Data Compression Transformations for Dynamically Allocated Data Structures *

Youtao Zhang and Rajiv Gupta

Dept. of Computer Science, The University of Arizona, Tucson, Arizona 85721

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. 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 12.5%.

1 Introduction

With the proliferation of limited memory computing devices, optimizations that reduce memory requirements are increasing in importance. 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.

* Supported by DARPA PAC/C Award. F29601-00-1-0183 and NSF grants CCR-0105355, CCR-0096122, EIA-9806525, and EIA-0080123 to the Univ. of Arizona.

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 Data Compression Transformations

As mentioned earlier, we have developed two compression transformations: one to handle pointer data and the other to handle narrow width non-pointer data. We illustrate the transformations by using an example of the dynamically allocated link list data structure shown below – the *next* and *value* fields are compressed to illustrate the compression of both pointer and non-pointer data. The compressed fields are *packed* together to form 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;	

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, we classify the *next* pointer as compressible. In this case we eliminate the common prefix from address `addr2` which is stored in the `next` pointer field. 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 an array element 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. Next we describe the handling of uncompressible data in partially compressible data structures. The implementation of partially compressible data structures require an additional bit for encoding information. This is why we compress fields down to 15 bit entities and not into 16 bit entities.

Partial compressibility. Our 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, we check 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 uncompressible fields we have to go through an extra step of indirection.

If the uncompressible data stored in the fields is modified, it is possible that the fields may now become compressible. However, we do not carry out such checks and instead we leave the fields in such cases 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 compressible and uncompressible kind. To avoid repeated allocation and deallocation of extra locations we simplify our approach so that once a field is assigned an uncompressible value, from then onwards, the data in the field is always maintained in uncompressed form.

We use the most significant bit (bit 31) in the word to indicate whether or not the data stored in the word is compressed or not. This is possible because in the MIPS base system that we use, the most significant bit for all heap addresses is always 0. 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.

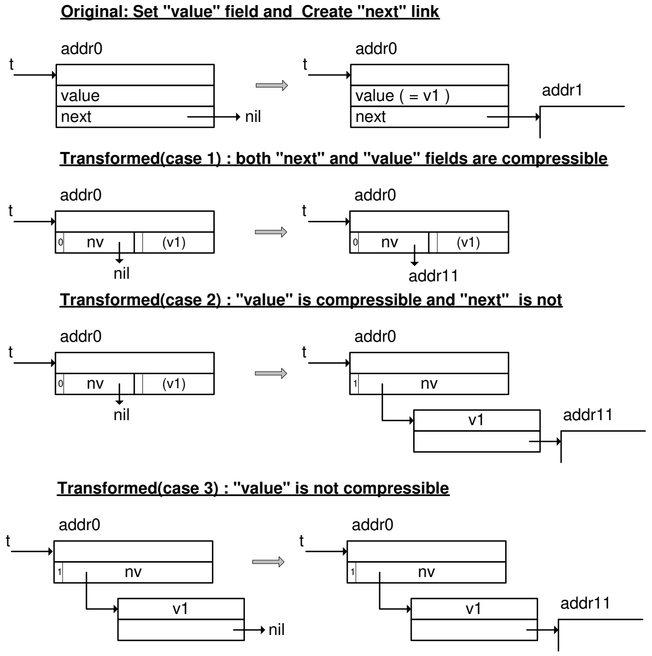

Fig. 1. Dealing with uncompressible data.

In Fig. 1 we illustrate 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 uncompressible 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 Fig. 1. 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 *next* field is found to be compressible, 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.

3 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 both the fields. However, the manipulation of the fields also creates additional overhead. To minimize this overhead we have design new RISC-style instructions. We have designed three simple instructions each 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 Fig. 2. 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 we actually compress a data item at runtime, we must first check whether the data item is compressible. Therefore the first instruction type we introduce allows efficient checking of data compressibility. We have provided the two new instructions that 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. The instruction also handles the case where `R2` contains a nil pointer which is represented by the value 0 both in compressed and uncompressed forms. Since 0 represents a nil pointer, the lower order 15 bits of an allocated address should never be all zeroes - to correctly handle this situation we have modified our `malloc` routine so that it never allocates storage locations with such addresses.
- `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.

Extract-and-expand. If a pointer is stored in compressed form, before it can be derefrenced, we must first reconstruct its 32-bit representation. We do the same for compressed non-pointer data before its use. Therefore the second instruction type that we introduce 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-poniter 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. We also handle the case when R3 contains a nil pointer. If the compressed field is a nil pointer, R1 is set to nil.

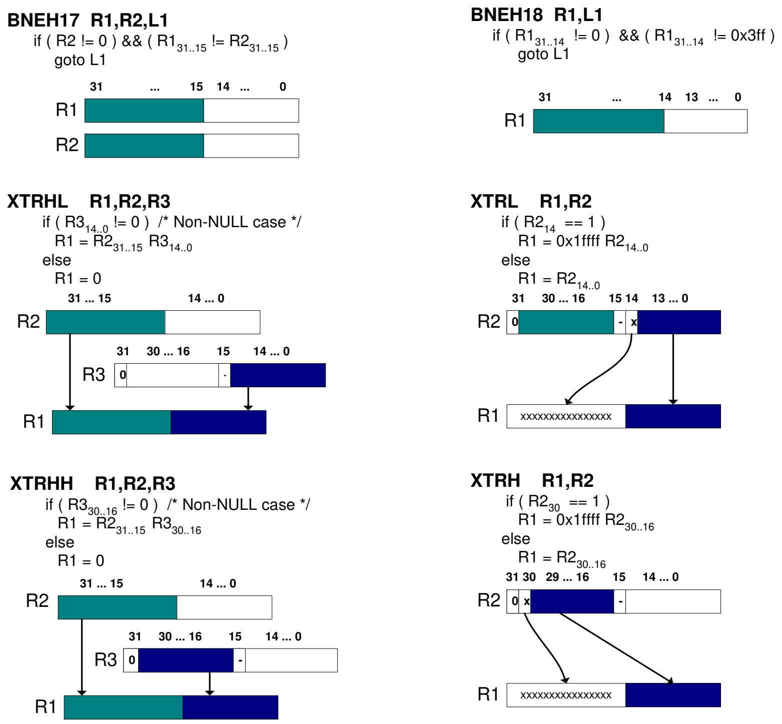

Fig. 2. DCX instructions.

`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 we 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, exapands it, and then stores the resulting 32 bit value in R1.

Next we give a simple example to illustrate the use of the above instructions. Let us assume that an integer field $t \to value$ and a pointer field $t \to next$ are compressed together into a single field $t \to value\_next$. In Fig. 3a we show how compressibility checks are used prior to appropriately storing *newvalue* and *newnext* values in to the compressed fields. In Fig. 3b we illustrate the extract and expand instructions by extracting the compressed values stored in $t \to value\_next$.

```
      ; $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
      swr    $18, 0($16)
      swr    $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
      xtrhl $18, $16, $3
      j    $L2
$L1:  ; load values from extra locations
      ...
$L2:  ...
```

(b) Illustration of extract and expand instructions.

Fig. 3. An example.

4 Compiler Support

Object layout transformations can only be applied to a C program if the user does not access the fields through explicit address arithmetic and also does not typecast the objects of the transformed type into objects of another type. Like prior work by Truong et al. [14] on field reorganization and instance interleaving, we assume that the programmer has given us the go ahead to freely transform the data structures when it is apprpriate to do so. From this step onwards the rest of process is carried out automatically by the compiler. In the remainder of this section we describe key aspects of the the compiler support required for effective data compression.

Identifying fields for compression and packing. Our observation is that most pointer fields can be compressed quite effectively using the common-prefix transformation. Integer fields to which narrow-data transformation can be applied can be identified either based upon knowledge about the application or using value profiling. The most critical issue is that of pairing compressed fields for packing into a single word. For this purpose we must first categorize the fields as *hot* fields and *cold* fields. It is useful 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. We used basic block frequency counts to identify pairs of fields belonging to the above categories and then applied compression transformations to them.

ccmalloc vs malloc. We make use of `ccmalloc` [6], a modified version of `malloc`, for carrying out *storage allocation*. This form of storage allocation was developed by Chilimbi et al. [6] 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. 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.

Register pressure. Another issue that we consider in our implementation is that of potential increase in *register pressure*. The code executed when the pointer fields are found to be uncompressible is substantial and therefore it can increase register pressure significantly causing a loss in performance. However, we know that this code is executed very infrequently since very few fields are uncompressible. Therefore, in this piece of code we first free registers by saving values and then after executing the code the values are restored in registers. In other words, the increase in register pressure does not have an adverse effect on frequently executed code.

Instruction cache behavior and code size. The additional instructions generated for implementing compression can lead to an increase in *code size* which can further impact the *instruction cache* behavior. It is important to note however that a large part of the code size increase is due to the handling of the infrequent case in which the data is found not to be compressible. In order to minimize the impact on the *code size* we can share the code for handling the above infrequent case across all the updates corresponding to a given data field. To minimize the impact of the performance on the *instruction cache*, we can employ a code layout strategy which places the above infrequently executed code elsewhere and create branches to it and back so that the instruction cache behavior for more frequently executed code is minimally affected. Our implementation currently does not support the above techniques and therefore we observed code size increase and degraded instruction cache behavior in our experiments.

Code generation. The remainder of the code generation details for implementing data compression are in most part quite straightforward. Once the fields have been selected for compression and packing together, whenever a use of a value of any of the fields is encountered, the load is followed by an extract-and expand instruction. If the value of any of compressed fields is to be updated, the compressibility check is performed before storing the value. When two hot fields that are packed together are to be read/updated, initially we generate separate loads/stores for them. Later in a separate pass we eliminate the later of the two loads/stores whenever possible.

5 Performance Evaluation

Experimental setup. We have implemented the techniques described 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` [3]. The evaluation is based upon six benchmarks taken from the *Olden* test suite [5] (see Fig. 4a) which contains pointer intensive programs that make extensive use of dynamically allocated data structures.

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 cache organization of simplescalar is shown in Fig. 4b. 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.

Impact on storage needs. The transformations applied and their impact on the node sizes is shown in Fig. 5a. 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

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

(a) Benchmarks.

Parameter	Value
Issue Width	4 issue, out of order
I cache	16K direct mapped
I cache miss latency	9 cycles
L1 data cache	16K direct mapped
L1 data cache miss latency	9 cycles
L2 unified cache	256K 2-way
Memory latency (L2 cache miss latency)	Configuration 1/2/3 = 100/200/400 cycles

(b) Machine configurations.

Fig. 4. Experimental setup.

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%.

We measured the runtime savings in heap allocated storage for small and large program inputs. The results are given in Fig. 5b. 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 33% storage savings for `treeadd` represents 4.2 Mbytes and 17 Mbytes of heap storage savings for small and large program inputs respectively. It should also be noted that such savings cannot be obtained by other locality improving techniques described earlier [14, 15, 6].

From the results in Fig. 5b we make another very important observation. The *extra locations* allocated when non-compressible data is encountered is 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 uncompressible 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 we had restricted the application of our technique to data fields that are always guaranteed to be compressible, we could not have achieved any compression and therefore no space savings would have resulted.

We also measured the increase in code size caused by our transformations (see Fig. 5c). 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, we expect that the increase in the size of user code will also be quite small.

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

(a) Reduction in node size.

Program	Before Linking	After Linking
`treeadd`	16.4%	0.04%
`bisort`	40.0%	0.01%
`tsp`	4.9%	0.18%
`perimeter`	21.3%	1.97%
`health`	33.7%	0.23%
`mst`	10.7%	0.06%
`average`	21.1%	0.41%

(c) Code size increase.

	Storage (bytes)					
	Small Input			Large Input		
Program	Original	Total (Extra)	Savings	Original	Total (Extra)	Savings
`treeadd`	12582900	8402040 (13440)	33.2 %	50331636	33605684 (51260)	33.2 %
`bisort`	786420	549880 (25600)	30.1 %	3145716	2301304 (204160)	26.8 %
`tsp`	5242840	4200352 (6080)	19.9 %	20971480	16800224 (23040)	19.9 %
`perimeter`	4564364	3265380 (5120)	28.5 %	20332620	14546980 (23680)	28.5 %
`health`	566872	510272 (320)	10.0 %	1128240	1015124 (320)	10.0 %
`mst`	3414020	2367812 (320)	30.6 %	54550532	37781828 (320)	30.7 %
`average`			25.4 %			24.9 %

(b) Reduction in heap storage for small and large inputs.

Fig. 5. Impact on storage needs.

Impact on execution times. Based upon the cycle counts provided by the simplescalar simulator we studied the changes in execution times resulting from compression transformations. The impact of L2 latency on execution times was also studied. The results in Fig. 6 are for small inputs. 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 for higher latencies are also similar.

We also compared our execution times with versions of the programs that use `ccmalloc`. Our approach outperforms `ccmalloc` in five out of the six benchmarks (our version of `mst` runs slightly slower than the `ccmalloc` version). On an average we outperform `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. We also pay additional runtime overhead 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. 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. Therefore we observe that as the latency of L2 cache is increased, compression outperforms `ccmalloc` by a greater extent. In summary our approach provides large storage savings and significant execution time reductions over `ccmalloc`.

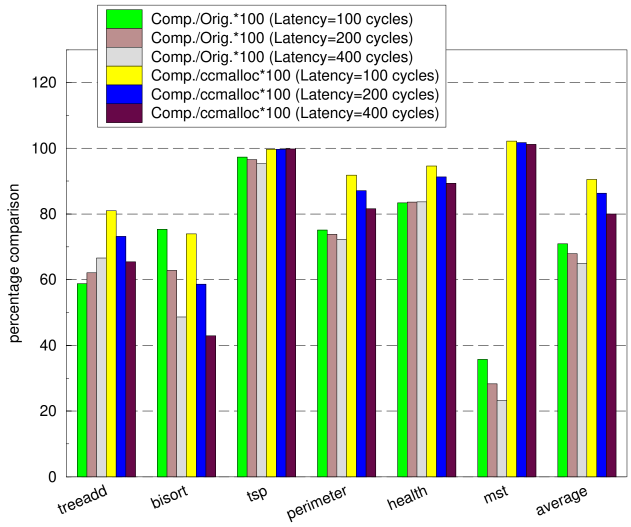

Fig. 6. Reduction in execution time due to data compression.

We would also like to point out that the use of special DCX instructions was critical in reducing the overhead of compression and extraction. Without DCX instructions the programs would have ran significantly slower. We ran versions of programs which did not use DCX instructions for L2 cache latency of 100 cycles. The average reduction in execution times, in comparison to original programs, dropped from 30% to 12.5%. Instead of an average reduction in execution times of 10% in comparison to `ccmalloc` versions of the program we observed an average increase of 9% in execution times.

Impact on power consumption. We also compared the power consumption for the compression based programs with that of the original programs and `ccmalloc` based programs (see Fig. 7). These measurements are based upon the `Wattch` [1] 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% for the small input. The reductions in power dissipation that compression provides over `ccmalloc` for the different cache latencies is also given. 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.

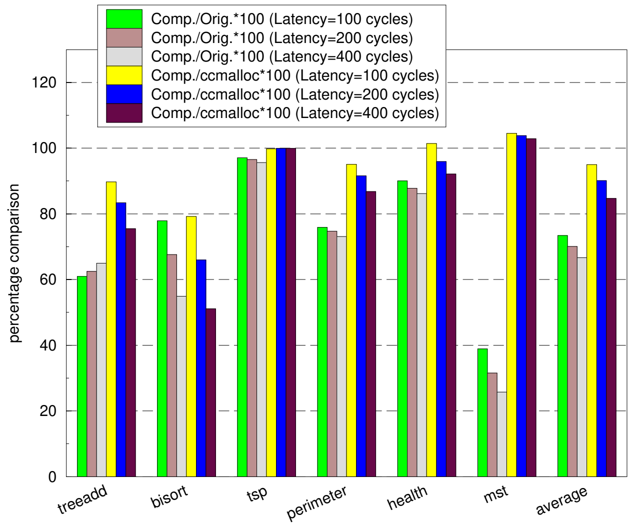

Fig. 7. Impact on in power consumption.

Impact on cache performance. Finally in Fig. 8 we present 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. However, the performances of D-cache and unified cache are significantly improved. This improvement in data cache performance is a direct consequence of compression.

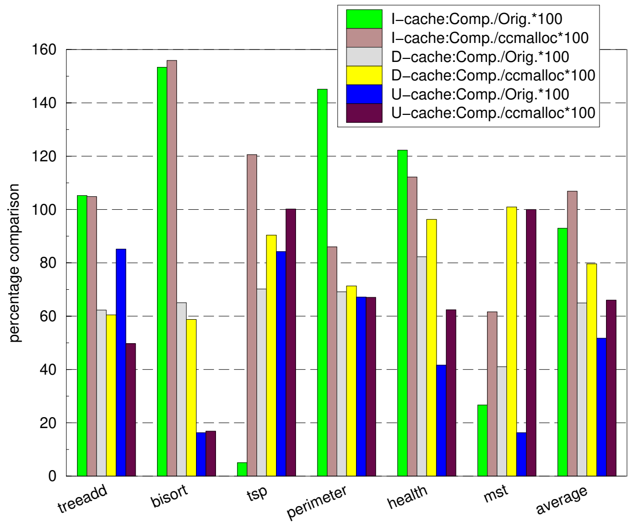

Fig. 8. Impact on cache misses.

6 Related Work

Recently there has been a lot of interest in exploiting narrow width values to improve program performance [2, 12, 13]. However, our work focusses on pointer intensive applications for which it is important to also handle *pointer data*. A great deal 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 [14, 6, 15]. However, none of these transformations result in space savings.

Existing compression transformations [10, 7] 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*. Our approach is not only more general but it is also simpler in one respect. We do 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.

ISA extensions have been developed to efficiently process narrow width data including Intel's MMX [9] and Motorola's AltiVec [11]. Compiler techniques are also being developed to exploit such instruction sets [8]. However, the instructions we require are quite different from MMX instructions because we must handle partially compressible data structures and we must also handle pointer data.

7 Conclusions

In conclusion we have introduced a new class of transformations that 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 we studied are very highly compressible, they contain small amounts of uncompressible data. Even for programs with fully compressible data structures our approach has one advantage. The application of compression transformations can be driven by simple value profiling techniques [4]. There is no need for complex compile-time analyses for identifying fully compressible fields in data structures.

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

References

1. D. Brooks, V. Tiwari, and D. Martonosi, "Wattch: A Framework for Architecture-Level Power Analysis and Optimizations," *27th International Symposium on Computer Architecture* (ISCA), pages 83–94, May 2000.
2. D. Brooks and D. Martonosi, "Dynamically Exploiting Narrow Width Operands to Improve Processor Power and Performance," *5th International Symposium on High-Performance Computer Architecture* (HPCA), pages 13–22, Jan. 1999.
3. D. Burger and T.M. Austin, "The Simplescalar Tool Set, Version 2.0," *Computer Architecture News*, pages 13–25, June 1997.
4. M. Burrows, U. Erlingson, S-T.A. Leung, M.T. Vandevoorde, C.A. Waldspurger, K. Walker, and W.E. Weihl, "Efficient and Flexible Value Sampling," *The Ninth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS), pages 160–167, Cambridge, MA, November 2000.
5. M. Carlisle, "Olden: Parallelizing Progrms with Dynamic Data Structures on Distributed-Memory Machines," PhD Thesis, Princeton Univ., Dept. of Comp. Science, June 1996.
6. T.M. Chilimbi, M.D. Hill, and J.R. Larus, "Cache-Conscious Structure Layout," *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), pages 1–12, Atlanta, Georgia, May 1999.
7. J. Davidson and S. Jinturkar, "Memory access coalescing : a technique for eliminating redundant memory accesses," *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), pages 186–195, 1994.
8. S. Larsen and S. Amarasinghe, "Exploiting Superword Level Parallelism with Multimedia Instruction Sets," *ACM SIGPLAN Conf. on Programming Language Design and Implementation* (PLDI), pages 145–156, Vancouver B.C., Canada, June 2000.
9. A. Peleg and U. Weiser, *MMX Technology Extension to Intel Architecture.* 16(4):42-50, August 1996.
10. M. Stephenson, J. Babb, and S. Amarasinghe, "Bitwidth Analysis with Application to Silicon Compilation," *ACM SIGPLAN Conf. on Programming Language Design and Implementation* (PLDI), pages 108–120, Vancouver B.C., Canada, June 2000.
11. J. Tyler, J. Lent, A. Mather, and H.V. Nguyen, "AltiVec(tm): Bringing Vector Technology to the PowerPC(tm) Processor Family," Phoenix, AZ, February 1999.
12. Y. Zhang, J. Yang, and R. Gupta, "Frequent Value Locality and Value-Centric Data Cache Design," *The Ninth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS), pages 150–159, Cambridge, MA, November 2000.
13. J. Yang, Y. Zhang, and R. Gupta, "Frequent Value Compression in Data Caches," *The 33nd Annual IEEE/ACM International Symposium on Microarchitecture* (MICRO), pages 258–265, Monterey, CA, December 2000.
14. D.N. Truong, F. Bodin, and A. Seznec, "Improving Cache Behavior of Dynamically Allocated Data Structures," *International Conference on Parallel Architectures and Compilation Techniques* (PACT), pages 322–329, Paris, France, 1998.
15. B. Calder, C. Krintz, S. John, and T. Austin, "Cache-Conscious Data Placement," *8th International Conf. on Architectural Support for Programming Languages and Operating Systems* (ASPLOS), pages 139–149, San Jose, California, October 1998.