Dynamic Page Sharing Optimization for the R Language

Helena Kotthaus[1] Ingo Korb[1] Michael Engel[2] Peter Marwedel[1]

[1]Department of Computer Science 12, TU Dortmund University, Germany
[2]School of Computing, Creative Technologies and Engineering, Leeds Metropolitan University, UK
Helena.Kotthaus@tu-dortmund.de, Ingo.Korb@tu-dortmund.de, M.Engel@leedsmet.ac.uk, Peter.Marwedel@tu-dortmund.de

Abstract

Dynamic languages such as R are increasingly used to process large data sets. Here, the R interpreter induces a large memory overhead due to wasteful memory allocation policies. If an application's working set exceeds the available physical memory, the OS starts to swap, resulting in slowdowns of a several orders of magnitude. Thus, memory optimizations for R will be beneficial to many applications.

Existing R optimizations are mostly based on dynamic compilation or native libraries. Both methods are futile when the OS starts to page out memory. So far, only a few, data-type or application specific memory optimizations for R exist. To remedy this situation, we present a low-overhead page sharing approach for R that significantly reduces the interpreter's memory overhead. Concentrating on the most rewarding optimizations avoids the high runtime overhead of existing generic approaches for memory deduplication or compression. In addition, by applying knowledge of interpreter data structures and memory allocation patterns, our approach is not constrained to specific R applications and is transparent to the R interpreter.

Our page sharing optimization enables us to reduce the memory consumption by up to 53.5% with an average of 18.0% for a set of real-world R benchmarks with a runtime overhead of only 5.3% on average. In cases where page I/O can be avoided, significant speedups are achieved.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors– interpreters; optimization; D.4.2 [*Operating Systems*]: Storage Management– main memory

Keywords R language; memory optimization; page sharing; paging; virtual memory

1. Introduction

When the amount of memory required for computations exceeds the physical memory available to the application, the execution is painfully slowed down by frequent page swaps that require disk I/O, a phenomenon also known as 'thrashing'. The performance penalty due to thrashing might render complex computations entirely infeasible. As a countermeasure, we propose an improved, more efficient memory allocation strategy for languages that are commonly used to process large vectors (e.g. APL, Matlab or R), reducing the overall memory consumption. We will use the R programming language [11] to demonstrate the effect of our improvement. R is a free, domain specific, dynamically typed programming language with functional features. It is a de–facto standard in statistics, e.g. for analyzing datasets using machine learning. A sample application could be to classify patient data according to genetic information, in order to determine an optimal medication strategy.

The basic features of R are extended by nearly 6000 packages in public repositories like CRAN[1]. There also exist packages for processing large datasets, particularly addressing more efficient memory management strategies (e.g. sparse matrices). However, this requires programmers to specifically adapt the code to invoke functionality provided by the respective package. Packages are often tailored to concrete applications, and thus cannot simply be used by arbitrary R programs. In contrast, our optimization takes place at the memory management layer between the R interpreter and the operating system, making it entirely transparent to the R programmer and applicable to any R program.

R is executed by an interpreter, which encapsulates all data structures into vectors. Depending on the size of these vectors, the interpreter chooses between multiple allocation strategies to reduce fragmentation. When the size is above a certain threshold, the interpreter allocates a large vector. For each of those, a dedicated block of memory is allocated, potentially spanning multiple pages.

These pages, even when unused, take up memory. Our optimization ensures that memory will only be allocated for pages that are definitely required by the program. Moreover, pages already allocated are shared, if possible, and may be reused. These optimizations incur a time-memory-tradeoff though: The more aggressive pages are shared, the more time must be spent to unshare them. Therefore blindly optimizing would incur a larger runtime penalty than a strategy that takes into account the expected use of the memory, avoiding sharing when it is expected that no memory could be saved. While there already exist similar OS level optimizations such as lazy page loading (meaning a page is only allocated when it is written to or read from for the first time) or sharing of pages with the same content (deduplication), these optimizations lack knowledge about the specific memory behavior of the runtime environment. This reduces the ability of the operating system to make qualified decisions about optimizing the memory usage. For instance, the operating system cannot know if a memory block requested by a function will be written to immediately or only at a later time. Our approach uses additional information available from the runtime environment, e.g. about the short-term usage pattern of a memory block to guide the efficient use of these optimizations.

Using the dynamic language R as an optimization target, this paper presents an optimization for reducing memory consumption

DLS '14, October 20–24, 2014, Portland, OR, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3211-8/14/10. . . $15.00.
http://dx.doi.org/10.1145/2661088.2661094

[1] Comprehensive R Package Network, `http://cran.r-project.org`

of R programs, which is based on dynamic sharing of memory contents on the page level. The key contributions of this paper are:

- its innovative memory allocation optimization, based on avoiding page duplication for large data structures
- page sharing related analyses which are used to refine the above page sharing optimization
- an experimental evaluation of the effect of our optimization on fifteen benchmarks, showing an up to 53.5% reduction in memory usage with an average runtime overhead of just 5.3%

Section 2 gives a brief survey of related work. Memory management of R programs and the R runtime environment is presented in Section 3. Section 4 presents this paper's main contribution: our novel page sharing strategies for R, which are based on avoiding page duplications by using code annotations and page content analysis. Section 5 presents the heuristics used for our optimization and implementation details. Benchmark results are presented in Section 6. Finally, Section 7 concludes with a summary of the paper, also giving an outlook on future work.

2. Related Work

As shown by Morandat et al. [8] and our previous work [9], the interpreter-based original R implementation has several drawbacks leading to slow and memory inefficient program execution.

A number of projects already work on diverse optimizations for R [1–5]. Some of these projects like FastR [3], Renjin [4] or Riposte [5] reimplement the original interpreter in another language such as Java or C++. These approaches benefit from optimizations available in their runtime environments. However, the reimplementations cannot yet guarantee full compatibility with existing R programs and packages due to the complex and evolutionary development of the R language and its missing formal specification.

Optimizations from other projects like pqR [2] and the Orbit VM [1] concentrate on specific bottlenecks on the R interpreter by using, e.g., function specialization or the introduction of scalar data types. While these provide full compatibility, they concentrate on language rather than systems effects as basis for their optimizations. For other interpreter-based dynamic languages there are similar approaches realized in projects like MaJIC [6] and PyPy [7].

In contrast to the approaches discussed above, our page sharing optimization works on a layer between the R interpreter and the operating system. This enables optimizations of arbitrary applications with only small modifications to the R interpreter including its built-in functions and without requiring application changes. Thus, in the following we provide a discussion of related system level approaches to reduce memory overhead.

In general, related work on reducing memory utilization can be categorized into two groups: memory compression approaches, often influenced by embedded systems resource constraints, and memory deduplication, which is mostly used in virtualization.

Memory compression tries to reduce the swapping activity of a system by compressing memory contents instead of swapping pages to secondary storage. For example, an adapted version of the approach by Wilson et al. [16] is used in MacOS X version 10.9. Wilson notes that the efficiency of memory compression largely depends on technology trends due to the large computational overhead. It will be increasingly attractive as CPU speeds increase faster than disk speeds.

More recent publications, such as Beltran et al. [17], Yang et al. [18] or Pekhimenko et al. [19], concentrate on using multicore systems and improved compression algorithms and on embedded devices. An analytical model for evaluating the efficiency of compression is presented by Chihaia and Gross [20].

All compression approaches share the drawback that a significant share of processor time is spent for compressing and decompressing memory contents. Typically, heuristics are employed to determine the memory contents for which compression is most rewarding. In contrast, our page sharing approach concentrates on zero-filled pages and proactively avoiding page duplication.

While some compression approaches like Nakar and Weiss [21] or Benini et al. [22] realize optimizations by incorporating profiled information on the application behavior or are specialized to rewarding compression targets such as sparse matrices by Lawlor [23]. These approaches still face the CPU overhead of compression and decompression.

In contrast, memory deduplication reduces the memory overhead by mapping virtual pages with identical contents to a single physical page. This is often beneficial in virtualized environments where large amounts of read-only memory, such as the code sections of shared libraries, are used in multiple virtual machines [25].

An often used implementation of deduplication is available in Linux as the Kernel Samepage Merging (KSM) [15]. Deduplication introduces significant computational overhead, since the contents of pages have to be scanned periodically in order to identify pages with identical content. KSM has been optimized by Chen et al. [13] using a classification of page access behavior to reduce page scanning for deduplication, while Miller et al. use cross-layer hints to improve the efficiency of deduplication scanners [24].

Valat et al. used application knowledge to reduce deduplication overhead [12], which uses a special version of mmap to avoid unnecessary page removal and enable page reuse in HPC environments. Jula and Rauchwerger use application-provided hints for more locality of data allocation in C++ applications [14]. A memory trace-based evaluation of different deduplication and compression approaches is presented by Deng et al. [26].

Compared to deduplication approaches, our page sharing optimization employs specific knowledge about the interpreter state to reduce the number of pages that need to be scanned for identical content. Scanning itself has low overhead, since only scans for zeroes have to be performed, which terminate at the first found nonzero element. Furthermore, deduplication approaches such as KSM are purely reactive, i.e., they can only reduce duplicates that already exist when it scans the memory. Our system is not just simply reactive, but it also proactively avoids the main sources of identical-content pages from object allocation and duplication. In addition, our optimization is aware of data structures and their contents used by the R interpreter. Thus, a better prediction and avoidance of unnecessary deduplication attempts is enabled.

In the next section, we will discuss the behavior of the R programming language in regard to memory allocation.

3. Memory Allocation in R

The life cycle of an object in the R interpreter starts with its allocation. The R interpreter assumes that vectors are laid out in consecutive pages in memory, unlike other languages that can construct a large array in memory using many smaller data structures. Depending on the size of the object, the interpreter uses a system of multiple memory pools for objects with a data size of up to 128 bytes. For objects larger than this threshold, memory allocated via the *malloc* C library function is used directly instead of pooling the allocations. This reduces the memory fragmentation when many small objects are created and some of them are released.

The R interpreter ensures that a newly allocated object is always initialized – either by explicitly initializing it or implicitly by writing the results of a computation to it.

After the object is allocated and initialized, it can now be used as input for various built-in interpreter functions like the plus operator. These functions may modify the object, at which point a copy-on-write optimization in the R interpreter is triggered: Instead of copying an object immediately, the R interpreter only marks the original object as shared with a flag in its header to delay the copy. The interpreter now has two references to the same object, either of which may be modified later. When this modification happens, the copy process is triggered and a full copy of the affected object (potentially spanning multiple pages) is created using the interpreter-internal *duplicate* function. R uses these mechanisms to implement a call-by-value semantic.

The R language does not require the programmer to explicitly manage memory, so it needs garbage collection to automatically free memory. The garbage collector in R is a mark-and-sweep, non-moving, generational collector. It can be manually triggered or runs when the interpreter's heap is in danger of running out of space.

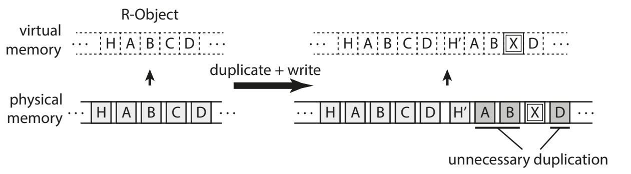

Figure 1. Example of Copy-on-Write in the standard R interpreter

When an R program allocates large objects, R's copy-on-write scheme implies that it has to use a large amount of memory for each copy of such an object. This is illustrated in Figure 1. On the left hand side a large R object consisting of an header *H* and four pages *A* to *D* is shown both in *virtual memory* on the top (marked with dotted lines) and its corresponding allocated *physical memory* on the bottom (solid lines). On the right hand side, the situation after a duplication that was triggered by a write access is shown. Now there are two R objects shown in the virtual memory on top and their corresponding physical memory on the bottom. In one of the copies, page *C* was modified and is now marked as *X*, and the copy has its own header *H'*. Although unmodified, the R interpreter needs to use additional memory to create duplicates of pages *A*, *B* and *D* (marked grey) because it assumes that objects are organized as continuous blocks of memory and thus it has to duplicate at object level granularity.

Our optimization attempts to reduce this memory overhead by copying only parts of the object instead, sharing the same memory pages between multiple objects as long as they are not modified. This scheme is transparent to the interpreter's memory management including the garbage collection, requiring only small changes in memory allocation and freeing, as well as in the duplicate function.

In the next Section we present the details of our page sharing strategies for the R interpreter.

4. Page Sharing Strategies

The general mechanism we use to avoid page duplications in R is described in Section 4.1 based on optimizing the allocation and duplication mechanisms. This optimization is further refined by using static annotations that reduce the optimization overhead, presented in Section 4.2 and by a dynamic refinement using a page content analysis to increase the amount of shared memory, introduced in Section 4.3.

4.1 Avoiding Page Duplications

As explained in the previous section, the R interpreter can only allocate complete objects. The first part of our optimizations for the R interpreter are implemented in the object allocation. To enable the sharing of memory at page granularity within the R interpreter, we employ a custom memory allocator when a large vector has to be allocated, as shown in Figure 2. When the internal function *allocVector* is called to allocate an object, our optimized interpreter selects between our custom memory allocator to share memory on page granularity or the standard *malloc* function if this is not required. In both cases the allocated memory is accessible within the address space of the R interpreter.

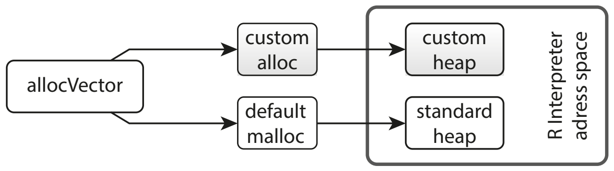

Figure 2. Memory allocation scheme for dynamic page sharing

Although our custom allocator uses a memory management scheme similar to standard virtual memory schemes commonly used in operating system kernels, it is completely implemented in user space for ease of implementation, although a kernel level implementation would be feasible. The downside of a user mode only implementation is that it needs to replicate certain data structures that are already present in the operating system like the mapping from virtual to physical memory, because the existing structures in the kernel are not sufficiently exposed to user space. Since we do not have direct access to physical memory in user space, we use a single file located on a RAM disk to allocate physical memory (custom heap in Figure 2). A simple free-bitmap based allocator is used to reserve pages from this file, and the file is dynamically enlarged when needed. Mapping these physical pages into the virtual address space of the interpreter can be accomplished using the *mmap* Unix system call and changing their access permissions is possible using the *mprotect* system call, which modifies the settings of the memory management unit of the processor. To create a memory management system from these operations in user space, we also need a page table to map from a virtual address to a physical page. There are multiple ways of implementing such a page table, for simplicity reasons we chose to use a hierarchical page table using the same four-level structure that is used by the CPU itself. Our system needs to map the same physical page to multiple locations in virtual memory to allow sharing of pages, therefore it needs a reference counter for each physical page. A reference count greater than one for a page indicates that it is shared between multiple objects or multiple times within one object.

The left side of Figure 3 illustrates an example of our optimized R object allocation that utilizes a global shared zeroed page. Here, our custom memory allocator was asked to allocate an object with a total size of five pages. While it has the requested size in virtual memory (dotted), physically it only consists of a single non-shared page in the beginning (marked *H* for header) followed by four pages that point to a shared page (marked with ***0***) called the zeroed page. The non-shared initial page is required as it will contain not just data but also the object header used by the R interpreter. The interpreter writes this object header to the front of the memory allocation area and since it will be frequently updated, for example during garbage collection, it cannot be shared between multiple objects. The shared zeroed page is known to be filled with zero-bytes. This allows us to avoid the zero-initialization of allocated memory in the interpreter and also ensures that memory is only allocated for those pages which are actually written to at a later time. This concept of prepared zeroed pages is also found in operating system kernels, but since the standard R interpreter immediately writes into all memory that it allocates for initialization, it does not benefit from this optimization.

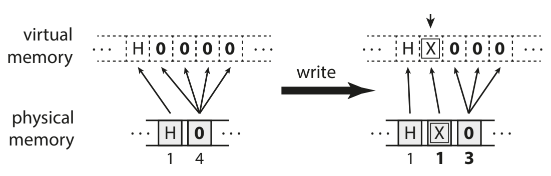

Figure 3. Page-shared memory allocation and copy-on-write

The numbers in small print below the physical pages in Figure 3 are the reference counters. Since the header page *H* is mapped only once, its reference count is one and the reference count of the zeroed page is four as it is shown to be mapped four times into virtual memory. The interpreter now has the illusion that it has received five pages of memory, even though only two pages are allocated physically. To sustain this illusion, we must ensure that any write access to a virtual page which points to a shared physical page can be detected and handled. If such a write access were not handled, from the interpreter's point of view a single write access would affect the intended virtual page and additionally all virtual addresses where the same physical page is shared. In Figure 3 this could mean that a write to one of the four instances of the zeroed page would be mirrored in its other three instances, resulting in incorrect object contents.

Therefore, we mark all pages with a reference count greater than one as read-only, ensuring that a write access triggers a segmentation fault. We catch this fault in a signal handler which performs unsharing of the affected page. This handler uses the virtual address of the write access to determine the affected physical page. It then allocates a new page, copies the contents of the original page to it and replaces the page that caused the segmentation fault with the new one. The resulting situation is shown on the right side of Figure 3: One of the instances of the zeroed page which was written to was replaced with a new page marked with *X*. This updates the reference count of the both zeroed page and the newly allocated page. Since the newly allocated page is only mapped once, it can now be marked read-write and further accesses do not require special handling anymore.

As noted in the discussion of R's memory management in Section 3, the R interpreter can only duplicate on the object level, even when the object consists of multiple pages and part of the copy may end up with identical content as the original (see Figure 1). To avoid this, we augmented the copy-on-write mechanism in the interpreter to take advantage of the page sharing capabilities of our system, improving the duplication granularity from object level to page level. While the allocation optimization avoids the immediate allocation of pages by using the global zeroed page, the duplication optimization allows us to reuse the already-allocated pages of the original object instead of allocating new pages.

An example of such a duplication optimization is shown in Figure 4. On the left side the situation before duplication is shown: An R object occupying five virtual pages, two of which reference the global zeroed page is to be duplicated. Unlike the original R interpreter which needs to allocate five new pages for the copy, our optimization reduces this to a single allocated physical page which is shown on the right side of the Figure with the original object on the top and the copy on the bottom. Although we could create a virtual-only copy of the first page containing the object header, the R interpreter updates this header in the copy immediately after duplication, which would trigger an unsharing of this page. To avoid the overhead of this event, our duplication immediately creates a physical copy of the page containing the header. Since most of the pages of the original object are now mapped twice in virtual memory, the reference counters of the corresponding pages must be updated and both the original and copy are marked as read-only to allow unsharing on write accesses.

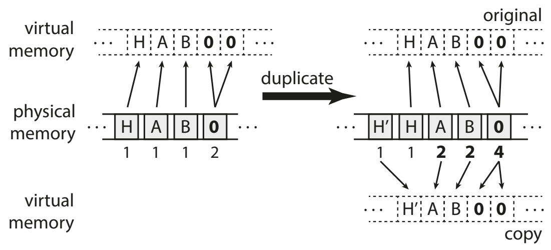

Figure 4. Duplication with page sharing

In total, the finer copy granularity of our implementation enables us to store both the original and copied objects from the example in just five pages of memory. In contrast, the original R interpreter would need ten pages of memory to store the same objects.

Although the mechanism of sharing pages during allocation and duplication described above always result in a valid view on memory for the interpreter, there are cases where they introduce additional overhead that can be avoided by further refinements described in the next Subsection.

4.2 Static Refinement using Annotations

Our static refinements consist of two kinds of annotation to reduce the run-time overhead caused by avoiding page duplication. The first is based on expected use of the object immediately after allocation, the second is based on the size of the allocated object.

The memory allocation strategy described in the previous Subsection (see Figure 3) saves memory by using a global zeroed page, assuming that not all pages of the allocated object will be written to immediately. This assumption is not always valid though, e.g. vector arithmetic functions in the R interpreter allocate a new object and immediately write to all pages of it to store their results. This will cause a segmentation fault for the first write of every page, triggering the allocation of memory for all pages of the object. These faults incur a run-time overhead for handling them which is not present when allocating an object with non-shared pages.

To avoid this run-time overhead, we have added annotations to the C source code of the built-in functions included in the R interpreter where we have determined that the newly allocated memory will be completely overwritten directly after allocation. For this situation, our custom allocation function returns an object where every virtual page references a new physical page, so no segmentation faults will be triggered by the write accesses. Although these R objects will not save any memory on allocation, they still have the opportunity for later optimizations, e.g. when they are duplicated. Thus those objects should still be managed by our page sharing.

Currently the annotations for these "full-overwrite" functions need to be placed manually in the R interpreter's C source code by locating calls to *allocVector*, followed by loop structures that write to every element of the newly-allocated object. Those manually placed annotations could also be automated by static code analysis checking for allocation calls followed by loops that write to the newly-allocated object similar to the pseudo-code in Figure 5.

The second kind of annotation for reducing overhead relates to the size of the object that the interpreter wants to allocate. The R interpreter can allocate objects with a variety of sizes, not all of which span multiple pages. Our custom allocator is only used for objects with a size that indicates potential for page sharing. This potential is limited for smaller objects. The first page of an object stores not just data but also the frequently modified object header which is why we avoid sharing this page. Therefore, our system passes R objects smaller than two pages of memory to the standard, non sharing memory allocator. This limit could also be used as a tunable parameter to select a trade-off between memory savings and the run-time overhead incurred by our optimizations.

```
object = allocVector(length)
for (i = 0; i < length; i++):
  object[i] = calculation(i)
```

Figure 5. Pseudo-code of a "full-overwrite" function

In addition to these static refinements we have also introduced a dynamic refinement for increasing the number of shared pages which will be described in the next Subsection.

4.3 Dynamic Refinement using Page Contents

During the execution of an R program, pages of objects are updated with the results of calculations. This can result in multiple distinct pages with the same content, which opens up the opportunity for sharing those pages. The general idea of locating identical objects in a system and saving memory by reducing them to a single object is known as deduplication (see Section 2).

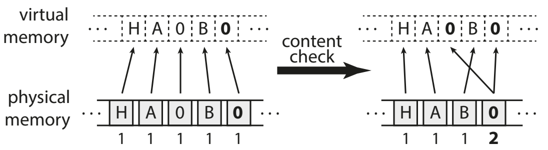

Figure 6. Page content check for reduction of memory usage

We currently implement a restricted version of the content scan which only checks for pages identical to the already existing global zeroed page. The deduplication of zeroed pages is illustrated in Figure 6. On the left side, the situation before the page content scan is shown where an object contains two identical zero pages. One of those pages is already mapped to the global zeroed page (shown in bold), while the other uses a separate physical page. On the right side, the situation after the content check is shown. Here, the content check has detected the separate copy and mapped its virtual page to the global zeroed page, freeing the memory that was used for the unnecessary duplicate.

Although we could implement a general scan that could detect duplication between pages of arbitrary content, such a scan would incur a significantly runtime overhead (e.g. due to the calculation of hash values) compared to scanning just for zeroed pages. A scan for arbitrary content would need to check the full content of all pages in the system, while a scan for zeroed pages can use an early abort condition as soon as a non-zero element is found.

The frequency of content checks and the number of pages that need to be scanned influences the overhead incurred by this optimization. In order to minimize the number of pages that need to be scanned, we chose to run the scan only after the completion of a garbage collection in the interpreter. At this point we know that all objects still in memory are alive. This enables us to avoid scanning pages for duplicate content that would soon be discarded. Triggering content checks from the garbage collector also provides a natural regulation mechanism for the frequency of content checks as the frequency of garbage collection calls depends on the memory requirements of the running R program.

With the described content check optimization we can dynamically share pages that were previously excluded from sharing the global zeroed page, e.g. arithmetic vector operations as described in Subsection 4.2. Thus both the static and the dynamic refinements of our page sharing optimization complement each other. In the next Section we will present the interaction of those refinement strategies and our general page duplication avoidance strategy.

5. Page Sharing Optimization

In this Section we will describe the heuristics used in the implementation of our page sharing optimization. Figure 7 shows the pseudo-code for the main functionality of our page sharing strategies. Lines 1 to 25 show our custom object allocator that implements the two static refinements described in Subsection 4.2. The first static refinement can be seen in line 3 where the size of the allocated object is used to decide if the allocation should be passed to the standard allocator or if it should handle the allocation with page sharing. If the size of the object is sufficiently large, our allocator first needs to allocate a region of virtual memory that is large enough to map the object (*alloc_virtspace*, line 8). In our implementation this is emulated using a *mmap* system call, which lets the kernel select a free region of virtual memory. Although the *mmap* call maps data into this region, we can override this mapping later using the *map_page* function in line 23. The second static refinement is implemented as a new parameter to the allocation function. Callers that completely overwrite the object after allocation set this parameter to indicate that they will not benefit from an initialization using the shared zero page. This parameter is used in line 16 to select if the allocator should return references to the global zeroed page or allocate memory for each virtual page of the object. Line 23 then calls the *map_page* function to map the selected page for the new object.

The pseudo-code of the *map_page* function is shown in figure 8 (line 1–13). It receives two parameters, a physical page and an address of a virtual page where the page should be mapped to. First, it adds a mapping from the virtual page to the physical page to the page table. Then the actual mapping in virtual memory is updated in line 4, which uses the *remap_file_pages* Linux system call to change an existing mapping. Finally, the reference counter of the page is incremented (line 7). If the reference counter is equal to one, the page is mapped just once into virtual memory and is marked as read-write (not shared), otherwise it is marked as read-only since it is shared between multiple objects.

Both static refinements used in the *alloc* function in 7 reduce the overhead of our optimization in cases where no gain would be expected. Compared to other OS level page sharing optimizations we can use additional information about the caller of the *alloc* function and the knowledge where the R interpreter stores the frequently modified object header. Therefore, we do not share the first page of the allocation (*first_page_flag* in line 16) saving the time required for the page fault taken when the interpreter first updates the header.

After an object was allocated by our custom allocator, it may need to be copied. Our duplication optimization strategies augment the existing copy-on-write mechanism in the R interpreter, enabling us to share pages other than the global zeroed page. Lines 28 to 53 show our *duplicate* function, which enables us to improve R's object level copy-on-write granularity to page level granularity, avoiding unnecessary copies. Similar to allocation, *duplicate* does not share the first page of the copy (lines 32 and 39-43) as it contains not just data but also the object's header. In line 36-51 the function iterates over the virtual pages of the original and copy and maps the same physical pages that are used for the original into the copy. Since these pages now must have a reference count of at least two as they are mapped in both the original and copy objects, *map_page* in line 51 automatically marks them as read-only. Additionally in line 49 the page in the original object is also marked as read-only. Both mappings of the page must be marked to ensure that a write access triggers a fault which can then be used to

```
1  alloc(object_size, expect_full_write):
2    # static refinement: object size
3    if object_size < 2_pages:
4      return standard_alloc(object_size)
5
6    else:
7      # reserve a virtual address space
8      object = alloc_virtspace(object_size)
9
10     first_page_flag = true
11
12     for_each (virt_pg in object):
13
14       # the first page is always physical,
15       # static refinement: full overwrite
16       if expect_full_write or first_page_flag:
17         new_pg = get_free_physpage()
18         first_page_flag = false
19
20       else:
21         new_pg = ZEROED_PAGE
22
23       map_page(new_pg, virt_pg)
24
25     return object
26
27
28 duplicate(orig_obj):
29   # reserve a virtual address space for copy
30   copy_obj = alloc_virtspace(sizeof(orig_obj))
31
32   first_page_flag = true
33
34   # map virtual pages of copy to
35   # physical pages of original
36   for_each (virt_orig_pg in orig_obj,
37             virt_copy_pg in copy_obj):
38
39     if first_page_flag:
40       # first page is copied
41       physpg          = get_free_physpage()
42       first_page_flag = false
43       copy_content(physpg, virt_orig_pg)
44
45     else:
46       # remaining pages are just mapped to the
47       # physical pages of the original object
48       physpg = pagetable_lookup[virt_orig_pg]
49       set_readonly(virt_orig_pg)
50
51     map_page(physpg, virt_copy_pg)
52
53   return copy_obj
54
55
56 # dynamic refinement:
57 content_check():
58   for_each (page in all_pages):
59
60     # check if the page is a copy of the
61     # global zeroed page
62     if pg != ZEROED_PAGE and
63        page_content_is_zero(pg):
64
65         # map the global zeroed page instead
66         unmap_page(pg)
67         map_page(ZEROED_PAGE, pg)
```

Figure 7. Core page sharing optimizations

```
1  map_page(phys_page, virt_page):
2    pagetable_add(virt_page, phys_page)
3
4    remap_file_pages(virt_page, phys_page)
5
6    # update reference counter
7    refcount[phys_page] += 1
8
9    # shared pages must not be write-able
10   if refcount[phys_page] > 1:
11     set_readonly(virt_page)
12   else:
13     set_readwrite(virt_page)
14
15
16 write_fault_handler(fault_pg):
17   phys_pg = pagetable_lookup(fault_pg)
18
19   # allocate new phys. copy if page was shared
20   if refcounts[phys_pg] > 1:
21     new_pg = get_free_page()
22     copy_content(new_pg, phys_pg)
23     unmap_page(fault_pg)
24     map_page(new_pg, fault_pg)
25
26   # page is now known to be non-shared and
27   # can be used directly
28   set_readwrite(fault_pg)
```

Figure 8. Page mapping and page fault handler

allocate a new physical copy of the affected page. Thus, our version of *duplicate* enables lazy page allocation for copied objects.

The dynamic refinement (see Section 4.3) we apply to save memory by sharing pages via a page content check is shown in the *content_check* function in lines 56 to 67. This optimization is enabled for each object that was allocated by our custom allocator and that is still alive after a garbage collection call. This reduces the number of pages that we have to analyze and thus the overhead of the optimization. Each page is scanned (line 58) for zero contents that are not already shared by our global zeroed page to free them afterwards. Therefore we map the global zeroed page to the pages that have zero contents and are not already shared (lines 62 and 63). The *unmap_page* function in line 66 removes the duplicated zero pages from the page table and frees the memory that was previously used by them. This function is the counterpart of *map_page*. Additionally, it decrements the reference counter of the previously-mapped page and marks the page as free if its reference counter is zero. The *map_page* function in line 67 then maps the global zeroed page into the virtual page occupied by the object and increases its reference counter by the number of saved pages.

The previously shown functions from Figure 7 all use the *map_page* function which marks shared pages (mapped more than once into virtual memory) as read-only and the function *write_fault_handler* shown in Figure 8 (line 16–28) is called. This handler allocates a new page (line 21), copies the content of the accessed page to it (line 22) and updates the virtual mapping (lines 23-24) at the page where the write occurred (called *fault_page*). This new page is not shared yet, so it can be marked as read-write (line 28). Since we remove a mapping of a virtual page, the reference counter of the corresponding physical page decreases, which is handled by *unmap_page*. If all but one virtual instance of a shared page were written to, the handler has created physical copies for all of these pages. The last instance which has a reference count of one is still marked read-only at this point, because the handler only updates the pages for which a fault occurred. On a write access of this last instance we leave out the allocation and copy steps: As the reference count indicates the page is not shared anymore (line 20), we can just mark it as read-write (line 28). After the handler has mapped a new read-write page at the fault location, the interpreter can resume its execution and the write will now succeed. Generally such a return from a custom segmentation fault handler is not supported by the POSIX standards, but Linux and other operating systems offer extensions to allow this. Our write fault handler is implemented with the existing *libsigsegv* library[2] which offers a common API for these OS-specific extensions.

In the next Section we will evaluate the memory savings and runtimes of the previously described page sharing optimizations.

6. Evaluation

This Section presents results obtained by applying the proposed dynamic page sharing optimization for R to real-world benchmarks. Subsection 6.1 describes the experimental setup and benchmark sets and the methodology used to perform the evaluation. Subsection 6.2 discusses the results in terms of memory consumption and Subsections 6.3 and 6.4 evaluate the runtime effects of our page sharing strategies.

6.1 Experimental Setup

We compared memory usage and runtime of the R interpreter including our optimizations against the standard R interpreter. For the following experiments we used a computer equipped with a 2.67 GHz Intel Core i5 M480 CPU and 6 GB of RAM, using a 64-bit version of Debian Linux 7.0 as the operating system. On this system, memory pages have a size of 4096 bytes. Although our implementation could use a larger page size than the system, we chose to use the same size as we expect this to maximize the amount of memory that can be shared.[3] We used R version 3.1.0 as the base for our modifications, utilizing the default configuration. Both the standard as well as our optimized interpreter were compiled using GCC version 4.7.2 with the default optimization flags (-O2) selected by the build system of R.

The standard memory measurement functions for user space functions in Linux only measure the virtual memory of a process. Since our system maps the same physical page multiple times into virtual memory, these functions cannot show the amount of physical memory our page sharing optimization saves because they count every virtual instance of a shared physical page. Therefore we had to create our own memory measurement functions. To measure the amount of memory reserved by the default allocator, we override the standard allocation functions like *malloc* with versions that track the current total amount of memory allocated and call the original function afterwards. In our custom allocator we can directly count the number of physical pages that needed to be reserved as well as the size of the management data structures. Using these measurements, we can accurately calculate the current amount of physical memory allocated.

For the evaluation of our dynamic page sharing optimization, we used two different benchmark sets. The first set shown in Table 1 is a shorter-running set of benchmarks. These were selected from the R benchmark 2.5 suite[4] that was originally developed to measure the performance of various configurations of the R interpreter plus one additional benchmark. The additional benchmark *glmnet* utilizes an existing sparse matrix optimization implemented as an R package and is included to analyze if our optimization can be beneficial for programs that already try to reduce memory usage by utilizing application-specific knowledge. For this benchmark set we selected iteration counts for the outer loop that result in a runtime of approximately 1 minute with the standard R interpreter.

Benchmark	Description
b25-1	Linear regression over a 3000x3000 matrix
b25-2	FFT of 2,400,000 random values
b25-3	Inverse of a 1600x1600 random matrix
b25-4	Grand common divisors of 400,000 pairs (recursive)
glmnet	regression using glmnet on a sparse 20000x1000 matrix

Table 1. Misc Benchmark Set

The second set of benchmarks is based on a set of long-running real-world machine-learning benchmarks, shown in Table 2. The choice of those machine learning classification algorithms is based on both the method's popularity and the availability of an implementation. These benchmarks were also used in [9] and are publicly available [10]. Most of the listed algorithms allow the user to adjust several parameters to increase predictive performance. We did not tune any parameters but instead either used the mostly meaningful defaults or, if available, used the implementation's internal auto-tuning process. To conveniently apply all learners on identical cross-validation splits we used the package `mlr`[5]. For the input data we used a dataset that fullfills the following criteria: (a) it is a 2-class classification problem, (b) it has a sufficiently large number of observations to achieve accurate results and (c) it has an even and realistic mixture of data types. We ensured that the data set contained no missing values as most of the algorithms would remove them from processing. The machine learning benchmarks were configured to use 20-fold cross validation. The size of the data set (15000 samples with 200 numeric features) was chosen to ensure that the fastest of the machine learning benchmarks has a runtime of approximately one minute on the standard interpreter. We chose to use the same data set for all algorithms to allow a better comparison of their memory requirements.

Benchmark	Description
ada	Boosting of classification trees
gbm	Gradient boosting machine
kknn	k-nearest neighbour classification
lda	Linear discriminant analysis
logreg	Logistic regression. Binary classification decision derived using a probability cutpoint of 0.5
lssvm	Least-squares support vector machine
naiveBayes	Naive Bayes classification
randomForest	Random classification forest
rda	Regularized discriminant analysis
rpart	Recursive partitioning for classification trees

Table 2. Machine Learning Benchmark Set

Each benchmark was run 10 times with both the standard as well as our page sharing interpreter, results are given as the arithmetic mean of these 10 runs. To reduce non-determinism, we have set the random number seed to a fixed value as the first statement in each of the benchmark programs. Each repetition was started in a fresh interpreter process, so initialization costs are included in the measurements (an expected overhead on the order of one second or less). The R interpreter does not use adaptive optimizations. All system services that might interfere with the measurements were

[2] `http://libsigsegv.sourceforge.net/`

[3] Using a smaller page size than the system size is inefficient since we rely on the hardware MMU for efficient page access protection.

[4] `http://r.research.att.com/benchmarks/R-benchmark-25.R`

[5] Version 1.0-2612. `http://CRAN.R-project.org/package=mlr`

disabled. Both runtime and memory usage were measured simultaneously. For the memory usage both the global peak memory usage for a benchmark run as well as a second average memory usage measurement which will be detailed in Subsection 6.2. We calculated a 95% confidence interval for these measurements as well as the ratio of their means using the percentile bootstrap method. Means over these ratios were calculated using geometric mean to reduce the influence of outliers.

In the next Section we will discuss the effects of our page sharing strategies on the memory consumption of the R interpreter with the setup and methodology described above.

6.2 Memory Consumption Analysis

In this Section we will analyze how our page sharing strategies influence the memory consumption of the R interpreter. Therefore we measured the peak memory consumption of the R interpreter during its execution. The peak memory consumption alone does not represent any information about changing memory usage over time though – the peak may occur only for an instant or for a longer period of time depending on the benchmark. Therefore we also measured short-term peak usage over periods of 1 second, resulting in a memory-over-time profile of the R program. These measurements are used to gain a complete view on the memory consumption of the program. We calculate the arithmetic mean of these one-second measurements as a second memory usage indicator besides the global peak to allow easier comparisons of the memory behavior, shown as *Average usage* in our results.

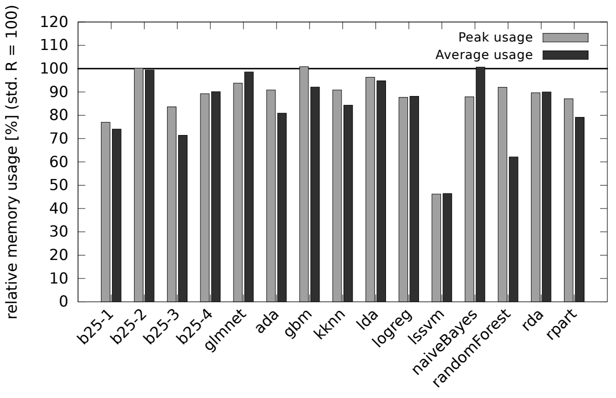

Figure 9. Relative memory usage with page sharing compared to standard R (lower is better). The 100% baseline represents the standard R interpreter (std. R) without optimizations. Geometric means of memory gains are 13.6% for peak and 18.0% for average memory usage. Error bars have been omitted as the confidence intervals were smaller than 0.5% for all values.

Figure 9 shows the peak (*Peak usage*) and average (*Average usage*) memory consumption of the R interpreter running with our page sharing optimization. The 100% baseline represents the standard R interpreter without optimizations. Values below this baseline indicate relative memory savings realized by our page sharing strategies. Error bars have been omitted as the confidence intervals were smaller than 0.5% for all value. The detailed values are presented in Table 3, which also shows the number of pages identified as shareable by the content check and thus indicates the optimization potential of this refinement. Confidence intervals have been omitted as they were smaller than 0.5% for all values.

The relative peak memory usage compared to the standard R interpreter ranges from -0.9% for *gbm* to 53.8% for *lssvm*. This shows that our optimizations do not realize any memory savings for some of the benchmarks and instead slightly increase their peak memory consumption because it needs additional data structures for page management. However, in the case of *gbm*, we can still achieve a reduction of the average memory usage by 7.9% compared to the standard R interpreter. For *naiveBayes* this situation is reversed: Our optimization saves 12.1% of its peak memory usage while it results in a slight increase of average memory usage (-0.6%). Since the amount of pages recovered by the content check (see Table 3) is small, the reduction of the peak memory usage must be caused by the optimizations for allocation and duplication.

In the case of *b25-2* our optimization cannot save memory in the peak case and no meaningful amount in the average case, but again the overhead incurred is very small. The reason why *b25-2* does not gain from our optimizations is that even though it uses large vectors with 2.4 million elements, it allocates a vector which is immediately filled with random numbers similar to the pseudo-code shown in Figure 5. Therefore it cannot gain from our optimizations in the allocation of the vector and the content check cannot find any all-zero pages either. This is also indicated by the low number of pages recovered by the content check shown in Table 3 as column *ZPG*. Furthermore, *b25-2* does not use any object duplication, so our optimizations for duplication cannot reduce its memory consumption either.

Even though our page sharing optimization results in a slight increase of peak or average memory usage for the three benchmarks described above, the twelve other benchmarks all gain from our optimizations in both the peak and average memory measurements. The geometric mean over all fifteen benchmarks shows a reduction of peak memory usage by 13.6% and a reduction of average memory usage by 18.0%. The benchmarks that profit the most from our optimizations are *lssvm* with 53.8% for peak usage and *randomForest* with 37.9% for average usage. Both of these benchmarks have very high values for the number of pages recovered by the content check. Thus for them the reduction of memory usage is not just caused by the allocation and duplication optimization but also the dynamic refinement of our optimization contributes to the savings.

Table 3 only shows summarized values for the memory consumption over the complete runtimes of all benchmarks. To gain additional insight about the memory usage behavior, we analyzed the memory consumption over time. Results for the four most interesting ones (*glmnet*, *gbm*, *randomForest* and *naiveBayes*) are shown in Figure 10. The confidence intervals for these measurements are all very small (less than 1%), so the figure shows only data from a single run. For each benchmark, we have selected the run whose execution time was closest to the average of the 10 runs. The x-axis represents the runtime in seconds, the y-axis the corresponding memory consumption of the benchmark. Both the profile for the standard R interpreter (grey curves) and the interpreter with our page sharing optimizations (black curves) are shown. The straight lines at the top mark the peak memory usage, while the dotted lines mark the average memory usage.

As mentioned in Section 6.1 we analyzed the *glmnet* because it utilizes an already-existing memory optimization in the form of an R package for sparse matrices. Our goal here is to determine if our optimizations can still offer additional memory savings even in the presence of such specialized application knowledge. In the top left of Figure 10 the memory-over-time behavior of this benchmark is shown. While we only have a small improvement of the average memory consumption (see dotted black line), we achieve a 6.2% improvement of peak memory consumption. The Figure shows that at all local memory peaks during the execution of *glmnet* we save a small amount of memory while the memory consumption during the remaining parts of runtime of the benchmark is largely unaffected. This results in only a minor reduction of the average memory consumption. Still, even in the presence of a very specific op-

Benchmark	Std R Peak [MB]	P-Sharing R Peak [MB]	Gain Peak [%]	Std R Avg [MB]	P-Sharing R Avg [MB]	Gain Avg [%]	ZPG [#]
b25-1	296.2	228.1	23.0	259.6	192.2	25.9	13
b25-2	131.1	131.4	-0.2	128.8	128.0	0.6	13
b25-3	197.2	164.8	16.4	157.7	112.6	28.6	37919
b25-4	134.2	119.7	10.8	127.2	114.6	9.9	194892
glmnet	354.9	332.8	6.2	249.5	246.0	1.4	46877
ada	187.2	170.1	9.1	156.0	126.2	19.1	2031992
gbm	191.5	193.2	-0.9	147.7	136.0	7.9	464
kknn	316.5	287.6	9.1	274.0	231.0	15.7	421
lda	216.2	208.2	3.7	184.8	175.1	5.3	20447
logreg	213.0	186.7	12.3	184.7	162.8	11.9	955
lssvm	1365.1	631.0	53.8	820.2	381.1	53.5	3972699
naiveBayes	143.6	126.2	12.1	80.8	81.3	-0.6	78
randomForest	565.5	520.4	8.0	390.8	242.7	37.9	1130650
rda	254.1	227.7	10.4	197.0	177.3	10.0	707
rpart	144.5	125.8	12.9	130.7	103.3	20.9	56214

Table 3. Memory Results of page sharing: Std R Peak - peak memory usage by the standard interpreter; P-Sharing R Peak - peak memory usage by optimized interpreter; Gain Peak - relative peak memory reduction achieved by optimized interpreter; Std R Avg - average memory usage by the standard interpreter; P-Sharing R Avg - average memory usage by optimized interpreter; Gain Avg - relative average memory reduction achieved by optimized interpreter; ZPG - number of zero pages found by the content check; Geo. mean of memory gain is 13.6% for peak and 18.0% for average memory usage; Confidence intervals have been omitted as they were smaller than 0.5% for all values

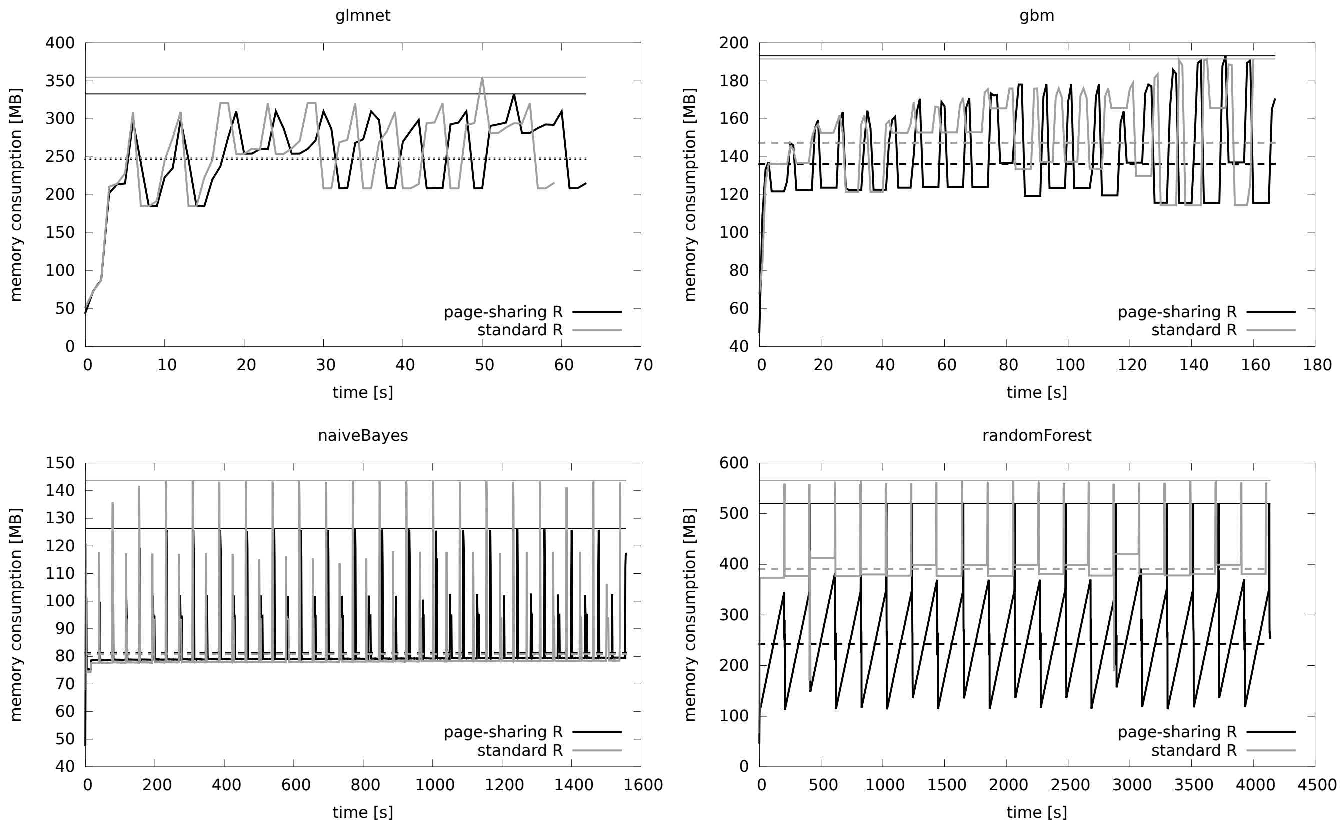

Figure 10. Memory consumption over time for benchmarks with different memory behavior for the standard R interpreter vs. optimized interpreter. Lines at the top mark the respective peak memory usage, dotted lines mark the average memory usage.

timization for sparse matrices we can still offer additional memory savings. As can be seen from column *ZPG* in Table 3 this is due to a large number of pages recovered by the content checks.

Not all benchmarks profit from the content checks though. For example, Table 3 shows that in *gbm* only 464 zeroed pages are recovered by the content checks, so this benchmark profits more from lazy allocation and duplication. The corresponding memory-over-time behavior is shown in the top right of Figure 10. For this benchmark, our optimization does not reduce the peaks of the memory consumption, but there is a marked reduction of memory usage in the valleys between the peaks, reducing the average memory consumption by 7.9%.

Another benchmark that not profit from the content checks is *naiveBayes* with just 78 zeroed pages recovered. The memory-over-time profile of *naiveBayes* in the bottom left of Figure 10 shows that its situation is reversed compared to *gbm*: In *naiveBayes* only the peak memory consumption is reduced by our optimization (large distance between the straight lines at the top), but not the average consumption (small distance between the dotted lines). The Figure shows that *naiveBayes* has much smaller peaks compared to *gbm*, so the large reduction of memory consumption at those peaks only has a small effect on the average memory consumption.

Finally, *randomForest* in the bottom right of Figure 10 provides an example of a benchmark which profits a lot from the recovery of zeroed pages by the content check. Over the full runtime of the benchmark, the content checking reclaims 1,130,650 pages, corresponding to slightly more than 4 GB of memory, which is more than the peak memory allocation of this benchmark. The memory-over-time profile shows a sawtooth curve for our optimized interpreter, so this benchmark uses large blocks of memory which are slowly written to. For our page sharing optimizations this represents an ideal memory usage pattern as we can delay the allocation of memory until the benchmark writes data to it. This results in a 37.9% improvement of the average memory consumption (large distance between dotted lines), which means that we reduce the average time during which the benchmark has a high memory consumption.

Looking back at Figure 10 for *glmnet* (top left), the black line which shows the profile for our optimized interpreter is longer than the grey line for the standard interpreter and there is an increasing shift between the peaks of both lines over time. The reason for this is that our page sharing needs additional CPU time to provide its optimizations. Therefore, we take a closer look at the runtime overhead of our page sharing optimization in the next Section.

6.3 Runtime Overhead of Page Sharing

Our page sharing optimization incurs a runtime overhead compared to the unmodified interpreter. Table 4 shows the runtime of our benchmarks for both the standard and page sharing R interpreters (*Std R*, *P-Sharing R*), the relative overhead of the page sharing interpreter (*Loss*) and the number of times the content check optimization was triggered (*CC*). Here the confidence intervals have been omitted as they were smaller than 1.0% for all values.

Benchmark	Std R [s]	P-Sharing R [s]	Loss [%]	CC [#]
b25-1	69.3	69.8	0.8	10
b25-2	61.6	62.0	0.7	58
b25-3	57.2	58.4	1.9	23
b25-4	60.2	70.3	16.9	1045
glmnet	59.3	64.0	7.9	156
ada	9874.3	10055.7	1.8	12113
gbm	160.4	168.0	4.7	287
kknn	2030.8	2063.6	1.6	486
lda	86.5	98.1	13.3	334
logreg	82.8	89.6	8.2	280
lssvm	530.5	601.2	13.3	1002
naiveBayes	1539.7	1555.6	1.0	31466
randomForest	4107.1	4135.5	0.7	224
rda	7617.8	7871.8	3.3	11849
rpart	61.5	64.2	4.4	304

Table 4. Runtime Results: Std R - runtime with standard R interpreter; P-Sharing R - runtime with optimized R interpreter; Loss - relative runtime overhead incurred by optimized R interpreter; CC - number of content checks executed by the optimized R interpreter; Geometric mean of runtime loss is 5.3%; Confidence intervals have been omitted as they were smaller than 1.0% for all values.

There are multiple reasons for this overhead. First, allocation itself is more complicated as both virtual and physical pages need to be managed. The static refinement (see Section 4.2) that limits our allocator to objects above a size limit of two pages avoids this overhead for small objects where our page sharing optimizations cannot realize any significant memory gains. Second, the first write access to a shared page requires intervention of the fault handler, which is an overhead that is not present in the standard R interpreter. We tried to reduce this overhead with a static refinement that excludes functions which are known to immediately overwrite their memory allocation, but this covers only the case where both the allocation and write access happen in the same interpreter function. If the immediate write is caused by calculations in the benchmark, our static refinement cannot detect this. A case where this happens is *b25-4* which has a runtime overhead of 16.9%. This benchmark recursively calculates the greatest common divisors for two vectors. The R interpreter duplicates those vectors passed as function parameters in each recursion and the benchmark updates these vectors.

Third, our dynamic refinement that checks the content of pages for deduplication also adds runtime overhead. Its overhead is expected to be small for programs where it does not contribute significantly to the memory savings, since it only checks for pages which contain just zero bytes, so it can abort its scan as soon as it finds the first byte that does not match this criterium. An example of a low overhead for content checks can be seen in the *ada* benchmark in Table 4 where the content check was triggered 12113 times, but the total overhead was just 1.8%. On the other hand, the *lssvm* benchmark shows an overhead of 13.3% with just 1002 content checks which found almost four million zero pages according to Table 3. This shows that the overhead of the content check depends not just on the number of times it is called but also on the number of pages it had to search through. The geometric mean of the runtime overhead for all benchmarks is just 5.3%.

Our static refinement that checks the object size to determine if the standard allocator or our custom allocator should be used gives us the opportunity to modify this size limit in order to change our runtime overhead. This results in a trade-off between memory savings and runtime overhead as we expect that an increase of the object size limit results in a decrease of the number of objects that are considered for our custom allocator.

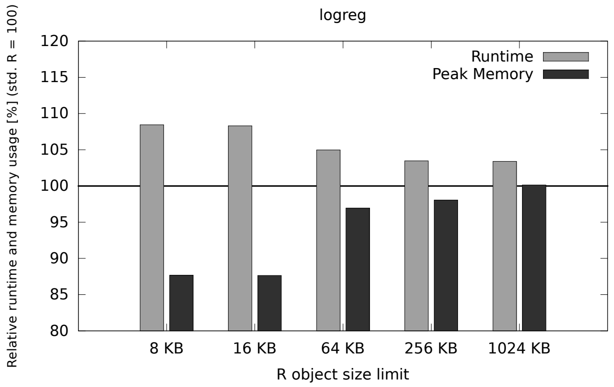

Figure 11. Static refinement using different object sizes; Error bars have been omitted as the confidence intervals were smaller than 0.3% for all values

This trade-off is illustrated in Figure 11 for *logreg* with five different object size limits for our custom allocator shown on the x-axis. The y-axis shows the runtime (grey) and peak memory consumption (black) of the optimized interpreter relative to the same values for the standard R interpreter. Error bars have been omitted as the confidence intervals were smaller than 0.3% for all values. For both values a lower percentage is better, since the 100% baseline represents the runtime and memory values of the standard

Benchmark	Std R Peak [MB]	P-Sharing R Peak [MB]	Gain Peak [%]	Std R Avg [MB]	P-Sharing R Avg [MB]	Gain Avg [%]	Std R [s]	P-Sharing R [s]	Speedup (CI)
logreg-2 1GB	1228.2	1094.8	10.9	965.7	789.6	18.2	6395.5	5785.6	$1.105^{1.144}_{1.071}$
logreg-2 6GB	1228.2	1094.8	10.9	967.8	823.2	14.9	579.8	598.5	$0.969^{0.971}_{0.967}$
lssvm 1GB	1365.1	631.1	53.8	970.0	381.3	60.7	3080.3	593.8	$5.188^{5.350}_{5.029}$
lssvm 6GB	1365.1	631.0	53.8	820.2	381.1	53.5	530.5	601.2	$0.882^{0.885}_{0.880}$

Table 5. Evaluation results with two configurations of RAM; see table 3 and 4 for column descriptions, except Speedup: Runtime speedup factor (Std. R / P-Sharing R). Confidence intervals for runtime are shown (CI), others have been omitted as they are smaller than 0.8%.

R interpreter without our optimization. The 8 KB limit is the same one that was used for the measurements shown previously.

The results in Figure 11 show that for *logreg* increasing the size limit for custom-allocated objects reduces the gain of our optimizations as fewer objects can be optimized. With a size limit of 1024 KB, our gain is slightly negative as we have a memory overhead for our page management data structures. This indicates that in *logreg* the savings of our page sharing system are triggered by R object allocations with a size smaller than 1024 KB. On the other hand, since the increase of the size limit reduces the number of objects handled by our optimization, its overhead contributes less to the overall runtime of the benchmark. With a small object size limit like 8 or 16 KB, the runtime overhead is larger (8.4%).

In all previous measurements the RAM available in the system was sufficient to hold all data used by the benchmark. If this is not the case, our runtime overhead can become insignificant which we will show in the next Section.

6.4 Runtime Reduction with Page Sharing

When the amount of RAM in the system is too small to hold all data required by the benchmark, there are situations where our optimizations can also reduce the runtime of the benchmark instead of adding overhead. This is due to frequent page swaps that require disk I/O when the total capacity of RAM is exceeded, a phenomenon also known as 'thrashing'. To analyze this situation, we will consider two benchmarks. The first one is the *lssvm* benchmark where our optimization provides a large reduction in memory consumption. The second benchmark is an instance of*logreg* where our optimization provides smaller memory gains. We had to increase the memory requirement of the benchmark beyond the capacity of RAM in the system. Therefore, we limited the system to just 1GB of RAM instead of increasing the data set size because the runtime of the benchmarks does not scale linearly with the data set size, resulting in excessively high runtimes. Since *logreg* has a much smaller memory consumption than 1 GB, we have additionally increased the data set size for *logreg* to 70000 samples with 300 numeric features which increases the memory requirements of this benchmark to approximately the same level as *lssvm*. This still results in acceptable runtimes for *logreg*.

Table 5 shows the results for the previous 6GB and the limited 1GB RAM configuration for both benchmarks. *logreg* is now shown as *logreg-2* because it was running with the previously described larger data set. In the 1GB configuration, the system had to swap for both the standard and optimized interpreters, resulting in a large increase in runtime over the 6GB configuration. The peak memory usage for the interpreters are identical in both configurations while the average memory usage differs as this value is time-dependent and thus influenced by swapping. This swapping also increases the variability in our runtime measurements, so we included the confidence intervals for the speedup factors in table 5.

Reducing the available memory from 6 GB to 1 GB has drastically increased the run time for both interpreters. Still, the reduction in memory usage by our optimizations has turned the slowdown (factor 0.969) in the 6 GB configuration into a small speedup (factor 1.105) when the RAM is limited to 1 GB. Depending on the benchmark and its memory usage pattern, a different situation could also happen: In the worst case, the content check of our optimized interpreter touches a large number of pages, forcing them to be swapped in. This additional swap activity can increase the runtime so much that the gains from a reduced memory footprint may become irrelevant. A kernel-level implementation of the content check which we plan for future work, could easily avoid this by not scanning pages that are currently swapped out.

The second benchmark *lssvm* shows something closer to the best case for our optimization: It manages to save enough memory to avoid swapping. In this case we can get significant speedups as shown in Table 5 for the 1 GB configuration of *lssvm*.

Similar to *logreg-2*, the memory usages do not vary much between both configurations. Considering the runtime results, our optimized interpreter only needs 593.8 seconds to run the benchmark which is almost unchanged from the 6 GB configuration. On the other hand, the standard interpreter has now increased its runtime to 3080.3 seconds (51.3 min.) when limited to 1 GB of RAM. This makes the overhead of our optimization irrelevant as the time gained by avoiding page I/Os is much larger. Thus our page sharing optimization allows us to speed up the benchmark by a factor of 5.2 by reducing the peak memory consumption by 53.8%.

This shows that reducing the memory consumption by our page sharing optimization can significantly improve the runtime for memory–hungry benchmarks if the available RAM is constrained. In turn, this can allow the processing of larger data sets.

7. Conclusion

Reducing the memory overhead of R application is a useful optimization that can benefit a large number of applications – especially in cases where the use of swap space can be avoided. We presented an application transparent memory optimization approach that employs page sharing at a memory management layer between the R interpreter and the operating system's memory management. By concentrating on the most rewarding optimizations, the sharing of zero-filled pages and improving the object duplication granularity of the R interpreter to page-level granularity, we avoid the overhead of OS level memory optimization approaches such as deduplication and compression. Using our approach, considerable reductions of the memory consumption for a large number of typical real-world benchmarks could be achieved, which allows the processing of larger data sets. In the case where swapping could be avoided, we managed to achieve a significant speedup.

Our approach has several parameters that would allow for dynamic tuning to speed up our optimization. In a future work we will utilize machine-learning techniques to optimize the trade-off between runtime and memory by tuning these parameters.

Currently, our approach has to replicate a subset of the virtual memory information that is already available in the OS kernel. Future work will concentrate on a hybrid user level/kernel mode implementation of our page sharing approach, which should result in a reduction of the overhead of our current user mode-only implementation and enable possibilities for the concurrent optimization of multiple applications with large memory footprints. This could result in a better utilization of multicore systems since it allows to execute a larger number of applications simultaneously without forcing the OS to swap.

Acknowledgments

This work was partly supported by Deutsche Forschungsgemeinschaft (DFG) within the Collaborative Research Center SFB876, project A3 and partly supported by Oracle Labs. The authors thank Tomas Kalibera, Michel Lang and Bernd Bischl for providing us with feedback.

References

[1] Wang H., Wu P., Padua D., Optimizing R VM: Allocation Removal and Path Length Reduction via Interpreter-level Specialization. In *Proceedings of the International Symposium on Code Generation and Optimization.* Orlando, Florida. 2014.

[2] Neal R., pqR - a pretty quick version of R. University of Toronto, 2014. URL `https://github.com/radfordneal/pqR`

[3] Kalibera T., Maj P., Morandat F., Vitek, J., A Fast Abstract Syntax Tree Interpreter for R. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments.* Salt Lake City, Utah, USA. pp.89–102. 2014.

[4] Bertram A., Renjin: JVM-based Interpreter for the R Language for Statistical Computing. 2014. URL `http://www.renjin.org`

[5] Talbot J., DeVito Z., Hanrahan P., Riposte: a trace-driven Compiler and Parallel VM for Vector Code in R. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques.* Minneapolis, Minnesota, USA. pp.43–52. 2012.

[6] Almasi G., Padua, D.A., MaJIC: A Matlab Just-In-Time Compiler. Languages and Compilers for Parallel Computing. Springer Berlin Heidelberg. pp.68–81. 2001.

[7] Bolz C.F., Cuni A., Fijalkowski M., Rigo, A., Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems.* Genova, Italy. pp.18–25. 2009.

[8] Morandat F., Hill B., Osvald, L., Vitek, J., Evaluating the Design of the R Language: Objects and Functions for Data Analysis. In *Proceedings of the 26th European conference on Object-Oriented Programming.* Beijing, China. pp. 104–131. 2012.

[9] Kotthaus H., Korb I., Lang M., Bischl B., Rahnenführer J., Marwedel P., Runtime and Memory Consumption Analyses for Machine Learning R Programs. Journal of Statistical Computation and Simulation. 2014.

[10] Lang M., Kotthaus H, BenchR: Set of Benchmark of R. TU Dortmund University. 2014. URL `https://github.com/allr/benchR`

[11] R Core Team, R: A Language and Environment for Statistical Computing, R Foundation for Statistical Computing, Vienna, Austria, 2014. URL `http://www.R-project.org`

[12] Valat S., Pérache M., Jalby W., Introducing Kernel-level Page Reuse for High Performance Computing. In *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness.* Seattle, Washington, pp.3:1–3:9. 2013.

[13] Chen L., Wei Z., Cui Z., Chen M., Pan H., Bao Y., CMD: Classification-based Memory Deduplication Through Page Access Characteristics. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments.* Salt Lake City, Utah, USA. pp.65–76. 2014.

[14] Jula A., Rauchwerger L., Two Memory Allocators That Use Hints to Improve Locality. In *Proceedings of the 2009 International Symposium on Memory Management.* Dublin, Ireland. pp.109–118. 2009.

[15] Arcangeli A., Eidus I., Wright C., Increasing memory density by using KSM. In *Proceedings of the Ottawa Linux Symposium.* Ottawa, Ontario, Canada. pp. 19–28. 2009.

[16] Wilson P., Kaplan S., Smaragdakis Y., The Case for Compressed Caching in Virtual Memory Systems. In *Proceedings of USENIX ATC* Monterey, California, USA. 1999.

[17] Beltran V., Torres J., Ayguade E., Improving disk bandwidth-bound applications through main memory compression. In *Proceedings of the 2007 workshop on MEmory performance: DEaling with Applications, systems and architecture (MEDEA '07)* ACM, New York, NY, USA, pp. 57–63. 2007.

[18] Yang L., Dick R., Lekatsas H., Chakradhar S., High-performance operating system controlled online memory compression. In *ACM Trans. Embed. Comput. Syst. 9, 4* ACM. 2010.

[19] Pekhimenko G., Seshadri V., Kim Y., Xin H., Mutlu O., Gibbons P., Kozuch M., Mowry T., Linearly compressed pages: a low-complexity, low-latency main memory compression framework. In *Proceedings of MICRO-46.* ACM, New York, NY, USA, pp.172–184. 2013.

[20] Chihaia I., Gross T., An analytical model for software-only main memory compression. In *Proceedings of the 3rd workshop on Memory performance issues (WMPI '04).* ACM, New York, NY, USA, pp.107–113. 2004.

[21] Nakar D., Weiss S., Selective main memory compression by identifying program phase changes. In *Proceedings of the 3rd workshop on Memory performance issues (WMPI '04).* ACM, New York, NY, USA, pp.96-101. 2004.

[22] Benini L., Macii A., Macii E., Offline Data Profiling Techniques to Enhance Memory Compression in Embedded Systems. In *Proceedings of the 12th International Workshop on Integrated Circuit Design, Power and Timing Modeling, Optimization and Simulation (PATMOS '02)* Springer-Verlag, London, UK, pp.314–322. 2002.

[23] Lawlor O., In-memory data compression for sparse matrices. In *Proceedings of the 3rd Workshop on Irregular Applications: Architectures and Algorithms (IA^3 '13).* ACM, New York, NY, USA. 2013.

[24] Miller K., Franz F., Rittinghaus M., Hillenbrand M., Bellosa F., XLH: more effective memory deduplication scanners through cross-layer hints. In *Proceedings of USENIX ATC'13.* USENIX Association, Berkeley, CA, USA, 279-290. 2013.

[25] Sharma P., Kulkarni P., Singleton: system-wide page deduplication in virtual environments. In *Proceedings of HPDC '12.* ACM, New York, NY, USA, pp.15–26. 2012.

[26] Deng Y., Song L., Huang X., Evaluating Memory Compression and Deduplication. In *Proceedings of the IEEE NAS '13.* IEEE Computer Society, Washington, DC, USA, pp.282–286. 2013.