Compression in Data Caches with Compressible Field Isolation for Recursive Data Structures

Masamichi Takagi[1] and Kei Hiraki[1]

Dept. of CS, Grad. School of Info. Science and Tech., Univ. of Tokyo
{takagi-m, hiraki}@is.s.u-tokyo.ac.jp

Abstract. We introduce a software/hardware scheme called the Field Array Compression Technique (FACT) which reduces cache misses due to recursive data structures. Using a data layout transformation, data with temporal affinity is gathered in contiguous memory, where the recursive pointers and integer fields are compressed. As a result, one cache-block can capture a greater amount of data with temporal affinity, especially pointers, improving the prefetching effect of a cache-block. In addition, the compression enlarges the effective cache capacity. On a suite of pointer-intensive programs, FACT achieves a 41.6% reduction in memory stall time and a 37.4% speedup on average.

1 Introduction

Non-numeric programs often use recursive data structures (RDS). For example, they are used to represent variable-length object-lists, trees for data repositories. Such programs using RDS make graphs and traverse them, however the traversal code often leads to cache misses. Many studies have proposed techniques effective for reducing these misses. These are (1) data prefetching [3], (2) data layout transformations, which gather data with temporal affinity in contiguous memory to improve the prefetch effect of a cache-block [1], and (3) data compression in caches, which compresses the data stored in the caches [4–6]. Not only does compression enlarge the effective cache capacity, but it also increases the effective cache-block size. Therefore applying it with the data layout transformation can produce a synergy effect that further enhances the prefetch effect of a cache-block. This combined method can be complementary to data prefetching. While we can compress data into $\frac{1}{8}$ or less of its original size in some programs, existing data compression methods in data caches limit the compression ratio to $\frac{1}{2}$, mainly due to the hardware complexity in the cache structure [4–6]. Therefore we propose a method which achieves a compression ratio over $\frac{1}{2}$.

In this paper, we propose a software/hardware scheme which we call the Field Array Compression Technique (FACT). FACT aims to reduce cache misses caused by RDS through data layout transformation of the structures and compression of the structure fields. This has several positive effects. (1) The data layout transformation improves the prefetch effect of a cache-block. (2) FACT compresses recursive pointer and integer fields of the structure, and this compression further enhances the prefetch effect by enlarging the effective cache-block

size. (3) This compression also enlarges the effective cache capacity. Since FACT utilizes a novel data layout scheme for both the uncompressed data and the compressed data in memory and utilizes a novel form of addressing to reference the compressed data in the caches, it requires only slight modification to the conventional cache structure. Therefore FACT exceeds the limit of existing compression methods, which exhibit a compression ratio of $\frac{1}{2}$.

2 Field Array Compression Technique

The detailed steps of FACT are as follows: (1) We first take profile-runs to inspect the runtime values of the recursive pointer and integer fields, and we locate fields which contain values that are often compressible. These compressible fields are the targets of the compression. (2) By modifying the source code, we transform the data layout of the target structure to isolate and gather the compressible fields from different instances in the form of an array of fields. (3) We replace the load/store instructions which access the target fields with special instructions, which also compress/decompress the data (we call these instructions `cld/cst`). (4) During runtime, the `cld/cst` instructions carry out the compression/decompression using special hardware, as well as performing the normal load/store job. Since this method utilizes a field array, we call it the Field Array Compression Technique (FACT). The compressed data is handled in the same manner as the non-compressed data, and both reside in the same cache.

2.1 Compression of Structure Fields

To compress the recursive pointer fields, we replace the absolute address of a pointer with a relative address in units of the structure size, which can be represented using a narrower bit-width than the absolute address. Figure 1 illustrates the compression. Assume we are constructing a balanced binary tree in depth-first order using RDS (1). We use the custom memory allocator which arranges the instances in contiguous memory (2). Therefore we can replace the pointers with relative orders (1). The compression/decompression is done when writing/reading pointer fields. Assume the compression ratio is $1/R$. Note that since the difference between the addresses of two contiguous instances is 8 bytes due to the data layout transformation, the relative order is equal to the address difference of the two instances divided by 8, and we use this as the $64/R$-bit codeword. We use a special codeword to indicate incompressibility, and which is handled differently by the `cld` instruction if the difference is outside the range that can be expressed by a standard codeword. As to the integer field compression, FACT utilizes two methods [7]. As to the selection of the compression target fields in a program, we use profiling method, whose details are described in [7]. In addition, we choose one compression ratio and one integer compression method used in a program using this profiling.

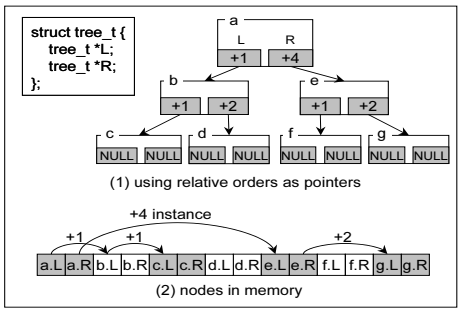

Fig. 1. Pointer compression.

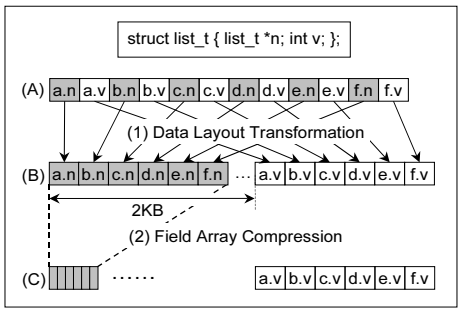

Fig. 2. Instance Interleaving (I2).

2.2 Data Layout Transformation for Compression

We transform the data layout of RDS to make it suitable for compression. The transformation is the same as Instance Interleaving (I2) proposed in [1]. We implement I2 by modifying the source code [7]. Since the compression shifts the position of the data, accessing the compressed data in the caches requires a memory instruction to translate the address for the uncompressed data into the address which points to the compressed data in the caches. When we use the different address space in the caches for the compressed data, the translation can be done by shrinking the address for the uncompressed data by the compression ratio. However, the processor must know the ratio which varies with the structure types. To solve this problem, we transform the data layout of RDS to isolate and group the compressible fields away from the incompressible fields. Assume as an example compressing a structure which has a compressible pointer `n` and an incompressible integer `v`. Figure 2 illustrates the isolation. Since field `n` is compressible, we group all the `n` fields from the different instances of the structure and make them contiguous in memory. We segregate `n` and `v` as arrays (B). Assume all the `n` fields are compressible at the compression ratio of $\frac{1}{8}$, which is often the case. Then we can compress the entire array of pointers (C). In addition, the address translation becomes a simple division-by-eight. We can also exploit temporal affinity between fields through the transformation. Assume two `n` pointers contiguous in memory have temporal affinity and each instance requires 16 bytes without I2. Using a 64-byte cache-block, 1 cache-block can hold 4 `n` pointers with temporal affinity (A). With I2, it can hold 8 of these pointers (B), and with compression at a ratio of $\frac{1}{8}$, it can hold 64 of these pointers (C). This transformation enhances the prefetch effect of a cache-block.

2.3 Address Translation for Compressed Data

Since we attempt to compress data which changes dynamically, we find it is not always compressible. Therefore we need area for both the compressed and uncompressed data. We allocate space for the both initially. This layout requires an address translation from the uncompressed data to the compressed data. We can calculate it using an affine transformation with the following steps: FACT uses a custom allocator, which allocates a memory block (for example, 2 KB) and

divides it into two for the compressed data and the uncompressed data. When using a compression ratio of $\frac{1}{8}$, it divides the total block into a $1:8$ ratio for the compressed data block and the uncompressed data block. This layout also provides the compressed data with spatial locality. However, this layout restricts the position of the compressed data, thus causing cache conflicts. Therefore we prepare new address space for the compressed data in the caches. We add a 1-bit tag to the caches to distinguish the address spaces. Figure 3 illustrates these address spaces. Consider the uncompressed data D and its physical address A (1), the compressed data d and its physical address a (3), and the compression ratio of $\frac{1}{R}$. We utilize $\frac{A}{R}$ in this new address space to point to d in the caches (2). We need only to shift A to get $\frac{A}{R}$. That is, `cld/cst` instructions that access D shift given address A into $\frac{A}{R}$ and access d in the caches using $\frac{A}{R}$ (X). When the compressed data needs to be written-back to or fetched from main memory, we translate address $\frac{A}{R}$ into address a (Y).

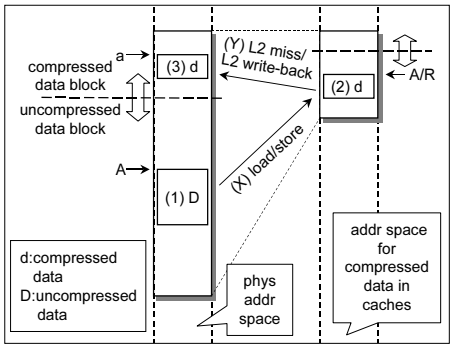

Fig. 3. Address translation in FACT.

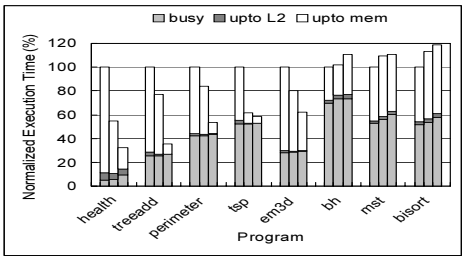

Fig. 4. Execution time comparison. In each group, each bar shows from the left, execution time of the baseline configuration, with I2, and with FACT, respectively.

2.4 Deployment and Action of `cld/cst` Instructions

FACT replaces the load/store instructions that access the compression target fields with `cld/cst` instructions. There are three types of `cld/cst` instructions, corresponding to the compression targets and methods, and we choose an appropriate type for each replacement. The `cst` instruction checks whether its data is compressible, and if it is, `cst` compresses it and puts it in the cache after the address translation which shrinks the address by the compression ratio. When `cst` encounters incompressible data, it stores the codeword indicating incompressibility, and then it stores the uncompressed data to the address before translation. The details of the operation of the `cst/cld` instruction are described in [7].

3 Evaluation Methodology, Results, and Discussions

We assume architecture employing FACT uses a superscalar 64-bit microprocessor, which uses 7-stage pipeline and whose load-to-use latency is 3 cycles. We assume the decompression performed by the `cld` instruction requires one additional cycle to the load-to-use latency. When `cst` and `cld` instructions handle

Table 2. Program used in the evaluation. Note that we modified `perimeter` and `bh` for simulation convenience [7]

Name	Input param. for evaluation	Inst. count	Cmp. ratio
health	lev 5, time 300	69.5M	1/4
treeadd	1M nodes	89.2M	1/8
perim.	16K×16K img	159M	1/8
tsp	64K cities	504M	1/8
em3d	32K nodes, 3D	213M	1/8
bh	4K bodies	565M	1/4
mst	1024 nodes	312M	1/4
bisort	256K integers	736M	1/4

Table 3. Dynamic memory access rate and compression success rate of compression target recursive pointers

Prog.	Access (%)			Success (%)		
ratio→	1/4	1/8	1/16	1/4	1/8	1/16
health	31.1	1.45	1.45	94.6	76.8	76.5
treeadd	11.6	11.6	11.5	100	98.9	96.5
perim.	17.6	17.5	17.6	99.8	95.9	85.6
tsp	10.2	10.2	10.2	100	96.0	67.1
em3d	.487	.487	.487	100	99.6	99.6
bh	1.56	1.56	.320	88.2	51.3	52.2
mst	5.32	5.32	0	100	28.7	0
bisort	43.0	41.2	41.0	90.8	65.6	59.2

incompressible data, they must access both the compressed data and the uncompressed data. We assume the penalty in this case is at least 4 cycles for `cst` and 6 cycles for `cld`. We assume other compression operations of `cld`/`cst` instructions do not require additional latency. We developed an execution-driven, cycle-level software simulator of a superscalar processor to evaluate FACT. Table 1 shows its parameters. We used 8 programs from the Olden benchmark [2], `health`, `treeadd`, `perimeter`, `tsp`, `em3d`, `bh`, `mst`, `bisort`. Table 2 shows characteristics and the compression ratio used. They make graphs using RDS as their nodes. Note that different input parameters are used for the profile-run and the evaluation-run. All programs were compiled using Compaq CC version 6.2-504 on Linux Alpha, using optimization "-O4".

Table 1. Simulation parameters

Fetch, Decode, Issue, Retire	up to 8 insts, 128-entry inst. window, 64-entry load/store queue, 256-entry ROB
Exec unit	4 INT, 4 LD/ST, 2 other INT, 2 FADD, 2 FMUL, 2 other FLOAT
L1 data cache	32 KB, 2-way, 64 B blk size 3-cycle load-to-use latency
L2 cache	256 KB, 4-way, 64 B blk size 13-cycle load-to-use latency
Memory	200-cycle load-to-use latency

First we show the dynamic memory accesses of the compression target pointers (A_{target}) normalized to the total dynamic accesses (access rate), and the dynamic accesses of compressible pointers normalized to A_{target} (success rate). Table 3 summarizes the results with various compression ratios. `treeadd`, `perim`, `em3d`, and `tsp` exhibit high success rates. This is because they organize the nodes in memory in the same order as the traversal. In these programs we can compress many pointers into a single byte. On the other hand, `bh`, `bisort`, `health`, and `mst` exhibit low success rates, because they organize the nodes in a different order to the traversal order. Figure 4 compares the execution times of the programs using the baseline configuration, with I2, and with FACT. Each component in the bar shows from the bottom, busy cycles other than stall cycles due to cache misses (busy), stall cycles due to accesses to the secondary cache (upto L2), and due to accesses to main memory (upto mem). FACT reduces the stall cycles due to cache misses by 41.6% on av-

erage. If the traversal order and the memory order of the nodes in the programs are the same, I2 can exploit temporal affinity between fields, which FACT can exploit further by compression. This is the case in `health`, `treeadd`, `perim`, and `em3d`. I2 distributes the fields within one data structure among multiple cache-blocks. It recovers this inefficiency by gathering the fields with temporal affinity in one cache-block. When the memory order and the traversal order of the nodes in the programs are different, I2 cannot recover this inefficiency thus lowering the utilization ratio of a cache-block, resulting in degradation of performance. This is the case in `bh`, `mst`, and `bisort`.

4 Summary

We proposed the Field Array Compression Technique (FACT) which reduces cache misses caused by recursive data structures. Through software simulation, we showed that FACT yields a 37.4% speedup and a 41.6% reduction of memory stall time on average. This paper has four main contributions. (1) FACT achieves the compression ratio of $\frac{1}{8}$. This ratio exceeds $\frac{1}{2}$, which is the limit of existing compression methods. (2) We represent that we can compress many recursive pointer fields into 8 bits. (3) We represent the notion of a split memory space, where we allocate one byte of compressed memory for every 8 bytes of uncompressed memory. Each uncompressed element is represented in the compressed space with a codeword placeholder. This provides compressed data with spatial locality and simplifies the address translation from uncompressed data address to compressed data address. (4) We represent the notion of a new address space for compressed data in the caches. which simplifies the addressing of compressed data in the caches and avoids cache conflict.

References

1. D. N. Truong, F. Bodin, and A. Seznec. Improving cache behavior of dynamically allocated data structures. *In Proc. of PACT*, pp. 322–329, Oct. 1998.
2. A. Rogers et al. Supporting dynamic data structures on distributed memory machines. *ACM TOPLAS*, 17(2):233–263, Mar. 1995.
3. A. Roth, A. Moshovos, and G. S. Sohi. Dependence based prefetching for linked data structures. *In Proc. of ASPLOS*, pp. 115–126, Oct. 1998.
4. J. Yang, Y. Zhang, and R. Gupta. Frequent value compression in data caches. *In Proc. of MICRO*, pp. 258–265, Dec. 2000.
5. S. Y. Larin. Exploiting program redundancy to improve performance, cost and power consumption in embedded systems. Ph. D. Thesis, ECE Dept., North Carolina State Univ., Raleigh, North Carolina, Aug. 2000.
6. Y. Zhang et al. Data compression transformations for dynamically allocated data structures. *In Proc. of Int. Conf on CC*, LNCS 2304, pp. 14–28, Apr. 2002.
7. M. Takagi et al. Compression in data caches with data layout transformation for recursive data structures. TR03-01, Dept. of CS, Univ. of Tokyo, May 2003.