mmapcopy: Efficient Memory Footprint Reduction using Application Knowledge

Ingo Korb
Department of Computer Science 12
TU Dortmund University
ingo.korb@udo.edu

Helena Kotthaus
Department of Computer Science 12
TU Dortmund University
helena.kotthaus@udo.edu

Peter Marwedel
Department of Computer Science 12
TU Dortmund University
peter.marwedel@udo.edu

ABSTRACT

Memory requirements can be a limiting factor for programs dealing with large data structures. Especially interpreted programming languages that are used to deal with large vectors like R suffer from memory overhead when copying such data structures. Avoiding data duplication directly in the application can reduce the memory requirements. Alternatively, generic kernel-level memory reduction functionality like deduplication and compression can lower the amount of memory required, but they need to compensate for missing application knowledge by utilizing more CPU time, leading to excessive overhead. To allow new optimizations based on the application's knowledge about its own memory utilization, we propose to introduce a new system call. This system call uses the existing copy-on-write functionality of the Linux kernel to avoid duplicating memory when data is copied. Our experiments using real-world benchmarks written in the R language show that our approach can yield significant improvement in CPU time compared to Kernel Samepage Merging without compromising the amount of memory saved.

CCS Concepts

•**Software and its engineering** → **Virtual memory;** •**Information systems** → *Deduplication;* •**Applied computing** → Mathematics and statistics;

Keywords

virtual memory, deduplication, memory optimization, duplication avoidance, system call, R interpreter

SAC 2016,April 04 - 08, 2016, Pisa, Italy

http://dx.doi.org/10.1145/2851613.2851736

1. INTRODUCTION

The size of the main memory can be an important factor limiting the size of a problem that can be solved. Therefore, it is desirable to use the available memory as efficiently as possible. There are cases though where the semantics of a programming language can lead to needless copying of memory – for example when the language specifies pass-by-value semantics, a single function call may require copying huge data structures. In this paper we propose to use the existing copy-on-write functionality of the kernel using a new system call to copy a block of virtual address space.

One current solution that also uses copy-on-write to reduce memory requirements is to use a deduplication system like Linux' Kernel Samepage Merging (KSM) [6], which scans for identical memory pages and combines them into a single copy. This approach is commonly used when running many virtual machines on the same system. Such optimizations are not limited to virtual machines, but can also be applied to programs with large memory requirements. Interpreters for programming languages can be seen as another kind of virtual machine which can benefit from memory optimizations. The R language [3] is an example of a programming language with pass-by-value semantics which is commonly used for statistical analysis and bioinformatics. It is commonly used for processing large vector data structures [1] and the memory pressure is further increased due to R's use of garbage collection, which delays the deallocation.

Although a deduplication system like KSM can be used to reduce the memory footprint for memory-hungry programs, it does not address the underlying problem: The duplicated memory content can only be identified and merged after the duplication has occurred, requiring a large runtime overhead for identifying the duplicated pages. The knowledge of the application about content that needs to be copied can be used to avoid the creation of duplicated pages and thus the overhead for identifying it is avoided. One solution that is applied for example in the R interpreter is the use of reference counting to determine if an object can be modified without copying it. If a copy is needed, the entire object must still be copied. Another solution that avoids large-scale changes in the application is to apply the application's knowledge to memory management. Instead of making unnecessary copies, a copy-on-write mapping of an already existing block of memory can be created to share the memory between the original and copy, utilizing the kernel's functionality to copy only the pages that are modified. Direct

control over this capability is not available to the user space though. In [10], a pure user-space system for this purpose has been described, but it had to replicate some key kernel data structures for virtual memory management.

In this paper, we will present a dynamic page sharing optimization that is implemented in kernel space. For our approach, we developed a system call named *mmapcopy* that avoids unnecessary memory copies by forwarding application knowledge to the existing copy-on-write functionality of the kernel. We demonstrate the benefits of our new system call by comparing the memory savings and runtime overhead against KSM on a set of real-world benchmarks written in the R language [2], showing up to 11.7 percentage points increased memory savings compared to KSM and a geometric mean runtime overhead of just 6.5% when compared to an unmodified R interpreter.

2. RELATED WORK

There are two commonly-used approaches to use RAM more efficiently, deduplication and compression. An evaluation of both approaches based on real memory traces was made by Deng et al. [9], showing that deduplication yields better results than memory compression. A commonly used deduplication scheme on Linux systems is KSM by Arcangeli et al. [6] which also has been the subject of multiple improvements. For example, Miller et al. [7] use I/O based hints to prioritize the scanning of pages that were recently read from disk, reducing the time needed to detect shareable pages. Chen et al. [5] introduce a classification scheme based on access characteristics, comparing only pages within the same class. The Difference Engine [12] attempts to reduce memory by compression and sharing of similar pages, but this requires de-sharing even for read accesses.

All of these improvements are still reactive though, they can only eliminate duplicates after they have occurred. Although some information is used to improve the time needed to detect duplicates, the knowledge of an application about data that is copied is ignored.

Sharing memory pages within a single process appears to be a rarely-used concept: On Linux, it is automatically used to map just a single page filled with null-bytes into new memory allocations which is replaced with as many new pages as needed. This can cause performance issues in high-performance environments, prompting an enhancement by Valat et al. [4] which avoids clearing memory when the application knows that it will overwrite it in the near future. Furthermore, by using memory-mapped file access multiple times on the same file, an application can create shared page mappings itself.

A more restricted version of copy-on-write that operates on whole objects instead of memory pages is sometimes implemented using reference counters. Tozawa et al. have analyzed the details of PHP's copy-on-write scheme and proposed improvements in [8]. R [3] also implements a copy-on-write scheme. Generally, the complete object is copied on modification, resulting in duplicated content for partial modifications. A specific modification to the R interpreter that manipulates the virtual memory mapping from user space to use copy-on-write for partial object modification has been shown in [10]. In addition to copy-on-write, the approach demonstrated there also implements a content-based deduplication similar to KSM.

None of the mechanisms described above can be used to generate a shared, copy-on-write mapping of an already existing block of anonymous (not file-backed) memory, which is the main functionality of the *mmapcopy* system call we describe in this paper. This enables sharing with low runtime overhead by utilizing the application's knowledge about the data to be copied.

3. DEDUPLICATION AND COPY-ON-WRITE

In this section we will describe the operation of memory deduplication as implemented by KSM, used for our evaluation and the concept of copy-on-write which is an important basis for our new system call.

KSM [6] is a subsystem in the Linux kernel that is able to optimize memory usage by merging identical content. It works by comparing the contents of pages with each other and merges duplicates to save memory. To avoid the overhead of scanning every page, KSM only checks those pages that applications have explicitly marked as potentially mergeable using the `mprotect` system call.

This is for example beneficial on systems that run multiple virtual machines for merging the RAM of these virtual machines, but KSM can be useful for any memory-intensive application. Since the same physical page frame must be made visible in multiple virtual locations after merging, it utilizes the memory management unit of the CPU and is thus restricted to memory blocks that are aligned to page boundaries. KSM runs as a kernel thread, so it can be scheduled independently from applications that utilize it. It offers two tunable parameters to control the overhead of scanning pages for identical content: The number of pages to scan per run and the number of milliseconds to sleep in between. By increasing the number of pages, the sharing efficiency can be increased at the cost of runtime overhead.

Since writing to a shared page should only result in a modification of the instance that was written to and not any other shared instance, KSM uses a copy-on-write scheme to ensure that the application's view on memory stays consistent.

Copy-on-write is an optimization for memory allocation that allows a system to reduce memory requirements for identical pages by mapping the same physical page in multiple virtual locations. Copies are made when data is written to one of the virtual locations, thus delaying the memory allocation until it becomes unavoidable. A situation where the same physical page is mapped to multiple virtual locations is not just created by deduplication systems like KSM, but can also occur with other functions of the kernel. One example is memory allocation by an application where the kernel supplies the application with a block of memory that references a single global zeroed-out page.

On the application level, copy-on-write is used when copies of an object may be needed to preserve semantics, but these copies are not always modified. In this case, a reference counting scheme can be used to determine if a given object in memory can be safely modified or if a copy needs to

be made first. Unlike the deduplication schemes mentioned previously, the resulting copy is usually made at object-level granularity instead of page-level, which can result in a significant overhead when partial modifications are made to large objects. Examples of this can be found in PHP and R (see [10] for more details about copy-on-write in R).

In the next section we will present a new system call called `mmapcopy` which makes the copy-on-write functionality of the kernel available to an application and allows it to modify copies of large objects with reduced memory overhead.

4. OPTIMIZING MEMORY FOOTPRINT

To allow an application to optimize its memory footprint by forwarding its knowledge about memory allocation and object modification to the kernel, we have implemented a new system call in the Linux kernel. The following subsections will explain how this system call is used in the application and how it is implemented.

4.1 Utilizing Application Knowledge

Our basic goal is to allow an application to leverage its own knowledge about its memory usage for reducing the memory footprint by utilizing the kernel's copy-on-write capabilities. Therefore, instead of allocating new memory for a copy operation, the application needs to tell the kernel to create additional virtual mappings with copy-on-write.

Due to the nature of the system call, it is best suited for applications that handle objects consisting of many pages. Savings can be realized if an application creates full copies of these objects and makes only partial modifications, so they can still share some of their content with the original. Such patterns can for example occur in programming language interpreters when the language requires pass-by-value semantics even for large array objects. We chose to focus on the R interpreter for our evaluation because its memory usage fits these conditions.

In the case of the R interpreter, we modified the object allocation and duplication functions. Ordinarily, these functions would request memory using the `malloc` function and copy data with `memcpy`. Since the kernel's copy-on-write system utilizes the memory management unit, copies must be aligned at page boundaries. To ensure this, we modified the interpreter with an additional option for allocating memory directly from the kernel using `mmap` instead of `malloc` and releasing it with `munmap` instead of `free`. Similar to [10], we have limited the application of our optimizations to objects that have a size of at least two pages to avoid wasting memory when small objects are allocated. The standard allocation function is used for smaller objects instead, as we expect that any possible savings would be negated by the overhead they would incur. The standard memory allocator in the GNU C library uses a similar scheme, but its threshold for allocating memory using `mmap` is larger, with a default of 128KB. Our system marks objects that are allocated directly from the kernel in a spare bit in R's object header to ensure that the correct function is used when they are freed. Bypassing the C library's allocation features instead of lowering their threshold for direct kernel allocation ensures that we do not accidentally corrupt the internal state of the standard memory allocator when we create additional mappings of an already-existing memory block.

When the interpreter needs to copy an object, it checks if it has been marked as allocated directly from the kernel. If so, it is possible to create a copy-on-write mapping using our `mmapcopy` system call to delay the copy; otherwise the interpreter uses the standard allocate-and-copy code path. Since the copy-on-write mechanism is transparent to the application, no further changes are required for the correct operation of the R interpreter.

4.2 The mmapcopy System Call

The `mmapcopy` system call has an interface similar to the `memcpy` memory copy function. Both functions expect pointers to the source and target addresses and a parameter giving the length of the memory area to be copied. `memcpy` simply copies the specified block of memory from the source address to the target, but `mmapcopy` utilizes paged virtual memory to generate a virtual copy without requiring additional physical memory. Therefore, both the source and destination address must be aligned to page boundaries when `mmapcopy` is used. The system call can also choose a suitable target address itself when a null pointer is supplied as target pointer.

Our envisioned use case for the system call starts with the allocation of memory using the `mmap` system call, using its `MAP_ANONYMOUS` flag. Such memory is guaranteed to be page-aligned and only managed by kernel data structures. If an application now needs to create a partial or full copy of such a memory block, it calls `mmapcopy` with a pointer to this memory as source, a suitable (or null) target pointer and the length of the area to be copied.

In the kernel, the `mmapcopy` system call first removes any mapping that might already exist at the target address. It then iterates over the source memory area and creates a new mapping to the same physical memory pages at the target memory area. Both the source and target memory areas are marked as read-only, which triggers the copy-on-write mechanism of the kernel when the application tries to modify data in either memory area. Finally, it adds the target memory area to the list of mapped memory areas of the process that called `mmapcopy` to ensure that the memory is correctly accounted for and can be freed using `munmap`.

5. EVALUATION

Since interpreted programming languages with pass-by-value semantics that are used to deal with large vectors like the R language suffer from memory overhead, we have chosen the R interpreter to show the benefits of our optimizations. To demonstrate the advantages of our application-guided memory deduplication scheme we compare our system to the generic, but unguided deduplication in KSM. Five different configurations of the R interpreter have been evaluated: A completely unmodified version, one that only uses the direct kernel memory allocation, two that utilize KSM and one with our optimizations. The R interpreter and the benchmarks we used are single-threaded.

5.1 Experimental Setup

The experiments were run on a system equipped with 4GB of RAM and a 2.67GHz Intel Core i5 M480 CPU which has two CPU cores. The page size on this system is 4096 bytes. To avoid influences caused by dynamic CPU frequency scaling, we disabled all power management and turbo functions in the BIOS. The system was running the 64 bit version of Debian 7 as its operating system, using Linux 3.13.0 as the kernel with an extension of about 400 SLOC that implements our `mmapcopy` system call. The R interpreter used for our measurements is R version 3.1.0, compiled with the default optimization flags and GCC 4.7.2.

Since we compare our system against KSM, we chose two settings for it: The default which scans 100 pages per run and sleeps 20 milliseconds between runs and a more aggressive setting that scans 1000 pages per run with the same sleep time. The more aggressive setting was included since our initial measurements had shown only small memory savings for KSM in some benchmarks. This setting has drastically higher CPU time requirements than the default though and is probably not realistic unless a CPU core can be completely dedicated to memory deduplication. Settings between the two presented in this paper could be used to find a better balance between memory savings and CPU usage of KSM, but we did not see any particular sweet spot in our initial measurements and chose the 1000 page setting to approximate a system where a large amount of CPU time can be dedicated to memory deduplication.

Both KSM and our optimization can only work on memory blocks that are page-aligned. This requirement is not met by the original memory allocation functions in the R interpreter, so we added an alternative code path for allocations with a size of at least two pages (see 4.1 for details) and used this as base for both of the KSM configurations as well as our optimizations. In the configurations used for KSM measurements, this memory is marked as mergeable and we apply no further optimizations. In the configuration used for evaluating `mmapcopy`, our optimizations including the use of this new system call are enabled. We have also included a configuration that uses only this changed memory allocation, but does not apply any further optimizations to evaluate the impact of these modifications. A completely unmodified R interpreter is used as the baseline.

We have selected two different sets of benchmarks. One was a subset extracted from the R benchmark 2.5 suite [11] which is a commonly used benchmark suite to evaluate the performance of the R interpreter. For accurate measurements, the scaling parameters were chosen to ensure a runtime of about 60 seconds. The second benchmark set applies a number of real-world machine learning algorithms used for classification [2] to a synthetic data set. The run times for the benchmarks in this set vary between 70 and 10000 seconds. Each combination of benchmark and configuration has been measured 10 times, the reported values are the arithmetic average of these 10 runs.

5.2 Memory and Time Measurement

Although the Linux kernel offers many different memory measurements for processes, to our knowledge none of them accounts for physical pages that are mapped multiple times in the same process. For example, the resident set size reported by the Linux kernel is calculated using the number of pages in a process that are currently mapped to a physical page – if a page is mapped to multiple virtual addresses, it is counted once for each mapping. Since we want to know how much physical memory is saved, we need to distinguish between the amount of virtual memory in a process and the actual number of physical pages allocated. We implemented our own memory measurement that reads the virtual-to-physical mapping for the target process from the `/proc` file system. When a page has been deduplicated by KSM or shared by our new `mmapcopy` system call, the same physical page frame will show up at multiple virtual memory addresses. To derive an accurate count of the number of allocated physical pages in the target process, our measurement therefore counts the number of distinct physical pages found in the process. The calculations are made in a separate process, so they can be scheduled on the second core of the system without influencing the CPU time of the target process. We sample the number of allocated physical pages once per second to monitor the changes over time and use the arithmetic average of these measurements to calculate the average memory consumption for each benchmark. As the calculations for these memory measurements are made using page counts, they include memory wasted due to forced page alignment.

To compare the runtime overhead of our system with the overhead of KSM, we measured the CPU time of the `ksmd` kernel thread which is scheduled separately from the application. This CPU time can be considered either separately from the R interpreter (wall time) or combined with the interpreter's CPU time (total CPU time), so our figures show the CPU time of `ksmd` using a split bar. For `mmapcopy`, a separate measurement is not necessary since it is called from the R interpreter and thus included in its CPU time.

5.3 Memory Savings

In this section we will compare the memory savings of our optimizations against the KSM configurations described in section 5.1. Figure 1 shows the memory savings of each of the four modified configurations relative to the baseline of a completely unmodified R interpreter. A configuration that allocates larger objects directly from the kernel, but does not attempt to use any sharing techniques is shown as *orig+alloc*. The default KSM setting that scans 100 pages per run is labeled *ksm* and the aggressive setting with 1000 pages per run is labeled *ksm-aggr*. *mmcp* represents our optimizations that utilize the new `mmapcopy` system call. In addition to the individual results for each of the 14 benchmarks, Figure 1 also shows the geometrical mean over all of them. Here, we can see that just modifying the memory allocation functions already reduces the mean memory usage to 74.7% of the unmodified R interpreter. This is caused by the direct allocation from the kernel instead of the usual pooling of allocations that the `malloc` function uses: With our modifications, free memory is immediately returned to the system while `malloc` retains it (within reason) to re-use for later memory allocations. Both `mmapcopy` and `ksm` can achieve additional reductions, reducing the mean memory usage to 64.6% (*mmcp*), 67.4% (*ksm*) and 60.1% (*ksm-aggr*)

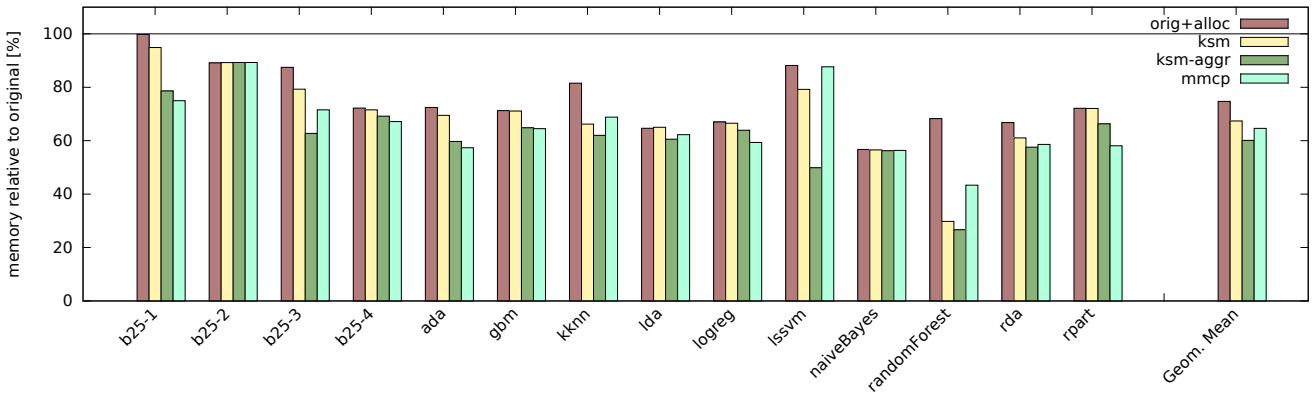

Figure 1: Relative memory usage for the interpreter with modified memory allocation (*orig+alloc*), the two KSM configurations (*ksm* and *ksm-aggr*) and our own optimizations (*mmcp*), compared to the original interpreter (100% line)

when compared to an unmodified R interpreter. This shows that our approach of using application knowledge to reduce memory usage is very competitive with generic approaches.

Compared to the default KSM configuration, our optimizations reduce the memory usage by an additional 2.8 percentage points (pp). Especially large gains for our optimizations compared to *ksm* were achieved for the benchmarks *b25-1* and *ada*. Here we could reduce the memory usage from 94.9% to 75.0% for *b25-1* and from 69.5% to 57.4% for *ada*. In seven of the benchmarks, our optimizations achieve a smaller amount of memory savings than *ksm-aggr*, especially in the case of *lssvm* and *randomForest*. This shows that our approach of avoiding duplicates by utilizing copy-on-write is less efficient than the arbitrary page matches utilized by KSM. Still, compared to the interpreter with modified allocation only, we save 12.7pp memory in *kknn* and 24.9pp in *randomForest*, but basically no savings were achieved in *lssvm*.

Although *ksm-aggr* realizes larger memory savings than our approach in seven of the 14 benchmarks, it does so at the cost of significant runtime overhead as we will show in the next section.

5.4 Runtime Influence

Both KSM as well as our memory optimizations can influence the runtime by cache effects, additional system calls and the CPU time required to scan page contents.

The scan for duplicated page contents in KSM is run as a kernel thread separate from the application, so we have shown it in Figure 2 as a cross-hatched bar segment stacked on top of the pure runtime of the R interpreter. Here, the relative runtime overhead for the interpreter with modified allocation (*orig+alloc*), the default configuration of KSM (*ksm*), the aggressive configuration (*ksm-aggr*) and our configuration (*mmcp*) is shown relative to an unmodified interpreter baseline (100%). The modified memory allocation already introduces a small runtime overhead due to the increased number of system calls, resulting in a mean 6.5% longer runtime compared to an unmodified interpreter. Our optimization improves the runtime of eight of the benchmarks compared to *orig+alloc*, even though it makes additional system calls to utilize `mmapcopy`. Preliminary measurements indicate that this is due to a reduced cache miss rate, since the same physical page needs to be loaded into the cache just once to provide cached access to any of its virtual copies. Conversely, the slightly increased runtimes for *ksm* and especially *ksm-aggr* appear to be due to increased cache miss rates as the scans made by KSM evict data from the CPU cache. The combination of these two effects give our optimizations a slight runtime advantage over the *ksm* and especially *ksm-aggr* configurations when only the interpreter runtime is considered. The additional CPU time of the page scanning employed by KSM is about 6.7% of the interpreter's CPU time in the mean case for *ksm* and 40.2% for *ksm-aggr*. This shows that the parameters chosen for the aggressive configuration of KSM are only viable if a CPU core can be dedicated to page scanning as there would otherwise be an unacceptably-large CPU overhead.

Altogether, our optimization achieves 2.8pp higher memory savings on average than the *ksm* configuration, but has a 3.4pp smaller runtime overhead when the time for page scanning is not considered and 10.4pp if the scan time is added to the interpreter runtime. Compared to the interpreter with modified allocation (*orig+alloc*), `mmapcopy` reduces the mean memory usage by 10.1pp and reduces the runtime overhead of the modified allocation by 2.9pp. This shows that our new system call can be efficiently utilized to reduce an application's memory footprint.

6. CONCLUSION AND FUTURE WORK

This paper explores an approach for reducing the memory footprint of memory intensive programs by utilizing application knowledge via a new system call. By applying the existing copy-on-write functionality of the kernel in a new way, we avoid duplicated memory pages when data is copied, e.g. due to call-by-value semantics. Our evaluation based on

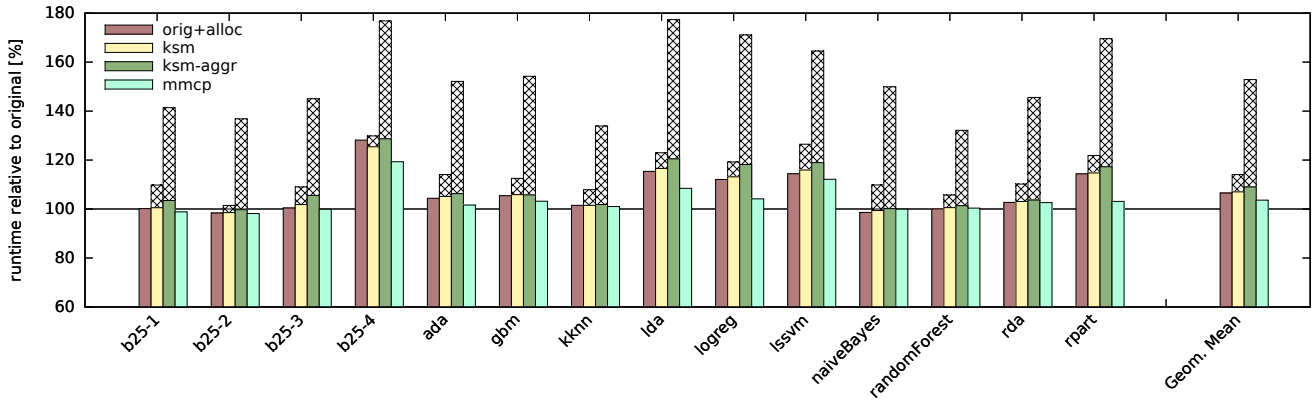

Figure 2: Relative runtime overhead for the interpreter with modified memory allocation (*orig+alloc*), the two KSM configurations (*ksm* and *ksm-aggr*) and our own optimizations (*mmcp*), compared to the original interpreter (100% line)

the R interpreter shows that our method increases memory savings by up to 11.7pp compared to the generic approach of KSM, but outperforms it on runtime. For KSM, 6.7–40.2% additional CPU time are required for page scanning, while our method decreases runtime slightly due to more efficient cache usage. In the future, we plan to explore other memory-hungry applications and investigate options to apply application knowledge to save memory between independent processes running on multiple cores. We also plan to investigate the possibility of implementing our system call in a library that overrides the standard memory allocation and copy functions to enable an application to profit from `mmapcopy` without modification to its source code.

7. ACKNOWLEDGEMENTS

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 would like to thank Michael Engel and Andreas Heinig for providing valuable feedback.

8. REFERENCES

[1] 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.

[2] Lang M., Kotthaus H, BenchR: Set of Benchmark of R. TU Dortmund University. 2015. URL https://github.com/allr/benchR

[3] R Core Team, R: A Language and Environment for Statistical Computing, R Foundation for Statistical Computing, Vienna, Austria, 2015. URL http://www.R-project.org

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] Tozawa A., Tatsubori M., Onodera T., Minamide Y., Copy-on-write in the PHP language. In *Proceedings of POPL '09*. ACM, New York, NY, USA, pp. 200–212, 2009.

[9] 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.

[10] Kotthaus H., Korb I., Engel M., and Marwedel P. Dynamic page sharing optimization for the R language. In *Proceedings of the 10th ACM Symposium on Dynamic languages*. ACM, New York, NY, USA, pp.79–90. 2014.

[11] Grosjean P., Urbanek S. R Benchmark 2.5 URL http://r.research.att.com/benchmarks/R-benchmark-25.R 2015.

[12] Gupta D., Lee S., Vrable M., Savage S., Snoeren A., Varghese G., Voelker G., Vahdat A. Difference Engine: Harnessing Memory Redundancy in Virtual Machines In *Communications of the ACM* ACM, New York, NY, USA, vol. 53, no. 10, pp.85–93. 2010.