Cooperative Memory Management of a JavaScript Virtual Machine with Datatype based Hardware Memory Deduplication

Zihan Li, Tomoharu Ugawa, Ryota Shioya

Recently, a memory deduplication hardware design called BCD was proposed. BCD leverages the similarity of the bit patterns between cache lines. Later, another improvement, allocating a single object aligned at the head of each cache line and filling the remaining part with zero, achieves better deduplication for object data. However, more consumption of virtual address space is introduced, resulting in a larger translation table for BCD mapping from addresses to real location in memory where deduplicated data is stored. In this paper, we allocate multiple small objects of the same type in a single cache line to reduce the consumption of virtual addresses. Furthermore, bit patterns of cache lines holding objects of the same types are expected to be similar because all objects in the same cache line are of the same type. We implemented this in a JavaScript virtual machine, eJSVM, and evaluated it. Furthermore, we confirmed that our proposal reduced memory consumption by using a simulator of the deduplication hardware.

1 Introduction

Nowadays, to improve the efficiency of programming in embedded systems, people are seeking to apply high-level languages such as JavaScript instead of traditional C or C++. Some JavaScript engines oriented to embedded systems such as eJS [1], QuickJS [2] are proposed to facilitate embedded system programming. Since embedded systems provide limited memory space, reducing the memory footprint is important.

To this end, Ri et al. proposed AOBD, a co-design of memory compression hardware and virtual machines (VMs) of object-oriented managed languages [3]. More specifically, AOBD uses base and compressed difference (BCD) deduplication [4]. It is a hardware design that compresses data in cache memory when they are written back to main memory. Because BCD decompresses data when it is loaded to cache, or cache fill, BCD is transparent to software. In the rest of this paper, we call a cache line-sized memory block in virtual memory space, which is subject to compression, simply a cache line.

BCD compresses memory by eliminating duplication from similar cache lines. It stores in memory the contents of some representative cache lines, and the differences between other cache lines and their similar representatives. Thus, a high compression ratio is achieved when the bit patterns of cache lines are categorized into a small number of categories, and when bit patters of cache lines in the same category are similar.

Regarding to the VM side, AOBD allocates objects at aligned addresses to the cache line size. Assuming that objects of the same type are likely to have similar bit patterns in the same field, which is stored at the same offset from the head of objects, this aligned allocation avoids divergence of bit patterns. Furthermore AOBD categorizes cache lines based on the types of the objects in the cache lines. Ri et al. implemented AOBD in a JavaScript VM for embedded systems, eJSVM [1], which is equipped with the optimization that represents objects of JavaScript as if they have static types.

However, aligned allocation costs large virtual memory space. Aligned allocation means that it allocates at least one cache line for each object, and it does not use the remaining area of cache lines. In BCD, large virtual memory space de-

Zihan Li, 鵜川 始陽, 塩谷 亮太, 東京大学情報理工学系研究科, Graduate School of Information Science and Technology, The University of Tokyo.

creases the end-to-end compression ratio because BCD has a translation table mapping from cache line-sized memory blocks in virtual memory to the location of its compressed data, and the table also consumes main memory.

We propose a technique to place multiple objects in a single cache lines while avoiding the divergence of the bit patterns. This reduces the consumption of virtual memory space, which finally contributes to a better compression ratio by reducing the size of the translation table.

The challenge is to avoid the divergence of the bit patterns introduced by the second object in a cache lines. We solve this problem by restricting each cache line to contain only the same type of objects. We implemented this idea in eJSVM. In the implementation, we managed unused areas of cache lines using segregated freelists. As a result of our evaluation, the proposal improves the compression ratio by 9.0% on average and by up to 13.6% compared to AOBD.

2 Hardware Deduplication

2.1 BCD

BCD deduplication [4] is a hardware design that eliminates duplicated bits in cache lines when data in cache memory is written back to main memory (in the rest of this paper, we denote memory for main memory). It is based on the observation that if two cache lines share the same bit pattern in certain bits, then the remaining part of these cache lines tend to have similar bit patterns. BCD calls the value in certain bits *signature*. BCD leverages this similarity to compress data; it stores only the differences in memory. The more bits that are the same in the remaining part, the fewer bits are needed to encode them. Therefore, the compression ratio is improved.

As Fig. 1 shows, BCD provides a new layer of addresses called the OS physical address (OSPA) to the memory system. OSPA is between the virtual memory address and the hardware memory address. As a result, when data is stored in memory, the OS is responsible for converting the virtual memory address to the OSPA, and BCD is responsible for converting the OSPA to the hardware memory address. BCD deduplicates when BCD converts addresses.

BCD uses the following four data structures to manage hardware memory.

- translation table
- base array
- difference array
- overflow region

The translation table is used to record the conversion from OSPAs to hardware memory addresses. Each table entry translates from an OSPA to a hardware memory address. The base array is a hash table to store uncompressed contents of representative cache lines. The key for the hash table is the signature of the cache line. The difference array is a hash table to store compressed contents. As for the overflow region, if the base or difference array is full for a certain signature, then the new uncompressed or compressed cache line data is stored in the overflow region.

When a cache line is evicted, BCD compresses the data in the following steps.

1. BCD extracts the signature of cache line to be evicted. The signature is computed by combining the leading 2 bytes of every 8-byte unit in one cache line, as Fig. 2 shows.
2. BCD looks up the base array by the signature. If a base entry with the same signature is not found, data in the evicted cache line are stored to a new base entry in the base array.
3. If a base entry is found, the remaining data (every ending 6 bytes of every 8-byte unit, see Fig. 3) are compared. If the base entry has exactly the same data as that of the evicted cache line, the base line entry is shared.
4. Otherwise, BCD computes the difference between the data from the evicted cache line and that of the base entry by bit-wise xor. BCD compresses it with the LZC compression algorithm and stores in the difference array. Again, if the difference array already has the exactly entry, the entry is shared.

BCD works well if, for cache lines with the same signature, the remaining parts have similar bit patterns.

2.2 AOBD

In object-oriented managed languages, all the data is represented by objects. Ri et al. proposed AOBD [3], a combination of an improved BCD and a JavaScript VM that achieves a high compression

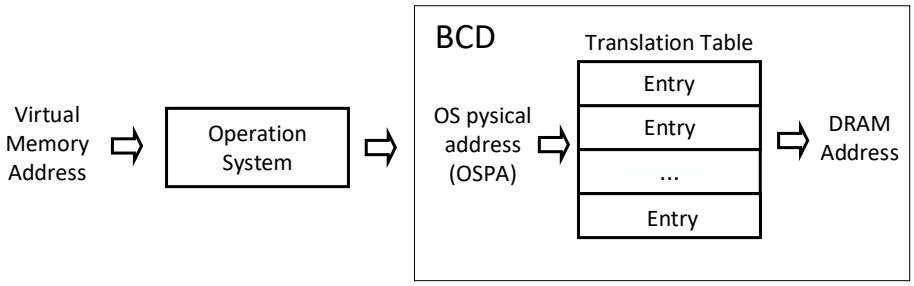

Fig. 1 Address Translation

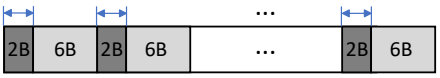

Fig. 2 BCD Signature

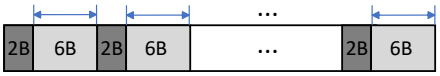

Fig. 3 Remaining Part For Compression

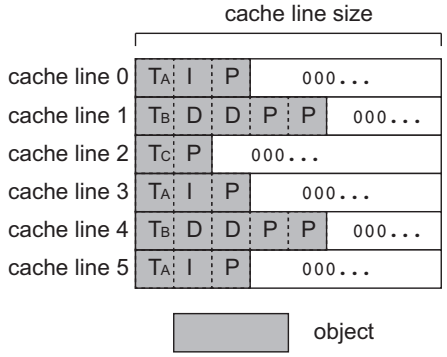

Fig. 4 Object Allocation in AOBD

ratio for the heap of the JavaScript VM. AOBD is based on the observation that objects of the same type have values of the same low level type, such as int, double, or a pointer, in the same fields, and the values of the same low level type tend to have similar bit patterns.

To leverage the similarity of bit patterns between objects of the same type, AOBD allocates objects at the head of cache lines as shown in Fig. 4. This figure shows six cache lines, each of which has an object of type T_A, T_B, or T_C. The dotted boxes with T_A, T_B, T_C are fields containing the object types. The symbols in other dotted boxes, I, D, and P, represent the low level types of the fields. Owning to aligned allocation, fields with the same low level type of objects of the same type are arranged at the same offset to the head of their cache lines. For example, each of cache lines 0, 3, and 5 has an object of type T_A. These cache lines have low level type I in the second word and P in the third word. As a result, cache lines with objects of the same type are likely to have similar bit patterns.

Furthermore, AOBD fills the remaining area of each cache line with zero to avoid diversity of bit patterns. Cache lines with objects of the same type have zero-filled areas at the same offset to the head of the cache lines. Thus, when BCD deduplicates, the difference of these zero-filled areas is computed and the difference is compressed to virtually zero bit. However, these zero-filled areas of cache lines registered to the base array would consume memory. To avoid this waste, AOBD has an option to compress trailing zero bits in the base array.

Finally, AOBD uses the value of the type information field of an object as the signature of cache line containing the object. As a result, signatures of cache lines with similar bit patterns are computed to the same value.

3 Allocate the Same Type of Objects within a Single Cache Line

Although AOBD improves the compression ratio, the cache line alignment enlarges the size of translation table. Since one cache line can hold only one object regardless of its size due to cache line alignment, each cache line consumes one virtual memory address in Fig. 1. Therefore, if we want to store two objects whose sizes are 32 bytes and the size of a cache line is 64 bytes, by cache line alignment, we consume two virtual memory addresses, each one

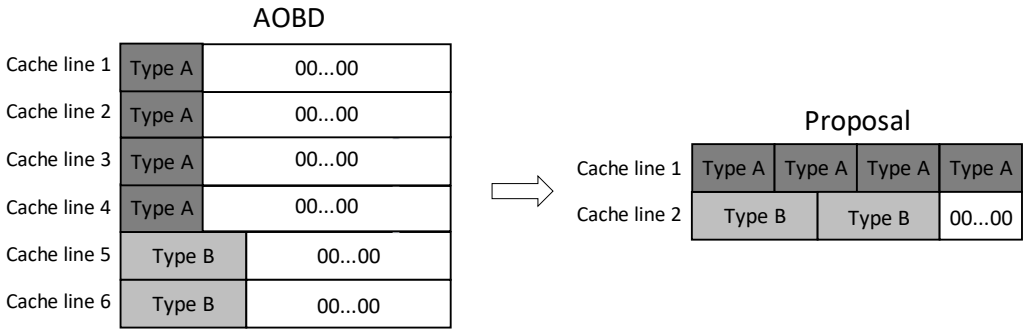

Fig. 5 Multiple Same Type of Objects in Single Cache Line

is for a cache line. However, originally, both objects can be stored in one cache line, and only one memory address is consumed. As a result, AOBD consumes more virtual memory addresses. Fig. 1 shows that each virtual memory address consumed is converted to an OSPA. For each OSPA, there is a corresponding entry in translation table to record the conversion from the OSPA to a hardware memory address. Therefore, the size of translation table is increased.

To solve the problem above, we propose the allocation of the same type of small objects within a single cache line. Here, the 'small object' means that its size is less than half of one cache line size. The main idea to reduce the consumption of virtual memory address in AOBD as well as keep the signature valid for memory compression is to allocate multiple same type of small objects into single cache line, as Fig. 5 shows. To do that, all the signature is captured from the first object. The way to calculate the signature is the same as that for AOBD. The signature is computed from special fields that contain the type information in the first object. Objects of the same type share the same signature. Therefore, we check only the first object. Since one cache line holds only the same type of objects sharing the same signature and the same signatures indicate similar bit patterns in the remaining part of the cache lines, therefore, the signature is valid for memory compression and deduplication. In addition, we apply the following strategies to avoid introducing more bit patterns for certain cache lines that contain the same types of objects. In AOBD, zero filling is applied to all unused space in the heap. However, here, we only do zero-filling if the whole cache line in the heap becomes unused. As long as at least one object in a cache line is alive after garbage collection (GC), we do not fill the block with zero for the unused part in it.

4 Implementation to eJSVM

4.1 eJSVM

We implemented our proposal in eJSVM as Ri et al. implemented AOBD in eJSVM. eJSVM is a JavaScript virtual machine (VM) for embedded systems. Because JavaScript is a dynamic language, objects do not have types, but properties are dynamically added to objects. Thus, AOBD does not work for a naive implementation of JavaScript VM. However, eJSVM uses optimizations to deal with objects as if they have types: hidden classes, pre-transitioning of the hidden classes, and in-object properties to implement JavaScript objects [5]. In this section, we describe the eJSVM equipped with AOBD.

4.1.1 JavaScript Objects

The layout information of an object is recorded in a separate meta-object, called *hidden class*. The hidden class is shared with objects with the same layout. Because JavaScript object may get a new property dynamically, the layout may change after the object is created. eJSVM feedbacks this change to the allocation site of the object (the location in the JavaScript program that created the object), so that hereafter objects are created with the new layout.

Figure 6 shows an example of the implementation of a JavaScript object. It comprises two parts. One is a `JSObject` data structure and the other is a hidden class. The hidden class is a mapping from the property names to the indexes of them in the storage. The `JSObject` has a pointer to the hidden

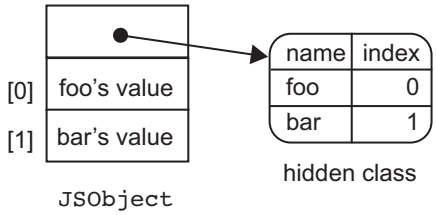

Fig. 6 JSObject

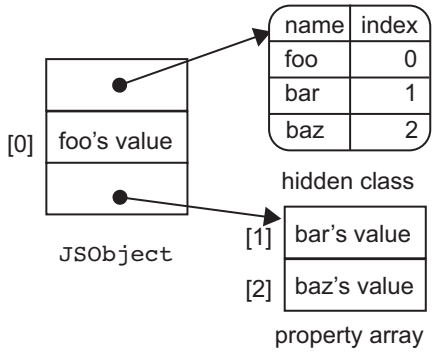

Fig. 7 Property Overflow

class and the storage for property values. For example, the index of the property `bar` is one according to the hidden class. Thus, its value is stored in the third word of the `JSObject`. Note that the first word of the `JSObject` is used to hold the pointer to the hidden class.

The size of `JSObject`s *differ* from object to object depending on the expected number of properties, which allocation site knows. When an unexpected property is added to a JavaScript object, however, the `JSObject` cannot be expanded. In such a case, an external *property array* is introduced to hold the overflowed properties, and the last slot of the `JSObject` is replaced with the pointer to the property array. Figure 7 shows the object shown in Figure 6 after a property `baz` is added. The pointer to the hidden class is updated, and a property array is created to hold the overflowed property `baz` and the property that used the last slot, which is overwritten by the pointer to the property array.

4.1.2 Non-JSObject Data

The memory manager of eJSVM deals with `JSObject`, hidden classes, and property arrays as separate data. In addition to hidden classes and property arrays, eJSVM uses several non-`JSObject` data: boxed primitives, property storages and VM internal data structures.

JavaScript's numbers are double-precision floating numbers, which requires 64 bits. To handle all type of data uniformly in one word, eJSVM *boxes* numbers except for small signed integers. Strings are also boxed. When such a primitive data is created, eJSVM allocates a chunk of memory to hold the data, and uses the pointer to the memory chunk as the primitive value.

In addition to the property arrays for overflowed properties, JavaScript's Array objects have separate property arrays. In eJSVM, Array objects' properties of integer indexes, which are technically string property names representing the integers, are stored to the array for quick access.

VM internal data structure including hidden classes is also allocated as separate data chunks. However, the number of them is small.

4.1.3 Identifying Data Type and Size

Each data of eJSVM has a header word that contains *type* and *size* fields. The type field identifies the type of the data, i.e., if it is `JSObject` or not, and if not, the type of non-`JSObject` data. Because some data structures, such as `JSObject`s and property arrays, have not fixed sizes, the header word also describes the size of the data.

Figure 8 shows a `JSObject`, a property array, and a boxed number with the header. The type fields of the headers describe corresponding types. The size field for the `JSObject` with n properties is $n + 1$, that for the property array with m properties is m, and that for the boxed number is one.

4.1.4 Signature of Cache Line

For `JSObject` data, AOBD uses the pointer to the hidden class as the signature of cache line that holds the `JSObject`. Fig. 9 shows the algorithm to compute the signature for the cache line whose address is `cache_line_addr`. To distinguish `JSObject` data from non-`JSObject` data, AOBD checks the value of `type` field in the header. For non-`JSObject` data, AOBD uses the combination of the `type` and `size` fields as the signature of the cache line.

For the second or following cache lines of a large object, which is larger than the cache line size, there are no bits describing type information. AOBD does not handle these cache lines correctly; it parses the first word of a cache line as if it is a header word even for the second or the following cache lines, and

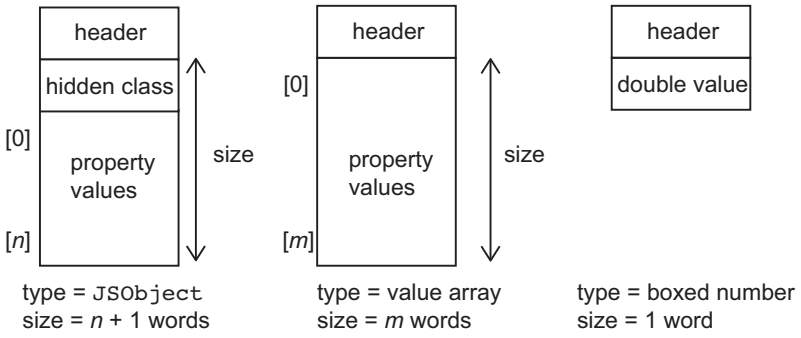

Fig. 8 Type and Size Fields of Header

```
compute_signature(cache_line_addr):
  header = *(HeaderType*) cache_line_addr;
  if header.type == JS_OBJECT:
    obj = (JSObject *) cache_line_addr;
    return obj->hidden_class;
  else:
    return (header.type, header.size)
```

Fig. 9 Computing Signature

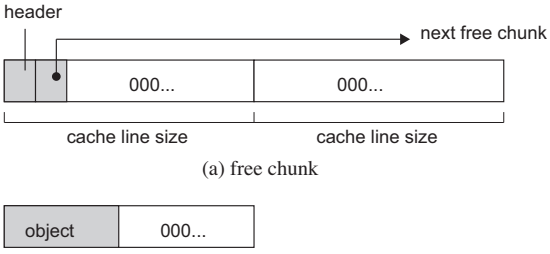

Fig. 10 Free chunk and memory block in use

if the false type field indicates that it is a `JSObject`, it uses the combination of the false type field and the second word, which is a false hidden class field, as the signature.

4.1.5 Memory Management

AOBD uses a modified version of mark-sweep GC with a first-fit free-list memory allocator implemented in eJSVM [6]. Because AOBD allocates a single object in each cache line, the memory manager manages memory at a granule of the cache line size. To avoid unnecessary variety of cache lines, the modified memory manager fills the unused area of both allocated and free memory chunks with zero. Furthermore, the modified mark-sweep GC uses bitmap marking.

More specifically, elements of the free list are one or more consecutive cache lines. Figure 10 (a) shows a free chunk of two cache lines. The first block has a header describing the size of this free chunk and a pointer to the next free chunk. The remaining area and following cache lines are filled with zero.

When eJSVM allocates memory of a certain size, it rounds up the size to the multiple of the cache line size. For a small object that fits in a single cache line, an entire single cache line is allocated for the object. The object is placed at the head of the cache line as shown in Figure 10 (b). The remaining area of the cache line is filled with zero.

The mark-sweep collector has a bitmap outside of the heap. Each bit of the bitmap corresponds to a cache line. When the mark phase finds a live object, it sets a bit corresponding to the cache line where the live object resides.

4.2 Implementation

Our implementation consists of three parts, free lists to manage partially used block, a two-layer memory allocator and modification to original mark sweep GC [6] in eJSVM.

4.2.1 Free lists for partially used block management

Our proposal allocates multiple small objects of the same type to a single cache line in the heap. However, the heap size is not infinitely large. To efficiently use the limited heap, memory manager collects spaces occupied by objects that are not not needed for application running to recycle spaces for future allocation, which is called garbage collection

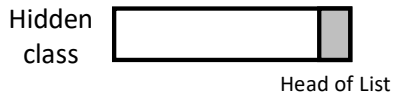

Fig. 11 Free List Head for JSObject

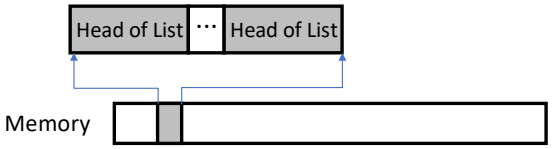

Fig. 12 Free List Head for Non-JSObject

(GC). The objects are not needed for application running during GC is called 'dead objects' and in opposite, the objects are still needed is called 'live object'. As a result, as Fig. 13 and Fig. 14 show, many partially used blocks in which only some slots are filled with live objects and can still hold more objects of the same type are generated after each GC cycle. We call the unused parts in a partially used block 'free slots'. To manage such partially used blocks with free slots, we use a free list for each signature of cache line, which is the pointer to hidden class for JSObject and the combination of the type and size field for non-JSObject.

Since the signatures for JSObject and Non-JSObject consist of different fields in the objects, we need a different way to manage the free lists. For JSObject, we put the head of such a free list into hidden class objects, as shown in Fig. 11. In this way, to find the free list to allocate a free slot in a partially used block for JSObject, we can access the hidden class reference in the JSObject to find the head of the free list. For Non-JSObject, we use a table to manage such free list head, as shown in Fig. 12. The index of such a table is computed by the combination of type and size field in small Non-JSObject headers, so we can also access the head from a Non-JSObject. When a new unused slot is added to a free list, the pointer of the slot is stored in its previous element in the free list.

Fig. 13 and Fig. 14 show what such free lists look like for JSObject and Non-JSObject. The reference to the next element in the free list, an free slot, is stored to a field in the first dead object in the free slot. However, the bit patterns between the reference value and the value of the field overwritten by the referent may be different. Therefore, some differences in the bit pattern are introduced into the block . Therefore, to both record the reference to the next unused slot and make as little modification to the dead object as possible, in our implementation, for Non-JSObject, as Fig. 14 shows, we make one unused slot contain more than one dead object by combining the adjacent dead objects for a larger slot. So, we create one element in the free list with only one modification to a dead object field instead of multiple elements with multiple modifications, which introduces more bit patterns.

As a result, there are two kinds of free lists to manage the free space in our memory manager; one is for partially used block for JSObject and Non-JSObject, the other is for completely unused block with cache line size. We call the former 'slot free lists' and the latter 'block free lists'.

4. 2. 2 Allocator

Here we introduce how the allocator works to allocate those partially used blocks managed by the free lists above. To implement our proposal, we applied two allocators, one is 'free block allocator' which allocates blocks whose sizes are integral multiplication of one cache line size, the other is 'partially used block allocator' for small objects.

When allocation for small object starts, depending on its type and size information, a proper allocator is used. If an object is a small JSObject, then first, a free list for such type of object is checked. We can access the head of such a free list by following the hidden class reference in JSObject and check that hidden class to access the free list head, as shown in Fig 11. If the free list is not empty, then we allocate the space occupied by the head of free list and set the next element in the free list as the head. This is the work of the partially used block allocator. Otherwise, if the free list is empty, then we use free block allocator to allocate a block with cache line size and record the unused slots in that block to the corresponding free list so that during the next allocation for the same type of small object, we can use partially used block allocator for it.

Fig. 15 shows the transfer of states of one cache line that holds at most two small objects of the same type. At state 1, the block is unallocated, managed by free block allocator, and filled with zero. Then when an allocation starts and free block

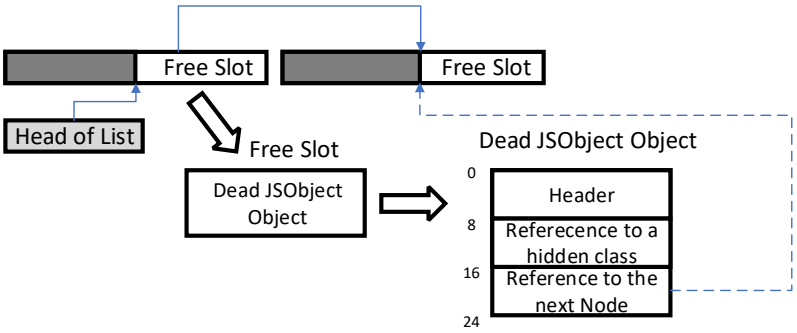

Fig. 13 Free List for JSObject

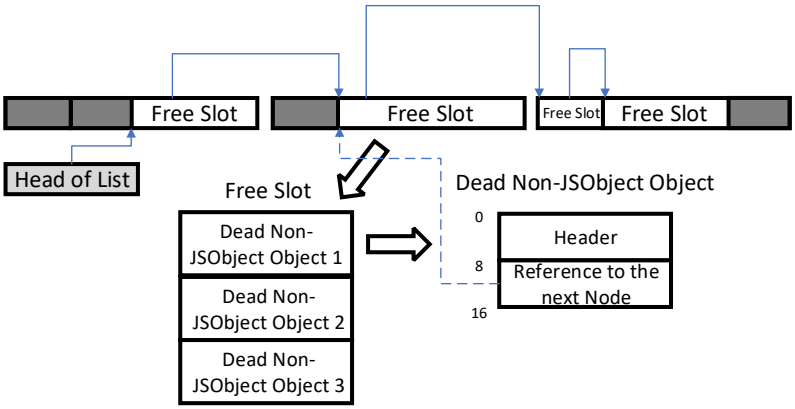

Fig. 14 Free List for Non-JSObject

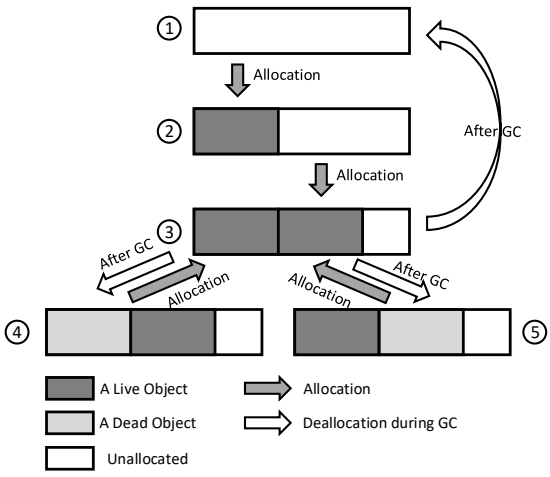

Fig. 15 States of a block

allocator is used, the block is allocated to a small object, as state 2 shows. Now, the remaining unallocated part is managed by partially used block allocator. Then, after one more allocation, the block is allocated with two objects and no more room for another object with the same type. After GC, depending on the whether those two objects in the block are live or not, the state changes. If there is no object alive, then the block goes to state 1, the block is filled with zero and managed by the block allocator again. Otherwise, if only one object is alive, then the block goes to state 4 or 5, the slot occupied by a dead object and not zero filled is managed by partially used block allocator, and the next time an allocation of the same type of object occurs, the same slot is allocated, and the block goes back to state 3.

However, if the size of allocated space is larger than the size of a small object (half of a cache line size), then we round up the requested size to multiple times one cache line size and use only free block allocator to allocate those space.

In short, if the allocation is for small objects, then we use the partially used block allocator for priority. When free slots in partially used block run out,

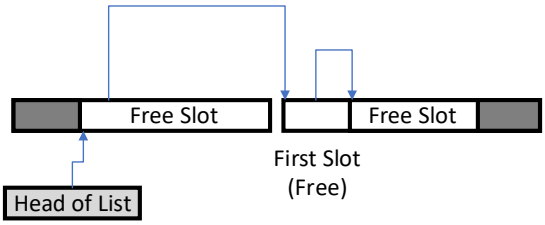

Fig. 16 First Slot (Free)

free block allocator is used and newly generated partially used block is allocated during the next allocation for small objects with the same type. Otherwise, if the allocation is not for small objects, the allocation is done by cache line allocator.

4.2.3 Modification to mark-sweep GC

Since we allocate a partially used block by our allocator, we need to recycle those blocks and rebuild free lists at appropriate time. And such work is finished during mark-sweep GC. Mark sweep GC is a non-moving GC, it marks all the live objects whose pointers are held by other objects or data structures during mark phase, then it sweeps free space during sweep phase, which means it scans all the objects in the heap one by one, combines all the adjacent small slots occupied by dead objects into a larger one and links all those larger slots into free lists, to manage all the free space available for next allocation.

In our implementation, we applied block marking, which means that if at least one object alive in a cache line, the whole block is considered as alive and such block may be a partially used block, otherwise, if no object alive in a block, then such block is considered as a free block. Partially used block is managed by partially used block allocator after GC. To do that, we need to rebuild the free lists for those partially used blocks. Details are introduced below. As for free block, it is managed by free block allocator after GC and the way to recycle such block is the same as that to recycle the space occupied by dead objects in original mark sweep GC.

To rebuild the free lists for partially used blocks, instead of directly scanning the heap based on object units by scanning objects in it one by one in original mark sweep GC, we scan the heap based on block units with cache line size. Then, in each block, we scan objects one by one and the free slots in partially used blocks are recorded to slot free lists. After the scan of a block, if no object alive in the block, then the whole block is recycled in the same way as how it is recycled by original mark sweep GC and zero-filling is applied to such block after GC. Otherwise, for all the free slots occupied by dead objects in a block, we combine the adjacent small slots into a larger one, as mentioned in previous section, except for the first slots of a cache line, as Fig. 16 shows,

the reason for the first slot cannot be combined with the one next to it is that combining free slots modifies the size field in its header. Since for NonJSObjects, the size field is a part for its signature, such modification may change the signature of a cache line unconsciously and provide false signature to memory compression hardware. In short, we should not change the header fields for signature in the first slot of a block.

Then the combined free slots are added to the head of the free list by substituting the original head to the current free slot. Unlike a free block with no object alive and zero filled, no zero filling is applied for free slots in a paritally used block.

5 Evaluation

We evaluated the compression ratio for our eJSVM with our implementation and the eJSVM modified to support AOBD for cache line alignment (called AOBD eJSVM) to check the improvement of our proposal. In addiction, we measured the execution time of three eJSVMs, the original eJSVM, AOBD eJSVM and the eJSVM for our proposal. We use Are We Fast Yet [7] benchmark suite to run each eJSVM for evaluation.

5.1 Compression Ratio Results

5.1.1 Methodology

To evaluate compression ratio, we run two eJSVMs with a heap size set to 3 times minimal heap size needed to run each benchmark. We use Pin [8] tools to monitor the running of the eJSVM to get memory trace, which records the memory read and write in the heap and the function call for Garbage collector. Then the obtained memory trace is analyzed by a simulator for the compression and deduplication hardware to get the result, which is com-

pression ratio.

5.1.2 Definition of compression ratio

The definition of compression ratio is the same as that for the evaluation of AOBD, shown by following equation.

$$\text{Compression Ratio} = \frac{1}{N}\sum_{i=1}^{N}\frac{Compress(R_i)}{R_i} \quad (1)$$

In the equation above, R means the size of live objects after each GC cycle, compress(R) means the size needed to store all the cache line containing those live objects after memory compression, and N means the number of GC cycles (how many times GC starts during the running of a eJSVM with a benchmark). After each GC cycle, we calculate a compression ratio and the average of all those compression ratios is the final compression ratio that we measured.

5.1.3 Results

Fig. 17 shows the comparison ratio of our eJSVM to AOBD eJSVM for compression ratio. It shows that our proposal improves compression ratio by 9.0% on average compared with AOBD method.

We perform evaluations for two eJSVM with the same AOBD memory compression methods, AOBD-HWSIG-ALIGNMENT and AOBD-COMP-HWSIG-ALIGNMENT in Fig. 17. The AOBD-HWSIG-ALIGNMENT is the AOBD method introduced in previous section, and AOBD-COMP-HWSIG-ALIGNMENT is the same as AOBD-HWSIG-ALIGNMENT except that it compresses the base array. Since AOBD allows only one object allocated to a cache line and applies zero-filling for unused part in the cache line, it is not necessary to store all those filled zero in base array as it is, those zero bits can be compressed by counting the number of zeros and encoding the counting in the base array to reduce the bits occupied by those zeros. This is how further compression to base array achieved. For both compression methods, our proposal outperforms AOBD.

Fig. 18 shows the proportion of the size of each data structure in memory compression and deduplication hardware. It shows that our proposal decreases the size of translation table, as expected, and diff array, which means that the compression of one cache line filled with multiple small objects of the same type outperforms the compression of multiple cache lines, each of which contains only one small object of the same type. And that is the reason why the size of diff array decreased.

cache line

Combined with the result in Fig. 17, Fig. 19 shows that the more small objects alive after each GC cycle, the better the compression ratio for our proposal, which is consistent with our anticipation because our proposal only aims to improve the compression for small objects. For Storage benchmark, the proportion of the size of small objects alive after each GC cycle is very small, around 10%. Therefore, there is no significant improvement for such benchmark. While for NBody benchmark, the proportion for small objects is large, around 40%, so more improvement (up to 13.6%) is achieved. As a result, for our proposal, the improvement for compression ratio is more significant for benchmarks with larger proportion of small objects alive after each GC cycle.

5.2 Execution Time

We also measured the execution time to see the overhead introduced by our proposal, as Fig. 20 shows. We set heap size to 100 MB to diminish the effect of GC to run time, and run those benchmarks by three eJSVM: the native eJSVM (represented by 'native' in the figure), AOBD eJSVM ('AOBD') and modified eJSVM for our proposal ('Proposal'). The result shows that when heap size is not large enough to run a benchmark without running many GC cycles (for Mandelbrot-small and NBody-small benchmarks), our implementation results in better performance because of the reduction of GC cycles, as Fig. 21 shows, which results in less time consumed by GC, as the orange bar in Fig. 20 shows. Even if the heap size is large enough to run a benchmark without GC (for other benchmarks), the overhead introduced by our memory manager is relatively low as shown by the blue bar in Fig. 20. Therefore, no significant overhead is introduced by our proposal compared with memory manager in original eJSVM.

6 Related Work

Many memory compression techniques have been proposed so far. However, most of them concentrate on hardware design and rarely consider how

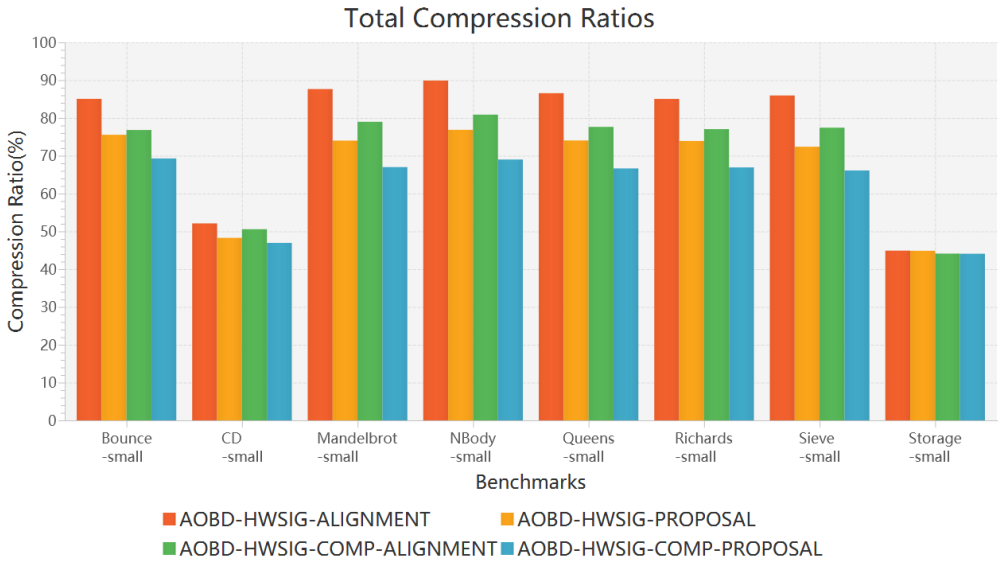

Fig. 17 Compression Ratios

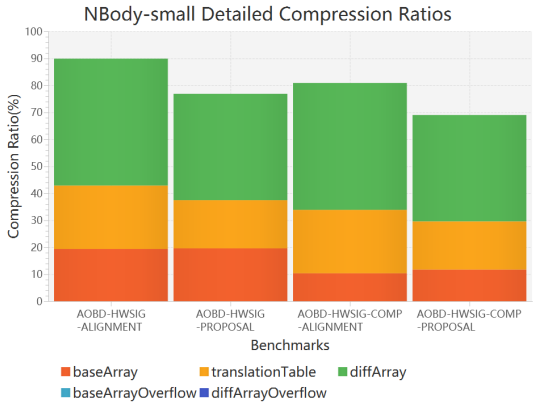

Fig. 18 Detailed Compression Ratios of NBody Benchmark

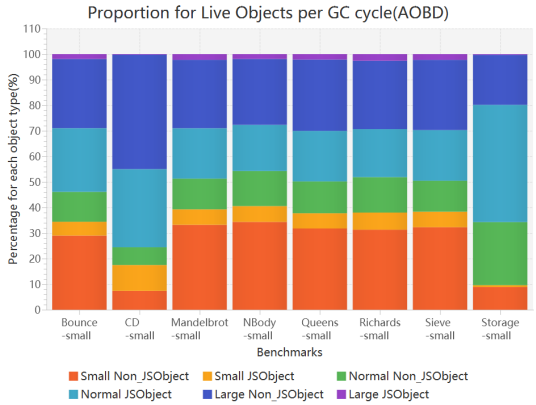

Fig. 19 Proportion for Live Objects per GC Cycle

memory allocator in virtual machine can affect the data layout in heap memory, which finally affect compression ratio and is considered in our proposal.

For general applications, there are many hardware designs that compress the data in the last level cache or main memory, like BDI [9], PSS [10], EPC [11], etc. For special use cases, some memory compression techniques are proposed. SMASH [12] handles the problem of accelerating sparse matrix computation by providing a hardware-software cooperative mechanism. Tavana et al. investigated the potential of applying data compression for multi-GPU architectures [13]. In addition, some methodologies for the evaluation of memory compression have been studied. For example, Sartor et al. analyzed Heap data compressibility [14] and provided some formulas to evaluate memory compression.

However, all of these methods do not consider much about memory compression for object-oriented managed programming language. Zippads [15] fills this blank by a design of a compressed

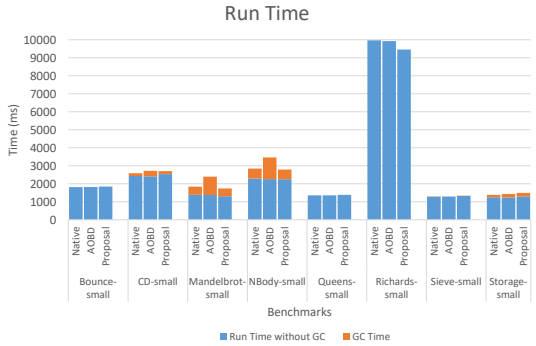

Fig. 20 Run Time for benchmarks

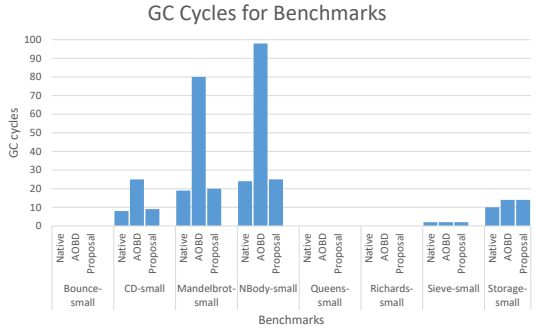

Fig. 21 GC Cycles for benchmarks

object-based memory hierarchy, which is specialized for object compression. Our proposal handles memory compression for object data from another perspective, the cooperation between hardware system and virtual machine.

7 Conclusion

In this paper, we propose a way to allocate multiple small objects of the same type into a single cache line to help a memory compression hardware to achieve better compression ratio for objects. The evaluation shows that our proposal reduces the compression ratio by 9.0% on average compared with AOBD method. Our proposal reduces the size of translation table in AOBD. In addition, we find that the compression for one cache line containing multiple objects with the same type is better than that for the multiple cache lines holding one object of the same type in each of them, this is shown by the reduction of the size of difference array in AOBD.

Acknowledgements This work was supported by JSPS KAKENHI Grant Number JP18KK0315 and JP20H00578.

References

[1] Tomoharu Ugawa, Hideya Iwasaki, and Takafumi Kataoka. ejstk: Building javascript virtual machines with customized datatypes for embedded systems. *Journal of Computer Languages*, 51:261–279, 2019.

[2] QuickJS. https://bellard.org/quickjs.

[3] Yukitada Ri, Zihan Li, Tomoharu Ugawa, and Ryota Shioya. オブジェクトの型を利用したキャッシュラインの重複除去によるハードウェアメモリ圧縮の高効率化 (efficient hardware memory compression by object-type based cache line deduplication). (ARC) 2022-ARC-248, March 2022.

[4] Sungbo Park, Ingab Kang, Yaebin Moon, Jung Ho Ahn, and G. Edward Suh. Bcd deduplication: effective memory compression using partial cache-line deduplication. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, pages 52–64. ACM, 2021.

[5] Tomoharu Ugawa, Stefan Marr, and Richard Jones. Caching hidden classes for pre-transitioning object memory layout in javascript, 2021. Workshop on Modern Language Runtimes, Ecosystems, and VMs (MoreVMs '21).

[6] Hiro Onozawa and Hideya Iwasaki. Customizing javascript virtual machines for specific applications and execution environments (in japanese). *Computer Software*, 38(3):23–40, 2021.

[7] Stefan Marr, Benoit Daloze, and Hanspeter Mossenbock. Cross-Language Compiler Benchmarking: are we fast yet? 2016.

[8] Intel. https://www.intel.com/content/www/us/en/developer/articles/ tool/pin-a-dynamic-binary-instrumentation-tool.html.

[9] Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. Base-delta-immediate Compression: Practical Data Compression for On-chip Caches. 2012.

[10] Daniel Rodrigues Carvalho and André Seznec. A case for partial co-allocation constraints in compressed caches. In Alex Orailoglu, Matthias Jung, and Marc Reichenbach, editors, *Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 65–77, Cham, 2022. Springer International Publishing.

[11] Jinkwon Kim, Mincheol Kang, Jeongkyu Hong, and Soontae Kim. Exploiting inter-block entropy to enhance the compressibility of blocks with diverse data. In *2022 IEEE International Symposium on High-Performance Computer Architecture*

(HPCA), pages 1100–1114, 2022.

[12] Konstantinos Kanellopoulos, Nandita Vijaykumar, Christina Giannoula, Roknoddin Azizi, Skanda Koppula, Nika Mansouri Ghiasi, Taha Shahroodi, Juan Gomez Luna, and Onur Mutlu. Smash: Co-designing software compression and hardware-accelerated indexing for efficient sparse matrix operations. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 600–614, New York, NY, USA, 2019. Association for Computing Machinery.

[13] Mohammad Khavari Tavana, Yifan Sun, Nicolas Bohm Agostini, and David Kaeli. Exploiting adaptive data compression to improve performance and energy-efficiency of compute workloads in multi-gpu systems. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 664–674, 2019.

[14] Jennifer Sartor, Martin Hirzel, and Kathryn McKinley. No bit left behind: the limits of heap data compression. pages 111–120, 01 2008.

[15] Po-An Tsai and Daniel Sanchez. Compress objects, not cache lines: An object-based compressed memory hierarchy. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 229–242, New York, NY, USA, 2019. Association for Computing Machinery.