.

- **gbdi_compress()**: This function takes uncompressed data, data count, a global base set, an output buffer, and an output size pointer as input. It compresses the input

Algorithm 1: establish_global_base_set_hb

```
Input: An array data of size data_count, a pointer to
       a GlobalBaseSet global_base_set, and a
       maximum number of bases max_bases.
Output: A populated GlobalBaseSet global_base_set
        containing the most common delta values.
global_base_set.count ← 0;
global_base_set.bases ← an array of size max_bases;
if data_count < 2 then
    return;
else
    delta_count ← data_count − 1;
    deltas ← an array of size delta_count;
    for i ← 0 to delta_count − 1 do
        deltas[i] ← data[i + 1] − data[i];
    end
    Sort deltas using quicksort with comparison
      function delta_compare;
    current_delta_count ← 1;
    max_delta_count ← 1;
    current_delta ← deltas[0];
    max_delta ← deltas[0];
    for i ← 1 to delta_count − 1 do
        if deltas[i] = current_delta then
            current_delta_count ←
              current_delta_count + 1;
        end
        if current_delta_count > max_delta_count
          then
            max_delta_count ← current_delta_count;
            max_delta ← current_delta;
            global_base_set.bases[global_base_set.count] ←
              max_delta;
            global_base_set.count ←
              global_base_set.count + 1;
        end
        current_delta_count ← 1;
        current_delta ← deltas[i];
        if i = delta_count − 1 and
          current_delta_count > max_delta_count then
            max_delta_count ← current_delta_count;
            max_delta ← current_delta;
            global_base_set.bases[global_base_set.count] ←
              max_delta;
            global_base_set.count ←
              global_base_set.count + 1;
        end
    end
    Free deltas;
end
```

data using the GBDI algorithm, leveraging the global base set. It outputs the compressed data into the output buffer and updates the output size.

Algorithm 2: gbdi_compress

```
Input: Data array data, size of data array data_count,
       global base set global_base_set, output buffer
       output_buffer, pointer to output buffer size
       output_size
Output: Compressed data array output_buffer,
        compressed data size output_size
// Initialize buffer and loop through
    input data
buffer_start ← output_buffer
for i ← 0 to data_count-1 do
    value ← data[i]
    best_delta ← 0
    best_delta_index ← -1
    best_compressed_value ← value
    for j ← 0 to global_base_set-¿count-1 do
        base ← global_base_set-¿bases[j]
        if value ≥ base then
            compressed_value ← value - base
            if compressed_value <
              best_compressed_value then
                best_compressed_value ←
                  compressed_value
                best_delta ← base
                best_delta_index ← j
            end
        end
    end
    // Write best delta index and best
        compressed value to output
        buffer
    if best_delta_index ≥ 0 then
        write_variable_length_integer(best_delta_index
          + 1, &output_buffer)
    end
    else
        write_variable_length_integer(0,
          &output_buffer)
    end
    write_variable_length_integer(best_compressed_value,
      &output_buffer)
end
*output_size ← output_buffer - buffer_start
```

- **gbdi_decompress**: The decompression algorithm reads a variable length integer from the input buffer to determine the base index and compressed value. If the base index is greater than zero, the decompressed value is calculated by adding the compressed value to the corresponding global base value. Otherwise, the decompressed value is just the compressed value. The decompressed values are stored in an output buffer, and the function updates a pointer to the size of the output buffer.

Algorithm 3: gbdi_decompress

```
Input: Compressed data compressed_data, size of
       compressed data compressed_data_size, global
       base set global_base_set, decompressed data
       array decompressed_data, pointer to
       decompressed data count
       decompressed_data_count
Output: Decompressed data array decompressed_data,
        decompressed data count
        decompressed_data_count
   // Initialize input buffer and output
      index
1  $input\_buffer \leftarrow compressed\_data$
2  $output\_index \leftarrow 0$
   // Decompress the data
3  while $input\_buffer <$
     $compressed\_data + compressed\_data\_size$ do
4      $base\_index \leftarrow$
         $read\_variable\_length\_integer(\&input\_buffer)$
5      $compressed\_value \leftarrow$
         $read\_variable\_length\_integer(\&input\_buffer)$
6      if $base\_index > 0$ then
7          $value \leftarrow global\_base\_set->$
             $bases[base\_index - 1] + compressed\_value$
8      end
9      else
10         $value \leftarrow compressed\_value$
11     end
12     $decompressed\_data[output\_index++] \leftarrow value$
13 end
14 $*decompressed\_data\_count \leftarrow output\_index$
```

- **main()**: The main function reads an ELF file, extracts memory segments, and processes cache lines. It calls **extract_cache_lines()** function to create a cache lines buffer from the segments. It then compresses the cache lines using GBDI compression, this is done by first calling **establish_global_base_set_hb()** which calculates the global bases and creates a global base set and then calling **gbdi_compress** which compresses the cache line as per the gbdi compression algorithm and outputs the compressed data to the output buffer. Finally, it prints the compression statistics.

IV. Results

We tested our implementation of the algorithm on the following memory dump files—

- parsec_freqmine5dump
- parsec_fluidanimate5dump
- 620.omnetpp_s_5.dump
- 600.omnetpp_s_5.dump

The following results of compression ratio were obtained by compiling and running the algorithm on the CRC1 machine provided by the Computer Science Dept. at Virginia Tech.

TABLE I
GBDI Compression Ratio (All sizes are in Bytes)

Memory Dump	Original Size	Compressed Size	CR
631.deepsjeng_s_5.dump	1803132928	503943605	3.58
parsec_fluidanimate5dump	137556992	74133444	1.86
620.omnetpp_s_5.dump	57908224	41464076	1.4
600.omnetpp_s_5.dump	51412992	43178335	1.19
TriangleCount_3.dump	892108800	250461143	3.56

As we can see from TABLE 1 the results we have obtained do not match the expected results as mentioned by the authors in their paper. There could be many reasons why the results we have observed are different from that of the authors. One of the reasons could be that we are not using any of the optimizations that the authors have mentioned which improve the compression ratio significantly.

From our results we can also observe that some results like that of 631.deepsjeng_s_5.dump and TriangleCount_3.dump produce compression ratio of 3.58 and 3.56 respectively, which are unusually high and are not correct. Unfortunately we were not able to find out the reason behind this behaviour but we know that the compression algorithm produces consistent results as the decompressed data matches the original input.

V. Conclusion and Future Work

In this project we focus on the Global Base-Delta-Immediate (GBDI) memory compression algorithm which is based on and is an extension of the Base-Delta-Immediate (BΔI) compression. We successfully implemented the GBDI memory compression algorithm using the C programming language. We also tested our implementation of the algorithm and calculated the compression ratio on the following memory dump files—parsec_freqmine5dump, parsec_fluidanimate5dump, 620.omnetpp_s_5.dump, and 600.omnetpp_s_5.dump.

Through our implementation and testing we observed that the results we have obtained do not match the results provided by the authors. From our testing the compression ratio (CR) varies between 1 and 2 and for some memory dumps the compression ratio (CR) is above 3.5 which is unusually high and is incorrect.

For future work we can improve the algorithm and improve its overall performance by implementing all the optimizations that the authors have shown like encoding the base pointers using Huffman Coding so the base pointers take up less space in the final compressed data to improve the compression ratio that would match more closely to the results shown by the authors. We could also add the ability to dynamically selecting between BDI and GBDI based on the input data for best compression performance as mentioned by the authors which would be an addition to baseline GBDI. Another thing that needs to be addressed are the outlier compression ratios for some memory dumps, the reason behind this behaviour needs to be investigated and the algorithm needs to be changed to mitigate this issue. Testing the algorithm on various other benchmarks which simulate real-world data patterns can also be done as future work.

References

[1] G. Pekhimenko, V. Seshadri, O. Mutlu, M. A. Kozuch, P. B. Gibbons and T. C. Mowry, ”Base-delta-immediate compression: Practical data compression for on-chip caches,” 2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT), Minneapolis, MN, USA, 2012, pp. 377-388.

[2] A. Angerd, A. Arelakis, V. Spiliopoulos, E. Sintorn and P. Stenström, ”GBDI: Going Beyond Base-Delta-Immediate Compression with Global Bases,” 2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA), Seoul, Korea, Republic of, 2022, pp. 1115-1127, doi: 10.1109/HPCA53966.2022.00085.