Compressing Heap Data for Improved Memory Performance

Youtao Zhang
Department of Computer Science
The University of Texas at Dallas
Richardson, TX 75083
zhangyt@utdallas.edu

Rajiv Gupta
Department of Computer Science
The University of Arizona
Tucson, AZ 85721
gupta@cs.arizona.edu

Abstract

We introduce a class of transformations which modify the representation of dynamic data structures used in programs with the objective of *compressing* their sizes. Based upon a profiling study of data value characteristics, we have developed the *common-prefix* and *narrow-data* transformations that respectively compress a 32 bit address pointer and a 32 bit integer field into 15 bit entities. A pair of fields which have been compressed by the above compression transformations are packed together into a single 32 bit word. The above transformations are designed to apply to data structures that are *partially compressible*, that is, they compress portions of data structures to which transformations apply and provide a mechanism to handle the data that is not compressible. The accesses to compressed data are efficiently implemented by designing *data compression extensions* (DCX) to the processor's instruction set. We have observed average reductions in heap allocated storage of 25% and average reductions in execution time and power consumption of 30%. If DCX support is not provided the reductions in execution times fall from 30% to 18%.

Keywords

data compression, common prefix pointers, narrow data, data packing, DCX instructions, data memory footprint, cache misses.

1 Introduction

Over the last decade, the memory and CPU performance gap has become a major performance bottleneck in modern computer architectures. Cache has been proposed as an effective component to bridge this gap. Since cache is usually much smaller than the main memory and the user space, it is very important to make good use of the cache memory in order to achieve good performance. Traditional approaches to improve cache performance such as doubling the size, increasing the associativity from hardware, or rearranging data objects or data fields within objects by compilers, do not change the data density in the cache. However, as we will see, a large percentage of the bits stored in both the cache and the main memory are redundant. By removing these redundant bits, more data items can be kept in a cache of given size and alleviate the memory bottleneck by reducing the number of cache misses.

The user space is divided into three areas: stack, global and heap. The data structures allocated in different areas show different cache and memory accessing behavior. Those allocated in stack usually have much better performance than the rest. Data layout optimizations can be used to optimize the behavior of global and heap data accesses. Those allocated in the global data space, even if they have bad performance, can be optimized well by existing compilers. However, those allocated in the heap have bad cache behavior and they are hard to optimize at compile time since they are allocated dynamically. The focus of this research is mainly on the dynamic data structures.

We introduce a class of transformations which modify the representation of dynamically allocated data structures used in pointer intensive programs with the objective of *compressing* their sizes. The fields of a node in a dynamic data structure typically consist of both *pointer* and *non-pointer* data. Therefore we have developed the *common-prefix* and *narrow-data* transformations that respectively compress a 32 bit address pointer and a 32 bit integer field into 15 bit entities. A pair of fields which have been compressed can be packed into a single 32 bit word. As a consequence of compression, the memory footprint of the data structures is significantly reduced leading to significant savings in heap allocated storage requirements which is quite important for memory intensive applications. The reduction in memory footprint can also lead to significantly reduced execution times due to a reduction in data cache misses that occur in the transformed program.

An important feature of our transformations is that they have been designed to apply to data structures that are *partially compressible*. In other words, they compress portions of data structures to which transformations apply and provide a mechanism to handle the data that is not compressible. Initially data storage for a compressed data structure is allocated assuming that it is fully compressible. However, at runtime, when uncompressible data is encountered, additional storage is allocated to handle such data. Our experience with applications from *Olden* test suite demonstrates that this is a highly important feature because all the data structures that we examined in our experimentation were highly compressible, but none were *fully* compressible.

For efficiently accessing data in compressed form we propose *data compression extensions* (DCX) to a RISC-style ISA which consist of six simple instructions. These instructions perform two types of operations. First since we must handle partially compressible data structures, whenever a field that has been compressed is updated, we must *check* to see if the new value to be stored in that field is indeed compressible. Second when we need to make use of a compressed value in a computation, we must perform an *extract and expand* operation to obtain the original 32 bit representation of the value.

We have implemented our techniques and evaluated them. The DCX instructions have been incorporated into the MIPS like instruction set used by the `simplescalar` simulator. The compression transformations have been incorporated in the `gcc` compiler. We have also addressed other important implementation issues including the selection of fields for compression and packing. Our experiments with six benchmarks from the *Olden* test suite demonstrate an average space savings of 25% in heap allocated storage and average reductions of 30% in execution times and power consumption. The net reduction in execution times is attributable to reduced miss rates for L1 data cache and L2 unified cache and the availability of DCX instructions.

2 Profiling Study for Compression Scheme Design

Before the design of a dynamic and effective data compression technique, we need to profile programs and collect information that would guide us in the development of new compression techniques. In this section, such a framework is presented for profiling dynamically allocated data objects. The framework allows us to analyze the value characteristics and lifetime of the dynamically allocated objects. More specifically, the framework allows us to carry out and answer the following questions.

- What data structures should be compressed?
- How should they be compressed?
- When should they be compressed?

2.1 Type based profiling

Usually, a program contains a large number of objects of a given type and there are a number of fields within the given type. For example, all the nodes of a linked list are of the same structure with several fields: a pointer field to link the nodes together and some other data fields. Often there exists a significant degree of value similarity across the same fields from different nodes. Space requirements could be reduced by taking advantage of this similarity. However, a compression strategy that treats uniformly all fields in a type is too coarse and would miss many opportunities in practice. A new type-based profiling technique which collects the following information is proposed to solve this problem.

- The lifetime of each object and the total number of load and store accesses to this object are found through profiling. This information identifies the overall behavior of each dynamically allocated object.
- The value characteristics for each field of each type in the program are determined. This information is organized as the value range summary of all field instances. Value characteristic information can be collected at different granularity and it is possible to keep an additional list of most frequently used N values and their access frequencies.

To collect the above information, a straightforward approach is a complete instrumentation of all access points including the creation point at *malloc()*, the deletion point at *free()* and all load and store accesses.

Although it is easy to trace at the high level the type information of objects through *malloc()* and *free()* function call points, a high level variable access can be translated to either a register access and a memory access. Since we are only interested in memory accesses, instrumentation at a high level is thus insufficient. One possible approach is to trace the load and store accesses by modifying the code generation part of the compiler but it is too expensive. In this section we describe a type-based profiling framework which combines the use of high level instrumentation and lower level simulation.

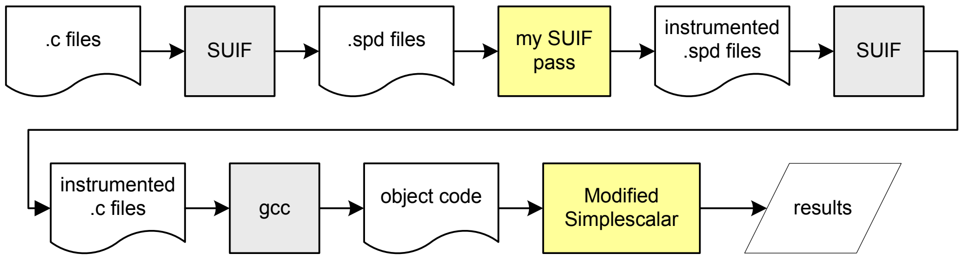


Figure 1: Type based profiling framework.

Figure 1 shows the framework which combines the use of SUIF 1.0 compiler [1] and Simplescalar simulator [2]. The original C programs are first converted to SUIF intermediate representations (IRs) by *scc*. A new pass is written to instrument these IRs and high level type of information is inserted into the instrumented code. The results of the new pass are still IRs and they are converted back to C programs by *s2c* (a conversion tool in SUIF). Then the instrumented version of C programs are compiled by *gcc* provided in Simplescalar and the MIPS-like executable code is generated. The Simplescalar simulator, which has been modified to process high level type information, is used to simulate the execution and collect profiles.

Since nearly all data items are accessed at word level, we only consider the accesses at word size level from now on. Although there are accesses to fetch double precision floating point values, double word-sized values, or subword level values, the overall percentage of such accesses is usually very small.

Next, let us discuss the experiments and their results in answering the *what*, *when* and *how* questions of data compression using this framework.

2.2 Compression Scheme Design

2.2.1 Selecting object types to compress

A program may contain multiple data types, each exhibiting different access patterns and different compressibility. The data types that an optimizing compiler should compress are those, by transforming which, positive impact on performance is expected. The fields in a data type should also be considered separately with the aim of maximizing benefit. Since different compression schemes might group fields differently, and thus affect the overall compressibility, this section will discuss how to separate and pick out different data structures for compression. The compression of different fields is left to subsequent sections.

As described, the collected profiles provide the information about the number of objects for each type and the information about the value ranges for all fields of a given type. Potential space savings can thus be calculated from this information. Experiments were conducted to identify the appropriate data types to compress in SPEC95int benchmark suite. Most programs from this benchmark suite have at least one of the following properties.

- There are a set of similar types and a *generic type*. The *generic type* is used to build up the data structure but each node is of a specific type. Object instances are accessed by *type casting* to a specific type. Programs 130.li and 126.gcc, which themselves are compilers, exhibit this property. These programs build up a syntax tree for each function and each node in the tree could be of a specific type (e.g., for an expression, a FOR statement, or an IF statement, etc).

- The program first allocates a large chunk and starting address of the node is recalculated (aligned) to a special address. Because of implicit address arithmetic, the compiler has difficulty in remapping the fields and their offsets. Program 124.m88ksim exhibits this property.

The above identified properties block further processing of SPEC95int benchmarks and we conclude that they are not good for automatic compression transformations.

The programs from Olden benchmark suites [3] were further studied. Olden is a pointer intensive benchmark suite (Table 1). Although the types of a program are divided into several groups, each group has only one type. The clear type isolation provides good opportunities for compiler-based compression. The main data structures that were considered for compression are given in Table 1.

Program	Application	Main data structure
`bh`	Barnes & Hut N-body force computation algorithm	Heterogeneous tree
`bisort`	Bitonic Sorting	Binary Tree
`health`	Columbian health care simulation	Doubly- linked lists
`mst`	Minimum spanning tree of a graph	Heterogeneous tree
`perimeter`	Perimeter of regions in images	Quad-tree
`treeadd`	Recursive sum of values in a balanced B-tree	Binary tree
`tsp`	Traveling salesman problem	Balanced binary tree
`voronoi`	Computes the voronoi diagram of a set of points	Balanced binary tree

Table 1: Olden Benchmark Summary.

2.2.2 Choosing the compression scheme

A major challenge in the design of a compression scheme is to balance the dynamic compression costs and the benefits of compression. A dynamic compression scheme should be simple and fast for most if not all of the accesses. If applying traditional compression techniques (e.g., a dictionary based approach or Huffman coding), there are at least two memory accesses: one to fetch the encoded data and the other to fetch the decoding data. As the cache and memory accesses are already the bottleneck and the major focus of applying compression dynamically is to reduce the total number of these accesses, these techniques are not appealing for dynamic compression. The new compression scheme that would be suitable to have in a dynamic environment should get all information about a value in one access for most if not all of the data accesses. Of course, subsequent computation to extract the decoded value might be inevitable. A logical comparison of the traditional and new compression schemes is shown in Figure 2.

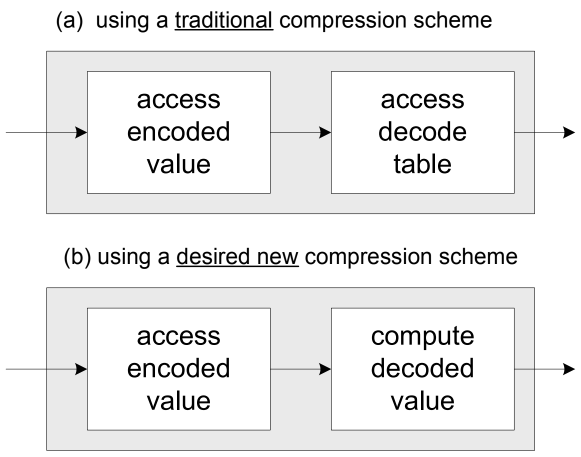

(a) using a traditional compression scheme
access encoded value
access decode table
(b) using a desired new compression scheme
access encoded value
compute decoded value

Figure 2: Access sequences with different compression schemes.

To design a compression scheme that can get all information from one memory access, no complicated encoding scheme should be used but rather we should discard directly the *redundant* bits from original word representation. The following two types of redundancy was identified (see Figure 3):

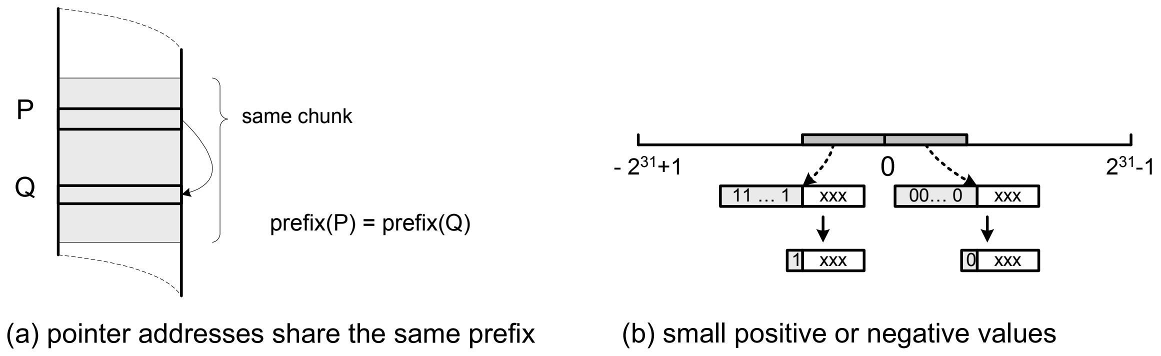

P
Q
same chunk
prefix(P) = prefix(Q)
- 2^31+1
0
2^31-1
11 … 1
xxx
00… 0
xxx
1
xxx
0
xxx
(a) pointer addresses share the same prefix
(b) small positive or negative values

Figure 3: Representing a 32-bit value with fewer than 32 bits.

- If a pointer is saved in a place that is close to the place it points to, the value of the pointer and the address of the pointer share the same prefix. Since the value is accessed always from its address, the prefix of the value can be considered as redundant as it can be constructed from its address easily. In this case, the prefix bits of the pointer can be safely discarded.

- If a value is close to zero, the higher order bits are sign extensions and they are either all 0s or all 1s. In either case, there is no need to remember all these identical bits and thus the prefix bits are considered as redundant. Only the sign bit should be remembered and the rest can be safely discarded.

With the compression opportunities identified, one could dynamically discard all redundant bits and use the least possible bits to represent a value, or discard some redundant bits but use fixed number of bits to represent a value. Since data values change dynamically, the former strategy would bring too much complexity to dynamic memory management and thus is not used. For the latter, we need to choose the fixed number of bits based upon the cost-benefit analysis of using this fixed number of bits to carry out compression.

Potential savings in space due to redundancy removal

The benefits of a dynamic compression scheme come from the *space savings* and the corresponding *cache miss reduction* due to the space savings. As a result, the benefit estimation is based upon space savings and the experiment is designed as follows. The original scheme always represents a word-sized value with 32 bits. The new scheme uses a fixed bit width L and if after removing the redundant prefix bits from a value's representation, the required number of bits is less that L, L bits are used to represent this value; otherwise, 32 bits are used to represent the value. The total number of bits required for representing values involved in all accesses for different values of L were collected.

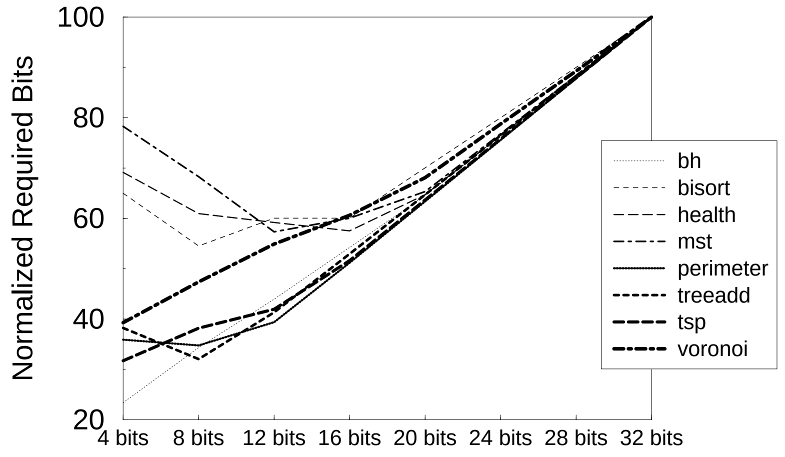


Figure 4: Required bits with fixed length.

The programs from the Olden suite with small inputs were executed to collect the profiling information. The results are shown in Figure 4 with the original required bits normalized as 100%. A smaller fixed length saves greater number of bits for each compressible value. However, if L is too small, the probability that a value will be represented using 32 bits is high. From the results in Figure 4, it can be seen that some benchmarks achieve the best profiling results at the point with fixed 4 bits while some achieve the best results with 8 or 16 bits. However, if the fixed length is bigger than 16 bits, the required bits increase almost linearly for all benchmarks.

Potential costs of redundancy removal

Let us now study the cost of dynamic compression. The real cost is implementation dependent and a precise estimation can only be done when both the compression scheme and the lower level architecture are all well defined. A coarse estimation is presented instead and it is sufficient to guide the design of the compression scheme. Since the benefits come from the compression of fields whose majority instances are compressible values, a successful compression scheme should speedup the accesses of compressible values; otherwise, the slowdown of majority accesses will significantly lower the overall performance. On the other hand, the accesses of incompressible values could be slower than those of compressible ones. Thus, the cost estimation is performed by answering the following question. If only accesses of incompressible values are slowed down, would the cost from accessing incompressible values be offset by the benefit obtained from accesses of compressible values? This in turn depends on the distribution of compressible and incompressible values. The results for Olden benchmark programs are shown in Figure 5. A memory access is considered to be a *fitting access* if its value can be represented by fixed 4, 8, 16 bits respectively. The results in Figure 5 show that more than half of the accesses could be expensive non-fitting accesses if using 4 bits. While with 16 bits, the percentages of fitting accesses are between 69% and 99%. As a result, *16-bit* is a cost-effective point and a good candidate to use.

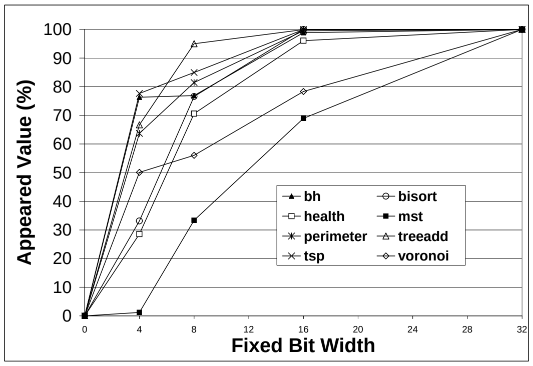


Figure 5: Distribution of values with fixed length.

Dynamic values change frequently and it is expensive to convert a compressible value to an incompressible one and *vice versa*. With dynamic expansion of values considered, a study of the benchmarks has been done

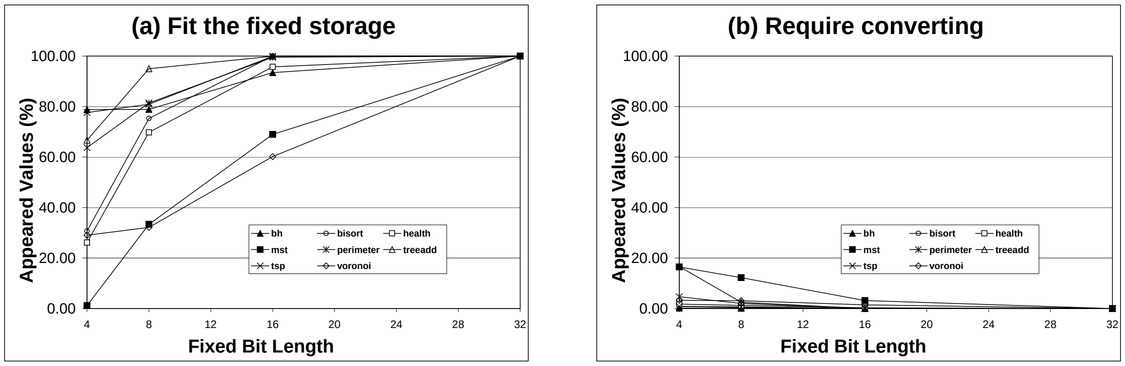


Figure 6: Distribution of values with fixed storage.

from the storage point of view and the results are shown in Figure 6. All values are initially allocated with fixed lengths, 4 bits, 8 bits and 16 bits respectively. If a value changes from compressible to incompressible, it gets expanded and stays as an incompressible value from then onwards. Therefore later accesses will be fitting-accesses even if they are accessing the incompressible value. The results show that with a fixed 16-bit representation, the majority memory accesses fit this length and dynamic conversion is very infrequent.

From the above analysis and the results in estimated costs and benefits, we conclude that a well-balanced compression scheme should represent a 32 bits value with fixed 16 bits, allow dynamic expansion, but not allow dynamic shrinking.

2.2.3 Choosing the time for compression

The next problem to be considered is that of determining *when* these types should be compressed. The following three possible schemes are studied.

- Complete compression at the beginning. The simplest scheme is to compress all fields of a type at the beginning. If implementing in a compiler, it means that the memory layout of the type is reduced to half of its original size at compile time. It has the simplicity that the offset of each fields is known at compile time and code generation is therefore simplified.

- Selective compression at the beginning. Since different fields exhibit different compression opportunities and some fields such as floating point value fields are general hard to compress. Consider the cost the program has to pay at runtime to access incompressible fields, it is more preferable to compress only those highly compressible fields and leave the rest as they are. This scheme still has the property that the offsets of the fields are known at compile time.

- Compression after last write. Since the compressibility of a value can only be changed by a write operation, after the last write of a field, its representation is fixed and more aggressive compression scheme can be used and we do not have to worry that the values might change later. This completely eliminates the cost that a compression scheme has to pay to handle conversions from compressible values to incompressible ones.

Figure 7 shows the experimental results in studying the Olden benchmark programs under different compression time. For the selective compression scheme at the beginning, a field is chosen if 80% of its instances are compressible. The x axis of the figure shows the execution time which has been normalized to 5000 units. The y axis shows the required heap space during the execution. "free()" is not considered during program execution and thus the memory requirements increase continuously and decrease to zero at the end of the execution.

The scheme that compresses an object after its last write can remove the dynamic conversion cost from compressible values to incompressible ones. It can also achieve best space savings for 4 out of 8 programs. Although it looks like an appealing approach, it is difficult to apply in practice. Usually at some program point, only a small number of nodes in a data structure are modified. However, the whole data structure might be traversed and nodes are dynamically selected for modification. It would be very expensive, and sometimes impossible, to predict the last write to a particular node.

Comparing the two schemes that compress data items at the beginning, the results show that although compressing all the fields achieves more space reduction at the beginning, it requires more data expansion during program execution. For example let us assume there are 1000 nodes each of which contains four fields. Two are highly compressible – 90% of them are compressible; the other two are almost uncompressible - 5% of them are compressible. At the beginning of the program execution, all fields contain zeros. Using the scheme that compresses all fields at the beginning, we need (1000_nodes $\times$ 4_fields/node $\times$ 2_words/4_fields) = 2000 words. However, as the program runs, we need 1000_nodes $\times$ 4_fields/node $\times$ ((90% $\times$ 1_word/2_fields + 10% $\times$ 3_words/2_fields) $\times$ compressible_2_fields/4_fields+ (5% $\times$ 1_word/2_fields + 95% $\times$ 3_words/2_fields) $\times$ other_2_fields/4_fields) = 4100 words. On the other hand, if we only compress two highly compressible fields, we need 1000_nodes $\times$ 4_fields/node $\times$ ((90% $\times$ 1_word/2_fields + 10% $\times$ 3_words/2_fields) $\times$ compressible_2_fields/4_fields+ (100% $\times$ 2_word/2_fields) $\times$ other_2_fields/4_fields) = 3200 words. Thus the scheme that compresses all fields requires more space than the selective compression scheme.

Moreover, the results show that selecting only highly compressible data fields for compression reduces both the space requirements and dynamic costs of accessing incompressible values. Overall, the scheme that compresses selected fields at the beginning achieves the best result for 5 out of 8 programs and almost the best for another 2 programs.

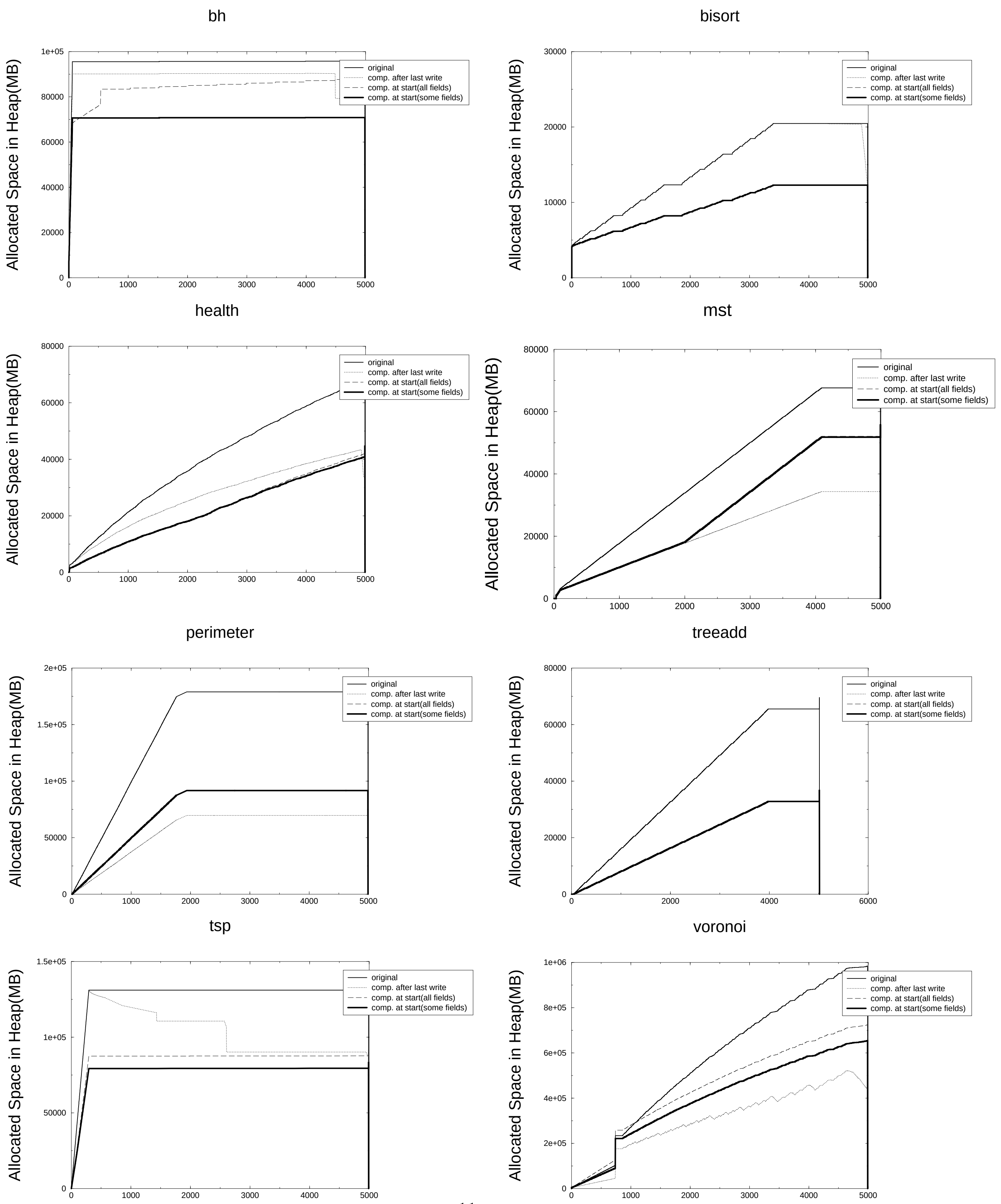


Figure 7: Deciding the time for compression.

3 Profile-guided data compression transformations

3.1 Data compression transformation

Based upon the results of experimental study in the preceding section we use the following strategy. We decided to compress all objects of a given type if there was no address arithmetic and no type casting. Promising fields were selected for compression at the beginning of the execution. A 32-bit value is represented using fixed 16 bits and while dynamic expansion is allowed, dynamic shrinking is not allowed. We will exploit two types of value representation redundancy: common prefix redundancy of pointer addresses and sign extension redundancy in small values. In the example below, a pointer field and a small value field are packed into a single 32-bit field *value_next*.

```
Original Structure:                  Transformed Structure:
  struct list_node {                     struct list_node {
    ···;                                         ···;
    int value;                                   int value_next;
    struct list_node *next;              } *t;
  } *t;
```

In this way, 4 bytes are saved from each node in the linked list. Although indicated by a type re-declaration, this transformation is not done at the source level and there is no need to generate the new declaration at source code level. Instead the optimizing compiler will change the memory layout before code generation and then generate new code sequences accordingly. As we see, there are two types of fields: pointer addresses and small value fields. They are handled differently through two types of data compression transformations.

Common-prefix transformation for pointer data. The pointer contained in the *next* field of the link list can be compressed under certain conditions. In particular, consider the addresses corresponding to an instance of *list_node* (`addr1`) and the *next* field in that node (`addr2`). If the two addresses share a common 17 bit prefix because they are located fairly close in memory, the *next* pointer is classified as compressible. In this case the common prefix from address `addr2` which is stored in the `next` pointer field is eliminated. The lower order 15 bits from `addr2` represent the representation of the pointer in compressed form. The 32 bit representation of a *next* field can be reconstructed when required by obtaining the prefix from the pointer to the *list_node* instance to which the *next* field belongs.

Narrow data transformation for non-pointer data. Now let us consider the compression of the narrow width integer value in the *value* field. If the 18 higher order bits of this value are identical, that is, they are either all 0's or all 1's, it is classified as compressible. The 17 higher order bits are discarded and leaving a 15 bit entity. Since the 17 bits discarded are identical to the most significant order bit of the 15 bit entity, the 32 bit representation can be easily derived when needed by replicating the most significant bit.

Packing together compressed fields. The *value* and *next* fields of a node belonging to an instance of *list_node* can be packed together into a single 32 bit word as they are simply 15 bit entities in their compressed form. Together they are stored in *value_next* field of the transformed structure. The 32 bits of *value_next* are divided into two half words. Each compressed field is stored in the lower order 15 bits of the corresponding half word. According to the above strategy, bits 15 and 31 are not used by the compressed fields.

To select fields for compression, the candidate fields are classified from the type-based profiling described in the previous section. A field is identified to be *highly compressible* if 90% of the fields instances are compressible. A pointer value is considered as compressible if it shares the 17 bits prefix with its address and a small value is considered as compressible if the higher order 18 bits are the same.

A critical issue is that of pairing compressed fields for packing into a single word. Based on the profiling information, fields are further categorized into *hot* fields and *cold* fields. With all categorized fields, there are two choices in packing. It is possible to pack *two hot fields* together if they are typically accessed in tandem. This is because in this situation a single load can be shared while reading the two values. It is also useful to compress any *two cold fields* even if they are not accessed in tandem. This is because even though they cannot share the same load, they are not accessed frequently. In all other situations it is not as useful to pack data together because even though space savings will be obtained, execution time will be adversely affected.

Next the handling of incompressible data in partially compressible data structures is described. The implementation of partially compressible data structures requires an additional bit for encoding information. This is why fields are compressed down to 15 bit entities and not into 16 bit entities.

Partial compressibility. The basic approach is to allocate only enough storage to accommodate a compressed node when a new node in the data structure is created. Later, as the pointer fields are assigned values, it is checked to see if the fields are compressible. If they are, they can be accommodated in the allocated space; otherwise additional storage is allocated to hold the fields in uncompressed form. The previously allocated location is now used to hold a pointer to this additional storage. Therefore for accessing incompressible fields the approach has to go through an extra step of indirection.

If the incompressible data stored in the fields is modified, it is possible that the fields may now become compressible. However, such checks are not carried out and instead the fields in such cases are left in uncompressed form. This is because exploitation of such compression opportunities can lead to repeated allocation and deallocation of extra locations if data values repeatedly keep oscillating between the compressible and incompressible kind. To avoid repeated allocation and deallocation of extra locations the approach is simplified so that once a field is assigned an incompressible value, from then on wards, the data in the field is always maintained in uncompressed form.

The most significant bit (bit 31) in the word is used to indicate whether or not the data stored in the word is compressed or not. It contains a 0 to indicate that the word contains compressed values. If it contains a 1, it means that one or both of values were not compressible and instead the word contains a pointer to an extra pair of dynamically allocated locations which contain the values of the two fields in uncompressed form. While bit 31 is used to encode extra information, bit 15 is never used for any purpose.

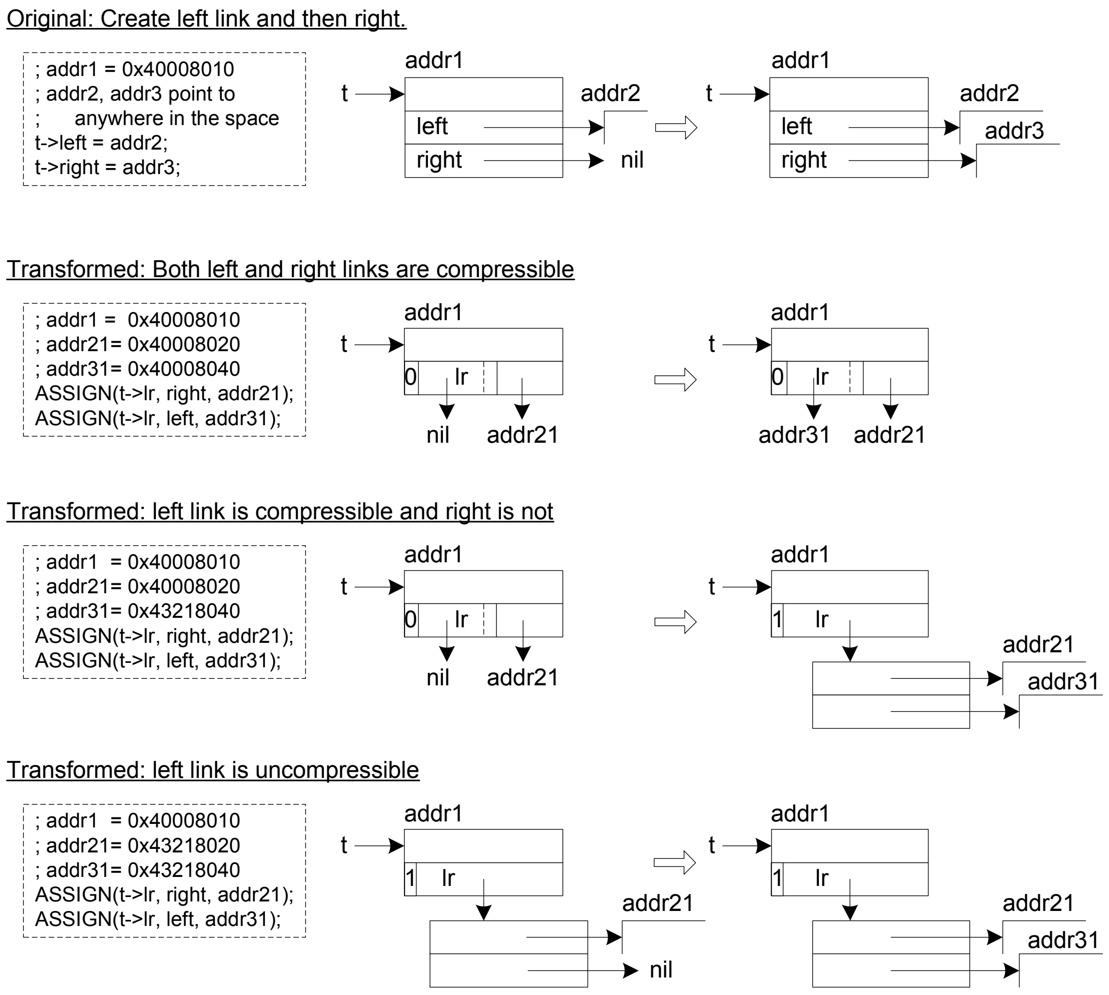
The example in Figure 8 illustrates the above method using an example in which an instance of *list_node* is allocated and then the *value* and *next* fields are set up one at a time. As we can see first storage is allocated to accommodate the two fields in compressed form. As soon as the first incompressible field is encountered additional storage is allocated to hold the two fields in uncompressed form. Under this scheme there are three possibilities which are illustrated in Figure 10. In the first case both fields are found to be compressible and therefore no extra locations are allocated. In the second case the *value* field, which is accessed first, is compressible but the *next* field is not. Thus, initially *value* field is stored in compressed form but later when the *next* field is found to be incompressible, extra locations are allocated and both fields are store in uncompressed form. Finally in the third case the *value* field is not compressible and therefore extra locations are allocated right away and none of the two fields are ever stored in compressed form.

Applicability. Similar to object layout optimization techniques, data compression transformations need to rearrange the fields in an object. A basic assumption of object layout transformations states that it is ensured by the programmer that application of layout transformations are safe (program correctness is ensured). Generally, if there is no address arithmetic and fields are accessed from their names, the assumption can be satisfied. Starting from this assumption, the optimizing compiler automatically transform the data types and generate corresponding code.

Memory allocation. `Ccmalloc` [4], a modified version of `malloc`, is used to carry out *storage allocation*. This form of storage allocation was developed by Chilimbi et al. [4] and as described earlier it improves the locality of dynamic data structures by allocating the linked nodes of the data structure as close to each other as possible in the heap. Compared to system *malloc*, it has one more pointer parameter which indicates the parent node of the new node. The new node is allocated in the same cache line chunk as it parent node if there are still enough space to hold the new one. Otherwise, a new chunk is allocated. As a consequence, this technique increases the likelihood that the pointer fields in a given node will be compressible. Therefore it makes sense to use `ccmalloc` in order to exploit the synergy between `ccmalloc` and data compression.

3.2 Instruction set support

Compression reduces the amount of heap allocated storage used by the program which typically improves the data cache behavior. Also if both the fields need to be read in tandem, a single load is enough to read


Original: Create left link and then right.
Transformed: Both left and right links are compressible
Transformed: left link is compressible and right is not
Transformed: left link is uncompressible
addr1
t
left
right
addr2
addr3
nil
lr
0
1
addr21
addr31

```
; addr1 = 0x40008010
; addr2, addr3 point to
;      anywhere in the space
t->left = addr2;
t->right = addr3;
```

```
; addr1 =  0x40008010
; addr21= 0x40008020
; addr31= 0x40008040
ASSIGN(t->lr, right, addr21);
ASSIGN(t->lr, left, addr31);
```

```
; addr1  = 0x40008010
; addr21= 0x40008020
; addr31= 0x43218040
ASSIGN(t->lr, right, addr21);
ASSIGN(t->lr, left, addr31);
```

```
; addr1  = 0x40008010
; addr21= 0x43218020
; addr31= 0x43218040
ASSIGN(t->lr, right, addr21);
ASSIGN(t->lr, left, addr31);
```

Figure 8: Dealing with incompressible data (ASSIGN() is a MARCO that assigns a given value (thrid parameter) to either left or right part (the second paramter) of a compressed field (the first parameter)).

both the fields. However, the manipulation of the fields also creates additional overhead. To minimize this overhead new RISC-style instructions are designed. Six simple instructions have been designed of which three each are for pointer and non-pointer data respectively that efficiently implement common-prefix and narrow-data transformations. The semantics of the these instructions are summarized in Figure 9. These instructions are RISC-style instructions with complexity comparable to existing branch and integer ALU instructions. Let us discuss these instructions in greater detail.

Checking compressibility. Since we would like to handle partially compressible data, before actually compressing a data item at runtime, first a check is made to determine whether the data item is compressible. Therefore the first instruction type that is introduced allows efficient checking of data compressibility. Two new instructions have been designed and they are described below. The first checks the compressibility of pointer data and the second does the same for non-pointer data.

`bneh17 R1, R2, L1` – is used to check if the higher order 17 bits of `R1` and `R2` are the same. If they are the same, the execution continues and the field held in `R2` can be compressed; otherwise the branch is taken to a point where we handle the situation, by allocating additional storage, in which the address in `R2` is not compressible. A nil pointer is represented by the value 0 in both compressed and uncompressed forms. To distinguish the compressed nil pointer from an allocated address whose lower order 17 bits are zeros, we modified our `malloc` routine so that it never allocates storage locations with such addresses to compressed fields.

`bneh18 R1, L1` – is used to check if the higher order 18 bits of `R1` are identical (i.e., all 0's or all 1's). If they are the same, the execution continues and the value held in `R1` is compressed; otherwise the value in `R1` is not compressible and the branch is taken to a point where we place code to handle this situation by allocating additional storage.

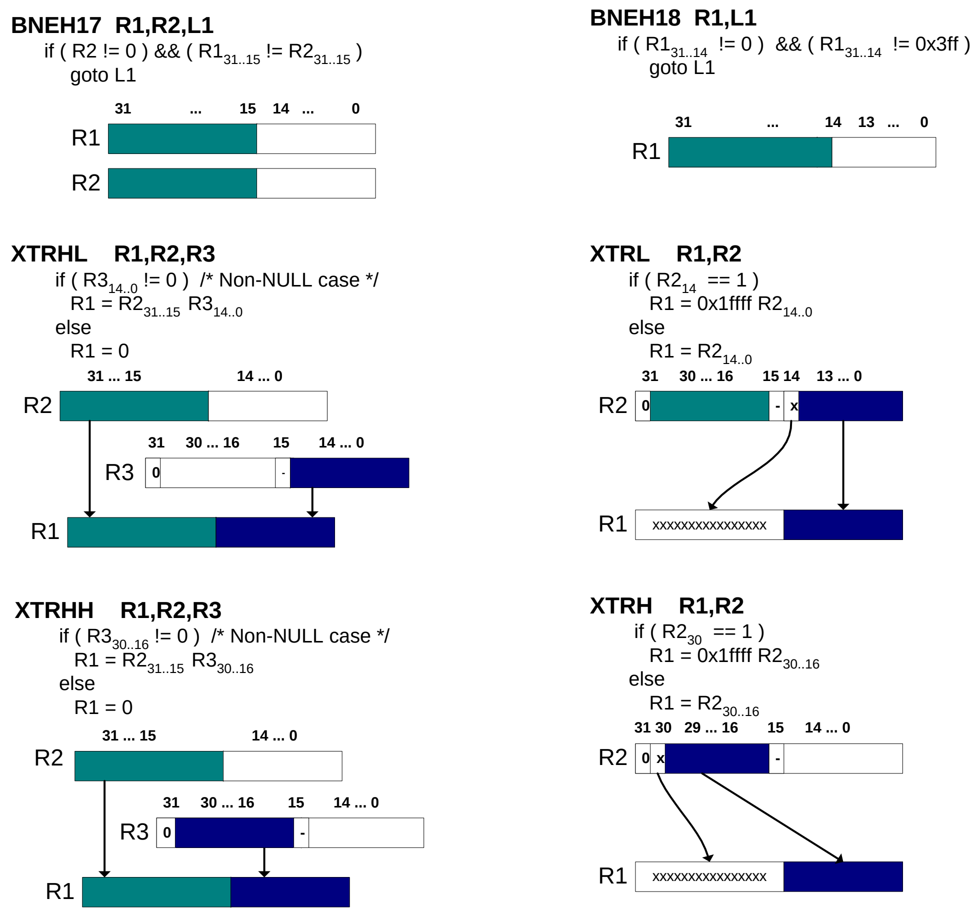


Figure 9: DCX instructions.

Extract-and-expand. If a pointer is stored in compressed form, before it can be dereferenced, its 32-bit representation must be reconstructed. Compressed non-pointer data should be handled similarly before its use. Therefore the second instruction type that is introduced carries out extract-and-expand operations. There are four new instructions that we describe below. The first two instructions are used to extract-and-expand compressed pointer fields from lower and upper halves of a 32-bit word respectively. The next two instructions do the same for non-pointer data.

`xtrhl R1, R2, R3` – extracts the compressed pointer field stored in lower order bits (0 through 14) of register `R3` and appends it to the common-prefix contained in higher order bits (15 through 31) of `R2` to construct the uncompressed pointer which is then made available in `R1`. The case when `R3` contains a nil pointer is also handled. If the compressed field is a nil pointer, `R1` is set to nil.

`xtrhh R1, R2, R3` – extracts the compressed pointer field stored in the higher order bits (16 through 30) of register `R3` and appends it to the common-prefix contained in higher order bits (15 through 31) of `R2` to construct the uncompressed pointer which is then made available in `R1`. If the compressed field is a nil pointer, `R1` is set to nil.
The instructions `xtrhl` and `xtrhh` can also be used to compress two fields together. However, they are not essential for this purpose because typically there are existing instructions which can perform this operation. In the MIPS like instruction set that was used in this work this was indeed the case.

`xtrl R1, R2` – extracts the field stored in lower half of the `R2`, expands it, and then stores the resulting 32 bit value in `R1`.

`xtrh R1, R2` – extracts the field stored in the higher order bits of `R2`, expands it, and then stores the resulting 32 bit value in `R1`.

Next a simple example is given to illustrate the use of the above instructions. Let us assume that an integer field $t \rightarrow value$ and a pointer field $t \rightarrow next$ are compressed together into a single field $t \rightarrow value\_next$. In Figure 10(a) it is shown how compressibility checks are used prior to appropriately storing *newvalue* and *newnext* values in to the compressed fields. In Figure 10(b) we illustrate the extract and expand instructions by extracting the compressed values stored in $t \rightarrow value\_next$. In the code `sh` is an existing MIPS instruction that stores the least-significant 16 bits of the register to an memory address.

3.3 Without instruction set support

The compaction overhead comes from checking, extracting and possible expanding of data fields at runtime. If DCX instructions are not provided, we can still perform the discussed data compression transformations but

```
     ; $16 :  &t-> value_next
     ; $18 :  newvalue
     ; $19 :  newnext

     ; branch if newvalue is not compressible
     bneh18   $18, $L1
     ; branch if newnext is not compressible
     bneh17   $16, $19, $L1
     ; store compressed data in t-> value_next
     ori      $19, $19, 0x7fff
     sh       $18, 0($16)
     sh       $19, 2($16)
     j        $L2
$L1:
     ; allocate extra locations and store pointer
     ; to extra locations in t-> value_next
     ; store uncompressed data in extra locations
     ...
$L2: ...
```

(a) Illustration of compressibility checks.

```
     ; $16:  &(t-> value_next)
     ; $17:  uncompressed integer t-> value
     ; $18:  uncompressed pointer t-> next

     ; load contents of t-> value_next
     lw       $3,0($16)
     ; branch if $3 is a pointer to extra locations
     bltz     $3,  $L1
     ; extract and expand t-> value
     xtrl     $17, $3
     ; extract and expand t-> next
     xtrhh    $18, $16, $3
     j        $L2
$L1:
     ; load values from extra locations
     ...
$L2: ...
```

(b) Illustration of extract and expand instructions.

Figure 10: An example.

expect greater overhead. The instructions executed when DCX support is not available is shown in Figure 11. In comparison to the DCX based code in Figure 10, the main difference is that the DCX instructions are expanded into multiple instructions that implement their logic which was presented in Figure 9. In the new code we use more registers (i.e., \$2 and \$20) to store intermediate results. This sometimes causes additional spill code to be generated if we run out of available registers. While there are several branch instructions generated in this code, as shown by the experiments presented later, we did not see large performance degradation due to bad branch prediction.

4 Experimental Results

4.1 Experimental setup

Techniques described have been implemented to evaluate their performance. The transformations have been implemented as part of the `gcc` compiler and the DCX instructions have been incorporated in the MIPS like instruction set of the superscalar processor simulated by `simplescalar` [2]. The evaluation is based upon six benchmarks taken from the *Olden* test suite which contains pointer intensive programs that make extensive use of dynamically allocated data structures. We used "-O3" compiler option when generating the object code for simulation. We did not apply the technique to SPECint benchmarks due to the limitations described in section 2.2.1.

In order to study the impact of memory performance we varied the input sizes of the programs and also varied the L2 cache latency. The programs were run for three input sizes – small (this is the standard input that is typically used to run the benchmark), medium and large (see Figure 12(a)). The cache organization of simplescalar is shown in Figure 12(b). There are first level separate instruction and data caches (I-cache and D- cache). The lower level cache is a unified-cache for instructions and data. The L1 cache used was a 16K direct mapped cache with 9 cycle miss latency while the unified L2 cache is 256K with 100/200/400 cycle miss latencies. Our experiments are for an out-of-order issue superscalar with issue width of 4 instructions and the *Bimod* branch predictor.

4.2 Impact on storage needs

The transformations applied for each program and their impacts on node sizes are shown in Figure 13. In the first four benchmarks (`treeadd`, `bisort`, `tsp`, and `perimeter`), node sizes are reduced by storing pairs of compressed pointers in a single word. In the `health` benchmark a pair of small values are compressed together and stored in a single word. Finally, in the `mst` benchmark a compressed pointer and a compressed small value are stored together in a single word. The changes in node sizes range from 25% to 33% for five of the benchmarks. Only in case of `tsp` is the reduction smaller – just over 10%.

The runtime savings in heap allocated storage are measured for each of the three program inputs. The

```
        ; $16 :  &t−>value_next
        ; $18 :  newvalue
        ; $19 :  newnext
        ; $20 :  0xffffc000 preloaded

        and  $2,  $18, $20
        beq  $2,  $0, $L1
        beq  $2,  $20, $L1 ; finish the logic of bneh18
        beq  $19, $0, $L1
        or   $2,  $16, $19
        srl  $2,  $2, 15
        bne  $2,  $L1 ; finish the logic of bneh17
        ori  $19, $19, 0x7fff
        sh   $18, 0($16)
        sh   $19, 2($16)
        j    $L2
   $L1:
        ; allocate extra locations and store pointer
        ; to extra locations in t−>value_next
        ; store uncompressed data in extra locations
        ...
   $L2:  ...
```

(a) Illustration of compressibility checks.

```
     ; $16:  &(t−>value_next)
     ; $17:  uncompressed integer t−>value
     ; $18:  uncompressed pointer t−>next
     ; $20 :  0xffffc000 preloaded

     lw   $3,  0($16)
     bltz $3,  $L1
     andi $2,  $3, 0x4000
     beq  $2,  $0, $L3
     addi $17, $3, 0x3fff
     j    $L4
$L3: or   $17, $3, $20 ; finish the logic of xtrl
$L4: srl  $2,  $3, 16
     bne  $2,  $0, L5
     mov  $18, $0,
     j    $L2
$L5  and  $18, $3, $20
     ori  $18, $18, $2 ; finish the logic of xtrhh
     j    $L2
$L1:
     ; load values from extra locations
     ...
$L2:  ...
```

(b) Illustration of extract and expand instructions.

Figure 11: The code without DCX instructions for the previous example.

Program	Application
`treeadd`	Recursive sum of values in a B-tree
`bisort`	Bitonic Sorting
`tsp`	Traveling salesman problem
`perimeter`	Perimeters of regions in images
`health`	Columbian health care simulation
`mst`	Minimum Spanning tree of a graph

Program	small input	medium input	large input
`treeadd`	20 1	21 1	22 1
`bisort`	32768 1	128000 1	312000 1
`tsp`	65536 1	131072 1	262144 1
`perimeter`	12 1	13 1	14 1
`health`	3 2000 1	3 3000 1	3 4000 1
`mst`	512 1	1024 1	2048 1

(a) Benchmarks and inputs used.

Parameter	Value
Issue Width	4 issue, out of order
Instruction cache	16K direct map
Icache miss latency	9 cycles
Level 1 data cache	16K direct map
Level 1 data cache miss latency	9 cycles
Level 2 unified cache	256K 2-way asso.
Memory latency (level 2 cache miss latency)	100/200/400 cycles

(b) Cache configurations used.

Figure 12: Experimental setup.

Program	Transformation Applied	Node Size Change (bytes)
`treeadd`	CommonPrefix/CommonPrefix	from 28 to 20
`bisort`	CommonPrefix/CommonPrefix	from 12 to 8
`tsp`	CommonPrefix/CommonPrefix	from 36 to 32
`perimeter`	CommonPrefix/CommonPrefix	from 12 to 8
`health`	NarrowData/NarrowData	from 16 to 12
`mst`	CommonPrefix/NarrowData	from 16 to 12

Figure 13: Applied transformations.

results are given in Figures 14(a-c). The average savings are nearly 25% while they range from 10% to 33% across different benchmarks. Even more importantly these savings represent significant levels of heap storage – typically in megabytes. For example, the 20% storage savings for `tsp` represents 1.0 Mbytes, 2.0 Mbytes, and 3.9 Mbytes of heap storage savings for small, medium and large program inputs respectively. It should also be noted that such savings cannot be obtained by other locality improving techniques described earlier [5, 6, 7, 4, 8].

From the results in Figure 14(a-c) another very important observation is made. The *extra locations* allocated when non-compressible data is encountered are non-zero for all of the benchmarks. In other words we observe that for none of the data structures to which our compression transformations were applied, were all of the instances of the data encountered at runtime actually compressible. A small amount of additional locations were allocated to hold a small number of incompressible pointers and small values in each case. Therefore the generality of our transformation which allows handling of *partially compressible* data structures is extremely important. If the application of compression was restricted to data fields that are always guaranteed to be compressible, no compression would have been achieved and therefore no space savings would have resulted.

The increase in code size caused by compression transformations was also measured (see Figures 15). The increase in code size prior to linking is significant while after linking the increase is very small since the user code is small part of the binaries. However, the reason for significant increase in user code is because each time a compressed field is updated, our current implementation generates a new copy of the additional code for handling the case where the data being stored may not be compressible. In practice it is possible to share this code across multiple updates. Once such sharing has been implemented, the increase in the size of user code will also be quite small.

4.3 Impact on execution time

Based upon the cycle counts provided by the simplescalar simulator we studied the changes in execution times resulting from compression transformations. The impact of input size and L2 latency on execution times was also studied. The execution time in number of cycles for the scheme using original malloc is shown

Program	Storage (bytes)		
	Original	Compressed nodes + Extra locations = Total	Space savings
`treeadd`	12582900	8388600 + 13440 = 8402040	33.2 %
`bisort`	786420	524280 + 25600 = 549880	30.1 %
`tsp`	5242840	4194272 + 6080 = 4200352	19.9 %
`perimeter`	4564364	3260260 + 5120 = 3265380	28.5 %
`health`	566872	509952 + 320 = 510272	10.0 %
`mst`	3414020	2367492 + 320 = 2367812	30.6 %
`average`			25.4 %

(a) Reduction in heap storage for small input.

Program	Storage (bytes)		
	Original	Compressed nodes + Extra locations = Total	Space savings
`treeadd`	25165812	16777208 + 26560 = 16803768	33.2 %
`bisort`	3145716	2097144 + 136320 = 2233464	29.0 %
`tsp`	10485720	8388576 + 12160 = 8400736	19.9 %
`perimeter`	9322572	6658980 + 10560 = 6669540	28.5 %
`health`	847584	762348 + 320 = 762668	10.0 %
`mst`	13643780	9453572 + 320 = 9453892	30.7 %
`average`			25.2 %

(b) Reduction in heap storage for medium input.

Program	Storage (bytes)		
	Original	Compressed nodes + Extra locations = Total	Space savings
`treeadd`	50331636	33554424 + 51260 = 33605684	33.2 %
`bisort`	3145716	2097144 + 204160 = 2301304	26.8 %
`tsp`	20971480	16777184 + 23040 = 16800224	19.9 %
`perimeter`	20332620	14523300 + 23680 = 14546980	28.5 %
`health`	1128240	1014804 + 320 = 1015124	10.0 %
`mst`	54550532	37781508 + 320 = 37781828	30.7 %
`average`			24.9 %

(c) Reduction in heap storage for large input.

Figure 14: Impact on storage.

Program	Code Size (bytes)		
	Original	Transformed	Increase
`treeadd`	5480	6376	16.4%
`bisort`	11944	16720	40.0%
`tsp`	18280	19172	4.9%
`perimeter`	14976	18160	21.3%
`health`	15952	21324	33.7%
`mst`	12768	14136	10.7%
`average`			21.1%

(a) Code size before linking.

Program	Code Size (bytes)		
	Original	Transformed	Increase
`treeadd`	228360	228444	0.04%
`bisort`	257552	257572	0.01%
`tsp`	238004	238448	0.18%
`perimeter`	233736	238340	1.97%
`health`	256608	257200	0.23%
`mst`	232296	232440	0.06%
`average`			0.41%

(b) Code size after linking.

Figure 15: Impact on object code size.

in Figure 16. Let us examine the results in Figure 17(a) – these results are for L2 cache latency of 100 cycles. The reduction in execution times in comparison to the original programs which use `malloc` range from 3% to 64% while on an average the reduction in execution time is around 30%. The reductions in execution times increase gradually with the input size.

The execution times are compared with versions of the programs that use `ccmalloc`. The new approach outperforms `ccmalloc` in five out of the six benchmarks (our version of `mst` runs slightly slower than the `ccmalloc` version). On an average it outperforms `ccmalloc` by nearly 10%. Our approach outperforms `ccmalloc` because once the node sizes are reduced, typically greater number of nodes fit into a single cache line leading to a low number of cache misses. Additional runtime overhead is incurred in form of extra instructions needed to carry out compression and extraction of compressed values. However, this additional execution time is more than offset by the time savings resulting from reduced cache misses; thus leading to overall reduction in execution time.

The experiments of Figure 17(a) were also repeated for higher L2 cache latencies. The results are presented in Figures 17(b-c). As the latency of L2 cache is increased, compression outperforms `ccmalloc` by a greater extent. The graph in Figure 17(d) plots the average reduction in execution time that compression provides over `ccmalloc` for the different cache latencies. As it can be seen, on an average, compression reduces the execution times by 10%, 15%, and 20% over `ccmalloc` for L2 cache latencies of 100, 200, and 400 cycles respectively. These numbers also improve gradually with input size. In summary our approach provides large storage savings and significant execution time reductions over `ccmalloc`.

Program	Input Size	Memory Latency		
		100 cycles	200 cycles	400 cycles
`treeadd`	small	187104548	213446048	266129161
	medium	374255422	426980319	532430305
	large	748581997	854125025	1065211532
`bisort`	small	88018305	110354193	155027555
	medium	220326074	267876903	362982855
	large	1086856767	1403106753	2035623908
`tsp`	small	1009746602	1068975750	1187436916
	medium	2058288126	2196364134	2472525949
	large	4187156823	4502658588	5133680969
`perimeter`	small	63872531	76008631	100280831
	medium	136728007	161660850	211523707
	large	310945412	365539012	474726218
`health`	small	480405345	810592426	1471076555
	medium	1626046228	2940667674	5570065464
	large	3482298578	6435455814	12341950613
`mst`	small	224191291	365504051	648130202
	medium	957099915	1580381682	2826953231
	large	3980486935	6501985456	11545026712

Figure 16: Execution time in cycle counts for different configurations.

4.4 Impact on power consumption

Experiments have also been done to compare the power consumption for the compression based programs with that of the original programs and `ccmalloc` based programs (see Figures 18(a-d)). These measurements are based upon the `Wattch` [9] system which is built on top of the `simplescalar` simulator. These results track the execution time results quite closely. The average reduction in power consumption over the original programs is around 30% which increases gradually with the size of the input. The graph in Figure 17(d) plots the average reduction in power dissipation that compression provides over `ccmalloc` for the different cache latencies. As we can see, on an average, compression reduces the power dissipation by 5%, 10%, and 15% over `ccmalloc` for L2 cache latencies of 100, 200, and 400 cycles respectively. These numbers further improve gradually as the input size is increased.

4.5 Impact on cache performance

Figure 19 presents the impact of compression on cache behavior, including I-cache, D-cache and unified L2 cache behaviors. As expected, the I-cache performance is degraded due to increase in code size caused by our current implementation of compression. Due to the change of code layout, in some case, the modified code reflects smaller number of I-cache misses e.g. TSP. This is consistent with that from ccmalloc. However, the performances of D-cache and unified cache are significantly improved. This improvement in data cache performance is a direct consequence of compression.

4.6 Impact on performance without DCX ISA support

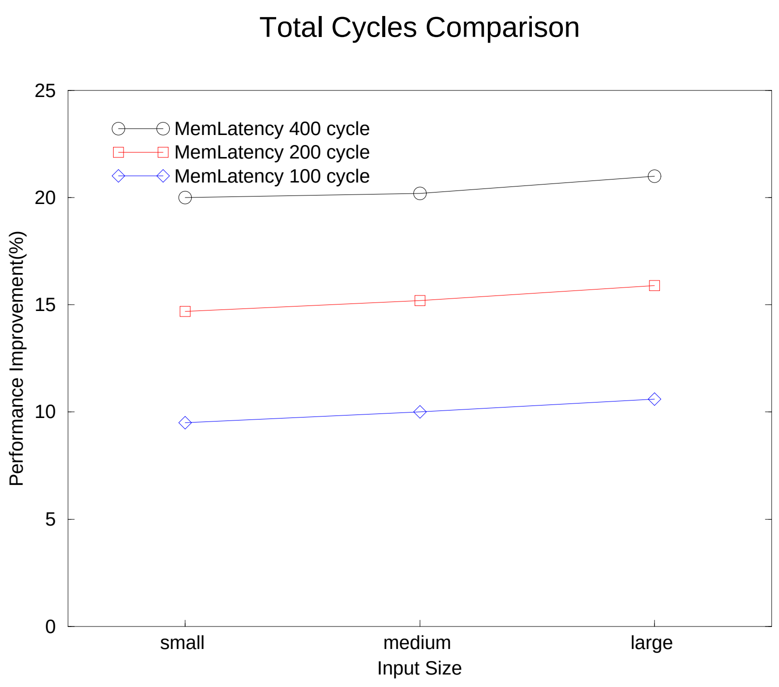
Figure 20 presents the impact on performance if DCX ISA support is not provided. As expected the performance improvment is eroded by the overhead that we have to pay for checking, compacting and extracing data items at the runtime. For example, with small input and memory latency 100 cycles we observed an average of 18% improvement over the orignal malloc scheme and 3% slowdown over *ccmalloc*. Luckily this overhead is offset by the performance gains from larger inputs and long memory latencies (which represents the trend). With medium input and memory latency 200 cycles, we have an improvment of 5.5% over `ccmalloc` compared to 14.7% improvemntment over `ccmalloc` if DCX instructions are provided.

4.7 Comparison with more cache configurations

Experiments have also been conducted to study the performance cahnges with different cache sizes and associativity. Figure 21 presents the results for medium input when repectively the L2 cache size is doubled, or the L1/L2 cache associativity is doubled, or L2 hit latency is doubled. From the figure we see comparable results as that in Figure 19. This is because all programs require much large space than the L2 cache size and the reduction in heap storage requirement has a consistent impact on performance. Similar results have

Program	Input Size	Memory Latency					
		100 cycles		200 cycles		400 cycles	
		$\frac{Comp.}{Orig.}$	$\frac{Comp.}{ccmalloc}$	$\frac{Comp.}{Orig.}$	$\frac{Comp.}{ccmalloc}$	$\frac{Comp.}{Orig.}$	$\frac{Comp.}{ccmalloc}$
`treeadd`	small	58.8 %	81.0 %	62.1 %	73.2 %	66.6 %	65.4 %
	medium	58.8 %	81.0 %	62.0 %	73.2 %	66.6 %	65.4 %
	large	58.7 %	81.0 %	62.0 %	73.2 %	66.5 %	65.4 %
`bisort`	small	75.3 %	73.9 %	62.8 %	58.6 %	48.6 %	42.9 %
	medium	69.3 %	69.5 %	54.8 %	53.4 %	40.7 %	38.7 %
	large	67.7 %	64.9 %	51.9 %	48.1 %	36.5 %	32.8 %
`tsp`	small	97.3 %	99.7 %	96.5 %	99.7 %	95.3 %	99.8 %
	medium	97.0 %	99.7 %	96.2 %	99.7 %	94.9 %	99.8 %
	large	96.8 %	99.5 %	95.9 %	99.6 %	94.5 %	99.6 %
`perimeter`	small	75.1 %	91.8 %	73.8 %	87.1 %	72.2 %	81.6 %
	medium	76.3 %	92.3 %	74.8 %	87.7 %	72.9 %	82.1 %
	large	77.6 %	93.1 %	76.0 %	88.5 %	73.9 %	82.9 %
`health`	small	83.4 %	94.6 %	83.6 %	91.3 %	83.7 %	89.3 %
	medium	68.1 %	95.5 %	66.3 %	93.1 %	65.3 %	91.8 %
	large	62.1 %	95.8 %	60.0 %	93.8 %	58.8 %	92.6 %
`mst`	small	35.7 %	102.2 %	28.3 %	101.7 %	23.2 %	101.2 %
	medium	37.8 %	102.1 %	31.2 %	101.6 %	26.8 %	101.0 %
	large	36.6 %	102.2 %	30.7 %	101.6 %	26.6 %	101.0 %
`average`	small	70.9 %	90.5 %	67.9 %	85.3 %	64.9 %	80.0 %
	medium	67.9 %	90.0 %	64.2 %	84.8 %	61.2 %	79.8 %
	large	66.6 %	89.4 %	62.7 %	84.1 %	59.5 %	79.0 %

(a) Change in cycle counts

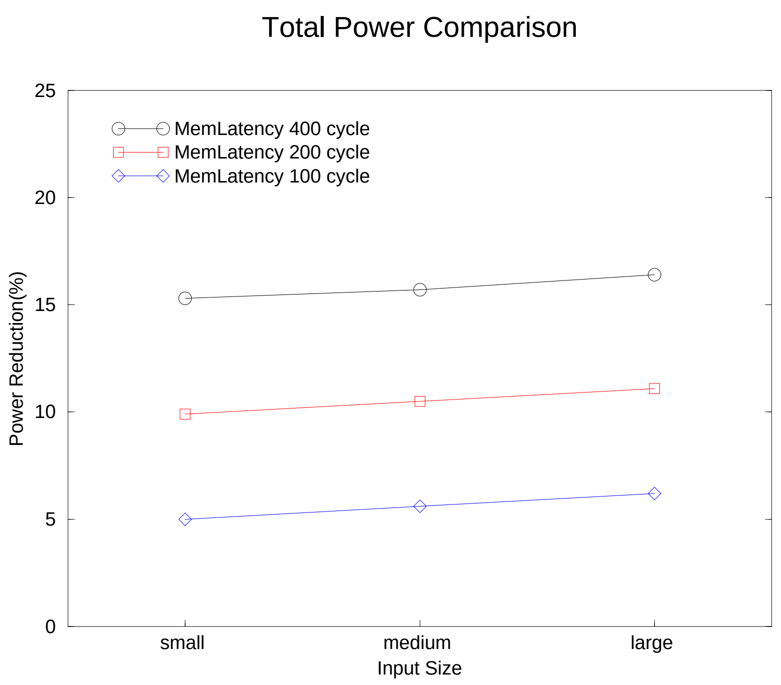


(b) Compression vs `ccmalloc`.

Figure 17: Change in execution time due to data compression.

Program	Input Size	Memory Latency					
		100 cycles		200 cycles		400 cycles	
		$\frac{Comp.}{Orig.}$	$\frac{Comp.}{ccmalloc}$	$\frac{Comp.}{Orig.}$	$\frac{Comp.}{ccmalloc}$	$\frac{Comp.}{Orig.}$	$\frac{Comp.}{ccmalloc}$
`treeadd`	small	60.9 %	89.7 %	62.5 %	83.4 %	65.0 %	75.5 %
	medium	60.9 %	89.7 %	62.5 %	83.4 %	65.0 %	75.5 %
	large	60.9 %	89.7 %	62.5 %	83.4 %	65.0 %	75.5 %
`bisort`	small	77.9 %	79.2 %	67.6 %	66.0 %	54.9 %	51.1 %
	medium	73.7 %	75.9 %	61.3 %	61.6 %	47.9 %	46.8 %
	large	73.3 %	72.1 %	59.5 %	56.9 %	44.5 %	41.3 %
`tsp`	small	97.1 %	99.8 %	96.5 %	99.9 %	95.6 %	99.9 %
	medium	96.7 %	99.8 %	96.1 %	99.8 %	95.1 %	99.9 %
	large	96.7 %	99.7 %	96.0 %	99.7 %	94.9 %	99.7 %
`perimeter`	small	75.9 %	95.1 %	74.7 %	91.6 %	73.1 %	86.8 %
	medium	77.1 %	95.6 %	75.8 %	92.2 %	74.1 %	87.4 %
	large	78.6 %	96.4 %	77.2 %	93.1 %	75.2 %	88.2 %
`health`	small	90.0 %	101.4 %	87.8 %	95.9 %	86.2 %	92.1 %
	medium	73.5 %	101.1 %	69.5 %	96.6 %	67.1 %	93.8 %
	large	66.8 %	100.9 %	62.7 %	96.8 %	60.2 %	94.3 %
`mst`	small	38.9 %	104.5 %	31.5 %	103.8 %	25.7 %	102.9 %
	medium	40.5 %	104.2 %	33.9 %	103.4 %	28.9 %	102.5 %
	large	39.7 %	104.2 %	33.5 %	103.4 %	28.7 %	102.5 %
`average`	small	73.4 %	95.0 %	70.1 %	90.1 %	66.7 %	84.7 %
	medium	70.4 %	94.4 %	66.5 %	89.5 %	63.0 %	84.3 %
	large	69.3 %	93.8 %	65.2 %	88.9 %	61.4 %	83.6 %

(a) Change in power consumption.



(b) Compression vs `ccmalloc`.

Figure 18: Impact on power consumption.

Program	Input Size	I-cache		D-cache		Unified-cache	
		$\frac{Comp.}{Orig.}$	$\frac{Comp.}{ccmalloc}$	$\frac{Comp.}{Orig.}$	$\frac{Comp.}{ccmalloc}$	$\frac{Comp.}{Orig.}$	$\frac{Comp.}{ccmalloc}$
`treeadd`	small	105.2 %	104.8 %	62.2 %	60.4 %	85.1 %	49.7 %
	medium	106.4 %	105.5 %	61.5 %	59.7 %	85.0 %	49.7 %
	large	107.3 %	104.7 %	60.0 %	59.8 %	84.9 %	49.7 %
`bisort`	small	153.3 %	155.9 %	65.0 %	58.7 %	16.2 %	16.8 %
	medium	228.2 %	234.1 %	68.7 %	63.1 %	15.5 %	16.6 %
	large	228.2 %	234.1 %	47.3 %	38.3 %	7.4 %	7.0 %
`tsp`	small	5.0 %	120.5 %	70.1 %	90.3 %	84.1 %	100.1 %
	medium	4.0 %	122.1 %	66.0 %	94.0 %	84.4 %	100.1 %
	large	3.6 %	124.9 %	62.4 %	84.3 %	84.4 %	100.1 %
`perimeter`	small	145.1 %	86.0 %	69.1 %	71.3 %	67.1 %	67.0 %
	medium	205.1 %	83.5 %	68.9 %	70.9 %	67.0 %	66.8 %
	large	321.8 %	78.0 %	69.1 %	70.3 %	67.0 %	66.8 %
`health`	small	122.2 %	112.1 %	82.2 %	96.2 %	41.6 %	62.3 %
	medium	133.8 %	116.6 %	82.2 %	97.8 %	46.4 %	67.3 %
	large	144.8 %	120.7 %	82.1 %	98.6 %	51.9 %	71.1 %
`mst`	small	26.6 %	61.6 %	41.0 %	100.9 %	16.2 %	100.0 %
	medium	16.8 %	48.2 %	49.0 %	96.3 %	21.3 %	100.0 %
	large	13.8 %	42.7 %	33.2 %	94.8 %	21.5 %	100.0 %
`average`	small	92.9 %	106.8 %	64.9 %	79.6 %	51.7 %	66.0 %
	medium	115.7 %	118.3 %	66.0 %	80.3 %	53.3 %	66.8 %
	large	136.6 %	117.5 %	59.0 %	74.3 %	52.8 %	65.8 %

Figure 19: Change in cache misses (memory latency = 100 cycles).

been observed for small and large inputs.

5 Related work

Recently there has been a lot of interests in exploiting narrow width values to improve program performance [9, 10, 11]. However, our work focuses on pointer intensive applications for which it is important to also handle *pointer data*. A lot of research has been conducted on development of locality improving transformations for dynamically allocated data structures. These transformations alter object layout and placement to improve cache performance [5, 4, 7]. However, none of these transformations result in space savings.

Existing compression transformations [12, 13] rely upon compile time analysis to prove that certain data items do not require a complete word of memory. They are applicable only when the compiler can determine that the data being compressed is *fully compressible* and they only apply to *narrow width non-pointer* data. In contrast, our compression transformations apply to *partially compressible* data and, in addition to handling narrow width non-pointer data, they also apply to *pointer data*. The approach introduced in this section is not only more general but also simpler in one respect. It does not require compile-time analysis to prove that the data is always compressible. Instead simple compile-time heuristics are sufficient to determine that the data is likely to be compressible. Data width has also been exploited in the hardware for reducing the

Program	Input Size	Memory Latency					
		100 cycles		200 cycles		400 cycles	
		$\frac{Comp.}{Orig.}$	$\frac{Comp.}{ccmalloc}$	$\frac{Comp.}{Orig.}$	$\frac{Comp.}{ccmalloc}$	$\frac{Comp.}{Orig.}$	$\frac{Comp.}{ccmalloc}$
`treeadd`	small	69.3 %	95.4 %	71.2 %	84.0 %	74.0 %	72.6 %
	medium	69.2 %	95.4 %	71.2 %	84.0 %	73.9 %	72.6 %
	large	68.8 %	94.8 %	70.8 %	83.6 %	73.6 %	72.3 %
`bisort`	small	112.9 %	110.8 %	92.8 %	86.6 %	69.9 %	61.8 %
	medium	85.0 %	85.2 %	66.4 %	64.7 %	48.4 %	45.9 %
	large	91.3 %	87.6 %	69.3 %	64.3 %	48.0 %	43.1 %
`tsp`	small	97.6 %	100.1 %	96.9 %	100.1 %	95.6 %	100.1 %
	medium	97.3 %	100.0 %	96.5 %	100.0 %	95.1 %	100.1 %
	large	97.1 %	99.9 %	96.2 %	99.9 %	94.8 %	99.9 %
`perimeter`	small	84.5 %	103.3 %	81.7 %	96.4 %	78.2 %	88.3 %
	medium	85.8 %	103.8 %	82.9 %	97.1 %	79.1 %	89.0 %
	large	87.5 %	104.9 %	84.4 %	98.3 %	80.4 %	90.1 %
`health`	small	91.8 %	104.1 %	88.6 %	96.8 %	86.4 %	92.3 %
	medium	73.7 %	103.2 %	69.4 %	97.4 %	66.9 %	94.0 %
	large	66.7 %	103.0 %	62.5 %	97.7 %	60.1 %	94.6 %
`mst`	small	36.2 %	103.6 %	28.6 %	102.8 %	23.4 %	101.9 %
	medium	38.2 %	103.3 %	31.5 %	102.4 %	26.9 %	101.6 %
	large	37.0 %	103.4 %	30.9 %	102.5 %	26.7 %	101.6 %
`average`	small	82.0 %	102.9 %	76.6 %	94.4 %	71.2 %	86.2 %
	medium	74.9 %	98.5 %	69.6 %	91.0 %	65.1 %	83.9 %
	large	74.7 %	98.9 %	69.0 %	91.0 %	63.9 %	83.6 %

Figure 20: Performance impact without DCX support.

Program	Large L2 cache (512K)		High associativity (2-way L1/4-way L2)		Long L2 hit latency (18 cycles)	
	$\frac{Comp.}{Orig.}$	$\frac{Comp.}{ccmalloc}$	$\frac{Comp.}{Orig.}$	$\frac{Comp.}{ccmalloc}$	$\frac{Comp.}{Orig.}$	$\frac{Comp.}{ccmalloc}$
`treeadd`	58.8 %	81.0 %	58.8 %	81.0 %	58.3 %	81.9 %
`bisort`	82.2 %	75.8 %	77.5 %	74.4 %	76.1 %	73.9 %
`tsp`	97.1 %	99.7 %	97.0 %	99.7 %	95.8 %	99.4 %
`perimeter`	76.2 %	92.3 %	76.3 %	92.3 %	74.5 %	91.5 %
`health`	103.4 %	98.8 %	67.5 %	95.5 %	68.1 %	96.4 %
`mst`	37.4 %	102.2 %	37.8 %	102.1 %	38.6 %	102.0 %
`average`	75.8 %	91.6 %	69.1 %	90.8 %	68.6 %	90.8 %

Figure 21: Performance changes with more cache configurations.

power consumption [14, 15].

ISA extensions have been developed to efficiently process narrow width data including Intel's MMX [16] and Motorola's AltiVec [18]. Compiler techniques are also being developed to exploit such instruction sets [17]. However, the instructions introduced in this section are quite different from MMX instructions because both partially compressible data structures and pointer data must be handled.

Compression techniques have also been proposed in reducing the memory footprint in object-oriented systems. New garbage collectors are proposed in [19, 20] which target for environments with severe memory constraints and reduce heap space footprint through object compaction and compression. Our technique is orthogonal to these techniques and can be combined with them for achieving further footprint reduction.

6 Conclusion

In this paper, two types of data compression transformations are introduced to apply data compression techniques to compact the sizes of dynamically allocated data structures. These transformations result in large space savings and also result in significant reductions in program execution times and power dissipation due to improved memory performance.

An attractive property of these transformations is that they are applicable to partially compressible data structures. This is extremely important because according to our experiments, while the data structures in all of the benchmarks studied in this section are very highly compressible, they always contain small amounts of incompressible data.

This approach is applicable to a more general class of programs than existing compression techniques: it can compress pointers as well as non-pointer data; and it can compress partially compressible data structures. Finally the DCX ISA extensions have been designed to enable efficient manipulation of compressed data. The same task cannot be carried using MMX type instructions. The main contribution of this work is that data compression techniques can now be used to improve performance of general purpose programs and therefore it takes the utility of compression beyond the realm of multimedia applications.

Acknowledgement

Supported by grants from IBM, Intel, Microsoft, and National Science Foundation awards CCR-0324969, CCR-0220334, CCR-0208756, CCR-0105535, and EIA-0080123 to the University of Arizona, and grants from National Science Foundations awardis CCR-0429986 and CCR-0447934 to the University of Texas at Dallas.

References

[1] Stanford SUIF. http://www.suif.org/.

[2] Doug Burger and Todd Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin-Madison, 1997.

[3] Martin C. Carlisle and Anne Rogers, Software Caching and Computation Migration in Olden, In *ACM Symposium on Principles and Practice of Parallel Programming*,pages 29-38, July 1995.

[4] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, Atlanta, GA, 1999.

[5] Dan N. Truong, Franois Bodin, and André Seznec. Improving cache behavior of dynamically allocated data structures. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 322–329, Paris, France, 1998.

[6] Chi-Keung Luk and Todd C. Mowry. Memory forwarding: Enabling aggressive layout optimizations by guaranteeing the safety of data relocation. In *The 26th Annual International Symposium on Computer Architecture*, pages 88–99, Atlanta, GA, 1999.

[7] Brad Calder, Chandra Krintz, Simmi John, and Todd Austin. Cache-conscious data placement. In *The Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 139–149, San Jose, CA, 1998.

[8] Trishul M. Chilimbi, Bob Davidson, and James R. Larus. Cache-conscious structure definition. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–24, Atlanta, GA, 1999.

[9] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *The 27th International Symposium on Computer Architecture*, pages 83–94, 2000.

[10] Youtao Zhang, Jun Yang, and Rajiv Gupta. Frequent value locality and value-centric data cache design. In *ACM 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 150–159, Cambridge, MA, 2000.

[11] Jun Yang, Youtao Zhang, and Rajiv Gupta. Frequent value compression in data caches. In *IEEE/ACM International Symposium on Microarchitecture*, pages 258–265, Monterey, CA, 2000.

[12] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. Bitwidth analysis with application to silicon compilation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 108–120, Vancouver B.C, Canada, 2000.

[13] Jack W. Davidson and Sanjay Jinturkar. Memory access coalescing: A technique for eliminating redundant memory accesses. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 186–195, 1994.

[14] Y. Cao and H. Yasuura, Low-Energy Design using Datapath Width Optimization for Embedded Processor-based Systems, IPSJ Journal, 43(5):1348-1356, May 2002.

[15] G. Loh, Exploiting Data-Width Locality to Increase Superscalar Execution Bandwidth, In *IEEE/ACM International Symposium on Microarchitecture*, November 2002.

[16] Alex Peleg and Uri Weiser. MMX technology extension to the intel architecture. *IEEE Micro*, 16(4):51–59, 1996.

[17] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 145–156, Vancouver B.C, Canada, 2000.

[18] Jon Tyler, Jeff Lent, Anh Mather, and Huy Van Nguyen. Altivec(tm): Bringing vector technology to the powerpc(tm) processor family. Phoenix, AZ, 2000.

[19] G. Chen, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, B. Mathiske, and M. Wolczko, Heap Compression for Memory-constrained Java Environments, In *18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 282-301, Anaheim, California, October, 2003.

[20] N. Vijaykrishnan, M. Kandemir, S. Kim, S. Tomar, A. Sivasubramaniam and M. J. Irwin, Energy Behavior of Java Applications from the Memory Perspective, In *USENIX Java Virtual Machine Research and Technology Symposium*, April 2001.