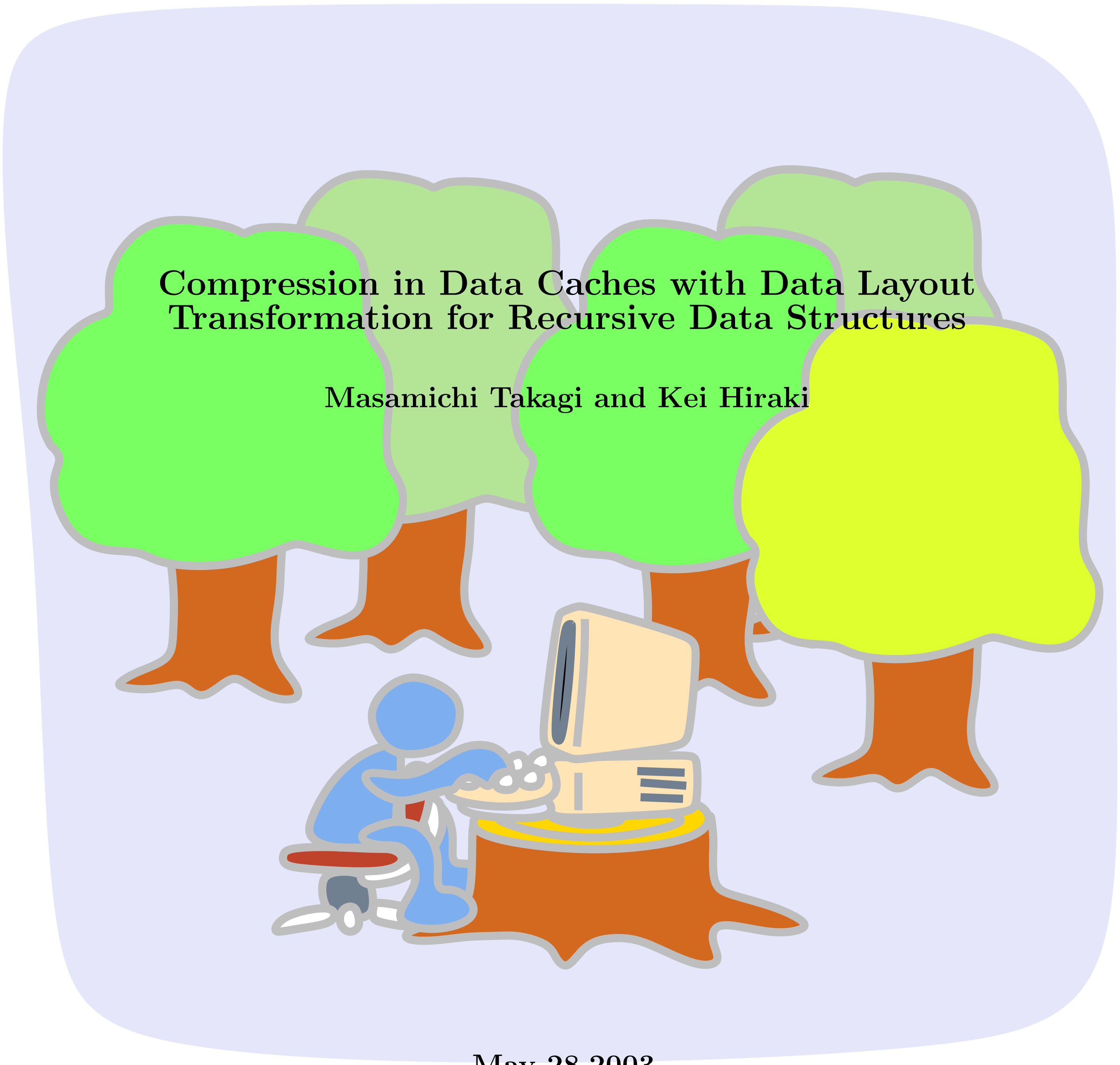
TECHNICAL REPORT 03-01

Compression in Data Caches with Data Layout Transformation for Recursive Data Structures

Masamichi Takagi and Kei Hiraki



May 28,2003

Department of Computer Science
University of Tokyo

7–3–1 Hongo, Bunkyo-Ku, Tokyo 113, Japan

TECHNICAL REPORT 03-01

Compression in Data Caches with Data Layout Transformation for Recursive Data Structures

Masamichi Takagi and Kei Hiraki

May 28,2003

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF TOKYO

7–3–1 HONGO, BUNKYO-KU TOKYO, 113 JAPAN

TITLE

Compression in Data Caches with Data Layout Transformation for Recursive Data Structures

AUTHORS

Masamichi Takagi and Kei Hiraki

KEY WORDS AND PHRASES

processor architecture, memory system and management, cache memory, hardware and software technique, data compression

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. 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.

REPORT DATE May 28,2003	**WRITTEN LANGUAGE** English
TOTAL NO. OF PAGES 16	**NO. OF REFERENCES** 18

ANY OTHER IDENTIFYING INFORMATION OF THIS REPORT

DISTRIBUTION STATEMENT

SUPPLEMENTARY NOTES

DEPARTMENT OF COMPUTER SCIENCE
University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan

Compression in Data Caches with Data Layout Transformation for Recursive Data Structures

Masamichi Takagi, Kei Hiraki
Department of Computer Science,
University of Tokyo
7-3-1, Hongo, Bunkyo-ku, Tokyo, Japan
{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. 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, trees for space cells, and trees for data repositories. Such programs using RDS make graphs and traverse them, however the traversal code often leads to cache misses. This is 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. Software and hardware data prefetching techniques are common [8][9]. Another technique is data layout transformations. These gather data with temporal affinity in contiguous memory to improve the prefetch effect of a cache-block [3]. Yet another technique is to enlarge the cache capacity, however this has limitations due to increasing the access time. A technique closely related to enlarging the cache capacity is data compression in caches, as compressing the data stored in the caches enlarges their *effective* capacity [10, 13, 15, 16]. Not only does compression enlarge the effective 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 the data accessed on the critical path 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. Therefore we propose a method which achieves a compression ratio over $\frac{1}{2}$ 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 data layout transformation along with recursive pointer and integer field compression. It enhances the prefetch effect of a cache-block and enlarges the effective capacity of the cache, leading to a reduction in the number of cache misses caused by RDS. 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 $\frac{1}{2}$, and achieves compression ratios of $\frac{1}{8}$ and over.

The remainder of this paper is organized as follows: In Section 2 we discuss related works, and in Section 3 we explain FACT in detail. Section 4 describes our evaluation methodology, we present and discuss our results in Section 5, and we give conclusions in Section 6.

2 Related Works

Several studies have proposed varying techniques for data compression in the caches. Yang et al. proposed a hardware method [13], which does not require modification of the source code. They compress each 32-bit word in a single cache-block and put it into the primary cache. Their method finds memory accesses which refer to a small number of distinct values in the entire program run, and compresses them using fixed-length coding and a static dictionary. We use this method for the compression of integer fields. Larin and Conte proposed another hardware method [15], which is also transparent to the program. Their method compresses each byte in the N-cache-block using Huffman coding, and puts the result into the primary cache. Lee et al. proposed a hardware method [10], which compresses 2 cache-blocks using the X-RL algorithm [11] and puts the result into the secondary cache.

Assuming the compression ratio is $\frac{1}{R}$, 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 conventional cache structure by limiting the compression ratio to $\frac{1}{2}$. FACT solves this problem by using a novel addressing scheme to point to the compressed data in the caches.

Zhang and Gupta proposed a combined hardware/software method for compressing dynamically allocated data structures [16]. 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 $\frac{1}{2}$ and makes a pair from them to put into the slot. Since the slot resides within the structure and the fields which are not the target of the compression require word alignment, the size of the slot cannot be smaller than one-word. In addition, each slot must gather fields from a single instance. These problems 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. The method of Zhang and Gupta allocates the area for the compressed data initially, and when it finds incompressible data, it allocates additional space for this data in uncompressed form. In contrast, our method initially allocates the space for both the compressed and uncompressed data.

Truong et al. proposed a transformation of the data layout for RDS, which they call Instance Interleaving [3]. It modifies the source code of the program to take identical fields from different instances of the data structure and make them contiguous in the memory. When they have temporal affinity, this transformation enhances the prefetch performance of a cache-block. Since the compression of data caches can enlarge the effective size 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 utilize this method to preprocess the data prior to compression.

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 detailed steps 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 (we call these compressible fields). These compressible fields are the target 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 by hand 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, and we choose an appropriate type for each replacement.

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.

Since we assume a commonly-used two-level cache hierarchy, the compressed data resides in the primary data cache and the secondary unified cache.

In the following sections, we describe details of the individual steps of FACT:

1. Compression method of structure fields
2. Selection of compression target fields
3. Isolation and gathering of compressible fields through data layout transformation of the data structures
4. Addressing the compressed data in the caches
5. Deployment of `cld`/`cst` instructions and their operation

3.1 Compression of Structure Fields

We compress recursive pointer fields since they are often accessed in the critical path. We also compress integer fields, since they often have exploitable redundancy.

3.1.1 Pointer Field Compression Using Sequence Number

Since we often allocate space for RDS in a group, the distance between memory addresses of two structures connected by a pointer is often small. In addition, one recursive pointer in the structure points to another instance of the same structure. 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. To facilitate the relative address calculation, we make a custom memory allocator for the structure. Its allocation step is similar to [1]. On allocation request, it allocates a pool of instances if a free instance is not available, and returns one instance from the pool. Then we modify the source code to make it use the allocator. Using this layout, the `cst` instruction replaces the pointers in runtime with the relative sequence numbers in the pool. Figure 1 illustrates the compression. Assume we are constructing a balanced binary

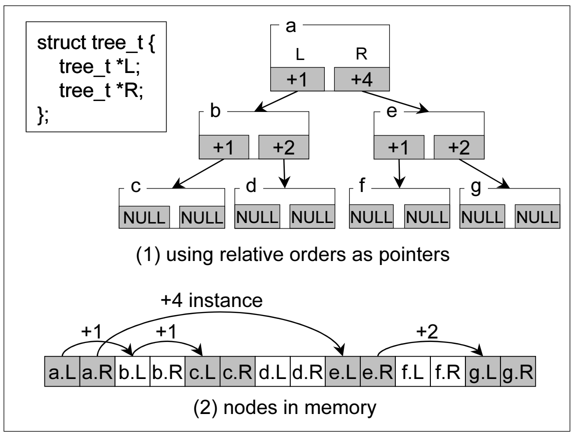


Figure 1: Pointer compression: we replace absolute addresses with relative sequence numbers in the instance pool.

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 compression algorithm of pointer fields. 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 $1/R$. Since the difference between the addresses of two consecutive instances is 8 bytes due to the data layout transformation described in Section 3.3.1, the relative sequence number in the pool is `(STDATA-BASE)/8` and we use it 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. Using hardware, the calculation for the compression can be done with arithmetic and the shift operation.

STDATA	Data to be stored
BASE	Base address
1/R	Compression ratio
INCMP	Codeword indicating incompressibility (-2^(64/R-1))
NULLCODE	Codeword indicating NULL (-2^(64/R-1)+1)

```
/* algorithm of pointer field compression */
compress_ptr(STDATA, BASE) {
   if(STDATA == 0) { return NULLCODE }
   DIFF = (STDATA – BASE)/8
   N = 64/R
   if(DIFF != NULLCODE && DIFF != INCMP &&
      -2^(N-1) <= DIFF && DIFF <= 2^(N-1)-1) {
      return DIFF
   } else {
      return INCMP
   }
}
```

Figure 2: Compression algorithm of pointer fields.

The decompression is done 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. In the case that the compression ratio is $\frac{1}{8}$, decompression requires an 8-to-1 MUX after fetching the word data from the cache, followed by 8-bit addition and a logical-shift operation.

3.1.2 Integer Field Compression

The integer fields often have exploitable redundancy. For example, there are fields which take only a small number of distinct values [12] or which use a narrower bit-width than is available [17, 18]. FACT exploits these characteristics by utilizing two methods.

In the first method, 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 [13]. 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 upper half of Figure 3 shows the algorithm. The compression is done when writing the integer field. Assume the compression ratio is $\frac{1}{R}$. The `cst` instruction searches the data in the dictionary, and if it finds the data, it uses the entry number as the 32/R-bit codeword. If the data is not found, the codeword indicating incompressibility is used. The dictionary can be implemented with CAM in a similar way to [14]. The decompression is done when reading the integer field. The `cld` instruction reads the dictionary with the compressed data. The dictionary itself can be implemented with a register file. When using a compression ratio of $\frac{1}{8}$, the compressed data goes through an 8-to-1 MUX after it is fetched from the cache, and is then used as the index when reading the 16-entry register file. In the second method, we locate 32-bit integer fields which use a narrower bit-width than is available, and replace them with narrower bit-width integers. The bottom half of Figure 3 shows the algorithm. On compression, the `cst` instruction checks the bit-width of the storing data, and it omits the upper bits if it can. On decompression, the compressed data is sign-extended.

While the first method includes the second, the second needs access to the hardware dictionary on decompression. Since a larger dictionary requires greater time, we use the first method when the dictionary contains less than or equal to 16 entries, otherwise we use the second method. That is, when we choose a compression ratio of $\frac{1}{8}$ or more in the entire program, we use the first method for the entire program, otherwise we use the second method.

3.2 Selection of Compression Target Field

In FACT, we locate the compressible fields of RDS in the program. Since the compressibility depends on the values in the fields which are determined dynamically, we gather runtime statistics of the load/store instructions through profiling. The profile-run takes different input parameters to the actual run, and collects the access

STDATA	Data to be stored
BASE	Base address
1/R	Compression ratio
INCMP	Codeword indicating incompressibility (-2^(32/R-1))

```
/* algorithm of int field compression (dictionary) */
compress_int_dict(STDATA) {
   if(STDAT in dictionary) {
      return entry number in dictionary
   } else {
      return INCMP
   }
}

/* algorithm of int field compression (narrow bit-width) */
compress_int_narrow(STDATA) {
   N = 32/R
   if(STDATA != INCMP &&
     -2^(N-1) <= STDATA && STDATA <= 2^(N-1)-1) {
      return STDATA
   } else {
      return INCMP
   }
}
```

Figure 3: Compression algorithm for integer fields.

count of the total instructions (A_{total}), the access count of the fields for each static instruction (A_i) and the occurrence count of compressible data for each static instruction (O_i). Then we mark the static instructions which 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 targets of the compression. We set $X = 0.1\%$ and $Y = 90\%$ through the experiments in the current implementation.

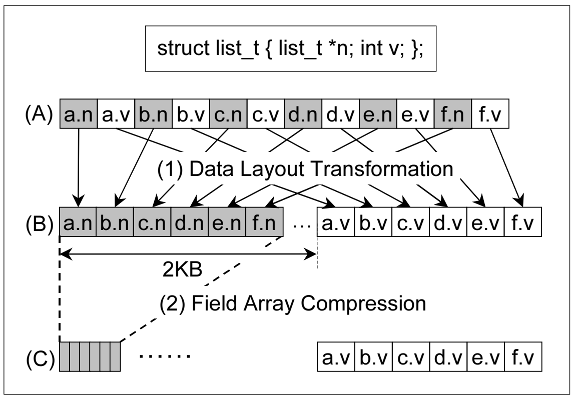
We take multiple profile-runs, which use different compression ratios. Taking three profile-runs, which have compression ratios of $\frac{1}{4}$, $\frac{1}{8}$, and $\frac{1}{16}$ respectively, we select one compression ratio to be used based on the compressibility numbers shown. This selection is done 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. This transformation also enables us to exploit any temporal affinity among the fields. The transformation is the same as Instance Interleaving proposed by Truong et al. [3]. Since the transformation produces an array of identical fields, we call it a Field Array Transformation (FAT).

3.3.1 Isolation and Gathering of Compressible Fields Through FAT

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. Assume the address of the instance is I and the compression ratio is $1/R$, then the translated address is I/R. However, the processor must know the value of R. To calculate R, we need the size of the incompressible part, and the size of the compressible part before and after compression. Therefore it 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 4 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 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 $I/8$, which can be done by a simple logical-shift operation.



struct list_t { list_t *n; int v; };
(A) a.n a.v b.n b.v c.n c.v d.n d.v e.n e.v f.n f.v
(1) Data Layout Transformation
(B) a.n b.n c.n d.n e.n f.n ... a.v b.v c.v d.v e.v f.v
2KB
(2) Field Array Compression
(C) ...... a.v b.v c.v d.v e.v f.v

Figure 4: Field Array Transformation (FAT) isolates and groups the compressible fields.

3.3.2 Exploiting Temporal Affinity Through FAT

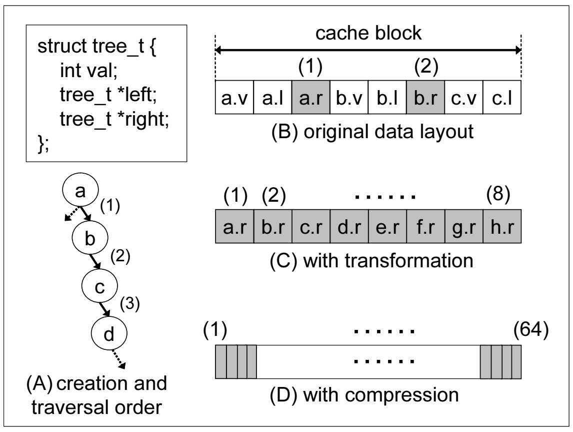


Figure 5: FAT groups fields with temporal affinity, and FACT allows one cache-block to contain more fields by compression.

We can exploit temporal affinity between fields through the transformation. Figure 5 illustrates this. Consider the binary tree in the program `treeadd`, which is used in the evaluation. The declaration of the structure is shown in the figure. After the 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 contiguous in memory have temporal affinity. Since each instance requires 24 bytes without FAT, using a 64-byte cache-block, 1 cache-block can hold 2 `right` pointers with temporal affinity (B). With FAT, it can hold 8 of these pointers (C), and with compression at a ratio of $\frac{1}{8}$, it can hold 64 of these pointers (D). This transformation enhances the prefetch effect of a cache-block.

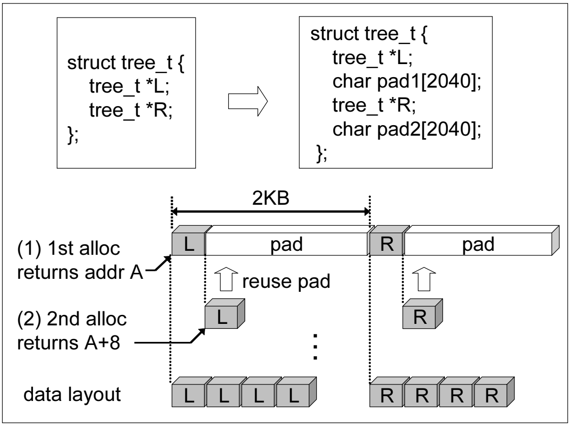
3.3.3 Implementation of FAT

We transform the data layout by modifying the source code. First, we modify the declaration of the structure to insert pads between fields. Figure 6 shows the declaration of `tree_t` in `treeadd` (1) before and (2) 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. Since the insertion of pads makes the address of the n-th field (structure head address)+(pad size)×$(n-1)$, 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 7 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. While this layout violates C semantics, it is rare this causes a problem (e.g. copying the structure).

(1)	(2)

```
typedef struct tree {
    int val;
    struct tree *left, *right;
} tree_t;
```
	```
	typedef struct tree {
	    int val;
	    char pad1[2044];
	    tree *left;
	    char pad2[2040];
	    tree *right;
	    char pad3[2040];
	} tree_t;
	```

Figure 6: Declaration of `tree_t` in `treeadd`, original (1) and after FAT (2).



```
struct tree_t {
    tree_t *L;
    tree_t *R;
};
```

```
struct tree_t {
    tree_t *L;
    char pad1[2040];
    tree_t *R;
    char pad2[2040];
};
```

Figure 7: FAT using a padded structure. By inserting pads between fields, it reserves the contiguous area, and places identical fields from other instances there.

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

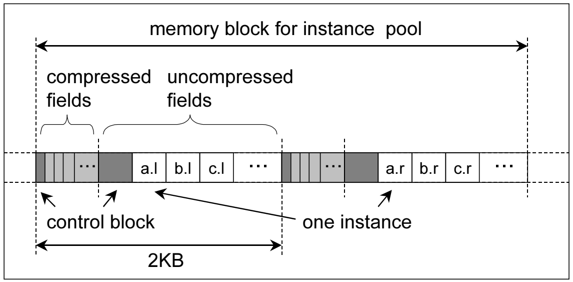


Figure 8: Internal organization of memory block managed by the custom allocator.

instance pool, which we call the arena. Figure 8 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. It then initializes the control block which is used to manage the free objects. The management scheme is similar to that described in [2].

We implemented another allocator using the instance pool technique, similar to [1, 2], and used it when evaluating the baseline configuration. The reason for this was so that any improvement in execution speed due to FACT could be determined separately from improvements due to the use of the instance pool technique.

3.4 Address Translation for Compressed Data

Since we attempt to compress data which changes dynamically, we find it is not always compressible. When we find incompressible data, area 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 [16]. 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 $\frac{1}{8}$, it divides each 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, as the compressed block is a reduced mirror image of the uncompressed block.

Consider the compressed data d and its physical address a, the uncompressed data D and its 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 $\frac{1}{R+1}$ 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 $\frac{R}{R+1}$ 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.

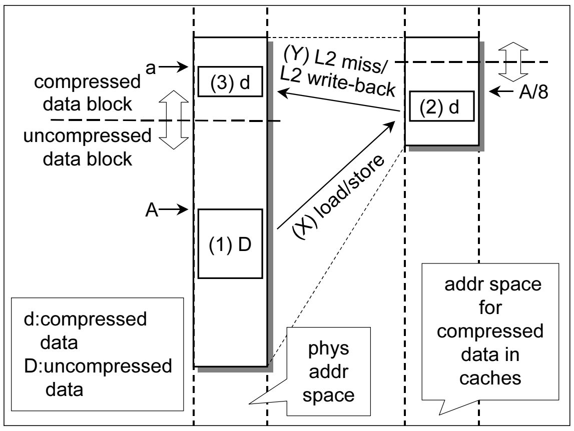


Figure 9: Address translation in FACT.

Figure 9 illustrates the translation steps. Assume we compress data at physical address $A(1)$ at the compression ratio of $\frac{1}{8}$. The compressed data is stored at the physical address $a(3)$. Then `cld/cst` instructions access the compressed data in the caches using the address $\frac{A}{8}$(X)(2). When the compressed data needs to be written-back to or fetched from main memory, we translate address $\frac{A}{8}$ into address a(Y)(3).

3.5 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 have two types of target, recursive pointer fields and integer fields, and use two methods of integer field compression, we have three types of `cld/cst` instructions. We choose one type depending on the type of the target field and the compression method chosen. We use only one integer compression method in the entire program. If we choose the compression ratio for the program to be higher than or equal to $\frac{1}{8}$, we use the dictionary method, otherwise we use the narrower bit-width method.

Figure 10 shows the operation of the `cst` instruction, and Figure 11 shows its operation in the cache. In the figures, we assume a cache hierarchy of one level for simplicity. We also assume a physically tagged, physically indexed cache. 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. 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 12 shows the operation of the `cld` instruction and Figure 13 shows its operation in the cache.

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)
  }
}
```

Figure 10: Operation of `cst` instruction.

ADDR	Address on 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}
}
```

Figure 11: Operation of `cst` instruction in the cache.

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 it is 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 7 stages: instruction cache access 1, instruction cache access 2, decode/rename, schedule, register-read, execution, write-back/commit. For load/store instructions, 4 stages follow the register-read stage: address-generation, TLB-access/data cache access 1, data cache access 2, write-back/commit. Therefore the load-to-use latency is 3 cycles. We assume the `cst` instruction can perform the compression calculation in its address-generation and TLB-access stages, therefore no additional latency is required. We assume the decompression performed by the `cld` instruction requires one additional cycle to the

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
}
```

Figure 12: Operation of `cld` instruction.

ADDR	Address on cache
FLAG	C:compressed data N:uncompressed data

```
/* operation of cld instruction in cache */
cache_read(ADDR, 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
        }
    }
    return memory data obtained
}
```

Figure 13: Operation of `cld` instruction in the cache.

load-to-use latency. When `cst` and `cld` instructions handle 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`. When `cld`/`cst` instructions access the compressed data, they shrink their addresses. We assume this is done within their address-generation and TLB-access stages, therefore no additional latency is required.

We developed an execution-driven, cycle-accurate software simulator of a superscalar processor to evaluate FACT. Contentions on the caches and buses are simulated. Table 1 shows its parameters. We set the instruction latency and the issue rate to be the same as the Compaq Alpha 21264 [6].

We used 8 programs from the Olden benchmark [4, 5], `health`, `treeadd`, `perimeter`, `tsp`, `em3d`, `bh`, `mst`, `bisort`, because they use RDS and they have a high rate of stall cycles caused by cache misses. Table 2 shows their profile-run input parameters, evaluation-run input parameters, characteristics, and the compression ratio used. All programs were compiled using Compaq C Compiler version 6.2-504 on Linux Alpha, using optimization option "-O4".

[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 1: Simulation parameters: parameters of processor and memory hierarchy for baseline configuration.

Processor	
Pipeline	7 stages, 6-cycle misprediction penalty
Fetch	fetch upto 8 insts, regardless of cache-block boundary, one request per cache-block, second taken branch terminates fetch, 24-entry request queue, 32-entry inst. queue
Branch pred.	16K-entry GSHARE, 256-entry 4-way BTB, 16-entry RAS
Decode/Issue	decode upto 8 insts, 128-entry inst. window, issue upto 8 insts
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 buffer, 16 MSHRs, oracle resolution of load–store addr. dependency
Retire	retire upto 8 insts, 256-entry reorder buffer
Memory hierarchy	
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 memory	200-cycle load-to-use latency, off-chip memory controler, 128-bit address/data muxed bus to L2, bus is clocked at 1/4 frequency of processor chip
TLB	256-ent, 4-way inst-TLB, 256-ent, 4-way data-TLB, pipelined, 50-cycle latency h/w miss handler

Table 2: Program used in the evalutaion: input parameters for profile-run, input parameters for evaluation, max. memory dynamically allocated, instruction count, ratio of stall cycles waiting for memory data due to cache-miss against the total execution cycles, data structures, compression ratio used, numbers and kinds of compression target fields in data structures (integer, pointer). The 4th to 6th columns show the numbers in the baseline configuration.

Name	Input param. for profile-run	Input param. for evaluation	Max dyn. memory	Inst. count	Mem. stall	Data structures	Compr. ratio	Compr. field
health	lev 5, time 50	lev 5, time 300	2.58MB	69.5M	95.0%	quad-tree, dbl-list	1/4	2, 3
treeadd	4K nodes	1M nodes	25.3MB	89.2M	74.7%	bin-tree	1/8	2, 1
perim.	128×128 img	16K×16K img[1]	19.0MB	159M	57.5%	quad-tree	1/8	5, 2
tsp	256 cities	64K cities	7.43MB	504M	47.6%	bin-tree, dbl-list	1/8	4, 1
em3d	1K nodes, 3D	32K nodes, 3D	12.4MB	213M	71.9%	single-list	1/8	1, 2
bh	256 bodies	4K bodies[2]	0.909MB	565M	30.2%	oct-tree, single-list	1/4	10, 6
mst	256 nodes	1024 nodes	27.5MB	312M	47.1%	array of single-lists	1/4	1, 0
bisort	4K integers	256K integers	6.35MB	736M	48.2%	bin-tree	1/4	2, 1

5 Results and Discussions

5.1 Compressibility of Recursive Pointer Fields and Integer Fields

First 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 $\frac{1}{4}$, $\frac{1}{8}$, and $\frac{1}{16}$. In these cases, 64-bit pointers are compressed into 16 bits, 8 bits, and 4 bits respectively, and 32-bit integers are compressed into 8 bits, 4 bits, and 2 bits respectively. Table 5.1 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, `treeadd`, `perimeter`, `tsp`, and `em3d` 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, `health`, `bh`, `mst`, and `bisort` exhibit low success rates, because they organize the nodes in a different order to the traversal order, or because they change the graph structure frequently.

With respect to integer compression, since `treeadd`, `tsp`, `em3d`, `bh`, and `bisort` have narrow bit-width integer fields, they exhibit high success rates. Among them, `treeadd` and `bisort` also exhibit high access rates. Since `perimeter` has enumeration type fields, it also has a high success rate. `bh` and `mst` have few accesses to the compression targets. In `perimeter` and `em3d`, when the compression ratio is high, the dictionary based method has a higher success rate than the narrow bit-width based method. This is because the former utilizes the narrow bit-width of codewords more efficiently than the latter. On the other hand, in healh and tsp, the narrow bit-width based method exhibits a higher success rate, because values not seen in the profile-run are used. In the following, we use a compression ratio of $\frac{1}{8}$ for treeadd, perimeter, em3d, tsp, and of $\frac{1}{4}$ for health, mst, bh, bisort.

5.2 Effect of Pointer Compression and Integer Compression

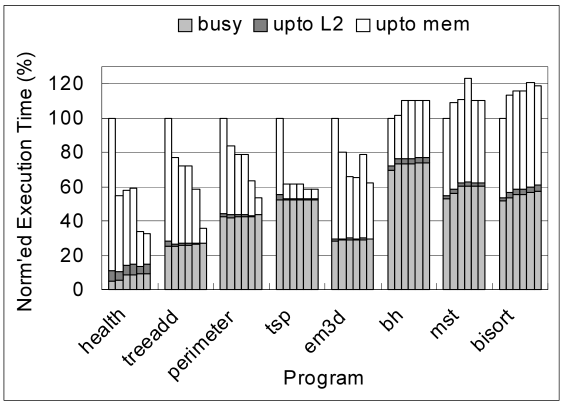


Figure 14: Execution time of the programs. In each group, each bar shows from the left, execution time of the baseline configuration, with FAT, with integer compression using narrow bit-width, with integer compression using the dictionary, with pointer compression, and with FACT, respectively

FACT uses both pointer field compression and integer field compression, and it uses two types of integer field compression. Therefore we show the individual effect of each, and the effect of combinations of the three methods. Figure 14 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 due to cache misses, stall cycles due to accesses to the secondary cache (upto L2), and stall cycles due to accesses to main memory (upto mem). All bars are normalized to the baseline configuration.

First we compare the compression of integer fields against the compression of recursive pointer fields. In health, treeadd, perimeter, and tsp, the pointer field compression 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, the integer compression 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 the two methods of integer compression. In health and mst, the method using the narrow bit-width shows more speedup than the method using the dictionary. 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 bisort, neither integer nor pointer compression lead to a reduction in memory stall cycles. This will be described in Section 5.3.

Figure 15 shows the results using a combination of pointer field compression and integer field compression. In all programs except health and mst, the difference between the two integer compression methods is small. In health and mst, the integer compression using the narrow bit-width method reduces memory stall cycles to a greater extent. When comparing Figure 14 and Figure 15, a combination of pointer and integer compression leads to greater performance than either method alone in all programs except bh, mst, and bisort. In addition, FACT is the best performing method in all programs except bh, mst, and bisort.

5.3 Execution Time Results of FAT and FACT

Figure 16 compares the execution times of the programs using FAT and FACT. As we can see from the figure, FACT reduces the stall cycles waiting for memory data by 41.6% on average, while FAT alone reduces the cycles by 23.0% on average.

The main data structure these programs use is the graph structure. If the traversal order and the memory order of the nodes in the programs are the same, FAT can exploit temporal affinity between pointers, which FACT can exploit further by compression. This is the case in health, treeadd, perimeter, and em3d. Especially in treeadd and perimeter, since the whole structure can be compressed into $\frac{1}{8}$ of its original size, FACT

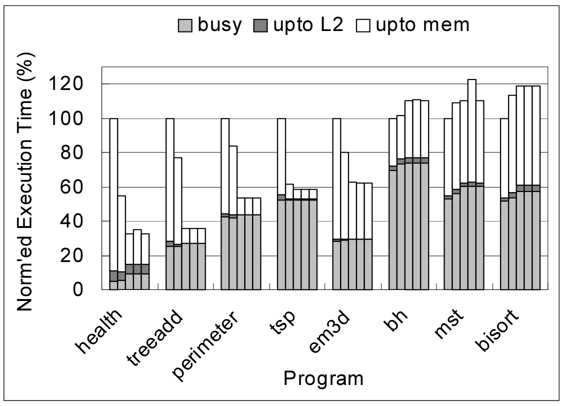


Figure 15: Execution time of the programs. In each group, each bar shows from the left, execution time of the baseline configuration, with FAT, with integer field compression using narrow bit-width and pointer field compression, with integer field compression using the dictionary and pointer field compression, with FACT, respectively.

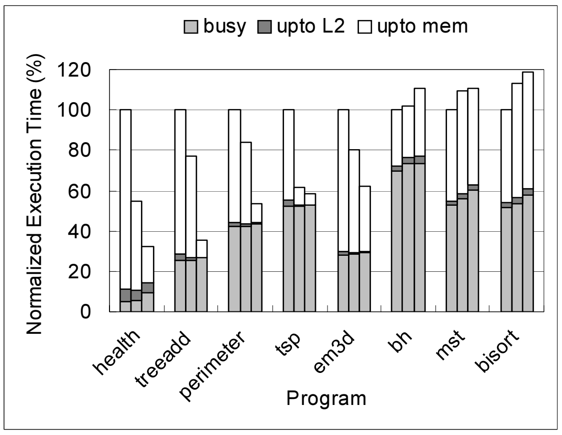


Figure 16: Execution time of the programs with FAT and FACT. In each group, each bar shows from the left, execution time of the baseline configuration, with FAT, and with FACT, respectively.

reduces most of the memory stall cycles which FAT leaves. In `tsp`, since FAT reduces most of the memory stall cycles, further reduction by FACT is small.

FAT 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, FAT cannot gather data with temporal affinity, and cannot recover this inefficiency thus lowering the utilization ratio of a cache-block, resulting in degradation of performance. In `bh`, the creation order and the traversal order of the nodes are different. In `mst`, since many lists allocate few elements in turn, nodes in each list are located far apart in memory. In `bisort`, the graph structure is changed frequently. As the memory order and the traversal order of the nodes in these programs are different, both FAT and FACT are ineffective.

When the processor using FACT finds incompressible pointers, three things can degrade the performance. These are, a serial chain of accesses to the compressed and uncompressed data, cache misses in the two references, and cache conflicts caused by the uncompressed data. In `bisort`, the frequent changes in the graph structure increase the incompressibility of the pointers, and in this case using FACT can actually slow down the program. Note that in `health`, `bh`, `mst`, and `bisort`, the number of busy cycles increase. This is because a low utilization ratio of the cache-block with FAT increases the TLB misses, and the incompressible pointers impose the overhead of making two references.

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. From the compression of recursive pointer fields and integer fields, it enlarges the effective cache-block size and enhances the prefetch effect further. It also enlarges the effective capacity of the cache. Through software simulation, we showed that FACT yields a 37.4% speedup and a 41.6% reduction of stall cycles waiting for memory data on average.

This paper has four main contributions. Assume a target compression ratio of $\frac{1}{8}$.

1. FACT achieves the compression ratio of $\frac{1}{8}$. This ratio exceeds $\frac{1}{2}$, which is the limit of existing compression methods. This ratio is achieved due to 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 due to 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, which simplifies the address translation from uncompressed data address to compressed data address and avoids cache conflict.

FACT exhibits poor performance with programs in which the memory order and the traversal order of graph nodes are different. We are currently investigating if graph reorganization at runtime, such as that used in [7], can help by making the memory and traversal orders the same. In addition, we are building a framework which automates the processes of FACT.

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. the SIGPLAN Conf. on Programming Language Design and Implementation*, 28(6):187–196, Jun. 1993.

[3] D. N. Truong, F. Bodin, and A. Seznec. Improving cache behavior of dynamically allocated data structures. *Proc. the 1998 Int. Conf. on Parallel Architectures and Compilation Techniques*, pp. 322–329, Oct. 1998.

[4] A. Rogers, M. C. Carlisle, J. Reppy, and L. Hendren. Supporting dynamic data structures on distributed memory machines. *ACM Transactions on Programming Languages and Systems*, 17(2):233–263, Mar. 1995.

[5] M. C. Carlisle and A. Rogers. Software caching and computation migration on olden. *Proc. ACM Symp. on Principles and Practice of Parallel Programming*, pp. 29–38, Jul. 1995.

[6] Compaq Computer Corporation. Alpha 21264 Microprocessor Hardware Reference Manual. Jul. 1999.

[7] T. M. Chilimbi, M. D. Hill, and J. R. Lalus. Cache-conscious structure layout. *Proc. the SIGPLAN Conf. on Programming Language Design and Implementation*, 1999.

[8] C-K. Luk and T. C. Mowry. Compiler based prefetching for recursive data structures. *Proc. the 7th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 222–233, Oct. 1996.

[9] A. Roth, A. Moshovos, and G. S. Sohi. Dependence based prefetching for linked data structures. *Proc. the 8th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 115–126, Oct. 1998.

[10] J. Lee, W-K. Hong, and S.D. Kim. An on-chip cache compression technique to reduce decompression overhead and design complexity. *Journal of Systems Architecture*, Vol. 46, pp. 1365–1382, 2000.

[11] M. Kjelso, M. Gooch, and S. Jones. Design and performance of a main memory hardware data compressor. *Proc. the 22nd EUROMICRO Conference*, pp. 423–430, Sep. 1996.

[12] Y. Zhang, J. Yang, and R. Gupta. Frequent value locality and value-centric data cache design. *Proc. the 9th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 150–159, Nov. 2000.

[13] J. Yang, Y. Zhang, and R. Gupta. Frequent value compression in data caches. *Proc. the 33rd annual IEEE/ACM Int. Symp. on Microarchitecture*, pp. 258–265, Dec. 2000.

[14] J. Yang and R. Gupta. Energy Efficient Frequent Value Data Cache Design. *Proc. the 35th annual IEEE/ACM Int. Symp. on Microarchitecture*, pp. 197–207, Nov. 2002.

[15] 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.

[16] Y. Zhang and R. Gupta. Data compression transformations for dynamically allocated data structures. *Proc. Int. Conf. on Compiler Construction*, LNCS 2304, Springer Verlag, pp. 14–28, Apr. 2002.

[17] D. Brooks, and D. Martonosi. Dynamically exploiting narrow width operands to improve processor power and performance. *Proc. the 5th Int. Symp. on High-Performance Computer Architecture*, pp. 13–22, Jan. 1999.

[18] M. Stephenson, J. Babb, and S. Amarasinghe. Bitwidth analysis with application to silicon compilation. *Proc. the SIGPLAN Conf. on Programming Language Design and Implementation*, pp. 108–120 Jun. 2000.

Prog. name	Accesses of compression target(%)			Compressible accesses(%)		
type→	pointer field, relative sequence number					
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

Prog. name	Accesses of compression target(%)			Compressible accesses(%)		
type→	integer field, narrow bit-width					
ratio→	1/4	1/8	1/16	1/4	1/8	1/16
health	24.2	1.51	.677	83.7	88.6	93.7
treeadd	5.80	5.78	5.77	100	100	100
perim.	12.4	12.4	4.33	100	100	83.1
tsp	.107	.107	.107	99.2	87.5	50.0
em3d	1.54	1.54	.650	100	99.1	68.8
bh	.0111	.0111	.0111	100	100	100
mst	0	0	0	0	0	0
bisort	27.8	0	0	100	0	0

Prog. name	Accesses of compression target(%)			Compressible accesses(%)		
type→	integer field, static dictionary					
ratio→	1/4	1/8	1/16	1/4	1/8	1/16
health	24.3	24.1	.827	46.9	18.9	90.3
treeadd	5.80	5.78	5.77	100	100	100
perim.	12.4	12.4	12.4	100	100	91.0
tsp	.107	.107	.107	50.0	50.0	50.0
em3d	1.54	1.54	1.14	100	100	72.6
bh	.0176	.0111	.0111	100	100	100
mst	6.08	0	0	29.8	0	0
bisort	27.8	0	0	100	0	0

Table 3: Dynamic memory accesses of compression target fields (A_{target}) normalized to the total dynamic memory accesses, and dynamic accesses of compressible data normailzed to A_{target}. Upper: compression of recursive pointer fields. Middle: compression of integer fields using narrow bit-width. Bottom: compression of integer fields using dictionary.