#

#

#

# **Study of Cache Compressibility**

**Aditya Venkataraman (adityav@cs.wisc.edu)**

**Deepinder Singh (dsingh24@wisc.edu)**

**CS752 Class Project, Fall 2014**

# Abstract

On-chip caches are vital for improving system performance by avoiding time-consuming fetches from off-chip memory. The efficacy of caches is due to the spatial and temporal locality inherent in most programs. Caches also play an important role in reducing effective system energy by avoiding many inefficient DRAM accesses. Hence, we desire larger caches to store more useful data closer to the processor. However, increasing cache sizes will result in an equivalent increase in cache area, power-consumption and latency of operation. We desire an increase in *effective* cache capacity without the concomitant undesirable effects. Cache compression is identified as a promising technique for achieving this goal. Compressed caches can store more useful data within the same cache capacity, but will take longer to service a request due to unavoidable decompression latencies. A number of works have explored various compressed cache designs espousing various compression algorithms. However, we felt a comprehensive study of the inherent *compressibility* of workloads was lacking.

In this work, we present a study of compressibility of cache contents across a range of workloads and characterize how this changes across time, compression algorithms and various cache design parameters. We develop an easy-to-use tool to analyze cache dumps and provide insights into the fundamental entropy, most recurring patterns that are ripe for compression at various granularities. We present a range of results which suggest that sufficient compressibility is found in most workloads, but the absolute compressibility varies widely. Some data semantics such as integers are more suited for compression while floating point values are not. We also identify the efficacy of a range of academic compression algorithms for these workloads and present that a combination of significance & dictionary algorithms are most suited for data compression. We also note the inability of these algorithms to compress instruction caches.

**Table of Contents**

 [1](#_Toc406450435)

[**Study of Cache Compressibility** 1](#_Toc406450436)

[Abstract 2](#_Toc406450437)

[1. Problem Statement 4](#_Toc406450438)

[2. Theoretical Compressibility 4](#_Toc406450439)

[3. ‘Software-heavy’ Algorithms 6](#_Toc406450440)

[4. Axes of Study 8](#_Toc406450441)

[5. Evaluation Methodology 8](#_Toc406450442)

[6. Cache Analyzer 10](#_Toc406450443)

[7. Cache Contents Decomposition 11](#_Toc406450444)

[8. Cache Compression Algorithms 11](#_Toc406450445)

[9. Benchmarks/Workloads tested 13](#_Toc406450446)

[10. Compression Results 14](#_Toc406450447)

[11. Code Compression 17](#_Toc406450448)

[12. Levels of Memory Heirarchy 18](#_Toc406450449)

[13. Conclusion 21](#_Toc406450450)

[14. Appendix A – Overview on Cache Compression Algorithms 22](#_Toc406450451)

[14.1 Frequent Pattern Compression 22](#_Toc406450452)

[14.2 Base Delta Immediate 23](#_Toc406450453)

[14.3 CPACK – Cache Packer 23](#_Toc406450454)

[14.4 Golomb-Rice Encoding 24](#_Toc406450455)

[15. Appendix B – Cache Analyzer Graphs 25](#_Toc406450456)

[16. References 26](#_Toc406450457)

**List of Figures**

[Figure 1 Theoretical compressibility of a cache snapshot 4](#_Toc406406993)

[Figure 2 Theoretical compressibility of graph-analytics in Zero Information Entropy model 5](#_Toc406406994)

[Figure 3 Theoretical compressibility of graph-analytics in Markov Zeroth Order model 5](#_Toc406406995)

[Figure 4 Compression efficiency of DEFLATE across dictionary sizes 6](#_Toc406406996)

[Figure 5 Variation in compression ratio vs input size for LZW 6](#_Toc406406997)

[Figure 6 L2 cache line decomposition for graph-analytics 10](#_Toc406406998)

# Problem Statement

What is the entropy of on-chip caches across workloads? What is the reason behind this (lack of) entropy and how can this be exploited to achieve compression? Can this entropy be exploited using conventional compression techniques? How does this compressibility & entropy vary across a range of factors such as cache design parameters, data semantics etc.?

# Theoretical Compressibility

In information theory, the entropy is the average amount of information in a source. If the information of the source is constant or deterministic, the entropy if zero. On the other hand, highly varying sources will have greater entropy. We can consider the cache contents to represent a source and study the variance of symbols across them. This will help us compute the entropy of cache contents and inversely, the compressibility of the information. Entropy is usually studied relative to an information model. In this work, we used two entropy models to calculate the theoretical upper limit of compression.

1. *Zero Information Entropy* – Based on the insight that the number of unique symbols in a source is usually lesser than the total number of symbols in the alphabet. The entropy is computed as -

*Entropy (H) = ∑log(no. of unique symbols).*

For example, in a text file which contains just 20 out of the 26 English alphabets, entropy for each character would be calculated as log2(20) rather than log2(26)

1. *Markov Zeroth Order Entropy* – Zero information entropy treats each occurring symbols are equally distributed across the source. This is usually not true. Markov Zeroth Order Entropy model recognizes this and calculates the entropy as a function of the probability of a symbol’s occurrence. The Entropy is computed as -

*Entropy = - ∑p(x)log2(p(x)).*

Where, p(x) represents the probability of a symbol’s occurrence.

The theoretical compressibility can be defined as a ratio of the entropy over the original symbol size, averaged over all files.

*Compression ratio = Avg( H/original symbol size)*

To calculate these theoretical models, we took periodic snapshots of cache contents throughout the test and passed them to Python scripts which implemented these models. One snapshot of the entropy for graph-analytics benchmark is presented below.



Figure 1 Theoretical compressibility of a cache snapshot

But this compressibility isn’t constant and varies over time as Figure 2 and Figure 3 suggest. But we note that across the duration of the test the compressibility is consistently > 2x. There are numerous instances where the compressibility is significantly higher.



Figure 2 Theoretical compressibility of graph-analytics in Zero Information Entropy model



Figure 3 Theoretical compressibility of graph-analytics in Markov Zeroth Order model

This trend was visible across the workloads we tested, suggesting that theoretical compressibility in the form of low-entropy is present.

#  ‘Software-heavy’ Algorithms

Considering the entropy of cache-contents, we next analyze the ability of conventional, industry-standard compression techniques in exploiting cache compression. We picked two industry-standard tools – DEFLATE, part of the widely used gzip utility and Lempel-Ziv-Welch compression algorithms. Standard open-source implementations were downloaded from the internet. We noted two major drawbacks in using such ‘software-heavy’ algorithms for cache compression. Firstly, these algorithms typically construct very large dictionaries (order of 32K to 64K entries). Such large dictionaries are infeasible for hardware implementation. So we reduced the dictionary sizes and studied the variation in compressibility. In the following pages, compression ratio is defined as the ratio of uncompressed size to compressed size of cache contents.



Figure 4 Compression efficiency of DEFLATE across dictionary sizes

As we see, at dictionary sizes of 16 entries, which are feasible for hardware implementation, these algorithms perform very badly, often with a compression ratio of < 1.

Secondly, we note that these algorithms perform much better with large input sizes, such as entire cache-contents. For cache compression, we expect the input sizes to be much smaller – in the order of a single block. So we studied the variation in compression ratio with decreasing input size.



Figure 5 Variation in compression ratio vs input size for LZW

From the above two analysis, we conclude that off-the-shelf compression techniques are not directly suited for cache compression.

# Axes of Study

Considering the scope for theoretical compressibility of caches and the apparent inability of conventional techniques to exploit them, we looked into alternate methods of exploiting cache compression. We present our analysis and results along the following axes -

1. Cache content composition and trends.
2. Performance of contemporary cache compression techniques.
3. Compressibility across workloads.
4. Effect of data semantics on compressibility – especially code vs data and integer vs floating point numbers.
5. Variation in compressibility across block sizes.
6. Variations in compression across levels of memory.

# Evaluation Methodology

All our results have been obtained from two environments:

a) GEM5 Full system simulations of our target workloads. The simulation environment is presented below:

Table 1 GEM5 simulation environment



We implemented our compression algorithms in the Ruby Cache Memory model. We added functionality into the model to periodically pass the entire cache snapshots into text files as well as the compression functions. The periodicity could be defined in terms of number of simulation ticks or number of allocates (writes) into the cache. For the former, we used a typical periodicity of 500,000 ticks and for the latter, we used a periodicity of 1024/2048 allocates. These numbers were chosen for reasonable simulation overhead without skewing trends.

b) The cache snapshot text files from the simulations were then passed into our Cache Analyzer tool which is capable of post-processing the images to reveal a number of insights and trends.

# Cache Analyzer

For our study of cache entropy and compressibility, we were unable to find any convenient, existing tools. Most cache tools focus on cache performance analysis such as hit-rate with little concern to the actual contents of the cache. So we developed an easy-to-use tool called ‘Cache Analyzer’ to analyze cache contents and present insights into cache contents. For its simplicity and powerful statistics & plotting support, we used Python to develop the tool. Given the standard cache dump file from GEM5, this tool is capable of following:

1. Identify frequently occurring patterns at intra-block & inter-block granularity. We expect the following frequent patterns.
	1. Zero lines
	2. Lines with same value repeated (same bytes, same words etc.)
	3. Narrow values (A 1B value stored in a 4B word)
	4. Aligned values (Pointers are typically aligned)
	5. Lines with low-gradient values
2. Frequency distribution of symbols at word, half-word, byte and interleaved half-word granularities
3. Frequent Pattern Compression potential – judge the potential of compression using the standard FPC algorithm
4. Base Delta Immediate Compression potential – judge the potential of compression using the standard BDI algorithm, using a single base and multiple bases.
5. Entropy analysis of symbols (at words and bytes granularity) using Zero Information Model and Markov Zeroth Order Model.
6. Plot insightful histograms & plots of above trends
7. Easily extendable for other functions due to modular design.

We were able to pass on this tool to another team working on cache compression and they were able to find it useful. We plan to open-source this tool shortly. We present some graphs generated using this tool in the Appendix B.

# Cache Contents Decomposition

Using the above tool we were able to decompose the contents of each cache in the system across workloads. We identify that a major chunk of the cache contents for data caches fit within the limited set of patterns we defined in the previous section. This analysis suggests that any compression algorithm that tries to compress *only* these common patterns should be capable of significant compression. However, we also note that the cache composition varies throughout the test and decomposed snapshots reveal a varying trend. However, we note a consistent trend of >30% of the cache contents fitting within these patterns across workloads and test duration. These results are consistent with a similar analysis done in [2]



Figure 6 L2 cache line decomposition for graph-analytics

# Cache Compression Algorithms

Algorithms developed for the purpose of cache compression broadly fall into two categories -

1. *Dictionary-based schemes* - Most proposals in hardware cache or memory compression are hardware implementations of dictionary-based software compression algorithms. Such hardware dictionary-based schemes depend mainly on (statically or dynamically) building and maintaining a per-block dictionary. The dictionary is used to encode words (or bytes) that match in the dictionary, while keeping words (bytes) that do not match in their original form with an appropriate prefix. Dictionary-based compression algorithms are effective in compressing large data blocks and files.
2. *Significance-based schemes* - Another class of compression algorithms, significance-based compression, depend on the fact that most data types can be stored in a fewer number of bits compared to their original size. For example, sign-bit extension is commonly used to store small integers (8-bit) into 32-bit or 64-bit words. In contrast to dictionary-based compression schemes, significance-based compression does not incur a per-line dictionary overhead. Hardware implementations of significance-based compression schemes can be simpler and faster when compared to dictionary-based schemes. Both of these properties make significance-based compression more suitable for the typically-short cache lines.

We chose the following four algorithms for our study as they represent four reasonably far-flung design points in the space of cache compression techniques. Please refer Appendix A for more details on these algorithms.

1. *Frequent Pattern Compression (FPC)* -Significance based, compress frequent patterns
2. *Base Delta Immediate Compression (BDI)* - Low-gradient within a line
3. *Golumb-Rice Encoding (GOL)* – Sparse cache contents with lots of 0s
4. *Cache-Packer (CPACK+Z)* – Significance + Dictionary based. Dynamic dictionary



Figure 7 Compression and Decompression Latencies for cache-compression algorithms

# Benchmarks/Workloads tested

*Cloudsuite Software testing* - It is a resource-hungry and time-consuming task that can leverage cloud computing. These are the applications that can potentially benefit from the abundance of resources in clustered systems. Cloudsuite SW testing basically runs Cloud9 - an automated software-testing platform that parallelizes symbolic execution and scales on clusters of commodity hardware.

*Cloudsuite Graph-analytics -* CloudSuite uses GraphLab machine learning and data mining software for the graph analytics benchmark. One of the implementations on GraphLab is TunkRank, which provides the influence of a Twitter user based on the number of that user’s followers.

*Specjbb -* This is a Java server benchmark. It is based on a world-wide supermarket company with an IT infrastructure that handles a mix of point-of-sale requests, online purchases and data-mining operations. Two metrics are important - a pure throughput metric and a metric that measures critical throughput under service-level agreements (SLAs) specifying response times.

*Speccpu (gcc and bwaves) -* This benchmarks is a CPU-intensive benchmark suite, stressing a system’s processor, memory subsystem and compiler. It provides a comparative measure of compute-intensive performance across the widest practical range of hardware using workloads developed from real user applications.

# Compression Results

We tested the aforementioned compression algorithms on the discussed benchmarks are able to make the following observations. (The various graphs are also shown below).

1. Compression ratio seen is the maximum in CPACK. Compression performance follows the trend - CPACK > FPC > BDI > GOL
2. Speccpu benchmarks are more compressible than specjbb which are more compressible than CloudSuite.
3. L1D cache is more compressible than L2. This is because L1D contains just data while the L2 contains both data and instructions.









**Integer vs fp**

We also performed an analysis of cache compression on integers vs floating point numbers. Compression ratios are, on average, higher for integer benchmarks compared to floating point benchmarks. However, some benefit is still possible for floating point benchmarks with a high percentage of zero words. This low compression ratio is because FP numbers (except 0) don’t fit into frequent patterns as much and their mantissa bits are highly irregular. The below shows this comparison (gcc is the tested integer workload while bwaves is the tested FP workload).

 

**Block sizes**

In increasing the block size (for our L2 runs) from 32kB to 128kB, we found that the amount of compression possible on cache contents goes up. Both CloudSuite and Specjbb show similar results in our runs.





# Code Compression

We tried running our compression algorithms for compressing of the L1 instruction cache. The compression ratio obtained were at best marginally better than 1.0 and in some cases was even less than 1.0. The below figure shows the results of these algorithms on Specjbb and CloudSuite.



Running the Cache Analyzer on the dumped contents of the L1-I cache, we found that most cache content assumptions don’t hold true for code. The below figure shows that only a small percentage of cache data falls into the ‘compressible categories’ of most conventional algorithms.



# Levels of Memory Heirarchy

Most works on cache compression focus exclusively on Last Level Caches as they have more tolerance for the decompression latency. However, from our study, we find that L1 Data cache is greatly suited for compression. Cache content decomposition reveals that a significantly greater portion of the L1D cache fits within common recurring patterns than the L2 cache. This suggests that compression algorithms that exploit these recurring patterns would perform better on L1D as compared to L2. Our results in the previous sections affirms this expectation.

As the following pie-chart tells us, often up to 90-92% of the L1D consists of easily exploitable recurring patterns, compared to <50% for the L2. We believe that the lower proportion for the L2 is due to our unified L2 design which stores less-compressible code along with highly compressible data.



Figure 8 L1D Cache Line Analysis

Given these results, we note the high compression potential for L1D and for systems where L1D latency tolerance is higher, such as GPUs, L1D compression could represent an exciting opportunity. However, our analysis also reveals a few new patterns in the L1D which are not found in L2 or L1I caches. These patterns suggest the viability of different algorithms suited more for L1D than L2 caches. In particular, two patterns are highly visible:

1. Frequent Value Locality
2. Endian-ness.

Frequent Value Locality reveals that among all the values in the L1D, a small subset of values are recurring with great periodicity. These values typically include small integers, FFFF and zeros. This presents the opportunity to have a small dictionary solely for holding these frequently occurring values. We could find just one work in the literature that has identified this locality in L1 cache contents [8]. Effects of Endian-ness can also be seen in L1D contents. Our simulations were for an X86 architecture which subscribes to Little Endian ordering of bytes within a word. When we analyzed entropy of upper-half-words and lower-half-words of each word in the L1D cache, we noted that the entropy of lower-half-words is significantly lower. Most of the half-words were solely 0000 or FFFF. We present these results through two intuitive figures.

In the figure below, we study the percentage of unique half-words in the L1D and the proportion of these unique half-words from the upper two bytes and the lower two bytes. This reveals the entropy of the upper and lower half-words. As we note below, a significant portion of unique half-words (70-95%) are contributed by the upper two bytes.



Figure 9 Entropy of L1D lower-half-words

Considering the low entropy of lower-half-words, we next proceeded to analyze the frequency distribution of unique lower-half-words. As expected 0000 and FFFF were the most dominant values, but the remaining values seem to exhibit sufficient reuse. The figure below shows the proportion of unique lower-half-words that could be fit into a 64 entry dictionary. We see that from 87-99% of the unique half-words are mere recurrences of a subset of 64 values. Considering that the L1D cache in our system has 2048 lines, with 16 lower half-words in each line, this was a very interesting result.



Figure 10 Repeated Lower-Half-words in L1D

The final question we hoped to answer was whether this recurrence has a *cache-line-locality* i.e. whether the lower-half-words are repeated often within each cache line. Intuitively one would expect such a behavior. For example, if we are storing an array of pointers to data structures, one expects the upper bytes of the pointers to remain the same, considering standard memory allocation policies. Hence, in each cache line we studied the repetition of lower half-words and whether a small subset of values are repeated consistently in each line.



Figure 11 Cumulative distribution of 4-most common lower-half-words

The above figure suggests that our expectation was right. For each cache line, 4 lower-half-word values are prevalent – 0000, FFFF and two other values picked from the line. We note that, except for a few aberrations, up to 14-16 lower-half-words are fitting within these 4 values. This suggests the scope for a small two-entry dictionary for each line to encode great chunks of the L1D cache.

# Conclusion

In this work, we have presented a comprehensive study of cache compression across various axes. Using standard entropy models, we studied the theoretical scope for compression and judged the inability of standard compression algorithms like gzip to effectively compress caches, emphasizing the need for specialized algorithms for compressing on-chip caches. We analyzed the potential for simple compression techniques by studying the cache contents of various workloads and noting that a significant chunk of the cache contents contain redundancies and narrow values. We implemented four widely different cache-compression algorithms in GEM5 and analyzed their efficacy for a range of workloads. We note that a combination of significance + dictionary based algorithms is the most effective at compression, however at a higher decompression latency. We also note that certain workloads from speccpu2006 exhibit significantly higher compression that other workloads such as specjbb2005 and cloud-suite. We also studied the compressibility of integers and floating-point benchmarks, with floating-point values being generally less compressible. The algorithms chosen perform very poorly for code compression, mainly due to instruction caches not exhibiting the common value patterns found in data caches. However, our theoretical entropy study suggests that code has sufficient compressibility, hence we conclude that different forms of redundancy should be employed to realize code compression. Lastly, we studied the variation of compression across levels of memory and note that L1D has significantly higher compression potential plus exhibits some unique and easy-to-exploit patterns for compressing lower-half-words in a Little Endian system. For a range of compressibility analysis, we developed an easy-to-use tool called Cache Analyzer.

# Appendix A – Overview on Cache Compression Algorithms

In this Appendix, we will examine the details of each compression algorithm implemented in this study. We will be providing the fundamental idea behind each algorithm as well as an overview on its implementation. The reader is advised to refer the relevant papers for more details.

## Frequent Pattern Compression

Frequent Pattern Compression is based on the insight that some data patterns are frequent in a cache and also compressible to a fewer number of bits. For example, many small-value integers can be stored in 4, 8 or 16 bits but are normally stored as 32-bits or 64-bits. These values are frequent enough to merit special treatment and storing them in a compressed form can increase the cache capacity. Special treatment is also given to runs of zeros as they are highly frequent. The key goal of FPC is to achieve dictionary-level compression without incurring heavy per-line overheads.

FPC compresses/decompresses on a cache line basis. Each cache line is divided into 32-bit words and each word is encoded as a 3-bit prefix plus data. The following table shows the different patterns for each prefix.

Table 2 FPC Patterns

|  |  |  |
| --- | --- | --- |
| **Prefix** | **Pattern Encoded** | **Data Size** |
| 000 | Zero Run | 3-bits (for runs upto 8 zeros) |
| 001 | 4-bit sign-extended | 4-bits |
| 010 | One byte sign-extended | 8-bits |
| 011 | Halfword sign-extended | 16-bits |
| 100 | Halfword padded with a zero halfword | The nonzero halfword (16-bits) |
| 101 | Two halfwords, each with a byte sign-extended | The two bytes (16-bits) |
| 110 | Word consisting of repeating bytes | 8-bits |
| 111 | Uncompressed word | Original word (32-bits) |

Each word in the cache line is encoded into a compressed format if it matches any of the patterns in the first six rows of the table. If a word doesn’t match any of these categories, then it is stored in its original 32-bit format. All prefix values as well as the zero-run length data bits are stored at the beginning of the line to speed up decompression. For an explanation on the choice of the patterns, please refer [1].

## Base Delta Immediate

BDI compression looks for compression granularities at cache line granularity such that it can store the entire cache line as a BASE value along with an array of DELTA values. The key idea behind this method is that many cache lines contain data with low dynamic range. As a result, the differences between words within such a line can be represented using fewer bytes than the words themselves. Since the detlas required fewer bytes, the combined size of base and array of deltas can be much smaller than the original line, achieving compression.

BDI views a cache line as a set of fixed-size values i.e. 8 8-byte, 16 4-byte or 32 2-byte values for a 64-byte cache line. It then determines if the set of values can be represented in a more compact form as a base value and a set of differences from the base value. The algorithm is capable of identifying up to two bases for a line with one base fixed as zero and the other base being determined from the cache line. To decompress a compressed line, the decompression algorithm needs the base value and the array of differences. The final values can be computed in parallel using a SIMD-style vector added. Hence, the decompression can be done in parallel for all words and with low latency.

## CPACK – Cache Packer

CPACK achieves compression by two means – (1) it uses statically decided, compact encodings for frequently appearing data words and (2) it encodes using a dynamically updated dictionary allowing adaptation to other frequently appearing words. The dictionary supports partial and full word matching. The patterns and the coding schemed employed in shown in the table below.

Table 3 Pattern Encoding for CPACK

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Pattern** | **Output** | **Length (b)** |
| 00 | zzzz | (00) | 2 |
| 01 | xxxx | (01)BBBB | 34 |
| 10 | mmmm | (10) bbbb | 6 |
| 1100 | mmxx | (1100)bbbBB | 24 |
| 1101 | zzzx | (1101)B | 12 |
| 1110 | mmmx | (1110)bbbbB | 16 |

‘z’ represents a zero-byte, ‘m’ represents a byte matched against a dictionary entry, ‘x’ represents an unmatched byte and B represents a byte, ‘b’ represents a bit. During compression, each word is compared with the patterns ‘zzzz’ and ‘zzzx’, if there is a match, the compression output is produced by combining the corresponding code and unmatched bytes. Otherwise, the word is compared with all dictionary entries and finds the one with the most matched bytes. The result is then obtained by combining code, dictionary entry index and unmatched bytes. Words that fail pattern matching are pushed into the dictionary. During decompression, the decompressor first extracts the codes for analyzing the patterns of each word. If the code indicates a pattern match, the original word is recovered by combining zeros and unmatched bytes. Otherwise, the decompression output is given by combining bytes from the input word with bytes from the dictionary entries, if the code indicates a dictionary match.

## Golomb-Rice Encoding

Golomb coding uses a tunable parameter M to divide an input value N into two parts: q, the result of the division by M and r, the remainder. The quotient is sent in unary coding, followed the remainder in truncated binary encoding. This encoding is particularly useful for situations in which the occurrence of small values in the input stream is significantly more likely than large values.

A simple encoding scheme would consist of:

1. For N, the number to be encoded, find quotient = q = int[N/M] and remainder = r = N modulo M.
2. Generate Codeword with format : <Quotient Code><Remainder Code>, where Quotient Code (in [unary coding](http://en.wikipedia.org/wiki/Unary_coding)) consists of a q-length string of 1s followed by a 0; and Remainder Code (in [truncated binary encoding](http://en.wikipedia.org/wiki/Truncated_binary_encoding)). If M is power of 2, code remainder as binary format. So  bits are needed. If  code r as plain binary using b-1 bits. If  code the number  in plain binary representation using b bits.

# Appendix B – Cache Analyzer Graphs

In this Appendix, we present graphs for some of the features the Cache Analyzer tool which were not explained in the main body. The tool itself has sufficient documentation & comments on reproducing these graphs for different workloads.

Cache Analyzer can analyze the occurrences of narrow and aligned values in cache lines and plot insightful histograms to reveal the distribution of these lines.



Figure 12 Cache Analyzer - Histogram of aligned & narrow values



Figure 13 Cache Analyzer - Most Common Words analysis



Figure 14 - Cache Analyzer - BDI size estimation

# References

[1] Alaa R. Alameldeen and David A. Wood. Frequent Pattern Compression: A Significance-Based Compression Scheme for L2 Caches. In Technical Report 1500, Computer Sciences Dept., UW-Madison, April 2004

[2] Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Michael A. Kozuch, Phillip B. Gibbons, Todd C. Mowry. Base-Delta-Immediate Compression: Practical Data Compression for On-Chip Caches. In PACT’12, September 19-23, 2012, Minneapolis, Minnesota, USA.

[3] Xi Chen, Lei Yang, Robert P. Dick, Li Shang, Haris Lekatsas. C-Pack: A High-Performance Microprocessor Cache Compression Algorithm. In IEEE TVLSI, August 2010.

[4] Alaa R. Alameldeen and David A. Wood. Adaptive cache compression for high-performance processors. In ISCA-31, 2004.

[5] D. Huffman. A method for the construction of minimum-redundancy codes. 1952.[6] Golumb-Rice encoding. <http://en.wikipedia.org/wiki/Golomb_coding>

[7] Sangheon Lee, Nakwoong Eum, Moo-Kyoung Chung, Chong-Min Kyung. Low Latency Variable Length Coding Scheme for Frame Memory Recompression. *In ICME, 2010 IEEE International Conference, Suntec City.*

[8] Georgios Keramidas, Konstantinos Aisopos, Stefanos Kaxiras. Dynamic dictionary-based data compression for level-1 caches. In *ARCS'06 Proceedings of the 19th international conference on Architecture of Computing Systems*