Instruction Cache Compression for Embedded Systems

by
Yujia Jin and Rong Chen

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 keep the instruction cache compressed as well. In this project we explored the issues for compressing the instruction cache. In the process, we constructed a high level implementation for a 64-bit, 5-stage pipeline MIPS like processor with compressed instruction cache. We developed a compression algorithm with instruction level random access within the compressed file. In addition we devised a branch compensation cache, a small cache mechanism to alleviate the unique branching penalties that branch prediction cannot reduce.

1. Introduction

Code compression is a general technique. A common method is the Compressed Code RISC Processor (CCRP)[6], which is show in *figure 1*. It breaks the program into blocks of n, where n corresponds to the cache line size. Each blocks is compressed individually by a suitable algorithm. A Line Address Table (LAT) is used to record the address of each compressed cache line within the compressed program. Cache Lookaside Buffer (CLB) is added to alleviate LAT lookup cost. During program execution, upon each cache miss, LAT entry cached in CLB is used to calculate the compressed cache line address. The compressed cache line is then fetched from memory, decompressed, and placed into the instruction cache. The most commonly used algorithms for block compression are variations of Huffman and arithmetic that compresses serially within the block, byte wise. A few of such algorithm can be found in [1], [5], and [6].

In this project we modify CCRP to compress the instruction cache. The organization of rest of the paper is as follow. In section 2, necessary modifications for the processor are discussed. In section 3, compression algorithm is discussed. In section 4, branch compensation cache is discussed. In section 5, results are shown. Finally, in section 6, our conclusions are stated.

2. Pipeline

Design is done on the 64-bit simple scalar instruction set, which is similar to MIPS. We assume a simple 5-stage pipeline processor for ease of development. In order to compress the instruction cache, decompression must be done after the cache. Therefore the CPU must now be modified to include 2 additional functions.

1) Address translation from the uncompressed version to the compressed version for each instruction. (CLB)
2) Instruction decompression. (DEC)

These 2 additional sections are added to the usual 5-stage pipeline as shown in *figure 2*. The CLB section implements 1) above, and DEC implements 2) above. The IF stage will now fetch the compressed instruction instead of the uncompressed instruction. The stage length for CLB and DEC are both dependent on the compression algorithm.

The additional CLB and DEC sections are both before the ID stage. Thus stages within these sections will add to branch penalty. In addition, these penalties cannot be reduced by branch prediction since they are before the ID stage, where instruction opcode is first being identified.

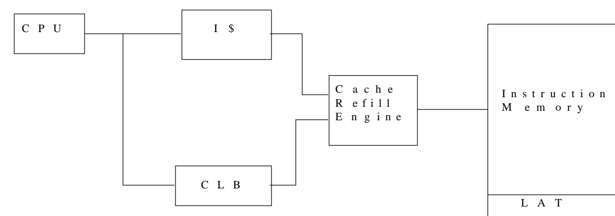


figure 1

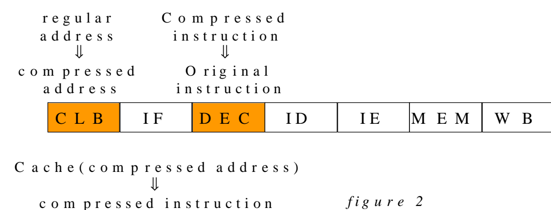


figure 2

3. Compression Algorithm

3.1. Problem

Compression algorithm used must have simple CLB and DEC sections, or it will lead to large branch penalty. The usual CCRP compression algorithm that compresses serially within the block, byte wise, will translate to 8 DEC stage, even with the assumption of single stage byte decompression, for our 64-bit instruction. Another problem with the standard CCRP compression algorithm is its granularity of random access. With CCRP, random access is only at block level. If the program counter jumps from one compressed block to another and it is the very last instruction in the block that it is jumping to, then all instructions within the block must be decompressed before the actual instruction will be fetched. This will lead to additional (n-1)*(|CLB|+|DEC|) stages of delay, where n is the number of instruction within the block, |CLB| and |DEC| are

number of stages within the CLB and DEC section, respectively. Therefore the best compression algorithm in term of branch penalty is one with instruction level random access, or n=1, and |CLB| and |DEC| of a single stage.

3.2. Algorithm

We devise such an algorithm. It can be thought of as a binary Huffman with only two compressed symbol length, 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.

1) Divide the 64-bit instruction into 4 sections of 16-bit.
2) Search content of table 1 for section 1 of the instruction. If found, record the entry number.
3) Repeat 2) for section 2 with table 2, section 3 with table 3, and section 4 with table 4.
4) If all four sections can be replaced by an entry number into the corresponding table, then the instruction is compressed by replacing it with the four, 8 bit entry numbers. Otherwise, the instruction remains uncompressed.
5) A 96 bit LAT entry is used for tracking every 64 instructions. It starts with a 32-bit base, which equals the compressed address for the first instruction. 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. Similar structure is used in [6]. They assume it could be done in one cycle, with CLB lookup taking half and adding taking another half. So we make the same assumption.

3.3. Details

With the Table compression algorithm, the compressed instructions can be either 4 or 8 byte long. This could lead to misalignment problem for instruction cache access in case of 8-byte instruction. For example, an 8-byte instruction can have its first 4 byte in the end of one cache line and its last 4 byte in the start of another cache line. This will result 2 cache accesses for a single instruction fetch. To solve this, instruction cache is divided into two banks, with each bank 4-byte wide. The first bank is used to store the upper half word of the 64-bit word. The second bank is used to store the lower half. Since every 8-byte instruction must have its two 4 byte halves in different banks, simple logic can be used to ensure each bank is accessed correctly to fetch the instruction in one cycle.

The determination of the content for the 4 tables is critical in achieving a good compression for this method. Between the 4 tables, they can keep 256^4, or 4 gig entries of distinct instructions. However, for every entry within a table, 256^3, or 16 meg entries of instructions will take on that same section. So it is quantity without quality. The problem is to determine the table content such that maximum number of program instructions can be compressed. Mathematically, we are maximizing $\Sigma_i\Sigma_j\Sigma_k\Sigma_l\, f(A_iB_jC_kD_l)$, where A_i, B_j, C_k, D_l are entries in table 1, 2, 3, 4, respectively, and $f(A_iB_jC_kD_l)$ is the occurrence frequency within the program for instruction $A_iB_jC_kD_l$. This is like an unate covering problem. So it is potentially NP complete. Therefore we use the following heuristic. The 256 most frequently occurring 16 bit symbols in section i of all the program instructions is placed into table i.

3.4. Result

In *figure 3* is the compression result for Table compression with the heuristic. Applications used are programs in spec95. On average Table can compress to 70% of the original size. In comparison to CCRP with Huffman byte serial compression at block size of 8, we loose about 20% on average. This 20% lost is partly due to the smaller granularity of random access we have achieved. In general the smaller granularity of random access, the less compression will be achieved. This can

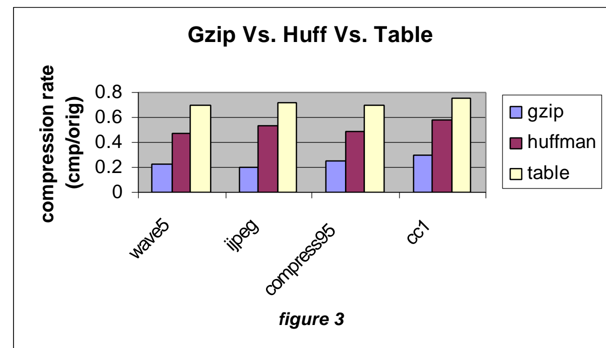


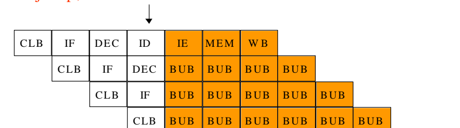
figure 3

be observed again with gzip's result, which does not provide any random access. On average it compresses to 20% of the original size, which is 30% better than CCRP with Huffman.

4. Branch Compensation Cache (BCC)

4.1. Problem

Find out it's a jump, has to flush



Restart the pipeline

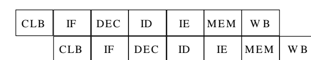


figure 4 more branch penalty for modified pipeline

As mentioned earlier, in order to perform decompression after instruction cache (between the I-cache and the CPU), we add two more stages into the pipeline before ID: CLB (cache lookaside buffer) and DEC

(decompression). As a result, branch penalty is increased to three cycles. This is illustrated in *figure 4*.
Originally, in regular five-stage pipeline, the branch penalty is only one stage (IF), but in our seven-stage pipeline, the branch penalty increases to three stages (CLB, IF, DEC). Obviously it is not good, so we need a solution.

4.2. Solution

The solution is to add a branch compensation cache (BCC) and try to pre-store the target instructions there. Whenever we encounter a branch (or more precisely, a PC jump), go to the BCC and check if the target instruction is there. The basic idea is shown in *figure 5*.
In *figure 5*, a PC jump is found at the ID stage, so we go to check BCC to see if the target instruction is pre-stored there. If no, then we have to go through CLB, DEC and ID pipeline stages; but if yes, then we simply fetch it and keep going, no branch penalty at all in such case.

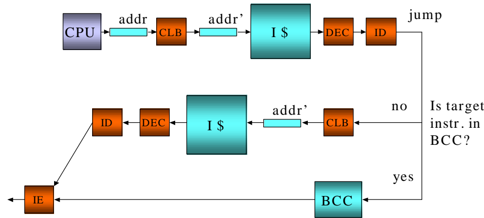


figure 5 use BCC to reduce branch penalty

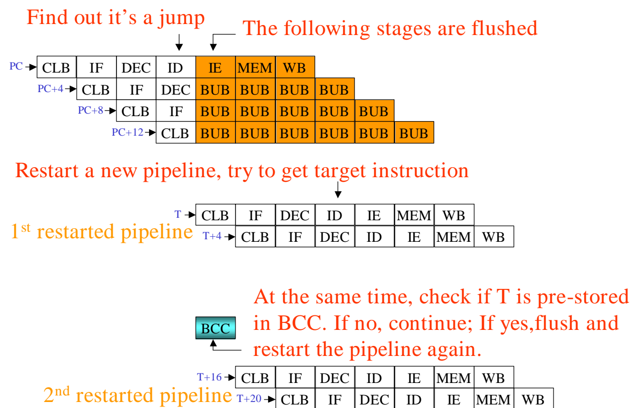


figure 6 details of BCC implementation

Figure 6 shows a more detailed pipeline implementation, where the key points are:

- At ID stage we find out a jump instruction being taken, so we must flush the current pipeline and restart.
- The restarted pipeline is used to get the target instruction through normal stages, i.e. using CLB to get the compressed address from the regular address, using IF to fetch the decompressed instruction, using DEC to decompress it to get target instruction. Simultaneously, we also go to the BCC to check if the required target instruction is pre-stored there. If no, then keep running the restarted pipeline; if yes, then we directly fetch the target instruction from BCC and flush/restart the pipeline for a 2nd time.
- The 2nd restarted pipeline is used to provide the sequential instructions after the jump. In order to completely eliminate the penalty incurred by a jump, we require T, T+8, T+16, T+24 instructions are all pre-stored in BCC upon a hit. So the 2nd restarted pipeline starts CLB stage with T+32 instruction, where T is the target instruction, T+8 is the next instruction, and so on.

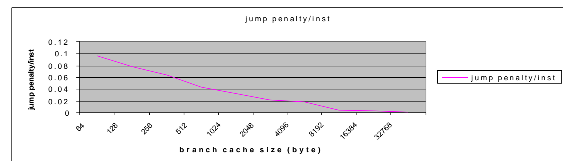


figure 7 compress in spec95

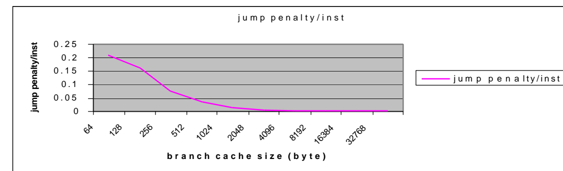


figure 8 ijpeg in spec95

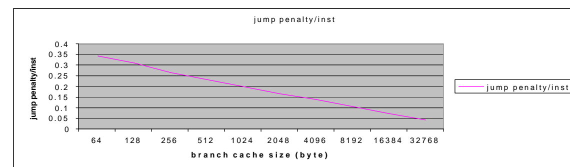


figure 9 gcc in spec95

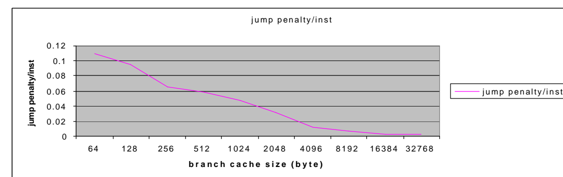


figure 10 wave5 in spec95

5. Result

5.1. Experiment Setup

Simple Scalar 3.0 is modified to simulate compressed instruction cache and branch compensation cache behaviors. CLB configuration of 8 entries is used where relevant. Instruction cache format of direct map, 8 byte block size with random replacement policy is used.

5.2. BCC Performance

To quantify the BCC performance, we simulate it on several applications such as spec95 compress, ijpeg, gcc

and wave5. In the simulation, the cache block size is kept as 8-byte long, while the block number varies from 1, 2, 4, 8, …, to 4096 to give different cache size. Directly mapped cache is used and the replacement policy is least recently used (LRU). *Figure 7 - 10* show the results. From the simulation results we can see that for a fairly small branch compensation cache (1KB ~ 2KB), the branch penalty is significantly reduced, which indicates our approach is very effective.

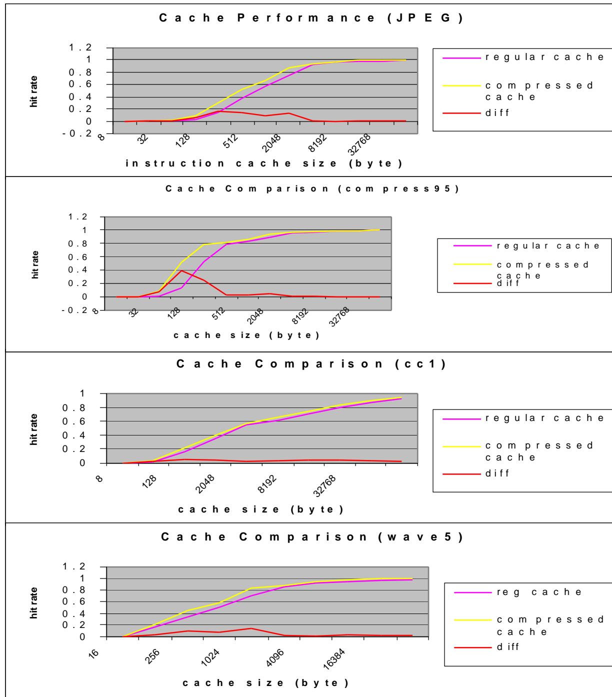


figure 11 cache performance analysis

5.3. Compressed Cache Performance

Experiments are performed on wave5, ijpeg, compress95, and cc1 within the spec95 benchmark. Results are displayed in graph *figure 11* which shows compressed cache with Table algorithm do outperform uncompressed cache, especially within a region of instruction cache size where uncompressed cache is getting from 10 to 90 % hit rate. The improvement could go as high as near 40%, which is shown in compress95's result. This is as expected since compressed cache contains more instructions and this translates to better hit rate. The reason that there is a window of sizes where cache compression is more effective is because this is the thrashing region and a small increase in cache size could give big improvement. Compression in cache is like a virtual cache increase. For example 70% compression in cache could give a cache that is 1.4 times the original size. Therefore cache compression is especially effective within this thrashing region. So the original idea that compressed instruction cache can lead to smaller instruction cache is valid.

5.4. Area Versus CPI Analysis

Area versus CPI analysis is carried out for the following configurations on wave5, ijpeg, compress95, and cc1 applications.

1) Regular 64-bit, 5-stage pipelined MIPS processor without any code compression.
2) Code compression only in instruction ROM by Huffman encoding as in CCRP.
3) Code compression with Table method to keep both the instruction ROM and instruction cache compressed.
4) Same as in configuration 3) with BCC added.

0.15 μm technology is assumed. Total die area is increase by increasing instruction cache size. Results are shown in *figure 12*. It is clear in all cases that code

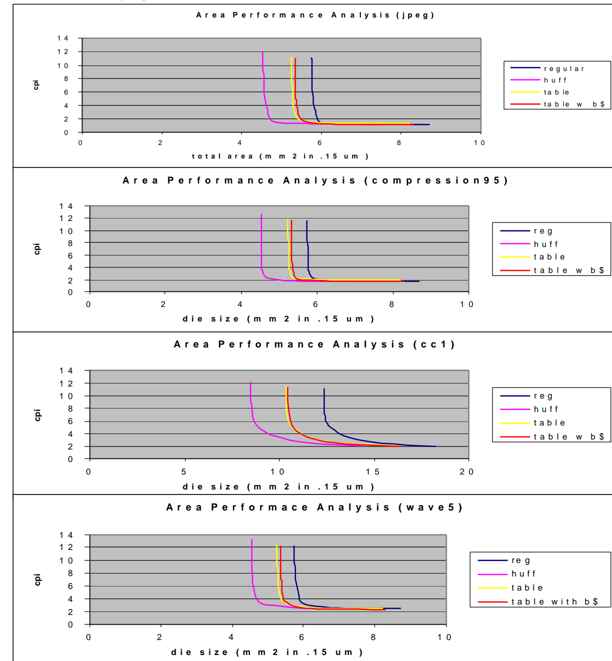


figure12 area performance analysis

compression at any level results smaller die size for similar CPI performance machines. The cache compression configurations are worse in die area against the CCRP configuration in 2), except in very low CPI region, because the excess loss in ROM offset the saving in instruction cache. The addition of BCC doesn't improve much except in the low CPI region because branching penalty is less of a factor than the cache miss when determining CPI outside this region.

6. Conclusions

We observe 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 (at cache line granularity) 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

and CPI is very low. Consequently, in the interest of embedded system, where cache size is always small relative to the overall system, instruction cache compression is not useful in comparison to the CCRP approach.

Although instruction cache compression is not effective in embedded system, it can be important in area where cache performance is more important than the instruction ROM die area. For example, in high performance computing, where large instruction cache is useful in improving hit rate, but painful in dealing with the critical path analysis. Here with instruction cache compression, one could potentially provide a cache performance that is 1/(compression rate-LAT) times the physical size.

Our current Table method can be improved by following.

1) Provide optimal table entries, which enable us to compress more instructions. Since table entries 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.

2) Provide multiple levels of tables, which can give us more compression. Current Table method has a lower compression limit of about 50% of the original size. 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 less bits.

7. Reference

[1] Haris Lekatsas and Wayne Wolf. Random access decompression using binary arithmetic coding. In Proc. Data Compression Conference, pages 306-315, March 1999.

[2] Y. Yoshida, B.-Y. Song, H. Okuhata, T. Onoye, and I. Shirakawa. An object code compression approach to embedded processors. In International Symposium on Low Power Electronics and Design (ISLPED), pages 265-268. ACM, August 1997.

[3] Martin Benes, Steven M. Nowick, and Andrew Wolfe. A fast asynchronous Huffman decoder for compressed-code embedded processors. In Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems, September 1998.

[4] Stan Liao, Srinivas Devadas, Kurt Keutzer, Steve Tjiang, and Albert Wang. Code optimization techniques for embedded DSP microprocessors. In Proc. ACM/IEEE Design Automation Conference, 1995.

[5] Haris Lekatsas and Wayne Wolf. Code Compression for Embedded Systems. In Proc. 35th Design Automation Conference, 1998.

[6] A. Wolfe and A. Chanin. Executing compressed programs on an embedded RISC architecture. In Proc. Int'l Symp. on Microarchitecture, 1992.

[7] C. Lefurgy, E. Piccininni and T. Mudge, Reducing code size with run-time decompression, EECS Dept., Univ. of Michigan, 2000.