Compressing Dynamic Data Structures in Operating System Kernels*

Haifeng He, Saumya Debray, Gregory Andrews

Department of Computer Science, The University of Arizona, Tucson, AZ 85721, USA

Abstract

Embedded systems are becoming increasingly complex and there is a growing trend to deploy complicated software systems such as operating systems and databases in embedded platforms. It is especially important to improve the efficiency of memory usage in embedded systems because these devices often have limited physical memory. Previous work on improving the efficiency of memory usage in OS kernels has mostly focused on reducing the size of code and global data in the OS kernel. This paper, by contrast, presents *dynamic data structure compression*, a complementary approach that reduces the runtime memory footprint of dynamic data structures. A prototype implementation for the Linux kernel reduces the memory consumption of the slab allocators in Linux by about 17.5% when running the MediaBench suite, while incurring only minimal increases in execution time (1.9%).

1 Introduction

The amount of memory available on typical embedded systems is usually limited by considerations such as size, weight, or cost. This makes it important to reduce the memory footprint of software running on embedded systems. The operating system on an embedded processor accounts for a significant part of its memory requirements. While there has been some work on reducing the memory usage of OS kernels, most of this has focused on reducing the size of code and global data in the OS kernels [4, 5, 8]. However, the *static* components of an OS kernel—its code and global data—account for only a portion of its total memory footprint. Just as significant are the *dynamic* data, namely, the stack and heap memory, which can easily exceed the size of static memory. We are not aware of any research on reducing the dynamic memory consumption of OS kernels.

It turns out that, in practice, there is quite often room to reduce the memory requirements for dynamic data. For example, integer-valued variables often do not require the full 32 bits allocated to them (on a conventional 32-bit architecture); other opportunities for memory usage reduction arise from redundancy in sets of pointer values whose high-order bits typically share a common prefix. However, such opportunities for data compression are usually not obvious statically, making it difficult to optimize programs to take advantage of them.

This paper presents a technique called *dynamic data structure compression* that aims to address the problem of reducing the dynamic memory requirements of programs. Our technique uses profiling to detect opportunities for dynamic data size reduction, then transforms the code so that data values are maintained using a smaller amount of memory. The technique is *safe*: if a runtime value is beyond the value range that can be accommodated in the compressed representation of some variable, our approach automatically "expands" that variable to its original (unoptimized) size. Our experiments show that this approach can be quite effective in reducing the dynamic memory footprint of the OS kernel: applying our technique to the slab allocator in Linux kernel reduces the dynamic memory consumption of slab allocator in Linux kernel by about 17.5% when running the MediaBench suite, while incurring only a 5.4% increase in code size and a 1.9% increase in execution time.

The remainder of this paper is organized as follows. Section 2 first gives a brief background of slab allocator in Linux kernel. Section 3 describes our approach in more detail. Section 4 describes the code transformations we use to maintain and use values in compressed form. Section 5 describes our experimental results. Section 6 describes related work, and Section 7 concludes.

2 Background: Linux kernel slab allocator

Slab allocation forms the core of dynamic memory allocation in the Linux kernel: the kernel memory allocation routines *kmalloc* and *kfree* (the kernel-level analogs of *malloc* and *free*) are built atop the slab allocator. For this reason, our prototype implementation targets the Linux slab allocator. This section gives a brief introduction to the slab allocator in the Linux kernel.

Slab allocation was adopted for the first time in the SunOS 5.4 kernel and introduced in Linux kernel since Linux 2.2 [7]. It provides an efficient mechanism to speed up dynamic memory allocation and reduce internal fragmentation in the kernel. The slab allocator groups objects into *caches* where each cache stores objects of the same type. The

*This work was supported in part by NSF Grants CNS-0410918 and CNS-0615347.

caches are then divided into *slabs* (hence the name of this system). Each slab consists of one or more physically contiguous pages but typically consists of only a single page. To avoid initializing objects repeatedly, the slab allocator does not discard the objects that have been allocated and then released but instead saves them in memory. When a new object is then requested, it can be taken from memory without having to be reinitialized. Our approach does not need to modify the internal implementation of slab allocator in Linux kernel. Instead, the memory consumption of the slab allocator is reduced by compressing only the data structures that are used by the slab caches.

3 Data Structure Compression

A data structure is transformed into a more space-efficient structure by statically compressing its compressible fields, which include scalars and pointers, based on profile information. Our approach consists of the following steps. We first use training inputs to obtain profile information about the values stored in each compressible field. Based on the profile data, we choose a compression scheme for each compressible field with the goal to use as few bits as possible. Finally, we modify the program source code to use compressed structures. Each statement that writes to a compressed field is modified so that the value being written is compressed appropriately before being stored. Similarly, each statement that reads from a compressed field is modified to extract the value of the field from the compressed representation and decompress it appropriately. To ensure safety of our optimization, we exclude from consideration structure fields with certain properties; this is discussed in more detail in Section 4.2.

It is important to note that since this compression is based on data obtained from profiling runs, it can happen that our scheme compresses a data field to k bits but some run of the program can produce a value for that field that requires more than k bits to represent it. To handle this, our approach uses a scheme to expand the compressed data field with additional storage, as necessary, to hold the incompressible data.

3.1 Data structure profiling

The goal of data structure profiling is to obtain information about the values stored in variables and in the fields of aggregate data structures, in particular `structs` (i.e., records), in order to determine whether and how to compress them. Data structure profiling is done by instrumenting all field reference expressions in the source code of a program. In the C programming language, the targets for profiling are fields that are referenced through the operators '`.`' and '`->`'. Three kinds of data are collected for a compressible field: 1) *value range*, i.e., the minimal and maximum values that are encountered during program execution; 2) *distinct values*, which record the top N distinct values presented in a profiled field [2, 13]; and 3) the number of references.[1] Based on the characteristics of the data obtained from profiling, the values taken on by a field in a data structure can be classified into the following categories:

Narrow width. This refers to a set of values that can be represented by a small number of bits. For example, values from the set $\{0, 2, 3, 5\}$ can be represented using only 3 bits.

Common prefix. This refers to a set of values whose binary representations have some number of high-order bits in common. In other words, there exists some $k > 0$ such that the top k bits of these binary representations are the same for all of these values. This situation is encountered primarily with pointers.

Small set. This refers to a set of values of "sufficiently small" cardinality. The idea here is that a value can then be referred to by its index in the set, and the index can be represented using only a small number of bits.

3.2 Data compression techniques

Based on the characteristics of profiling data, as described above, we consider four kinds of compression techniques: (i) compression with narrow-width data (NW); (ii) compression with common-prefix data (CP); (iii) compression with compression table, which is used for small-set data (CT); and (iv) a combination of 2 and 3 (CT+CP). The first scheme is mainly used to compress non-pointer scalar type fields and the other three schemes are mainly used to compress pointer fields.

Figure 1 illustrates one of the Linux kernel data structure called `dentry` (directory entry), which is used to describe the name of a file in the Linux file system. For simplicity, we only list four compressible fields in `dentry` to explain our compression approach. Table 1 shows the profile data collected for these four fields. In Table 1, the second column is the type of each field and the third column is the bit width of each field. The fourth and fifth columns indicate the value range of each field. The fifth column is the number of distinct values of a field[2]. The last column shows the

[1] This number is used later in our experimental evaluation to avoid compressing frequently-used fields and thereby reduce the runtime overhead.
[2] a value table with size $= 4096$ is used in experiments

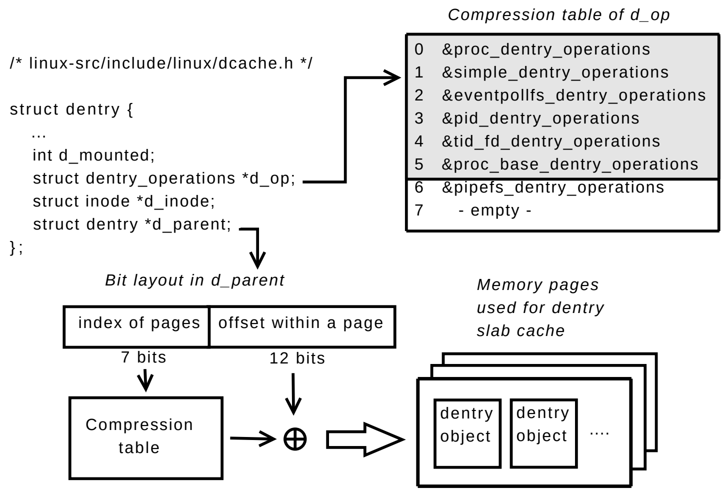


Figure 1: Data structure `dentry` in Linux kernel

Field	*Type*	*Size(bits)*	*Min*	*Max*	*# of values*	*# of pg. prefix*
`d_mounted`	Integer	32	`0x0`	`0x1`	2	-
`d_op`	Pointer	32	`0x0820b78c`	`0x0820ccd4`	6	-
`d_inode`	Pointer	32	`0x086480b4`	`0x09c82e34`	4096	737
`d_parent`	Pointer	32	`0x084fb1a4`	`0x0903cf74`	349	99

Table 1: Profiling data of the four compressible fields in `dentry` structure (with value table $size = 4096$)

data for "page prefixes," a special case of value profiling in which the values are prefixes of the addresses of memory pages. Page prefix information is used for a particular type of data compression that combines the common-prefix and compression-table techniques; this is discussed in more detail in Section 3.2.4.

3.2.1 Compression with narrow width data

Narrow width data is common in many non-pointer scalar type fields. The field `d_mounted` in structure `dentry` in Figure 1 illustrates this: it is defined to be of type `int`, which has value range $[-2^{31}, 2^{31} - 1]$ in a 32-bit machine. `d_mounted` is used to represent the number of file systems that are mounted on one particular directory. Table 1 shows that the value range of `d_mounted` is $[0, 1]$. This is because normally there is at most one file system that is mounted on one directory. It is, therefore, very unlikely that there are $2^{31} - 1$ file systems mounted on a single directory. A field with narrow width data is compressed by selecting the least possible bit width to represent the value ranges. In the above example, a single bit is enough to represent the value range of field `d_mounted`.

3.2.2 Compression with common prefix

A set of addresses that share a common prefix can be compressed by factoring out the common prefix and keeping the remaining bits as the compressed value. To decompress, the common prefix of the pointer is simply added back to the compressed value. For example, consider the value range of field `d_op` in Table 1. The minimum and maximum addresses of field `d_op` share a 17-bit common prefix (`0x08208---`), which means all the addresses presented in `d_op` during profiling also share 17 bit common prefix. We can therefore represent each value for this field using $32 - 17 = 15$ bits, with the 17-bit prefix stored separately. Whenever the field is used, the 17-bit prefix and 15-bit compressed representation are concatenated to obtain the original 32-bit representation.

3.2.3 Compression with compression table

The basic idea of this compression scheme is to keep the actual data value of a field in a table, which we call the *compression table*, and to use the index into the table as the compressed value in the field. Let V_f be the set of distinct values of a field f from profiling, then we need $\lceil log_2|V_f|\rceil$ bits to represent the index for V_f. For instance, in Table 1, d_op takes on 6 distinct values, and we need 3 bits to represent the indexes for this field in the compression table.

Using a compression table can achieve better compression results than using a common prefix. However, maintaining the compression table may introduce considerable overhead at runtime, especially when the size of the compression table is large. To limit the cost of using a compression table, our implementation limits the size of the table to at most 256 values; thus, a value needs at most 8 bits for the index. When the size of the compression table is larger than this limit, the common-prefix scheme, described in Section 3.2.2 above, is used instead. For example, consider the profiling data of field d_inode in Table 1. The number of distinct values (addresses) of d_inode is 4096,[3] which is larger than the limit. Therefore, compression with compression table is not applied to d_inode.

3.2.4 Combining common prefix and compression table

There are situations where the compression results can be further improved by combining the common-prefix and compression-table approaches. As an example, since a memory page in the Linux kernel is 4KB in size, the addresses of all the objects inside of a memory page of a slab cache differ only in the lower 12 bits, which give the offset within the page; the top 20 bits of these addresses are therefore a common prefix—the page prefix. The page prefixes themselves can be further compressed by keeping them in a compression table and using the index into the table plus page offset as compressed value.

For example, consider field d_parent in dentry structure: this is a pointer to objects in the dentry slab cache, as shown in Figure 1. The number of distinct page prefixes of d_parent (shown in the last column in Table 1) is 99, which is less than our 256 value limit for compression table size. Therefore, field d_parent can be compressed to use $\lceil log_2 99\rceil = 7$ bits for the index and 12 bits for page offset, for a total of 19 bits. This is illustrated in Figure 1.

3.2.5 Choosing the compression scheme

To determine which compression scheme to use for each field of a data structure, we compute, for each compression scheme, the number of bits that would be necessary to represent the training set of values obtained for that field using that compression scheme. For each field, we choose the scheme that achieves the smallest bit width for that field.

The choice of a compression scheme for a field in this manner effectively determines the representation for the values of that field at runtime. Runtime values for that field that can be accommodated within the chosen representation for that field are said to be *compressible*, while values that cannot be accommodated within that representation are said to be *incompressible*. For example, suppose we decide to use narrow data to represent a field f with 6 bits. Then, the runtime value 58 for f is compressible, but the value 68 is incompressible.

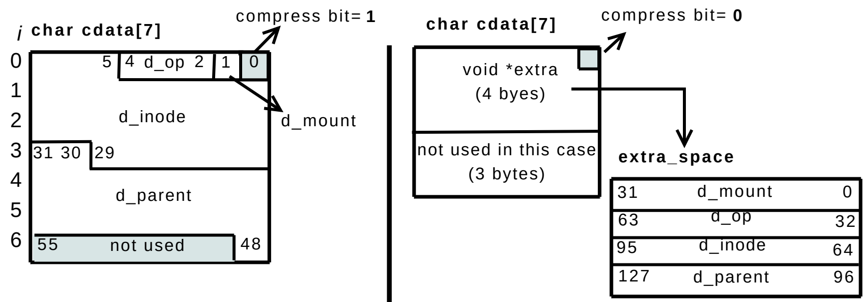
4 Data Structure and Source Code Transformation

After the compression scheme for each compressible field in a data structure has been determined, all the compressed fields are packed into an array *cdata* that is just large enough to accommodate the total number of bits in the compressed data structure. We use char as the type of array $cdata$ because char is the smallest data type in C.

Figure 2 a) shows the memory layout of array $cdata$ in the compressed dentry structure. We use one bit, called *compressed bit*, per *cdata* array(the lowest bit in the first byte of $cdata$) to indicate whether it contains compressed data or whether *cdata* contains a forwarding pointer to an uncompressed representation of that structure (i.e., some field value was incompressible). Initially, this bit is set to 1, which means $cdata$ contains compressed values. Later, if an incompressible value is encountered for any compressed field of that structure, this bit is set to 0 and the first word of *cdata* is set to point to an uncompressed representation of the structure. More detail is discussed in Section 4.1.

For each compressed data structure, two tables, shown in Figure 2, are created automatically. The first table, *compress access table*, stores the information about how to handle compressed values for each compressed field. The second table, *expansion access table*, contains information about how to handle a value once an instance of a compressed data structure is expanded to store an incompressible value. The compress access table is used to access the compressed representation *cdata* of a compressed structure as long as all the data values are compressible. Consider the compress access table shown in Figure 2 a). fid is a unique id assigned to each compressed field for fast look-up

[3] In fact, the total number of distinct values that appeared in d_inode is larger than 4096. However, the table used for value profiling is set to hold a maximum of 4096 values. Note that even though the number of distinct values recorded saturates at 4096, this does not compromise soundness because in this case, saturation simply means compression-table scheme is not applied.



compress access table

fid	*field*	*start*	*bits*	*scheme*
0	d_mount	1	1	NW
1	d_op	2	3	CT
2	d_inode	5	25	CP
3	d_parent	30	19	CP+CT

expansion access table

fid	*field*	*start*	*bits*	*extra*
0	d_mount	0	32	*True*
1	d_op	32	32	*True*
2	d_inode	64	32	*True*
3	d_parent	96	32	*True*

a) Memory layout of compressed fields in cdata array and the compression access table.

b) Memory layout of cdata array and extra allocated space after expansion and the expansion access table.

Figure 2: Memory layout of compressed data structure before and after expansion.

Procedure $compress(S, v)$
if $(S.type = NW)$ **then** /* narrow width */
  $v' \leftarrow v$
else if $(S.type = CP)$ **then** /* common prefix */
  $v' \leftarrow v \;\&\; S.offset\_mask$
else /* compression-table + common prefix */
  $prefix \leftarrow v \& S.prefix\_mask$
  $idx \leftarrow$ index of $prefix$ in $S.compress\_table$
  $v' \leftarrow (idx \ll S.offset\_bits) \mid (v \;\&\; S.offset\_mask)$
return v'

Procedure $decompress(S, v)$
if $(S.type = NW)$ **then** /* narrow width */
  $v' \leftarrow v$
else if $(S.type = CP)$ **then** /* common prefix */
  $v' \leftarrow S.prefix \mid v$
else /* compression-table + common prefix */
  $idx \leftarrow$ extra index from v
  $prefix \leftarrow S.compress\_table[idx]$
  $v' \leftarrow S.prefix \mid v$
return v'

Note: The operators &, |, and << denote bitwise-and, bitwise-or, and left-shift operations.

Figure 3: Procedures for compression and decompression a value v according to compression scheme S.

into the table. $start$ is the bit location where a compressed field starts in *cdata* and $bits$ is the bit width of a compressed field. Lastly, $scheme$ is the compression scheme of a compressed field as we discussed in Section 3.2.

The size of array *cdata* can be computed as $\lceil \frac{\sum Bits_f + 1}{8} \rceil$, where $Bits_f$ is the bit width of a compressed field f. For example, the size of *cdata* for compressed `dentry` structure is $\lceil \frac{1+3+25+19+1}{8} \rceil = 7$

4.1 Maintaining compressed data

The main issue in dealing with compressed structures is that while the decision to compress specific fields of a structure are made statically, the actual runtime representation of such a structure may or may not be compressed, depending on the values that have been stored into it. When accessing a compressed data structure at runtime, we therefore have to check the compress bit to determine the actual representation of that structure. Suppose that we have a compressible data structure T whose compressed representation is T'. As execution progresses, instances of T' are accessed and maintained as follows:

Allocate and free. Allocation and freeing of dynamically allocated data proceeds as expected: when a compressed structure is allocated, the compress bit is set to 1; when it is freed, its compress bit is checked, and if this bit is found to be 0, i.e., the structure was expanded, then the expanded representation is freed as well.

Read from a compressed field. When loading the value of a field within an instance of T', if the structure is in compressed form, the compress access table is looked up to determine the location within the *cdata* array of the compressed bits for that field as well as the compression scheme used. This information is then used to access the compressed bits for the field. The $decompress()$ routine, shown in Figure 3, is then used to transform these bits to an uncompressed value that can be used in a computation. If the structure is in uncompressed form, the forwarding pointer in *cdata* is used to access the uncompressed data, and the value of the field is accessed using location and size information obtained from the expansion access table.

Write to a compressed field. When storing a value v to a compressed field f in T', we have the following cases. (1) If T' is compressed and v is small enough to fit in the compressed representation of that field, then we use the function $compress()$, shown in Figure 3, to create the compressed representation v' of v. We then use the compress access table to determine the location and size of the compressed field within *cdata*, and write the compressed bits there. (2) If T' is compressed but v is too large to fit into the compressed representation of the field, we have to first create an expanded representation. This is done as described below. The value v is then written to the appropriate location within the expanded representation. (3) If T' is not compressed, we use the expansion access table to determine the location and size of the field in the expanded representation, and write v at that location.

If an incompressible value is encountered for any compressed field at runtime, all of the compressed fields are expanded to their original size according to the expansion access table. Consider the example shown in Figure 2 b), which illustrates how an instance of compressed `dentry` structure is expanded. First, extra space is allocated to hold decompressed values. The first four bytes of *cdata* are used to store the address of the extra allocated space. The remaining space in *cdata* is reused as much as possible, but in the example of `dentry`, the remaining three bytes are left unused. The compressed values stored in *cdata* are decompressed and stored in new locations according to the expansion access table. Assigning the address of extra space to the first four bytes in *cdata* also has the effect of setting the compress bit in *cdata* to 0.[4]

We made the design decision to expand all the compressed data in an instance of compressed data structure whenever any of the fields within that structure is assigned an incompressible value. The reason for this decision is that there is at most one expansion operation for each instance and only one pointer is needed to keep the addresses of extra space. It is possible to expand compressed fields separately, but that can lead to multiple allocations of extra space (which is expensive) and also require multiple pointers. Once an instance of a compressed data structure is expanded, it is always maintained in uncompressed form. This is to simplify the maintenance of an expanded instance, and also to avoid repeatedly expanding and converting back to compressed form.

4.2 Soundness considerations

Data structure compression changes the way in which structure fields are represented and accessed. To preserve safety, we have to make sure that such changes do not affect the observable behavior of the program. Intuitively, the requirement for this is that a field f of a structure S is considered for compression only if the only way in which f can be accessed in the program is via expressions of the form '$S.f$' or 'p->f', where p is a pointer to S. This property ensures that we can use type information to ensure that all accesses to a compressed field have the appropriate compression/decompression code added to them. We enforce this requirement as follows: a field f of a structure S is excluded from compression if any of the following hold: (1) the address of f is taken, e.g., via an expression of the form $\&(S.f)$; (2) a pointer p to the structure S or to the field f is cast to some other type (in either case, it would be possible to bypass the compression/decompression code when accessing f); or (3) an offset is used to access a field within S, e.g., $(\&S)+4$. These restrictions exclude from compression any field of a structure that can have a pointer to it. This is important because the process of comparison can change the relative order of fields that are compressed and fields that are not compressed: the former get pulled into the *cdata* array, the latter do not. The conditions given above ensure that code that may be sensitive to the layout of fields within the structure are precluded from compression.

5 Experimental Evaluation

We evaluated our ideas using the Linux kernel version 2.6.19. In order to emulate an embedded system environment, the experiments were conducted on an old laptop machine with Intel Pentium III 667MHZ processor and 128MB of memory. The data structure profiling is done by modifying GCC (4.2.1) to insert profiling code for every statement

[4] This assumes that dynamic memory allocation routines such as *malloc* return addresses that are at least even-address aligned (common implementations of *malloc* satisfy this requirement, e.g., the GNU C library returns blocks that are at least 8-byte aligned).

Cache Name	*Object type*	*Size (KB)*	*Ratio (%)*	*Cache Name*	*Object type*	*Size (KB)*	*Ratio (%)*
ext2_inode_cache	`ext2_inode_info`	1034	46.8	inode_cache	`inode`	51	2.3
dentry_cache	`dentry`	450	20.4	sysfs_dir_cache	`sysfs_dirent`	37	1.8
proc_inode_cache	`proc_inode`	137	6.2	bio	`bio`	18	0.8
buffer_head	`buffer_head`	67	3.0				

Table 2: Slab caches that are compressed (sorted by cache size in non-increasing order)

that contains field referencing expression in a program. The source code transformations are done manually at present. However, it is possible to automate the process using a source-to-source transformation tool, such as *CIL* [11].

5.1 Selecting the dynamic data structures to compress

In our current implementation, we compress part of the slab caches in the Linux slab allocator (recall that, as discussed in Section 2, this forms the core of dynamic memory management in the Linux kernel). The compressed slab caches are listed in Table 2. Column 1 gives the names of compressed slab caches; column 2 is the data structure type used by each slab cache; column 3 shows the size of each slab cache based on profiling; and column 4 is the ratio of the size of each slab cache to the total memory space in the slab allocator. Overall, the slab caches in Table 2 account for over 81% of all memory space used by the Linux slab allocator. This is also the reason why we selected them. We ignore the remaining slab caches for two reasons: first, most of the remaining slab caches consume only small amount of memory even though they can be compressed; second, there are several slab caches(account for about 14% of the total memory space) that can not be compressed with our current implementation, e.g., we currently do not compress array fields, and the main data field in the slab cache `radix_tree_node`, which is also the largest one that we do not compress, is an array.

Data structure compression can save memory space but it can also bring cost—compression/decompression of a compressed field are much more expensive than the store/load operations of an uncompressed field. Based on the profile information, all the fields can be classified into two categories: *hot* fields, which are the fields used frequently, and *cold* fields, which are the fields used infrequently. Conceptually, we should not compress hot fields because it may cause significant amount of overhead even though more memory space can be saved. To control this cost-benefit tradeoff, we use a user-specified threshold $r \in [0.0, 1.0]$ to determine the fraction of compressible fields (i.e., hot fields) that should not be compressed.

Let *cost* be the number of load/store of a compressible field based on profiling. Let C be the total number of load/store of all compressible fields in a program.[5] Given a value of r, we consider all the compressible fields f in the program in decreasing order of $cost(f)$ and determine the smallest value of N such that

$$\sum_{f:cost(f)>N} cost(f) \leq C \cdot r.$$

Any compressible field whose number of load/store is larger than N is avoided from compression. For example, r=0.0 means all compressible fields are compressed; and r=1.0 means nothing is compressed.

5.2 Experimental results

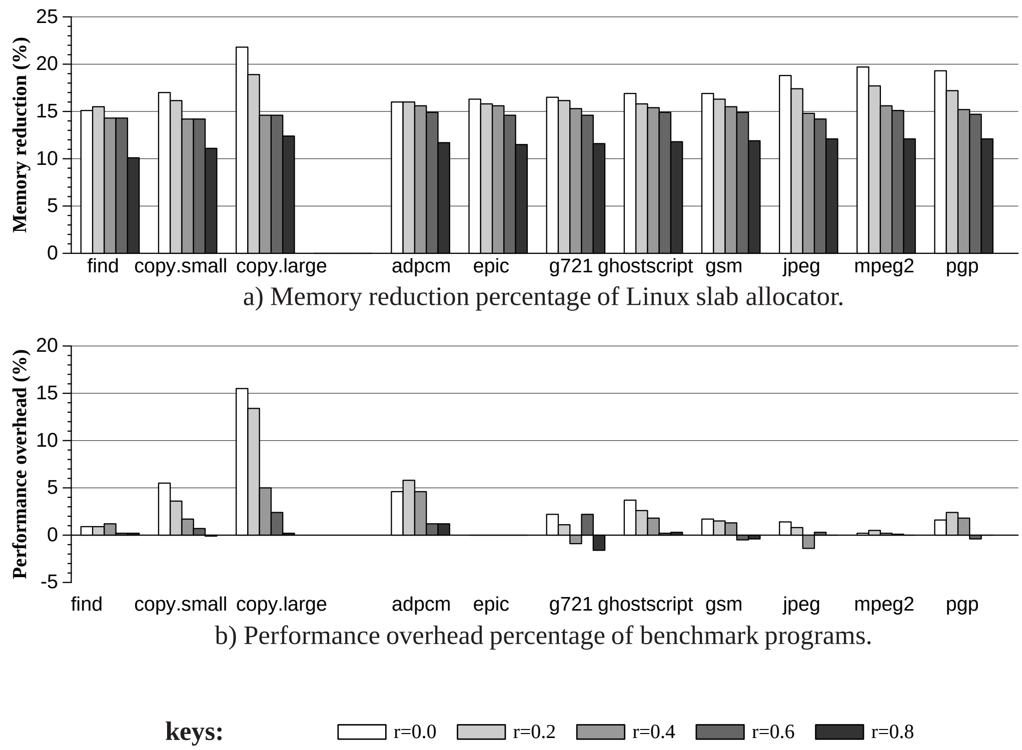
We used two sets of benchmarks to evaluate the runtime impacts of our approach: (1) a set of kernel-intensive benchmarks: *find*, which runs the `find` command at the root directory to scan all files in the file system, *copy.small*, which makes a copy of 5MB file and *copy.large*, which makes a copy of 20MB file; and (2) a collection of eight application programs from the MediaBench suite [10], used for evaluating multimedia and communications systems. These two sets of benchmarks were tested separately while all programs in each set were executed sequentially (for instance, the execution sequence of kernel-intensive benchmarks is *find*, *copy.small*, *copy.large*).

Table 3 shown the average percentage of memory reduction in Linux kernel slab allocator and average percentage of performance overhead for kernel-intensive benchmarks and MediaBench benchmarks. The values of r considered in our experiments are {0.0, 0.2, 0.4, 0.6,0.8}. As we can see in Table 3, both the average memory reduction and average performance overhead decrease when the value of r increases. When r=0.0, there is about 18% memory reduction

[5] In case of our experiment in the Linux kernel, all compressible fields are defined as the compressible fields in the data structures in Table 2.

Threshold r	Kernel-intensive		MediaBench	
	Memory reduction	*Speed Overhead*	*Memory reduction*	*Speed Overhead*
0.0	18.0%	7.3%	17.5%	1.9%
0.2	16.9%	6.0%	16.5%	1.8%
0.4	14.3%	2.7%	15.4%	0.9%
0.6	14.3%	1.1%	14.7%	0.4%
0.8	11.2%	0.1%	11.9%	-0.1%

Table 3: Average memory savings and overhead of kernel-intensive benchmarks and MediaBench.



a) Memory reduction percentage of Linux slab allocator.

b) Performance overhead percentage of benchmark programs.

Figure 4: Runtime impacts of dynamic data structures compression on memory saving and performance overhead.

for both benchmarks while the overhead of MediaBench (1.9%) is much lower than the overhead of kernel-intensive benchmarks (7.3%). However, even for kernel-intensive benchmarks, the overhead reduces to only 1.1% when r=0.6 and there is still about 14% of memory reduction.

The detailed results of memory reduction percentage and performance overhead percentage for each program are shown in Figure 4 a) and b) respectively. Generally, there is more overhead for kernel-intensive benchmarks than MediaBench benchmarks. *copy.large* has the largest overhead(over 15%) among all programs when r=0.0. But its overhead reduces to below 5% when $r \geq 6$. The results shown in Figure 4 a) also include the extra allocated space from expansion and the size of extra allocated space is relative small (less than a page (4KB)) for both sets of benchmarks.

Table 4 shows the static impacts of our approach on size reduction of compressed kernel data structure and kernel code size. On average, there is 28.7% size reduction of all the compressed data structures in the Linux kernel when r=0.0. The number drops to 21.5% when r=0.8. The increase of code size caused by code transformation is about 5.4% when r=0.0 and the increase reduces to about 2.5% when r=0.8.

5.3 Discussion

The reason for low overall performance overhead in the Linux kernel is that, even though data structure compression causes a performance overhead within the Linux kernel, the actual time spent within the kernel is only a small portion

Threshold r	0.0	0.2	0.4	0.6	0.8
Avg. data structure size reduction	28.7%	27.0%	25.9%	24.6%	21.5%
Increase of code size	5.4%	4.9%	3.9%	3.6%	2.5%

Table 4: Static impacts on data structure size and code size.

of the total running time of the whole system. For programs in MediaBench, most of the running time is spent in the user processes. Even for the kernel-intensive benchmarks, a significant amount time is spent in data copying between the kernel and the user applications. In order to evaluate how data structure compression can affect computation-intensive applications, we applied our approach to several benchmarks in the Olden test suite [3],[6] which consists of a set of pointer intensive programs that make extensive use of dynamic data structures. Due to space constraints, we only summarize the results here.

When compression is applied to on all fields regardless of how heavily they are used, we see significant performance degradation. The average reduction in memory consumption ranges from a little over 30%, for small inputs, to about 23% for large inputs. This is accompanied by slowdowns ranging, on average, from about 183% for small inputs to 194% for large inputs. When the most frequently accessed field is excluded from compression, there is a drop in the amount of memory usage reduction obtained, as one would expect: about 20% on average for small inputs and 6% for large inputs. However, because the remaining fields that are compressed are still accessed quite frequently, this still incurs significant runtime overhead, averaging about 88% for small inputs and 116% on large inputs. Interestingly, in the second case one of the benchmarks consumed more space with data compression than the original version: the reason for this is that the set of values encountered in the profiling data (small inputs) did not reflect the range of values encountered in the large inputs, leading to many compressed representations having to be expanded at runtime, thereby incurring an extra cost of 4 bytes per expanded representation to hold a pointer to the expanded representation.

The performance results for the Olden benchmarks illustrate that data structure compression can lead to significant performance overheads if applied to heavily used data. In order to be a good candidate for data structure compression, a program should use plenty of compressible data (to obtain significant memory size reductions) which are used relatively infrequently (to keep the runtime performance overhead low). The OS kernel on embedded systems typically meets these requirements. Our experiments with the Linux kernel bear this out, with good reductions in dynamic memory usage with only a very small performance overhead.

6 Related work

The introduction of 64-bit architectures has led a number of researchers to investigate the problem of compressing pointers into 32 bits [1, 12, 9]. Our work differs from these in that we can compress both pointer data and non-pointer scalar data, so a wider set of applications can be benefit from our approach.

Zhang and Gupta [14] use a hardware-based scheme to reduce the memory usage of dynamic data by compressing a 32-bit integer and a 32-bit pointer into 15-bit values, which then are packed into a single 32-bit field. Since their approach compresses each field in a uniform way, it can potentially cause two problems: first, space is wasted if compressible fields require few bits than 15; and second, more expansions happen at runtime if compressible fields require more bits than 15. Also, it requires specialized hardware to improve performance.

Cooprider and Regehr [6] apply static whole-program analysis to reduce a program's data size including statically allocated scalars, pointers, structures, and arrays. However, their work targets small systems on microcontrollers that do not support dynamic memory allocation. Priot work on reducing the memory requirement of OS kernels focuses only on static memory data in the OS kernels, i.e., code and global data [4, 5, 8]. We are not aware of previous work that exploits the opportunities of compressing dynamic data in the context of OS kernels.

7 Conclusions

Embedded systems are usually memory-constrained. This makes it important to reduce the memory requirements of embedded system software. An important component of this is the memory used by the operating system. This paper describes an approach to reducing the memory requirements of the dynamic data structures within the operating system. We use value profiling information to transform the OS kernel code reduce the size of kernel data structures. Experimental results show that, on average, our approach reduces the memory consumption of the slab allocators

[6] The Olden benchmarks we considered are *perimeter*, *treeadd*, and *tsp*.

in Linux by about 17.5% when running the MediaBench suite, while incurring only minimal increases in code size (5.4%) and execution time (1.9%).

References

[1] A.-R. Adl-Tabatabai *et al.* Improving 64-bit Java IPF performance by compressing heap references. In *CGO '04: Proc. International Symposium on Code Generation and Optimization*, pages 100–110, 2004.

[2] B. Calder, P. Feller, and A. Eustace. Value profiling and optimization. *Journal of Instruction Level Parallelism*, vol. 1, March 1999.

[3] Martin Christopher Carlisle. *Olden: parallelizing programs with dynamic data structures on distributed-memory machines*. PhD thesis, Princeton, NJ, USA, 1996.

[4] D. Chanet, B. De Sutter, B. De Bus, L. Van Put, and K. De Bosschere. System-wide compaction and specialization of the Linux kernel. In *Proc. 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'05)*, pages 95–104, June 2005.

[5] D. Chanet, B. De Sutter, B. De Bus, L. Van Put, and K. De Bosschere. Automated reduction of the memory footprint of the linux kernel. *Trans. on Embedded Computing Sys.*, 6(4):23, 2007.

[6] N. Cooprider and J. Regehr. Offline compression for on-chip ram. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming language Design and Implementation*, pages 363–372, June 2007.

[7] B. Fitzgibbons. The Linux slab allocator. `http://citeseer.ist.psu.edu/fitzgibbons00linux.html`.

[8] H. He, J. Trimble, S. Perianayagam, S. Debray, and G. Andrews. Code compaction of an operating system kernel. In *Proc. International Symposium on Code Generation and Optimization (CGO)*, pages 283–295, March 2007.

[9] C. Lattner and V. S. Adve. Transparent pointer compression for linked data structures. In *MSP '05: Proceedings of the 2005 workshop on Memory system performance*, pages 24–35, New York, NY, USA, 2005. ACM.

[10] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communications systems. In *Proc. 30th IEEE International Symposium on Microarchitecture (Micro '97)*, pages 330–335, December 1997.

[11] George C. Necula, Scott McPeak, Shree Prakash Rahul, Shree Prakash Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 213–228, London, UK, 2002. Springer-Verlag.

[12] K. Venstermans, L. Eeckhout, and K. De Bosschere. Object-relative addressing: Compressed pointers in 64-bit java virtual machines. In *In Proc. ECOOP '07*, volume 4609, pages 79–100. Springer-Verlag, 2007.

[13] S. Watterson and S. K. Debray. Goal-directed value profiling. In *Proc. Tenth International Conference on Compiler Construction (CC 2001)*, April 2001.

[14] Y. Zhang and R. Gupta. Data compression transformations for dynamically allocated data structures. In *CC '02: Proc. 11th International Conference on Compiler Construction*, pages 14–28. Springer-Verlag, 2002.