International Journal of Research Publication and Reviews

Journal homepage: www.ijrpr.com ISSN 2582-7421

"Instruction Cache Compression"

[1]S Bhargavi, [2]Jashwanth R S

[1,2]Dept. of ECE, SJCIT, Chickaballapur, India

ABSTRACT—

Code compression could lead to less overall system die area and therefore less cost. This is significant in the embedded system field where cost is very sensitive. In most of the recent approaches for code compression, only instruction ROM is compressed. Decompression is done between the cache and memory, and instruction cache is kept uncompressed. Additional saving could be achieved if the decompression unit is moved to between CPU and cache, and keeps the instruction cache compressed as well. The most commonly used algorithms for block compression are variations of Huffman and arithmetic that compresses serially within the block, byte wise. A feature of Huffman coding is how the variable length codes can be packed together. A more sophisticated version of the Huffman approach is called arithmetic encoding. In this scheme, sequences of characters are represented by individual codes, according to their probability of occurrence. The discussion is carried with the help of a compression algorithm with instruction level random access within the compressed file. In addition we present a branch compensation cache, a small cache mechanism to alleviate the unique branching penalties that branch prediction cannot reduce.

Keywords— Compression, Instruction cache, Huffman coding, Arithmetic encoding.

I. Introduction

A compressed instruction set, or simply compressed instructions, are a variation on a microprocessor's instruction set architecture that allows instructions to be represented in a more compact format. In most real-world examples, compressed instructions are 16 bits long in a processor that would otherwise use 32-bit instructions. The 16-bit ISA is a subset of the full 32- bit ISA, not a separate instruction set. The smaller format requires some tradeoffs, generally, there are fewer instructions available, and fewer processor registers can be used. Supplementary memory system that temporarily stores frequently used instructions and data for quicker processing by the CPU of a computer. Cache memory is an extremely fast memory type that acts as a buffer between RAM and the CPU.

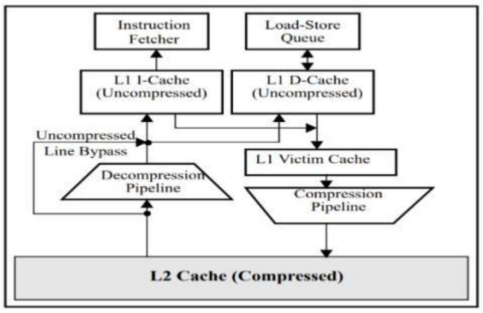

Fig 1: Compressed Cache Hierarchy[1].

Fig. 1 illustrates the proposed cache hierarchy. L1 instruction and data caches store uncompressed lines, eliminating the decompression overhead from the critical L1 hit path. This design also completely isolates the processor core from the compression hardware. It holds frequently requested data and instructions so that they are immediately available to the CPU when needed. Cache memory is used to reduce the average time to access data from the Main memory. Compression is the process of encoding, restructuring or modifying data in order to reduce its size. Cache compression is a promising technique to increase on-chip cache capacity and to decrease on-chip and off- chip bandwidth usage. The L1 data cache uses a write back, write allocate policy to simplify the L2 compression logic. On L1 misses, the controller checks an uncompressed victim cache in parallel with the L2 access. On an L2 hit, the L2 line is decompressed if stored in compressed form. Otherwise, it bypasses the decompression pipeline. On an L2 miss, the requested line is fetched from main memory. We assume uncompressed memory, however, this is largely an orthogonal decision. The L1 and L2 caches maintain exclusion and lines are allocated in the L2 only when replaced from the L1. In addition to its normal function, the victim cache acts as a rate-matching buffer between the L1s and the compression pipeline. For design simplicity, we assume a single line size for all caches. The new design had two instruction sets, one giving access to the entire ISA of the new design, and a smaller 16-bit set known as SH- compact that allowed programs to run in smaller amounts of main memory. This cache memory is divided into levels which are used for describing how close and fast access the cache memory is to main or CPU memory.

II. Methodology

Any compression algorithm will not work unless a means of decompression is also provided due to the nature of data compression. When compression algorithms are discussed in general, the word compression alone actually implies the context of both compression and decompression. Browser caching is probably the most common type of caching that you will find in the wild. It's a collaboration between the browser and the web server, used to speed up the loading time of repeated visits to the same site. Keep in mind that browser caching is an extensive topic, spanning multiple specification documents, so I will not cover every detail here. Then, when the page is reloaded, a few things can happen in terms of caching: If the Expires header is set to a date in the future, the resource will be loaded from the cache and no server request will be made. If there is no Expires header or if it has a date in the past, the resource is requested again. The browser will send any other caching headers back with the request. Then if the server determines that the resource hasn't changed, it will return a response with HTTP code 304 and no data. This saves the server some work and some bandwidth. When a browser receives the 304 'Not modified' code, it will load the resource from the cache. For example, movies, photos, and audio data are often compressed once by the artist and then the same version of the compressed files is decompressed many times by millions of viewers or listeners.

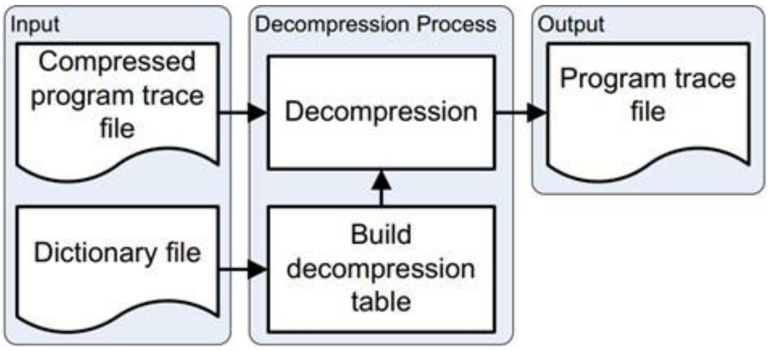

Fig 2: Decompression process.

Fig. 2 shows Decompression process. Building the matrix M for a window of size n is quadratic in n; building the counting matrix S is cubic in n. Note that this is an extremely pessimistic estimate and in practice the time complexity is essentially quadratic in n, because the matrix M is very sparse, and the largest sequence of consecutive ones to be counted is very small compared to the window size n. If the trace is of size t, then the number of windows to process is 2t=n. Thus the total work prior to encoding is O(t n), O(t n2). Finally, encoding is linear in t. Thus, as verified in practice, the expected run time is proportional to t n. Notice that the compression efficiency, for as long as reasonably feasible, is not very critical. This is because the compression is just a preprocessing step that is done only once for each program trace. Subsequently, a set of stored pre-processed traces is reused many times for different statistical analyses of various applications with various inputs. To verify and guarantee the correctness of the compression algorithm, a decompression algorithm was built to reproduce the trace file given the dictionary file and the compressed trace file. Note that in actual use, decompression is not needed; our analysis algorithm executes on the compressed trace.

III. Technology

Compression Algorithm

Run length encoding is a basic form of compression that converts consecutive identical values into a code consisting of the character and the number marking the length of the run. RLE is lossless compression. Redundancy reduction, used during lossless encoding, searches for patterns that can be expressed more efficiently. An image viewed after lossless compression will appear identical to the way it was before being compressed. Lossless compression techniques can reduce the size of images by up to half. The resulting compressed file may still be large and unsuitable for network dissemination. However, lossless compression does provide for more efficient storage when it is imperative that all the information stored in an image should be preserved for future use.

aaaaabbbbbbbbbbbbccccddddddddddeeeeeeeeee (Uncompressed)

5a12b4c9d10e (Compressed)

Compression algorithm used must have simple CLB and DEC sections, or it will lead to large branch penalty. The problems we face with the usual CCRP compression algorithm that compresses serially within the block byte wise are:

For a 64-bit instruction, we require 8 DEC stages to decompress.

Granularity of random access.

Consider a screen containing plain black text on a solid white background, over hypothetical scan line, it can be rendered as follows:

12W1B12W3B24W1B14W

This can be interpreted as a sequence of twelve Ws, one B, twelve Ws, three Bs, etc., and represents the original 67 characters in only 18. While the actual format used for the storage of images is generally binary rather than ASCII characters like this, the principle remains the same. Even binary data files can be compressed with this method; file format specifications often dictate repeated bytes in files as padding space. However, newer compression methods such as DEFLATE often use LZ77-based algorithms, a generalization of run-length encoding that can take advantage of runs of strings of characters. Run-length encoding can be expressed in multiple ways to accommodate data properties as well as additional compression algorithms. For instance, one

popular method encodes run lengths for runs of two or more characters only, using an "escape" symbol to identify runs, or using the character itself as the escape, so that any time a character appears twice it denotes a run. On the previous example, this would give the following:
WW12BWW12BB3WW24BWW14
This would be interpreted as a run of twelve Ws, a B, a run of twelve Ws, a run of three Bs, etc. In data where runs are less frequent, this can significantly improve the compression rate.
One other matter is the application of additional compression algorithms. Even with the runs extracted, the frequencies of different characters may be large, allowing for further compression; however, if the run lengths are written in the file in the locations where the runs occurred, the presence of these numbers interrupts the normal flow and makes it harder to compress.

Modified Algorithm

Huffman coding is a method of compression that is independent of the data type, that is, the data could represent an image, audio or spreadsheet. This compression scheme is used in JPEG and MPEG-2. Huffman coding is lossy compression. Huffman coding is done with the help of the following steps. Calculate the frequency of each character in the string. Sort the characters in increasing order of the frequency. These are stored in a priority queue Q . Make each unique character as a leaf node. Create an empty node z. Huffman Codes are Data can be encoded efficiently using Huffman Codes, It is a widely used and beneficial technique for compressing data and Huffman's greedy algorithm uses a table of the frequencies of occurrences of each character to build up an optimal way of representing each character as a binary string.

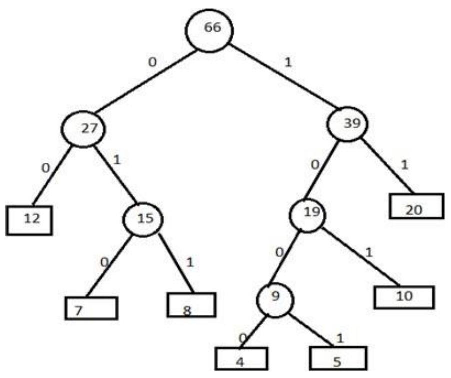

Fig 3: Huffman Coding Tree

Modified algorithm can be thought of as a binary Huffman with only two compressed symbol lengths, but due to its similarity to table lookup we refer to it as Table. It uses 4 tables, each with 256, 16-bit entries. Each 64-bit instruction is compressed individually as follow:

- Divide the 64-bit instruction into 4 sections of 16-bit.
- Search content of table 1 for section 1 of the instruction. If found, record the entry number.
- Repeat Step (ii) for section 2 with table 2 and so on with remaining sections.
- If entry number in the corresponding table replaces all the sections, then the instruction is compressed. Otherwise, the instruction remains uncompressed.
- A 96-bit LAT entry tracks every 64 instructions. 32- bits equals the compressed address. The rest of 64 bits are on or off depending on whether the corresponding instruction is compressed or not. Since this method tracts each instruction individually, the inter-block jump problem is gone. For the DEC section, decompression is either nothing or 4 table lookups in parallel. So, one stage is sufficient. For the CLB section, it is simply a CLB lookup follow by some adding in a tree of adders. This could be done in one cycle, with CLB lookup taking half and adding taking another half.

The Least Recently Used

Least Recently Used (LRU) algorithm is a page replacement technique used for memory management. According to this method, the page which is least recently used is replaced. Therefore, in memory, any page that has been unused for a longer period of time than the others is replaced. Size of primary storage has increased by multiple orders of magnitude.

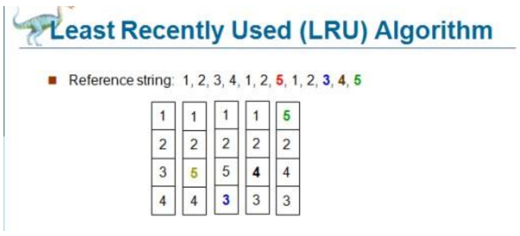
Least Recently Used (LRU) Algorithm
Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
1 2 3 4
1 2 5 4
1 2 5 3
1 2 4 3
5 2 4 3

Fig 4: LRU Algorithm.

With several gigabytes of primary memory, algorithms that require a periodic check of each and every memory frame are becoming less and less practical. Memory hierarchies haveImproves application performance grown taller. The cost of a CPU cache miss is far more expensive. This exacerbates the previous problem. The goal of any page replacement mechanism is to minimize the number of page faults. Page replacement algorithms were a hot topic of research and debate in the 1960s and 1970s. That mostly ended with the development of sophisticated LRU (least recently used) approximations and working set algorithms. Moreover, as the goal of page replacement is to minimize total time waiting for memory, it has to take into account memory requirements imposed by other kernel sub-systems that allocate memory. As a result, page replacement in modern kernels tends to work at the level of a general purpose kernel memory allocator, rather than at the higher level of a virtual memory subsystem.

The Least Frequently Used

Least Frequently Used (LFU) is a type of cache algorithm used to manage memory within a computer [12]. The standard characteristics of this method involve the system keeping track of the number of times a block is referenced in memory. When the cache is full and requires more room the system will purge the item with the lowest reference frequency. There are some downsides to this cache like if a new element is accessed, it will require significant number of repeated access to make it enter the cache as the frequency should be higher than already existing elements. In some cases, LFU Cache can be the best performing cache as well.

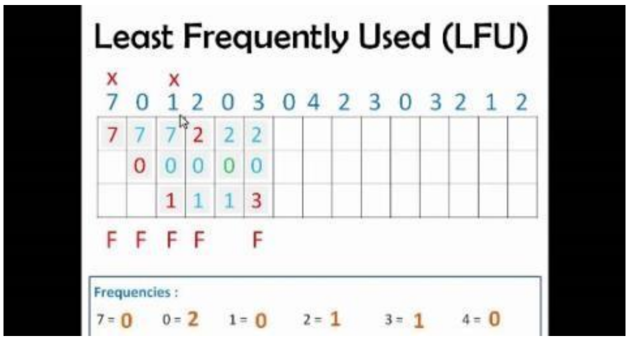

Fig 5: Least Frequently Used.

LFU is sometimes combined with a Least Recently Used algorithm and called LRFU. The simplest method to employ an LFU algorithm is to assign a counter to every block that is loaded into the cache. Each time a reference is made to that block the counter is increased by one. When the cache reaches capacity and has a new block waiting to be inserted the system will search for the block with the lowest counter and remove it from the cache.

- Ideal LFU: there is a counter for each item in the catalogue
- Practical LFU: there is a counter for the items stored in cache.

IV. Application

- Improves application performance
- Reduces database cost
- Reduces load on the backend
- Predictable Performance
- Eliminates database hotspots
- Increase Read Throughput

V. Advantages

- Improve Application Performance.
- Compression reduces the cost of storage, increases the speed of algorithms.
- Compression is achieved by removing redundancy, that is repetition of unnecessary data.
- Reduce the Load on the Backend.
- A Statistical compression cache scheme.
- A High-Performance microprocessor compression algorithm.
- Predictable Performance.
- Eliminate Database Hotspots. Increase Read Throughput.

VI. Conclusion

The observation a tradeoff between granularity of random access and compression rate, where smaller granularity implies less compression. To avoid access branch penalty, smaller granularity of random access than CCRP is forced in order to compress the cache. Consequently, compression is less in instruction ROM than CCRP scenario. This drawback offsets any saving in instruction cache, except in region where instruction cache is big relative to overall system. Instruction cache compression is effective in embedded system, related to area where cache performance is more important than the instruction ROM die area. Our current method can be improved by following: Provide optimal entries, which enable us to compress more instructions. Since table entry calculations are done in preprocessing, the potential exponential amount of time required may not be a problem. If it does become a time problem, better heuristics can definitely be devised. If multiple levels of tables are used, for example 4 entries tables, 16 entries tables and 256 entries tables, then we can compress with 4 entries tables first. If it falls then use the next level of tables. The advantage here is higher level tables with lease entries can compress more since pointer into the table has fewer bits.

References

1) S. Sardashti, A. Seznec and D. A. Wood, Yet Another Compressed Cache, ACM Trans. Archit. Code Optim., vol. 13, no. 3, Sep. 2018, pp. 1–25.
2) B. Panda and A. Seznec, Dictionary sharing: An efficient cache compression scheme for compressed caches, in 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture, vol. 28, 2016, pp. 1–12.
3) Qi Zeng, Rakesh Jha, Shigang Chen, and Jih-Kwon Peir, Data Locality Exploitation in Cache Compression, 2018 IEEE 24th International Conference on Parallel and Distributed Systems, 21 February 2019.
4) Uthayakumar Jayasankara, Vengattaraman Thirumal and Dhavachelvan Ponnurangam, A survey on data compression techniques: From the perspective of data quality, coding schemes, data type and applications, Journal of King Saud University - Computer and Information Sciences Vol. 33, Issue 2, February 2021, pp. 119-140.
5) G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, Basedelta-immediate compression, in Proceedings of the 21st international conference on Parallel architectures and compilation techniques - PACT ’12, 2012, pp. 377-389.
6) B. Panda and A. Seznec, Dictionary sharing: An efficient cache compression scheme for compressed caches, in 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture, 2016, pp. 1– 12.
7) Arelakis and P. Stenstrom, SC2: A statistical compression cache scheme, 2014 ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, pp. 145– 156.
8) Arelakis, F. Dahlgren, and P. Stenstrom, HyComp: A Hybrid Cache Compression Method for Selection of Data-typespecific Compression Methods, Proc. 48th Int. Symp. 7 Microarchitecture, 2016, pp. 38–49.
9) N. Binkert, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. Hill, and D. Wood, The gem5 Simulator, Comput. Archit. News, vol. 39, no. 2, 2011, pp. 1-11.