Field Array Compression in Data Caches for Dynamically Allocated 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 caused by recursive data structures. Using a data layout transformation, data with temporal affinity are gathered in contiguous memory, where recursive pointer and integer fields are compressed. As a result, one cache-block can capture a greater amount of data with temporal affinity, especially pointers, thereby improving the prefetching effect. In addition, the compression enlarges the effective cache capacity. On a suite of pointer-intensive programs, FACT achieves a 41.6% average reduction in memory stall time and a 37.4% average increase in speed.

1 Introduction

Non-numeric programs often use recursive data structures (RDS). For example, they are used to represent variable-length object-lists and trees for data repositories. Such programs using RDS make graphs and traverse them, however the traversal code often induces cache misses because: (1) there are too many nodes in the graphs to fit entirely in the caches, and (2) the data layout of the nodes in the caches is not efficient. One technique for reducing these misses is data prefetching [7, 8]. Another technique is data layout transformations, where data with temporal affinity are gathered in contiguous memory to improve the prefetch effect of a cache-block [3, 15]. Still another technique is enlargement of the cache capacity, however this has limitations caused by increased access time. Yet another technique is data compression in caches, which compresses data stored in the caches to enlarge their *effective* capacity [9, 11–13]. Not only does compression enlarge the effective cache capacity, it also increases the effective cache-block size. Therefore, applying it with a data layout transformation can produce a synergistic effect that further enhances the prefetch effect of a cache-block. This combined method can be complementary to data prefetching. We can compress data into 1/8 or less of its original size in some programs, but existing compression methods in data caches limit the compression ratio to 1/2, mainly because of hardware complexity in the cache structure. Therefore, we propose a method which achieves a compression ratio over 1/2 to make better use of the data layout transformation.

In this paper, we propose a compression method which we call the Field Array Compression Technique (FACT). FACT utilizes the combined method of a data layout transformation along with recursive pointer and integer field compression. It enhances the prefetch effect of a cache-block and enlarges the effective cache capacity, leading to a reduction in cache misses caused by RDS. FACT requires only slight modification to the

conventional cache structure because it utilizes a novel data layout scheme for both the uncompressed data and the compressed data in memory; it also utilizes a novel form of addressing to reference the compressed data in the caches. As a result FACT surpasses the limits of existing compression methods, which exhibit a compression ratio of 1/2. The remainder of this paper is organized as follows: Section 2 presents related works, and Sect. 3 explains FACT in detail. Section 4 describes our evaluation methodology. We present and discuss our results in Sect. 5 and give conclusions in Sect. 6.

2 Related Works

Several studies have proposed varying techniques for data compression in caches. Yang et al. proposed a hardware method [11], which compresses a single cache-block and puts the result into the primary cache. Larin et al. proposed another method [12], which uses Huffman coding. Lee et al. proposed yet another method [9], which puts the results into the secondary cache. These three techniques do not require source code modification. Assuming the compression ratio is $1/R$, these three methods must check R address-tags on accessing the compressed data in the caches because they use the address for the uncompressed data to point to the compressed data in the caches. These methods avoid adding significant hardware to the cache structure by limiting the compression ratio to 1/2. FACT solves this problem by using a novel addressing scheme to point to the compressed data in the caches.

Zhang et al. proposed a hardware and software method for compressing dynamically allocated data structures (DADS) [13]. Their method allocates word-sized slots for the compressed data within the data structure. It finds pointer and integer fields which are compressible with a ratio of 1/2 and produces a pair from them to put into the slot. Each slot must occupy one-word because of word alignment requirement; in addition, each slot must gather fields from a single instance. These conditions limit the compression ratio of this method. FACT solves these problems by using a data layout transformation, which isolates and groups compressible fields from different instances.

Truong et al. proposed a data layout transformation for DADS, which they call Instance Interleaving [3]. It modifies the source code of the program to gather data with temporal affinity in contiguous memory. This transformation enhances the prefetch performance of a cache-block. Compression after the transformation can improve the prefetch performance further.In addition, the transformation can isolate and group the compressible fields. Therefore we apply this method before compression.

Rabbah et al. automated the transformation using the compiler [15]. They proposed several techniques using compiler analysis to circumvent the problems caused by the transformation. We consider automating our compression processes as future work and we can utilize their method for the data layout transformation step.

3 Field Array Compression Technique

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. First, FACT transforms the data layout of the structure such that fields with temporal affinity are contiguous in memory. This transformation improves the prefetch performance of a cache-block. Second, FACT compresses recursive pointer and integer fields of the structure. This compression further enhances the prefetch performance by enlarging the effective cache-block size. It also enlarges the effective cache capacity.

FACT uses a combined hardware/software method. The steps are as follows: **(1)** We take profile-runs to inspect the runtime values of the recursive pointer and integer

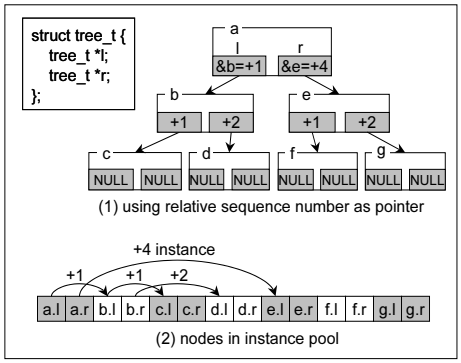

Fig. 1. Pointer compression

STDATA	Data to be stored
BASE	Base address
OFFSET	Address offset
1/R	Compression ratio
INCMP	Codeword indicating incompressibility

```
/* operation of cst instruction */
cst(STDATA, OFFSET, BASE) {
   PA = physical address of (BASE + OFFSET)
   CA = PA/R
   CDATA = call compress_ptr(STDATA, BASE)
         or its family according to type of cst
   cache_write(CA, CDATA, C)
   if(CDATA == INCMP) {
      /* STDATA is incompressible */
      cache_write(PA, STDATA, N)
   }
}
```

Fig. 2. Compression algorithm of pointer fields

fields, and we locate fields which contain values that are often compressible (we call these compressible fields). 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 of the structure in the form of an array of fields. This modification step is done manually in the current implementation. **(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`.) There are three types of `cld/cst` instructions corresponding to the compression targets and methods; we choose an appropriate type for each replacement. **(4)** At runtime, the `cld/cst` instructions perform compression/decompression using special hardware, as well as performing the normal load/store job.

We call this method the Field Array Compression Technique (FACT) because it utilizes a field array. The compressed data are handled in the same manner as the non-compressed data and both reside in the same cache. Aassuming a commonly-used two-level cache hierarchy, the compressed data reside in the primary data cache and the secondary unified cache. In the following, we describe details of the steps of FACT.

3.1 Compression of Structure Fields

We compress recursive pointer fields and integer fields because they often have exploitable redundancy.

Recursive Pointer Field Compression While we use RDS for dynamic allocation of instances in different memory locations on demand, elaborate and fast memory allocators create pools of instances for fast management of free objects and to exploit spatial locality [1]. For example, on an allocation request, the slab allocator [1] allocates a pool of instances if a free instance is not available; it then returns one instance from the pool. In addition, a graph created by RDS often consists of small sub-graphs each of which fits in a pool (e.g. binary tree). Therefore the distance between memory addresses of two structures connected by a pointer can often be small. In addition, one recursive pointer in the structure points to another instance of the same structure; moreover, the memory allocators using the instance pools can align the instances. Therefore we can replace the absolute address of a structure with a relative address in units of the structure size, which can be represented using a narrower bit-width than the absolute address. We abbreviate this method to **RPC**. The detailed steps of RPC are as follows: (i) We make a custom memory allocator for the structure, similar to

[1], using instance pools as described above. (ii) We modify the source code to make it use the allocator. (iii) Using this layout, the `cst` instruction replaces the pointers at runtime with the relative sequence numbers in the pool. Figure 1 illustrates the compression. Assume we are constructing a balanced binary tree in depth-first order using RDS (1). When we use the memory allocator which manages the instance pool, the instances are arranged contiguously in memory (2). Therefore we can replace the pointers with relative sequence numbers in the pool (1).

Figure 2 shows the algorithm of RPC. The compression is done when writing the pointer field. Assume the head address of the structure which holds the pointer is `BASE`, the pointer to be stored is `STDATA`, and the compression ratio is $1/R$. Because the difference between the addresses of two neighboring instances is 8 bytes (pointer size) as a result of the data layout transformation described in Sect. 3.3, the relative sequence number in the pool is `(STDATA-BASE)/8`; we use this as the 64/R-bit codeword. A `NULL` pointer is represented by a special codeword. We use another 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. The address `BASE` can be obtained from the base-address of the `cst` instruction. Decompression is performed when reading pointer fields. Assume the address of the head of the structure which contains the pointer is `BASE` and the compressed pointer is `LDDATA`. The decompression calculates `BASE+LDDATA`$\times$`8`, where `BASE` can be obtained from the base-address of the `cld` instruction.

Integer Field Compression The integer fields often have exploitable redundancy. For example, there are fields which take only a small number of distinct values [10] or which use a narrower bit-width than is available [14]. FACT exploits these characteristics by utilizing two methods. In the first method, which we abbreviate to **DIC**, we locate 32-bit integer fields which take only a small number of distinct values over an entire run of the program, then compress them using fixed-length codewords and a static dictionary [11]. When constructing the N-entry dictionary, we gather statistics of the values accessed by all of the load/store instructions through the profile-run, and take the most frequently accessed N values as the dictionary. These values are passed to the hardware dictionary through a memory-mapped interface or a special register at the beginning of the program. In our evaluation, this overhead is not taken into account. The compression is done when writing the integer field. Assume the compression ratio is $1/R$. The `cst` instruction searches the dictionary, and if it finds the required data item, it uses the entry number as the 32/R-bit codeword. If the data item is not found, the codeword indicating incompressibility is used. The decompression is done when reading the integer field. The `cld` instruction reads the dictionary with the compressed data. In the second method, which we abbreviate to **NIC**, we locate 32-bit integer fields which use a narrower bit-width than is available, and replace them with narrower bit-width integers. On compression, the `cst` instruction checks the bit-width of the data to be stored, and it omits the upper bits if it can. On decompression, the compressed data are sign-extended. While DIC includes NIC, DIC needs access to the hardware dictionary on decompression. Since a larger dictionary requires greater time, we use DIC when the dictionary contains less than or equal to 16 entries, otherwise we use NIC. That is, when we choose a compression ratio of 1/8 or more in the entire program, we use DIC for the entire program; otherwise we use NIC.

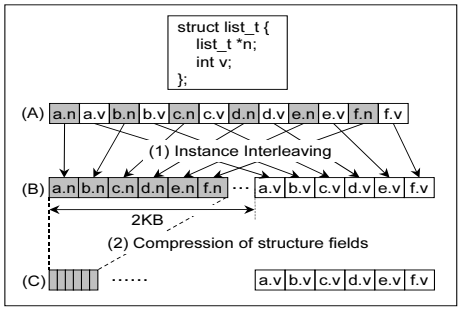

Fig. 3. Instance Interleaving (I2) and FACT

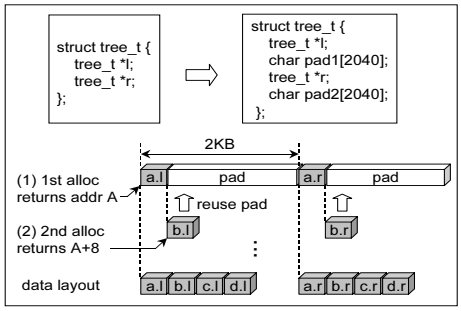

Fig. 4. I2 using a padded structure

3.2 Selection of Compression Target Field

In FACT, we locate the compressible fields of RDS in the program. We gather runtime statistics of the load/store instructions through profiling because the compressibility depends on the values in the fields that are determined dynamically. The profile-run takes different input parameters to the actual run. We collect the total access count of the memory instructions (A_{total}), the access count of fields for each static instruction (A_i) and the occurrence count of compressible data for each static instruction (O_i). Subsequently, we mark the static instructions that have A_i/A_{total} (access rate) greater than X and O_i/A_i (compressible rate) greater than Y. The data structures accessed by the marked instructions are the compression targets. We set X=0.1% and Y=90% empirically in the current implementation. We take multiple profile-runs that use different compression ratios. Taking three profile-runs with compression ratios of 1/4, 1/8, and 1/16, we select one compression ratio to be used based on the compressibility numbers shown. This selection is done manually and empirically in the current implementation.

3.3 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 by Truong et al. [3].

Isolation and Gathering of Compressible Fields through I2 FACT compresses recursive pointer and integer fields in RDS. 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. Assuming the address of the instance is I and the compression ratio is $1/R$, the translated address is I/R. However, the processor must know the value of R which varies with 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 3 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 fill one segment with `n` and the next segment with `v`, segregating them as arrays (B). Assume

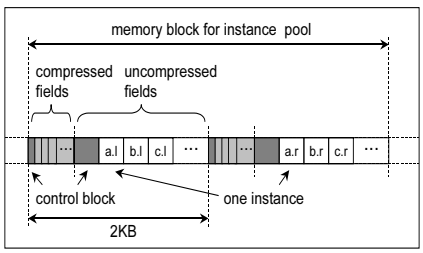

Fig. 5. Internal organization of one arena

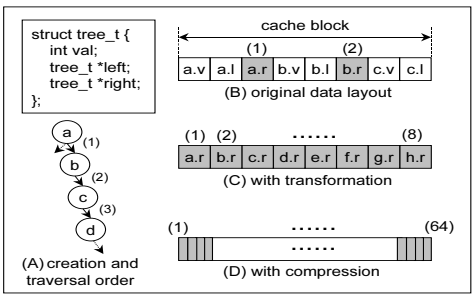

Fig. 6. I2 and FACT can exploit temporal affinity

all the `n` fields are compressible at the compression ratio of 1/8, which is often the case. Then we can compress the entire array of pointers (C). In addition, the address translation becomes $I/8$, which can be done by a simple logical-shift operation.

Implementation of I2 We transform the data layout by modifying the source code. First, we modify the declaration of the structure to insert pads between fields. Figure 4 shows the declaration of example structure for binary tree before and after the modification. Assume the load/store instructions use addressing with a 64-bit base address register and a signed 16-bit immediate offset. Compilers set the base-address to the head of the structure when accessing a field with an offset of less than 32KB, and we use this property in the pointer compression. Because the insertion of pads makes the address of the n-th field (structure head address)+(pad size)$\times(n-1)$, we limit the pad size to 2KB so that the compiler can set the base address to the structure head when referencing the first 16 fields. Figure 4 illustrates the allocation steps using this padded structure. When allocating the structure for the first time, a padded structure is created, and the head address (assume it is A) is returned (1). On the next allocation request it returns the address $A + 8$ (2) to reuse the pad with fields of the second instance. Second, we make a custom memory allocator to achieve this allocation, and we modify the memory allocation part of the source code to use it. The allocator is similar to that used in [3]. It allocates a memory block for the instance pool, which we call the arena as in [2]. Figure 5 illustrates the internal organization of one arena. The custom allocator takes the structure type ID as the argument and manages one arena type per structure type, which is similar to [1]. Since one arena can hold only a few hundred instances, additional arenas are allocated on demand. The allocator divides the arena by the compression ratio into compressed and uncompressed memory.

Problems of I2 The manual I2 implementation using padding violates C semantics. There are harmful cases: (1) copying an instance of the structure causes copying of other neighboring instances; (2) pointer arithmetic causes a problem because `sizeof` for the transformed data structure returns a value which does not match the offset to the neighboring instance; in addition, `sizeof` for one field does not match the offset to the neighboring field within one instance. However, (1) and (2) rarely occurs for programs using RDS. In cases where we find these problems in the current implementation, we simply stop applying the transformation. Alternatively, we can use a compiler to automate the transformation and circumvent these problems. Rabbah et al. studied this compiler implementation [15]. In the level the type information is

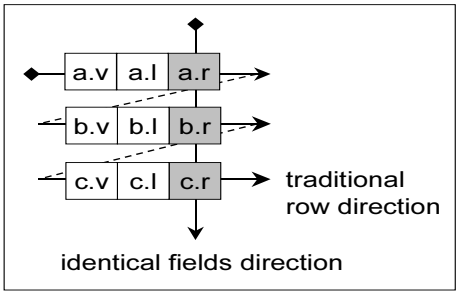

Fig. 7. Instances seen as a matrix and two directions in which we can exploit temporal affinity

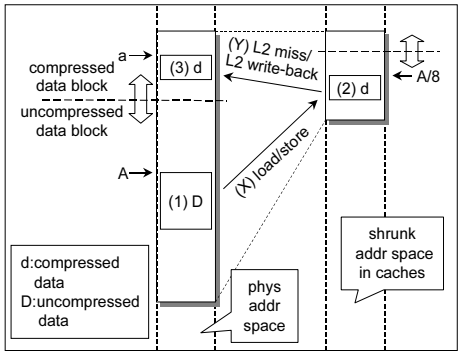

Fig. 8. Address translations in FACT

available, their compiler can virtually pad target structures by modifying the address calculation function. In addition, it can modify memory allocation part of the program. Their technique modifies instance copy functions to circumvent case (1), and recognize and handle typical pointer arithmetic expressions in case (2) such as `p+c` and `(field_type*)((long)p+c)` (assume `p` is a pointer to a structure and `c` is a constant.). There is another problem: (3) the precompiled library cannot handle a transformed data structure. Rabbah et al. suggested that they need to access the entire library source code, or they need to transform an instance back to the original form before they pass it to a library function and transform it again after the function call.

3.4 Exploiting Temporal Affinity through I2 and FACT

We can exploit temporal affinity between fields through I2 and FACT. Figure 6 illustrates an example. Consider the binary tree in the program `treeadd`, which is used in the evaluation. The figure shows the declaration of the structure. After declaration of `val`, the structure has a 4-byte pad to create an 8-byte alignment. `treeadd` creates a binary tree in depth-first order, which follows the `right` edges first (A). Therefore the nodes of the tree are also organized in depth-first order in memory (B). After making the tree, `treeadd` traverses it in the same order using a recursive call. Therefore two `right` pointers that are contiguous in memory have temporal affinity. Using a 64-byte cache-block, 1 cache-block can hold two `right` pointers with temporal affinity (B) because each instance requires 24 bytes without I2. With I2, it can hold 8 of these pointers (C). With compression at a ratio of 1/8, it can hold 64 of these pointers (D). In this way, I2 and FACT enhance the prefetch effect of a cache-block.

The above example is an ideal case. In general there are two main conditions that programs must meet so that I2 and FACT can exploit temporal affinity. The main data structures used in the programs using RDS are graph structures. These programs make graphs using instances of RDS as the graph nodes and then traverse these instances. We can see a group of these instances as a matrix where one instance corresponds to one row and the fields with the same name correspond to one column. This matrix is shown in Fig. 7. While traditional data layout exploits temporal affinity between fields in the row direction, I2 and FACT exploit temporal affinity between fields in the column direction. Therefore, the first condition is: **(C1)** there exists greater temporal affinity between fields in the column direction (e.g. recursive pointers of instances) than between fields in the traditional row direction. In addition, we need high utilization

in this direction. Therefore, the second condition is: **(C2)** in a short period, a single traverse or multiple traverses utilize many nodes which are close in memory. When a program does not meet either condition, the traditional data layout is often more effective and I2 might degrade the performance.

3.5 Address Translation for Compressed Data

Because we attempt to compress data which changes dynamically, we find it is not always compressible. When we find incompressible data, space for storing the uncompressed data is required. There are two major allocation strategies for handling this situation. The first allocates space for only the compressed data initially, and when incompressible data is encountered, additional space is allocated for the uncompressed data [13]. The second allocates space for both the compressed and uncompressed data initially. Since the first approach makes the address relationships between the two kinds of data complex, FACT utilizes the second approach. While the second strategy still 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 memory blocks and divides each into two for the compressed data and the uncompressed data. When using a compression ratio of 1/8, it divides each block into the compressed data block and the uncompressed data block with a 1 : 8 ratio. This layout also provides the compressed data with spatial locality, as the compressed block is a reduced mirror image of the uncompressed block.

Consider the compressed data d and their physical address a, the uncompressed data D and their physical address A, and the compression ratio of $1/R$. When we use a to point to d in the caches, the compressed blocks only occupy $1/(R+1)$ of the area of the cache. On the other hand, when we use A to point to D in the caches, the uncompressed blocks occupy $R/(R+1)$ of the area. Therefore we prepare another address space for the compressed data in the caches and use A/R to point to d. We call the new address space the shrunk address space. We need only to shift A to get A/R, and we add a 1-bit tag to the caches to distinguish the address spaces. In the case of the write-back to and the fetch from main memory, since we need a to point to d, we translate A/R into a. This translation can be done by calculation, which although requires additional latency, slows execution time by no more than 1% in all of the evaluated programs. Figure 8 illustrates the translation steps. Assume we compress data at physical address $A(1)$ at the compression ratio of 1/8. The compressed data are stored at the physical address $a(3)$. Then `cld/cst` instructions access the compressed data in the caches using the address $A/8(\mathrm{X})(2)$. When the compressed data need to be written-back to or fetched from main memory, we translate address $A/8$ into $a(\mathrm{Y})(3)$.

3.6 Deployment and Action of `cld/cst` Instructions

FACT replaces the load/store instructions that access the compression target fields with `cld/cst` instructions. They perform the compress/decompress operations as well as the normal load/store tasks at runtime. Since we use three types of compression we have three types of `cld/cst` instructions. We choose for each replacement one type depending on the target field type and the compression method chosen. We use only one integer compression method in the entire program as described in Sect. 3.1.

Figure 9 shows the operation of the `cst` instruction, and Fig. 10 shows its operation in the cache. In the figures, we assume a cache hierarchy of one level for simplicity. We

STDATA	Data to be stored
BASE	Base address
OFFSET	Address offset
1/R	Compression ratio
INCMP	Codeword indicating incompressibility

```
/* operation of cst instruction */
cst(STDATA, OFFSET, BASE) {
   PA = physical address of (BASE + OFFSET)
   CDATA = call compress_ptr(STDATA, BASE)
         or its family according to type of cst
   CA = PA/R
   cache_write(CA, CDATA, C)
   if(CDATA == INCMP) {
      /* STDATA is incompressible */
      cache_write(PA, STDATA, N)
   }
}
```

Fig. 9. Operation of `cst` instruction

ADDR	Address in cache
DATA	Data to be stored
FLAG	C:compressed data N:uncompressed data

```
/* operation of cst instruction in cache */
cache_write(ADDR, DATA, FLAG) {
   if(cache-miss on {ADDR, FLAG}) {
      if(FLAG == C) {
         PA = calculate address of compressed data
              in main memory from ADDR
         access main memory with address PA
              and cache-fill
      } else {
         access main memory with address ADDR
              and cache-fill
      }
   }
   store DATA using {ADDR, FLAG}
}
```

Fig. 10. Operation of `cst` inst. in the cache

BASE	Base address
OFFSET	Address offset
1/R	Compression ratio
INCMP	Codeword indicating incompressibility

```
/* operation of cld instruction */
cld(OFFSET, BASE) {
   PA = physical address of (BASE + OFFSET)
   CA = PA/R
   MDATA = cache_read(CA, C)
   if(MDATA != INCMP) {
      DST = decode MDATA according to type of cld
   } else {
      DST = cache_read(PA, N)
   }
   return DST
}
```

Fig. 11. Operation of `cld` instruction

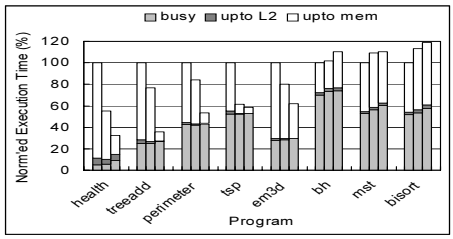

Fig. 12. Execution time of the programs with I2 and FACT. In each group, each bar shows from the left, execution time of the baseline configuration, with I2, and with FACT, respectively

also assume a physically tagged, physically indexed cache. The `cst` instruction checks whether its data are compressible, and if they are, it compresses them and puts them 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. When the `cst` instruction misses the compressed data in all the caches, main memory is accessed after the second address translation, which translates the address of the compressed data in the caches to the address of the compressed data in main memory. Figure 11 shows the operation of the `cld` instruction. When the `cld` instruction accesses compressed data in the caches, it accesses the caches after the address translation and decompresses it. The translation shrinks the address by the compression ratio. When `cld` fetches compressed data from the cache and finds they are incompressible, it accesses the uncompressed data using the address before the translation. When `cld` misses the compressed data in the caches, it accesses main memory in a similar way to the `cst` instruction. Since the compressed and uncompressed memory can be inconsistent, we must replace all load/store instructions accessing the compression targets with `cld/cst` instructions so that the compressed memory is checked first.

4 Evaluation Methodology

We assume the architecture employing FACT uses a superscalar 64-bit microprocessor. For integer instructions, the pipeline consists of the following seven stages: instruction cache access 1, instruction cache access 2, decode/rename, schedule (SCH),

Table 1. Simulation parameters for processor and memory hierarchy

Fetch	fetch up to 8 insts, 32-entry inst. queue
Branch pred.	16K-entry GSHARE, 256-entry 4-way BTB, 16-entry RAS
Decode/Issue	decode/issue up to 8 insts, 128-entry inst. window
Exec. unit	4 INT, 4 LD/ST, 2 other INT, 2 FADD, 2 FMUL, 2 other FLOAT, 64-entry load/store queue, 16-entry write buf, 16 MSHRs, oracle resolution of load–store addr. dependency
Retire	retire up to 8 insts, 256-entry reorder buffer
L1 cache	inst: 32KB, 2-way, 64B block size data: 32KB, 2-way, 64B block size, 3-cycle load-to-use latency
L2 cache	256KB, 4-way, 64B block size, 13-cycle load-to-use latency, on-chip
Main mem.	200-cycle load-to-use latency
TLB	256-ent, 4-way inst TLB and data TLB, pipelined, 50-cycle h/w miss handler

register-read (REG), execution (EXE), and write-back/commit (WB). For load/store instructions, four stages follow REG stage: address-generation (AGN), TLB-access (TLB)/data cache access 1 (DC1), data cache access 2 (DC2), and WB. Therefore the load-to-use latency is 3 cycles. We assume the `cst` instruction can perform the compression calculation in its AGN and TLB stages; therefore no additional latency is required. 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 incompressible data, they must access both the compressed data and the uncompressed data. In this case, the pipeline stages for the first attempt to access the compressed data are wasted. Therefore we assume the penalty in this case is at least 4 cycles for `cst` (SCH, REG, data compression 1, and data compression 2) and 6 cycles for `cld` (SCH, REG, AGN, DC1, DC2, and decompression). When `cld`/`cst` instructions access the compressed data, they shrink their addresses. We assume this is done within their AGN or TLB stage; therefore no additional latency is required. We developed an execution-driven, cycle-accurate software simulator of a processor to evaluate FACT. Contentions on the caches and buses are modeled. Table 1 shows its parameters. We set the instruction latency and the issue rate to be the same as the Alpha 21264 [5].

We used 8 programs from the Olden benchmark [4], `health (hea)`, `treeadd (tre)`, `perimeter (per)`, `tsp`, `em3d`, `bh`, `mst`, and `bisort (bis)`, because they use RDS and they have a high rate of stall cycles caused by cache misses. Table 2 summarizes thier characteristics. All programs were compiled using Compaq C Compiler version 6.2-504 on Linux Alpha, using optimization option "-O4". We implemented a custom allocator using the instance pool technique, and modified the source codes of the programs to use it when evaluating the baseline configuration. This allocator is similar to [1, 2]. We did so to distinguish any improvement in execution speed caused by FACT from improvements resulting from the use of the instance pool technique, since apply-

Table 2. Program used in the evalutaion: input parameters for profile-run and evaluation-run, max. memory dynamically allocated, instruction count, The 4th and 5th columns show the numbers for the baseline configuration

Name	Input param. for profile-run	Input param. for evaluation	Max dyn. mem	Inst. count
hea	lev 5, time 50	lev 5, time 300	2.58MB	69.5M
tre	4K nodes	1M nodes	25.3MB	89.2M
per	128×128 img	16K×16K img[1]	19.0MB	159M
tsp	256 cities	64K cities	7.43MB	504M
em3d	1K nodes, 3D	32K nodes, 3D	12.4MB	213M
bh	256 bodies	4K bodies[2]	.909MB	565M
mst	256 nodes	1024 nodes	27.5MB	312M
bis	4K integers	256K integers	6.35MB	736M

[1] We modified `perimeter` to use a 16K×16K image instead of 4K×4K.

[2] The iteration number of `bh` was modified from 10 to 4.

Table 3. Access rate and compression success rate for compression target fields. Left table shows numbers for RPC; Middle for NIC; Right for DIC

Prog.	Access(%)			Success(%)		
ratio	1/4	1/8	1/16	1/4	1/8	1/16
hea	31.1	1.45	1.45	94.6	76.8	76.5
tre	11.6	11.6	11.5	100	98.9	96.5
per	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
bis	43.0	41.2	41.0	90.8	65.6	59.2

Access(%)			Success(%)		
1/4	1/8	1/16	1/4	1/8	1/16
24.2	1.51	.677	83.7	88.6	93.7
5.80	5.78	5.77	100	100	100
12.4	12.4	4.33	100	100	83.1
.107	.107	.107	99.2	87.5	50.0
1.54	1.54	.650	100	99.1	68.8
.0111	.0111	.0111	100	100	100
0	0	0	0	0	0
27.8	0	0	100	0	0

Access(%)			Success(%)		
1/4	1/8	1/16	1/4	1/8	1/16
24.3	24.1	.827	46.9	18.9	90.3
5.80	5.78	5.77	100	100	100
12.4	12.4	12.4	100	100	91.0
.107	.107	.107	50.0	50.0	50.0
1.54	1.54	1.14	100	100	72.6
.0176	.0111	.0111	100	100	100
6.08	0	0	29.8	0	0
27.8	0	0	100	0	0

ing this allocator speeds up the programs. Similarly, when evaluating the configuration with I2, we apply a custom allocator which implements the data layout transformation but does not allocate memory blocks for the compressed data.

5 Results and Discussions

Compressibility of Fields FACT uses three types of compression; RPC, NIC, and DIC. For each method, we show the dynamic memory accesses of the compression target fields (A_{target}) normalized to the total dynamic accesses (access rate), and the dynamic accesses of compressible data normalized to A_{target} (success rate). We use compression ratios of 1/4, 1/8, and 1/16. In these cases, 64-bit pointers (32-bit integers) are compressed into 16, 8, and 4 bits (8, 4, and 2 bits) respectively. Table 3 summarizes the results. Note that since the input parameters for the profile-run and the evaluation-run are different, success rates under 90% exist. The main data structures used in the programs are graph structures. With respect to pointer compression, `tre`, `per`, `em3d`, and `tsp` exhibit high success rates. This is because they organize the nodes in memory in a similar order as the traversal. In these programs we can compress many pointers into single bytes. On the other hand, `bh`, `bis`, `hea`, and `mst` exhibit low success rates, because they organize the nodes in a considerably different order to the traversal order, or because they change the graph structure quite frequently. With respect to integer compression, because `tre`, `per`, `tsp`, `em3d`, `bh`, and `bis` have narrow bit-width integer fields or enumeration type fields, they exhibit high success rates. Among them, `tre`, `per` and `bis` also exhibit high access rates. In the following, we use a compression ratio of 1/8 for `tre`, `per`, `em3d`, and `tsp`, and of 1/4 for `hea`, `mst`, `bh`, and `bis`.

Effect of Each Compression Method FACT uses three types of compression, therefore we show the individual effect of each, and the effect of combinations of the

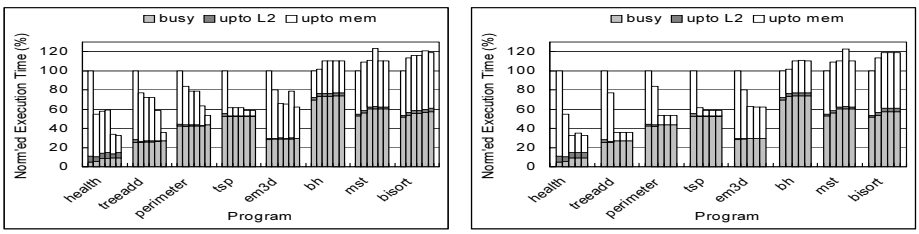

Fig. 13. Execution time of the programs. In each group, each bar shows from the left; Left graph: execution time of the baseline configuration, with I2, with NIC, with DIC, with RPC, and with FACT, respectively; Right graph: execution time of the baseline configuration, with I2, with NIC and RPC, with DIC and RPC, with FACT, respectively

three methods. The left side of Fig. 13 shows the results of applying each compression method alone. Each component in the bar shows from the bottom, busy cycles (busy) other than stall cycles waiting for memory data because of cache misses, stall cycles caused by accesses to the secondary cache (upto L2), and stall cycles caused by accesses to main memory (upto mem). All bars are normalized to the baseline configuration. First we compare NIC/DIC with RPC. In `hea`, `tre`, `per`, and `tsp`, RPC is more effective. This is because the critical path which follows the pointers does not depend on integer fields. On the other hand, in `em3d`, NIC/DIC is more effective. This is because the critical path depends on the integer fields, and there are more compressible integer fields than compressible pointer fields. Second we compare NIC with DIC. In `hea` and `mst`, NIC shows more speedup than DIC. This is because values not seen in the profile-run are used in the evaluation run. In other programs, the two methods exhibit almost the same performance. In `bh`, `mst`, and `bis`, no compression method reduces memory stall cycles. This will be explained in Sect. 5. The right side of Fig. 13 shows the results using a combination of pointer field compression and integer field compression. This combination leads to greater performance than either method alone in all programs except `bh`, `mst`, and `bis`. In addition, FACT is the best performing method in all programs except `bh`, `mst`, and `bis`.

Execution Time Results of FACT Figure 12 compares execution times of programs using I2 and FACT. As the figure shows, FACT reduces the stall cycles waiting for memory data by 41.6% on average, while I2 alone reduces them by 23.0% on average. `hea`, `tre`, `per`, `tsp`, and `em3d` meet both conditions (C1) and (C2) in Sect. 3.4. Therefore, both I2 and FACT can reduce the memory stall cycles of these programs, with the latter reducing them more. FACT reduces most of the memory stall cycles which I2 leaves in `tre` and `per` because the whole structure can be compressed into 1/8 of its original size. `bh` builds an oct-tree and traverse it in depth-first order, but it skips the traversal of child nodes often. In addition, nodes are organized in a random order in memory. `mst` manages hashes using singly linked-lists and traverses multiple lists. The length of node-chain which one traversal accesses is two on average. In addition, nodes of each list are distributed in memory. `bis` changes the graph structure frequently. Therefore these programs do not satisfy condition (C2) and thus I2 is less efficient than the traditional data layout. In this case, (T1) I2 increases the memory stall cycles; (T2) in addition, it increases the TLB misses, which results in the increase in the busy cycles. These effects can be seen for these programs. These programs have many incompressible fields. A processor using FACT must access uncompressed data along with compressed data when it finds these fields. In this case, (S1) this additional access increases the busy cycles; (S2) it can increase the cache conflicts and cache misses, which results in the increase in the memory stall cycles. These effects further slow down these programs. Note that (S2) accounts for the increase in the busy cycles in `hea` because `hea` also has many incompressible fields.

Decompression Latency Sensitivity We compare the execution time of I2 while varying the cycles taken by the compression operation of a `cst` instruction (L_c) and those taken by the decompression operation of a `cld` instruction (L_d). Note that up to this point, we have assumed that $(L_c, L_d)=(2, 1)$. In addition, note that two cycles

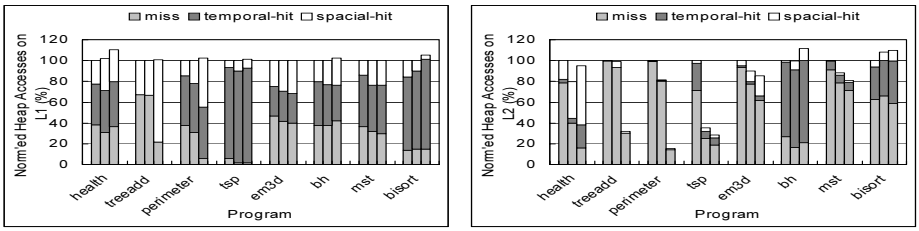

Fig. 14. Breakdown of accesses to the data caches, which refers to heap data. In each group, each bar shows from the left, the breakdown of accesses of the baseline configuration, with I2, and with FACT, respectively. Left: primary cache. Right: secondary cache

of L_c are hidden by the address-generation stage and the TLB-access stage of the `cst` instruction, but no cycle of L_d is hidden. The increase in the execution time of the cases $(L_c, L_d) = (8, 4), (16, 8)$ over the case $(L_c, L_d) = (2, 1)$ is 3.95% and 11.4% on average, respectively. FACT is relatively insensitive up to the case $(L_c, L_d) = (8, 4)$ because an out-of-order issue window can tolerate additional latency of a few cycles.

Prefetch Effect vs. Capacity Effect Both I2 and FACT reduce cache misses in two distinct ways: they increase the prefetch effect of a cache-block and they increase the reuse count of a cache-block. To show the breakdown of these effects, we use two cache-access events, which correspond to the two effects. We call them **spatial-hit** and **temporal-hit** and define them as follows: In the software simulator, we maintain access footprints for each cache-block residing in the caches. Every cache access leaves a footprint on the position of the accessed word. The footprints of a cache-block are reset on a cache-fill into a state which indicates that only the one word that caused the fill is accessed. When the same data block resides in both the primary and the secondary caches, cache accesses to the block in the primary cache also leave footprints on the cache-block in the secondary cache. We define the cache-hit on a word without a footprint as a spatial-hit and on a word with a footprint as a temporal-hit. Details of the relationship between temporal-hits and spatial-hits and the effects of I2/FACT are as follows: As is described in Sect. 3.4, I2 enhances the prefetch effect of a cache-block and increases the utilization ratio of a cache-block. The former reduces cache-misses and increases spatial-hits (we call this effect the prefetch effect). The latter reduces the number of cache-blocks used in a certain period of time, which increases temporal-hits. FACT enhances the prefetch effect further by compression, which increases spatial-hits (large-cache-block effect). In addition, compression reduces the number of cache-blocks used, which increases temporal-hits (compression effect).

We observe cache accesses to the heap because I2 and FACT handle heap data. Figure 14 shows the results. First we compare I2 with the baseline. In all programs except `bis`, misses decrease and spatial-hits increase in either or both cache levels. In `bh`, temporal-hits also increase in the secondary cache. In summary, I2 mainly increases the number of spatial-hits, therefore its main effect is the prefetch effect. Second we compare FACT with I2 with respect to the compression of FACT. In `hea`, `tre`, `per`, `em3d`, and `bis`, misses decrease and spatial-hits increase in either or both cache levels. In `hea`, `per`, `tsp`, `em3d`, `mst`, and `bis`, temporal-hits increase in either cache levels. The spatial-hits increase have the larger impact. In summary, the compression of FACT exhibits both the large-cache-block effect and the compression effect, with the large-cache-block effect having the greater impact.

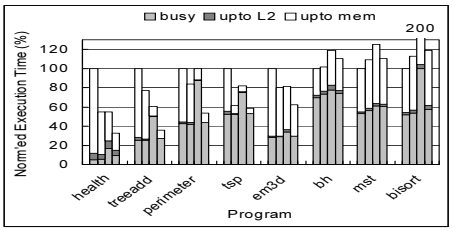

Fig. 15. Execution time of the programs with software FACT (FACT-S). In each group, each bar shows execution time of the baseline configuration, with I2, with FACT-S, and with FACT, respectively

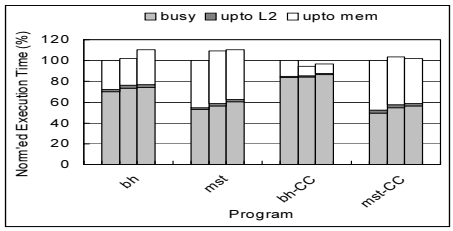

Fig. 16. Execution time of the programs with CC, I2, and FACT. The "bh" and "mst" groups are a subset of Fig. 12. In the "bh-CC" and "mst-CC" groups, each bar shows execution time of the baseline configuration with CC, with CC+I2, and with CC+FACT, respectively

FACT Software-Only Implemention FACT introduces new instructions and special hardware for compression and decompression operations to reduce the overhead associated with these operations. To depict the amount of this overhead, we implement FACT by modification of the source code only (we call this FACT-S) and compare FACT-S with FACT. We first apply I2 and then replace definitions and usages of the compression target fields with custom inline compression and decompression functions to implement FACT-S. In typical cases, the compression function requires 23, 49, and 15 instructions for RPC, DIC, and NIC, respectively. The decompression function requires 19, 23, and 13 instructions. Figure 15 shows the FACT-S execution time. We can see the software compression/decompression overhead as the increase in busy cycles. FACT-S is slower than I2 in all programs except `tre` because the overhead offsets the reduction in memory stall time. This observation confirms that additional instructions and hardware for FACT reduce the overhead effectively.

FACT with Cache-Conscious Data Layout Techniques FACT exhibits poor performance in `bh`, `mst`, and `bis` because they do not meet condition (C2) in Sect. 3.4. There are two reasons that `bh` does not meet the condition: (B1) `bh` skips graph nodes, and (B2) its nodes are organized in memory in a random order. There are two reasons for `mst`: (M1) `mst` traverses a small number of nodes at a time, and (M2) the nodes in each list are distributed in memory. There is one reason for `bis`: (S1) `bis` changes the graph structure quite frequently. However, we can organize these nodes in memory in a similar order to the traversal order using cache-conscious data layout techniques (CC) [6] for those programs which do not modify the graph structures quite frequently. Using CC, we can correct properties (B2) and (M2) and make FACT exploit more affinity in these programs. Therefore we apply these techniques and show how the execution time with FACT changes. For `bh`, we applied `ccmorph`, which inserts a function in the source code, to reorganize the graph at runtime. We reorganize the graph in depth-first order in memory because depth-first order is the typical order of traversal. We applied `ccmalloc` for `mst`; it replaces the memory allocator and attempts to allocate a graph node in the same cache-block as the "parent" node. The "parent" node is specified through an argument. We select the node next to the allocating node in the list as the "parent" node.

Figure 16 shows the results: the "bh" and "mst" groups are a subset of Fig. 12. The "bh-CC" and "mst-CC" groups show the results with CC normalized to the baseline configuration running programs with CC. Using CC, FACT can reduce the memory

stall time for `bh` and achieve speed beyond the baseline. In addition, FACT shows almost no degradation in the execution speed for `mst`.

6 Summary and Future Directions

We proposed the Field Array Compression Technique (FACT), which reduces cache misses caused by recursive data structures. Using a data layout transformation, FACT gathers data with temporal affinity in contiguous memory, which enhances the prefetch effect of a cache-block. It enlarges the effective cache-block size and enhances the prefetch effect further from the compression of recursive pointer and integer fields. It also enlarges the effective cache capacity. Through software simulation, we showed that FACT yields a 41.6% reduction of stall cycles waiting for memory data on average.

This paper has four main contributions. **(1)** FACT achieves a compression ratio of 1/8 and over. This ratio exceeds 1/2, which is the limit of existing compression methods. This ratio is achieved because of the data layout transformation and the novel addressing of the compressed data in the caches. **(2)** We are able to compress many recursive pointer fields into 8 bits, which is achieved partly because of grouping the pointers. **(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. Additionally, the addresses of the compressed elements and the uncompressed elements have an affine relationship. **(4)** We represent the notion of an address space for compressed data in the caches. The address space for uncompressed data is shrunk to be used as the address space for compressed data in the caches. This simplifies the address translation from the address for the uncompressed data to the address which points to the compressed data in the caches, and avoids cache conflicts.

Future work will entail building a framework to automate FACT using a compiler.

References

1. J. Bonwick. The slab allocator: An object-caching kernel memory allocator. *Proc. USENIX Conference*, pp. 87–98, Jun. 1994.
2. D. A. Barret and B. G. Zorn. Using lifetime prediction to improve memory allocation performance. *Proc. PLDI*, 28(6):187–196, Jun. 1993.
3. D. N. Truong, F. Bodin, and A. Seznec. Improving cache behavior of dynamically allocated data structures. *Proc. PACT*, pp. 322–329, Oct. 1998.
4. A. Rogers et al. Supporting dynamic data structures on distributed memory machines. *ACM TOPLAS*, 17(2):233–263, Mar. 1995.
5. Compaq Computer Corp. Alpha 21264 Microprocessor Hardware Reference Manual. Jul. 1999.
6. T. M. Chilimbi et al. Cache-conscious structure layout. *Proc. PLDI*, 1999.
7. C-K. Luk and T. C. Mowry. Compiler based prefetching for recursive data structures. *Proc. ASPLOS*, pp. 222–233, Oct. 1996.
8. A. Roth, A. Moshovos, and G. S. Sohi. Dependence based prefetching for linked data structures. *Proc. ASPLOS*, pp. 115–126, Oct. 1998.
9. J. Lee et al. An on-chip cache compression technique to reduce decompression overhead and design complexity. *Journal of Systems Architecture*, Vol. 46, pp. 1365–1382, 2000.
10. Y. Zhang, J. Yang, and R. Gupta. Frequent value locality and value-centric data cache design. *Proc. ASPLOS*, pp. 150–159, Nov. 2000.
11. J. Yang, Y. Zhang, and R. Gupta. Frequent value compression in data caches. *Proc. MICRO*, pp. 258–265, Dec. 2000.
12. 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., 2000.
13. Y. Zhang et al. Data compression transformations for dynamically allocated data structures. *Int. Conf. on Compiler Construction*, LNCS 2304, Springer Verlag, pp. 14–28, Apr. 2002.
14. D. Brooks et al. Dynamically exploiting narrow width operands to improve processor power and performance. *Proc. HPCA*, pp. 13–22, Jan. 1999.
15. R. M. Rabbah and K. V. Palem. Data remapping for design space optimization of embedded memory systems. *ACM TECS*, 2(2):186–218 May 2003.