RUMA has it: Rewired User-space Memory Access is Possible!

Felix Martin Schuhknecht Jens Dittrich Ankur Sharma

Information Systems Group
infosys.cs.uni-saarland.de

Abstract

Memory management is one of the most boring topics in database research. It plays a minor role in tasks like free-space management or efficient space usage. Here and there we also realize its impact on database performance when worrying about NUMA-aware memory allocation, data compacting, snapshotting, and defragmentation. But, overall, let's face it: the entire topic sounds as exciting as 'garbage collection' or 'debugging a program for memory leaks'.

What if there were a technique that would promote memory management from a third class helper thingie to a first class citizen in algorithm and systems design? What if that technique turned the role of memory management in a database system (and any other data processing system) upside-down? What if that technique could be identified as a key for *re-designing various core algorithms* with the effect of outperforming existing state-of-the-art methods considerably? Then we would write this paper.

We introduce RUMA: Rewired User-space Memory Access. It allows for *physiological* data management, i.e. we allow developers to freely rewire the mappings from virtual to physical memory (in user space) *while* at the same time exploiting the virtual memory support offered by hardware and operating system. We show that fundamental database building blocks such as array operations, partitioning, sorting, and snapshotting benefit strongly from RUMA.

1. INTRODUCTION

Database management systems handle memory at multiple layers in various forms. The allocations differ heavily in size, frequency, and lifetime. Many programmers treat memory management as a necessary evil that is completely decoupled from their algorithm and data structure design. They claim and release memory using standard `malloc` and `free` in a careless fashion, without considering the effects of their allocation patterns on the system.

This careless attitude can strike back heavily. A general example to counter this behavior is *manual pooling*. With classical allocators (like `malloc`) it is unclear whether an allocation is served from pooled memory or via the allocation of fresh pages, requested from the kernel. The difference, however, is significant. Requesting fresh pages from the system is extremely expensive as the program must be interrupted and the kernel has to initialize the new pages with zeroes, before the program can continue the execution. Thus, careful engineers implement their own pooling system in order to gain control over the memory allocation and to reuse portions of it as effectively as possible. However, manual pooling also complicates things. To write efficient programs, engineers rely on *consecutive memory regions*. Fast algorithms process data that is stored in large continuous arrays. Data structures store that data as compact as possible to maximize memory locality. This need is anchored deeply in state-of-the-art systems. For instance, the authors of [13] argue against storing the input to a relational operator at several memory locations for MonetDB: "*It does not allow to exploit tight for-loops without intermediate if-statements to detect when we should skip from one chunk to the next during an operator.*"

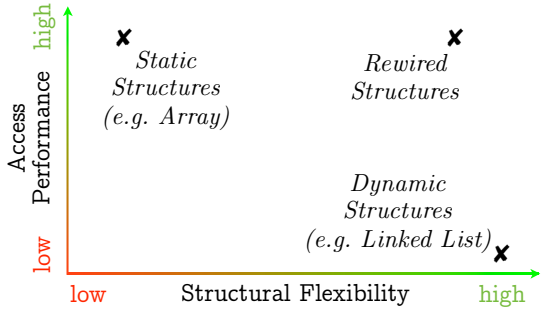


Figure 1: ***Structural Flexibility*** *vs* ***Lookup Performance*** *— While static structures like the array provide fast and convenient access performance, their structure is hard to modify (extend, shrink). While dynamic structures like the linked list are easy to modify, the lookup of entries is indirect and slow. Rewired structures offer direct access and high structural flexibility at the same time.*

Unfortunately, it is not always possible to gather large consecutive memory regions from the pool due to fragmentation. To work around this problem, memory can be claimed as chunks from the pool, using a simple software-based indirection. Allocations of memory are served by glueing together individual memory chunks via a directory. Thus, instead of accessing the entry at offset i by `a[i]`, the access is performed indirectly via `dir[i / chunkSize][i % chunkSize]`. Of course, this relaxes the definition of continuous memory, as every access has to go through the indirection now. As we will see, depending on the usage of the memory, this can incur significant overhead.

Obviously, demonstrated at the example of pooling, we face a general trade-off in memory management: *flexibility* vs *access performance*. Apparently, these properties seem to be contradictory to each other. On the one hand, a static fixed-size array is extremely efficient to process in tight loops, but hard to extend, shrink, or modify structurally. On the other hand, a chunk-based structure as

Proceedings of the VLDB Endowment, Vol. 9, No. 10

the linked-list is highly flexible, but suffers from an access indirection. Figure 1 visualizes this problem. The goal of RUMA is precisely to end this duality as rewired structures offer both the access performance of a consecutive memory region *while* providing the flexibility of a chunk-based structure.

To do so, we exploit a mechanism that is already present in todays operating systems: the virtual to physical page mapping, realized by a page table maintained by every process. Thus, instead of replicating this concept on the user level via a software-indirection, we make the existing virtual memory facility work for us. To do so, we introduce the concept of *rewiring memory*, which consists of two parts: (1) we reintegrate the concept of physical memory into user space without modifying the kernel in any way; (2) we actively manipulate the mapping from virtual to physical pages during runtime. This allows us to both reclaim individual pages from a pool and still form actual continuous memory regions that can be accessed with little to no overhead. Apart from the benefits in memory allocation, flexible rewiring of individual pages can come very handy in numerous situations. We believe that the design of efficient data management algorithms is only possible with the knowledge of how the internals of the memory system work — and consequently exploit them as much as possible. This is exactly what we cover in this paper, by making the following contributions:
(1) RUMA. We introduce **Rewired User-space Memory Access** (RUMA or 'rewiring' for short) allowing us to rewire the mapping of virtual to physical memory addresses at runtime. This remapping happens at the granularity of pages (both small or huge pages). The technique does not need any modifications of the operating system and also does not interfere with process isolation or user security. In particular, we do not disable any virtual memory mechanisms, i.e. all of the benefits of virtual memory management are still available. The trick is to piggyback on the memory mappings that are maintained by the operating system anyways and exploit and manipulate the existing mappings for higher-level tasks. We can realize the entire technique in user space and purely in software, without modifying the linux kernel. We will investigate the toolset provided by the kernel to separate virtual and physical memory in the user space and we will show how to freely manipulate the mapping from virtual to physical pages at runtime (see Section 2 and 3).
(2) Micro-Benchmarking. We benchmark the costs of rewiring in depth. First, we perform a set of *micro-benchmarks* to analyze both behavior and performance for memory allocation and access of rewiring in comparison to their traditional counterparts. Then, we inspect the costs and types of individual page faults to see their impact on the techniques and look at the expenses of shuffling items using rewiring respectively copying. Finally, we measure the impact of different access patterns on the techniques to understand the need for rewiring. We will learn that a major effect of rewiring is that it allows us to push down one storage indirection down to the operating system. At the same time, rewiring drastically increases the flexibility of storage management without introducing considerable overhead. This has a dramatic runtime effect for all mass-operations reading or writing memory (see Section 4).
(3) Applications. Rewiring can be applied in many places in data management. It applies whenever data is copied or moved around, e.g. for resizable data structures, partitioning, merging, hiding of data fragmentation, and realizing multiple views on the same physical memory. Rewiring also applies in situations where data is read-only or very hard to update in place, e.g. when applying a set of changes collected in a differential file or when adaptively refining a part of an index. Investigating all of the possible applications of RUMA is beyond the scope of a 12-page paper. From the long list of possible applications of rewiring we will focus on three applications that are central to many data managing tasks (see Sections 5.1 to 5.3) and leave other promising work to the future (see Section 6).
(4) Rewired Vector. We demonstrate the concept of rewiring at one of the most fundamental structures in data management: the array. It serves as a building block for fundamental main-memory structures like columns, hash tables, and indexes. One array operation that is particularly painful is resizing. We cannot enlarge an array without triggering costly physical copy operations, fresh page allocations, or wrapping the array in software into a list of arrays (leading to an additional storage indirection that is painful to handle). This resizing problem is also at the heart of several read-optimized structures like column layouts and read-optimized indexes including differential indexes. Using STL's resizing array implementation (vector) as a pivot for this problem we will introduce a rewired vector. This data structure grows and shrinks pagewise without any copying of old entries, thus significantly improving over STL vector. Further, we are still able to use pooled pages underneath, in contrast to our second baseline using the system call `mremap` for the resizing. Finally, we compare the rewired vector with our third baseline using a software-indirection. We show that only our structure manages to integrate comfortable flexibility into an array-based data structure (see Section 5.1).
(5) Rewired Partitioning & Sorting. We now pick an important building block in database systems: radix-based partitioning. It is a fundamental technique for indexing, join processing, and sorting. We investigate two state-of-the-art out-of-place partitioning algorithms [18], which either perform a histogram generation pass beforehand or maintain a linked list of chunks inside the partitions to handle the key distribution. We also test a version enlarging the partitions adaptively using `mremap`. We propose a rewired partitioning algorithm that manages to avoid the histogram generation altogether while still producing a contiguous output array by embedding our rewired vector. We show that partitioning and accessing the partitions in a separate sorting phase is significantly faster for rewired partitioning than for all the baselines (see Section 5.2).
(6) Rewired Snapshotting. Executing short running OLTP queries and long running read-only OLAP queries is a major challenge in database management. A way to efficiently pack them in one system is to provide different isolated copies of the transactional data to long running read only queries so that short running queries can perform fast updates without blocking. One way to snapshot data is to make a copy using functions such as `memcpy`, but as the size of data grows, duplicating data can have significantly higher costs. Systems like Hyper [8] use standard Linux features like copy-on-write to lazily snapshot transactional data only when it is needed. Although they improve the cost of snapshotting to a great extent, they usually pay a significant cost of allocating fresh memory pages while performing actual copy-on-write. We use rewiring for reusing physical pages to overcome this overhead. We show that reducing this overhead can improve throughput of query processing systems by 15% to 95% depending upon snapshotting frequencies and workload compared to the approach used in HyPer [8] (see Section 5.3).

2. VIRTUAL MEMORY MANAGEMENT

Let us first briefly recap virtual memory management as supported by the *Linux* kernel. We focus on Linux as we believe this is by far the most common platform for data management applications and particularly servers.

Virtual memory has many advantages: processes may allocate more memory than physically available, the memory belonging to different processes may easily be protected, and all binaries may be compiled using static virtual addresses. Traditionally, the programmer works solely on the level of virtual memory. Physical

memory is simply not accessible to him. In most situations, the view solely on virtual memory is convenient and handy for the user. However, in recent years, programmers repeatedly identified situations in which an awareness of the underlying physical memory can be beneficial. For instance, the KISS-tree [10], a radix trie, implements an extremely wide root-level node of 256MB as only a portion of it is actually physically allocated. Another example is the previously mentioned HyPer [8], that exploits implicit copy-on-write performed between processes related using `fork`. The authors of [19] engineer a virtual-memory aware counting sort, that avoids the initial counting pass by exploiting massive over-allocation of the output array. Finally, in [4] the need for a user mode page allocation system is claimed, that intends to separate virtual and physical memory in user space. The author requests the OS developers to rewrite the system calls such that physical pages can be allocated manually by the programmer and mapped freely.

In the following, we will demonstrate that such a deep change of the system is not necessary at all: we show how *existing features* in Linux can be exploited to achieve exactly that: separate physical and virtual memory in user space and freely manipulate the mappings between them. How could we get access to those mappings?

2.1 Physical Memory: Main-Memory Files

There is a way to work around this limitation in form of *main memory files*. Linux provides main memory filesystems that are mounted and used like traditional disk-based filesystems, but backed by volatile main memory. Widely used representatives are `tmpfs` and `hugetlbfs` which are typically used for shared memory objects that can be backed both by small and huge pages. To evaluate the impact of the page size, we perform the micro benchmarks in Section 4 both with small and huge pages backed files where appropriate. We can now create a fresh file using a simple call to the standard `open` system call as follows:

```
int fd = open("/mnt/mmfs/f", FILE_FLAGS, MODE);
```

This call tries to open[1] the file that is named `f` and creates it, if it does not exist. It returns the file descriptor in form of an integer, which will serve as a handle to identify the file later on. As a newly created file has a length of zero bytes by definition, we also have to set its size in the next step to s bytes. This can be done using a system call to the function `ftruncate(fd, s)`.

From this point on, we have a programmatic representation of physical memory. Although we do not directly handle physical memory pages, we know that file `f` is backed by physical memory. Specifically, for a page size of p, this means that in file `f`, the memory at offsets $[0; p-1]$ is backed by *some* physical page, say physical page $ppage_{42}$ and the memory at offsets $[p; 2 \cdot p - 1]$ is backed by another physical page, say $ppage_7$ and so on. Figure 2 visualizes the concept. Notice, that the mapping from offsets to physical files is handled by the file system and no reimplementation is necessary.

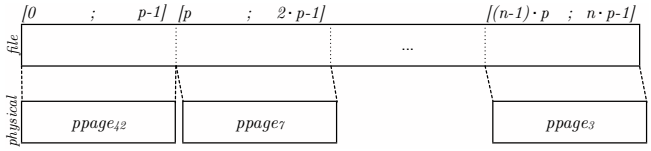


Figure 2: ***Backing of a main memory file by physical pages when using a main-memory file system****. The file offsets in interval $[i \cdot p, (i+1) \cdot p - 1]$ are mapped to some physical page by the file system. This mapping is not accessible for the user.*

Using a main-memory files system, we could now reimplement our own virtual memory mechanism. However, like that we would lose the existing hardware and operating system support for virtual memory management including all their benefits. For instance, we would neither be able to exploit the translation lookaside buffer of the CPU that caches translations from virtual to physical addresses nor could we use the hardware page walker in case of a TLB miss. Hence, we need to perform one additional step.

[1] An alternative is `memfd_create`, introduced in kernel 3.17.

2.2 Virtual Memory: mmap

We need to bring virtual memory into the game. To do so, we create a region of virtual memory and map it to the file. It is important to note that in the following all accesses will still be performed entirely through virtual memory — there will not be any direct data accesses to physical memory. On linux, the way to create a virtual memory mapping onto a main memory file is performed by a system call to `mmap` as follows:

```
void* vmem = mmap(b=NULL, s,
                  PROT_READ | PROT_WRITE,
                  MAP_SHARED, fd, 0);
```

In this example, the `mmap` call creates a virtual memory area of size s and maps it to the main memory file `fd` (as created in Section 2.1). Passing `b=NULL` as the first argument indicates, that the kernel should decide on the start address b of the new virtual memory area, which is guaranteed to be at a page boundary. Later on, we will pass an existing address b to remap existing mappings (see Section 3). `MAP_SHARED` states that changes through the mapping will be propagated to the underlying file. The alternative option would be `MAP_PRIVATE`, that does not propagate the changes but instead creates a private copy of the page to modify. The last argument, which is `0` in this example, specifies the start offset for the mapping into the file. By mapping a virtual memory area of size s into file `fd` at offset `0`, we basically map s/p many virtual pages to the physical pages backing the main-memory file (we consider s to be a multiple of the page size). Thus, we have created a handmade virtual memory to file offset mapping, which is then again indirected by the file system to physical pages. Figure 3 visualizes this mapping. In total, we have created a two-level mapping of virtual memory to physical pages. Sounds expensive? Don't worry. In the following, the filesystem will only be called very rarely. Only when accessing a virtual page for the first time, the file system is involved. The second access to that virtual page does not differ from accessing normal virtual memory. Hence, all calls except the first one have to resolve a one-level mapping only.

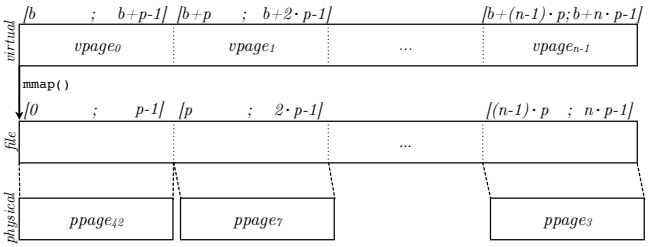


Figure 3: ***Mapping of virtual memory to main memory file****. A single call to `mmap` maps multiple virtual memory pages to the main-memory file. The start address of the area is denoted as b. The virtual page $vpage_i$ starting at address $b + i \cdot p$ is mapped to the file at offset $i \cdot p$ which in turn is backed by some physical page.*

2.3 Virtual Memory: Reserving vs COW

Let us go back to the normal memory allocation using `mmap` on anonymous memory, e.g. as used my `malloc` for large allocations:

```
char* data = mmap(NULL, 42 * p,
                  PROT_READ | PROT_WRITE,
                  MAP_ANONYMOUS | MAP_PRIVATE,
                  -1, 0);
```

The incautious programmer might think that this call allocates 42 physical main-memory pages of size p. What actually happens is the following: the operating system simply *reserves* a virtual address space of size $42 \cdot p$. This is done by creating a *conceptual memory mapping* in form of a `vm_area_struct`. By maintaining these structures, the kernel keeps track, for each process independently, which virtual address ranges have been *reserved* for that process. Notice that the operating system makes only *one* entry for all 42 pages at this point; no entry is inserted into the page table. The process' page table will only be populated on demand whenever the data is accessed. This works as follows.

Let us assume we try to read from some virtual address, e.g. by accessing `data`$[7 \cdot p + 16]$. In that case, the operating system will try to retrieve the mapping of that virtual page to its physical page at the virtual address `data` $+ 7 \cdot p$. If that mapping is neither in the TLB cache, nor in the page table, this will trigger a page fault. The operating system will now inspect the `vm_area_struct` instances to determine how this page fault can be resolved. It finds the previously created `vm_area_struct` instance which determines that `data` $+ 7 \cdot p$ is a valid virtual address as it was reserved by this process. Hence, the operating system will insert a new mapping into the page table from the virtual page starting at `data` $+ 7 \cdot p$ to the so called *zero-page*. The zero-page is a read-only page where all bytes are set to zero. Now, we have a valid entry for this virtual page, hence we can serve the read request to `data`$[7 \cdot p + 16]$. It will return 0. Any read request touching a virtual page that does not have an entry in the page table will be mapped like this. Now let's assume we write to the same page `data` $+ 7 \cdot p$ for the first time say at address `data`$[7 \cdot p + 20]$. What happens? In this case the operating system will apply copy-on-write (COW). It will get a new physical page, copy the contents of the zero-page over that physical page, write the new value to that page, and update the entry in the page table: the virtual page starting at `data` $+ 7 \cdot p$ now maps to the new physical page. This happens only for the first write to any virtual memory page. In summary, a call to `mmap` does neither allocate physical memory nor does it modify the page table.

2.4 Main-Memory File: Reserving vs COW

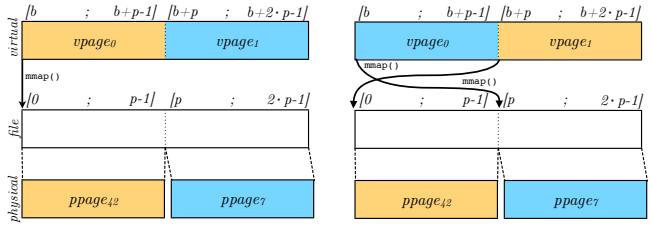
As we have discussed the behavior in the two-layered case in the previous Section 2.2, let us now inspect the differences in the three-layered memory-mapping, which we have in the case of a file-backing, as displayed in Figure 3.

Just like in the two-layered mapping the call to `mmap` simply creates a `vm_area_struct` without modifying the page table in any sense. What happens now if we access a virtual page for the first time, i.e. we trigger a page fault? In that case, the operating system will inspect the `vm_area_struct` and notice that this address space is backed by a file at a certain offset, not by anonymous memory. The operating system will then request the file system to return the address of the physical page it uses to back that offset. As the physical page does not yet exist, the file system allocates it (equivalent to the page-fault handling for anonymous memory), and returns the physical address of it. With this address, the page table entry can be created. Of course, the page fault handling mechanism employed by the file system adds some overhead compared to a page fault on anonymous memory. But as the page fault through the file system happens only for the first access to that virtual page, the three-layered mapping is basically turned into a two-layered mapping after this. Subsequent access costs are equal to those of anonymous memory. We will micro-benchmark this in detail in Section 4.3.

3. REWIRING MEMORY

Up to this point, we have discussed virtual memory that is backed by main memory files instead of anonymous memory. We did this in order to reintroduce the concept of physical memory into the user-space, to get separate handles for virtual and physical pages. However, so far, we did not exploit the power of this new freedom and what we believe to be the strongest aspect of this approach: the ability to actively modify, i.e. *to rewire* the mappings from virtual to physical pages. Doing so will allow us to *use rewiring as a building block for designing algorithms and data structures*.

To get started, let us discuss the following example. Assume we have an array occupying two pages of size p each. Now, we want to swap the contents of the two pages backing the array. Traditionally, this is done with a three-way-swap with one additional page as a helper. These operations trigger actual physical copy operations of three pages. Using rewiring, we can perform this swap using neither physical data copying nor any allocation of helper memory.



(a) Before: $vpage_0$ shows the contents of $ppage_{42}$, $vpage_1$ shows the contents of $ppage_7$.

(b) After: $vpage_0$ shows the contents of $ppage_7$, $vpage_1$ shows the contents of $ppage_{42}$.

Figure 4: ***Rewiring two pages.*** *Notice that no data is physically copied across physical pages.*

Figure 4 shows the concept. Initially, in Figure 4(a), a linear mapping of a two page sized virtual memory area starting at address b into the file exists. We created it as follows:

```
// (0) create a linear mapping:
char* b = (char*) mmap(NULL, 2 * p,
                       PROT_READ | PROT_WRITE,
                       MAP_SHARED, fd, 0);
```

Precisely, we map the virtual page $vpage_0$, starting at address b, to the file at offset 0, which is backed by physical page $ppage_{42}$. Consequently, we map the virtual page $vpage_1$, starting at address $b+p$, to the file at offset p, that is backed by physical page $ppage_7$. We now perform the actual page swap solely by rewiring the mapping from virtual pages to the file respectively the physical pages. Figure 4(b) shows the state after rewiring. No data was physically copied across pages. All we did is update two links using the following two calls to `mmap` in combination with `MAP_FIXED`:

```
// (1) Remap 1. vpage
// to file offset p
mmap(b, p,
 PROT_READ | PROT_WRITE,
 MAP_SHARED | MAP_FIXED,
 fd, p);
```

```
// (2) Remap 2. vpage
// to file offset 0
mmap(b + p, p,
 PROT_READ | PROT_WRITE,
 MAP_SHARED | MAP_FIXED,
 fd, 0);
```

The first `mmap` call (1) remaps the virtual page $vpage_0$, starting at address b, to the file at offset p, which is backed by the physical page $ppage_7$. The key component of this control is using `mmap` in combination with the argument `MAP_FIXED`. By providing this option, we can remap a virtual page to a different physical one by providing a new offset into the file. The second call (2) remaps the virtual page $vpage_1$, starting at address $b + p$, to the file at offset 0, which is backed by the physical page $ppage_{42}$. After this, the content visible through the virtual pages is swapped without copying any physical pages.

Obviously, we could have achieved a similar effect by implementing a software-directory by ourselves. Like that we could have swapped pointers to the two pages just like we swapped the two

mappings through `mmap`. However, doing so *we would have introduced one additional indirection for every access to an array slot*. Traversing this indirection can be very costly under certain access patterns. The beauty of rewiring is that virtual memory maintains one indirection *anyways*. Instead of adding an auxiliary indirection level, we let the existing one work for us. Hence, the central question in the following is: when can we piggy back on the existing virtual to physical page mappings *in order to get rid of one level of indirection in our algorithms?* In other words, how could we change state-of-the-art data processing techniques to delegate some of their implementations to operating system and hardware?

Before approaching these questions, let us outline how rewiring could be wrapped inside of a lightweight library, that simplifies the integration of the technique into existing applications. At the heart of rewiring is the separation of physical and virtual memory and the mapping between these two memory types. Thus, the core of the library consists of the following three components: (1) The allocation of physical memory that wraps the creation, maintenance, and initialization of main memory files. (2) The allocation of virtual memory that wraps the calls to `mmap` and keeps track of the state of the current mappings. (3) The rewiring functionality, which internally calls `mmap` to establish and modify the mappings between virtual and physical pages. The library can further increase the usability by rewiring on page identifiers instead of raw memory addresses, while still exposing them to the user if requested.

Overall, these components already suffice to form the core of a rewiring library, that can be assembled into existing code. However, before we can actually integrate rewiring into applications, we need to understand the possible overheads of rewiring in depth.

4. MICRO-BENCHMARKS

In this section, we will perform a set of micro-benchmarks to understand the impact of virtual memory and rewiring. Firstly, we inspect the costs of allocation and access. Secondly, we evaluate the page fault mechanism in detail. Thirdly, we dynamically rewire existing mappings and measure the cost. Finally, we look at the impact of different access patterns on the memory.

4.1 Experimental Setup

We run all experiments on a two-socket server consisting of two quad-core Intel Xeon E5-2407 with a clock speed of 2.2 GHz. The CPU does neither support hyper-threading nor turbo mode. The sizes of the L1 and L2 caches are 32KB, respectively 256KB; the shared L3 cache has a size of 10MB. The processor offers 64 slots in the fast first-level data-TLB to cache translations of virtual to physical 4KB pages. In a slightly slower second-level TLB, 512 more translations can be stored. For 2MB huge pages, the TLB cache can store 32 translations in L1 dTLB. In total, the system is equipped with 48GB of main memory, divided into two NUMA regions. For all experiments, we make sure that all memory (both file-backed and anonymous) is allocated on a single NUMA region and that the thread is running on the socket attached to that region. The operating system is a 64-bit version of Debian 8.1 with Linux kernel version 3.16. The codebase is written in C99 and compiled using gcc 4.9.2 with optimization level 3. Throughout the following micro benchmarks, we will use a dataset of 1 billion entries, where each entry is of type `uint64_t` and has a size of 8B leading to a total size of roughly 7.45GB, unless mentioned otherwise.

4.2 Allocation Types

Before we can start with the evaluation, we have to define what an allocation actually means in our context. In the following, we distinguish three different types of allocations, that will serve as the competitors in our evaluation:

(a) Private Anonymous Memory — Allocating a memory area of n pages means mapping a consecutive virtual memory area of n pages to n (unfaulted) anonymous physical pages. The operating system resolves all page faults with fresh physical pages, we can not use a pool. We can access the memory directly.
(b) Software-Indirected Memory — Allocating a memory area of n pages using a software-indirection means creating a directory of n slots where each slot contains the virtual address of a page in the pool. We realize the pool using (faulted) virtual pages. We translate and redirect any access through the directory.
(c) Rewired Memory — Allocating a memory area of n pages using rewiring means mapping a consecutive virtual memory area of n pages to n (faulted) pages in the pool. We realize the pool using a main-memory file and we access the memory directly.

When we use a page pool in (b) and (c), the way in which we select the mapped pages can have an influence on the mapping and access performance. In the best case, where the pool is non-fragmented, the n requested pages can be gathered consecutively. In the worst case of a highly fragmented pool, we have to gather each page individually. Thus, we test both extremes in the following where meaningful. Note that we do not count the effort to find and maintain unused pages in the pool.

4.3 Allocation & Access

The first step in understanding the runtime costs of memory usage is measuring memory allocation and its implication on access performance. To analyze the total cost and impact of allocation using different techniques, we perform the following simple experiment: Firstly, we *allocate* a memory area. The allocation of a memory area highly differs depending on the used memory management, see Section 4.2. Secondly, we *write* random values to the area sequentially from start to end. Finally, we *repeat* the sequential write pass. We compare the total time of the three steps under the allocation types (a), (b), and (c), where for (b) and (c) using the page pool, we test both a sequential and random assignment of pages. Figure 5 shows the results. For the allocation type (a) using private anonymous memory, we focus solely on huge pages as the page fault costs for small pages are significant and render them inferior over huge pages in basically all scenarios. For the types (b) and (c) we test both small and huge pages, as the page size influences the flexibility of the memory and both sizes can come handy in certain situations.

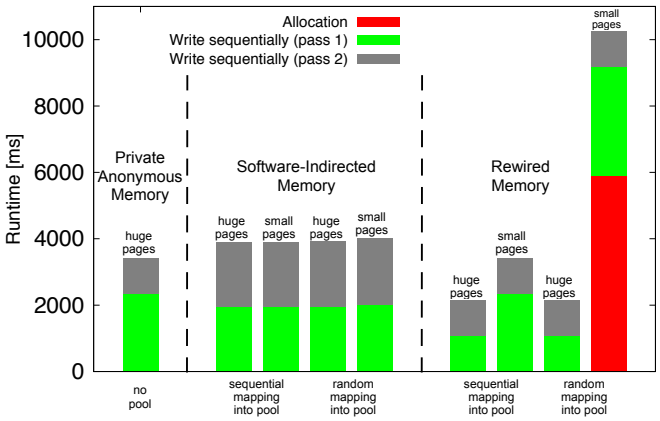


Figure 5: Allocation costs (red), sequential write costs for the first pass (green), and sequential write costs for the second pass (grey). The graph compares the allocation types of Section 4.2 using both small and huge pages where appropriate.

We can see that the allocation phase is basically for free in all cases, except when randomly rewiring individual small pages. When

rewiring on a per huge page basis (random pool), the 3815 necessary calls to `mmap` are negligible. When mapping small pages, the almost 2 million calls (and the generated `vm_area_struct` instances) are painful. However, mapping each page individually simulates the extreme case of the most fragmented pool that is possible. With a good pool management scheme underneath, this is very unlikely to occur [7].

More interesting is the runtime of the first sequential write pass. Here, rewired memory clearly outperforms the remaining two methods as it manages to combine the reusing of pages as well as direct access. During the first writing to the rewired memory, soft page-faults are triggered, that are only significant for small pages. We will inspect the different types of page faults (soft vs hard) in Section 4.4 in detail. The software-indirected memory suffers from the offset translation and the lookup into the directory. Overall, rewired memory is around 1.8x faster than the software-indirected memory for huge pages. For private anonymous memory, the first write performs even worse than in the software-indirected case and is 2.2x slower than rewired memory using huge pages. As the mapping does not rely on a pool, the first write to a virtual pages triggers a costly hard page-fault, that is served by the operating system with a zeroed physical page.

The second sequential write pass shows for rewired memory the same runtime as the first pass in the case of huge pages, for small pages it is significantly cheaper. The difference between the passes is that the second one can already exhibit a populated page table. For huge pages, as the number of entries is small, this difference is negligible. For small pages however, it becomes visible, where the second pass is 2.2x (sequential pool) to 3.1x (random pool) faster than the first one. The software-indirected memory still suffers from the translation and directory access costs. The write runtime of the private anonymous memory is also significantly faster than in the previous pass and equals the one of rewired memory. The reason for this is that the second pass, in contrast to the first one, does not suffer from page-faults anymore. Overall, we clearly see that rewiring combines the advantages of both base-lines while avoiding the drawbacks. Rewiring offers the flexibility of individual page re-usage while enabling direct access without a slowdown.

4.4 Costs of an Individual Page Fault

In order to understand the high costs of the first write of private anonymous memory and the advantages of using a page pool in more detail, we perform an experiment where we trigger a number of page faults and compute the average costs. We use a dataset of 2GB that is either backed by 1024 huge pages or by $524,288$ small pages. We simply loop over these pages and write to the first byte on each. Here, we can evaluate only two out of our three allocation types of Section 4.2. Firstly, we look at (a) (Private Anonymous Memory), which is the two-level memory mapping where the virtual pages are not yet backed by any anonymous physical pages. Thus, the first access will trigger a kernel request for a zeroed page. Again, we focus on huge pages due to the limited usability of small pages in this context. Secondly, we look at type (c) (Rewired Memory) where the physical pages come from our pool of initialized pages. Again, we divide between sequentially and randomly mapped pages and test both small and huge pages. Additionally, we look at both `MAP_SHARED` and `MAP_PRIVATE` options, as it influences the costs and both can be useful in certain situations. Note that we do not evaluate (b) (Software-Indirected Memory) in this context, as it is not connected to page faults in any way.

Table 1 shows the results. We repeated all measurements 100 times, each run performed in a fresh process. Let us focus on the 'per page' results. We see that the costs of faulting a page of private anonymous memory are significant with around $600\mu s$. In comparison with its direct counterpart, rewired memory using huge pages with `MAP_SHARED` option that faults in $710ns$, private anonymous memory is three orders of magnitude slower, i.e. this clearly shows that zeroing the page is the dominant part of the page fault, not setting up and inserting the page table entry, which is done by both methods. Therefore, we have to distinct between two types of page faults: *hard* and *soft* page faults. If the page is freshly claimed from the kernel, we have a hard page fault. If the page exists already, for instance in a file representing a pool, and only the page table entry must be inserted, we face a soft page fault. The expensive page cleaning is again confirmed when using the `MAP_PRIVATE` option, where a fresh anonymous page is used to resolve the COW. The difference between mapping the pooled pages sequentially and randomly is only visible for small pages, where in total $524,288$ pages are mapped. Here, a sequential mapping is up to 1.5x faster than mapping them individually, as only a single `vm_area_struct` is queried in contrast to hundreds of thousands.

Backing Type	Pooled Pages?	Page Size	Mapping Type	Amortized Time for page-fault [ns] per page	per KB
Private Anonymous Memory	no	huge	private	600 251	293
Rewired Memory	yes (sequential)	small	shared	686	172
			private	2 048	512
		huge	shared	710	< 1
			private	526 519	257
	yes (random)	small	shared	1 053	263
			private	2 727	682
		huge	shared	810	< 1
			private	528 910	258

Table 1: ***Cost of a single page-fault*** *for the allocation types (a) (Private Anonymous Memory) in comparison with (c) (Rewired Memory) from a page pool. We trigger page-faults for a dataset of 2GB and report the average costs amortized per page and per KB.*

The major takeaway message of this experiment is: the actual page fault costs are negligible if the physical page pointed to was already created before (soft page fault), which is the case when we use a user-managed page pool.

4.5 Costs of Rewiring Memory

So far, we have seen how memory allocation, access, and faulting behaves for rewired memory in comparison with the traditional approaches. Let us now see how expensive the actual rewiring of memory is. To answer this question, we perform the following experiment: we form chunks of size $2^x \cdot 4$KB and randomly shuffle the data at this granularity. The exponent x is varied[2] from 0 to 19.

For the traditional approach using private anonymous memory and `memcpy` shuffling implies physically copying the chunks into a separate, fresh array. For the rewired approach, we establish a new virtual memory area that is mapped. All shuffling can be done through rewiring virtual memory. Figure 6 shows the results for both small and huge pages for rewired memory. The rewired versions are additionally tested with manual population, i.e. directly after mapping a chunk of memory, we access the first byte of each page of the chunk to trigger the page fault.

We can see that the chunk size has a large impact on the runtime of the shuffling for rewired memory. For very high granularities (e.g. a chunk size of 4KB), the calls to `mmap` create a significant overhead ($1,953,125$ calls for 4KB chunks). Recall that for a single mapping of a contiguous virtual memory area to *one* physical offset in the file we only require a single `vm_area_struct` which is kept in a separate tree structure maintained by the kernel. For

[2]Notice that $2^{19} \cdot 4\text{KB} = 2\text{GB}$ is the largest chunk size fitting two times into the array.

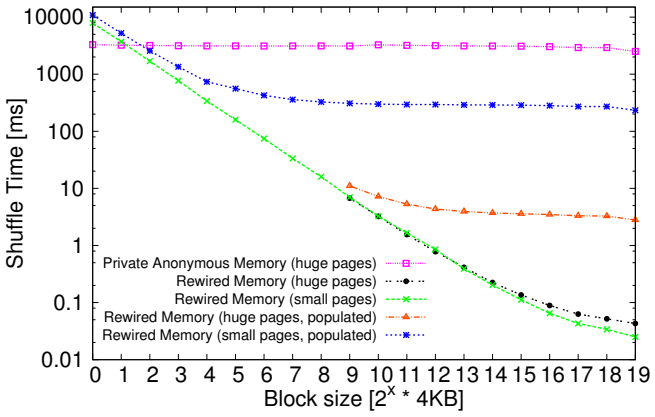


Figure 6: ***Time to shuffle*** *chunks of $2^x \cdot 4KB$ size. As the test array consists of $1,953,125$ chunks of $4KB$, we can shuffle at most chunks of size $2^{19} \cdot 4KB$. We compare the traditional shuffling using private anonymous memory and* `memcpy` *with rewired memory (with and without explicit page table population). For huge pages, the highest granularity we can test is $2^9 \cdot 4KB = 2MB$.*

the shuffling of chunks however, we need (in the worst case) one `vm_area_struct` per chunk. This overhead can be observed in Figure 6. We can see that for small chunk sizes of about 4KB the costs of rewiring are more expensive than the actual physical copy operation of private anonymous memory. However, as soon as we start increasing the size of the chunks, rewired memory clearly outperforms private anonymous memory. We can also observe that the page table population by triggering soft page faults is not for free and observable, at least for small pages.

In summary, when using reasonable large chunks sizes, rewiring is considerably cheaper than the actual copying of the data.

4.6 Impact of Access Pattern

Previously, we have seen that a page-wise software-indirection is a possible alternative to rewiring when implementing a pool-based memory allocation system. However, when it comes to accessing the memory, overhead must be factored in due to additional index translation and access of the auxiliary directory. In Figure 5 of Section 4.3 we already saw that sequentially reading through indirection is considerably more expensive than scanning a flat array. The question arises whether accessing memory through the software-indirection is generally more expensive than direct memory access. Thus, we now test the following access patterns on both rewired memory and a software-indirection. For all tests, page faults are already resolved.

We distinct the following five patterns, which we visualize in Figure 7: for *random, independent* pattern (7(a)), we access the memory at random places where the access index is generated individually. For the *sequential, independent* pattern (7(b)), we scan the memory from beginning to end where the access index is the iteration variable of the loop. For *random dependent* access (7(c)), we access the memory at random places where the access index is the result of the previous access. There is only one cycle in the dependency chain. For *sequential dependent* access (7(d)), we access the memory at sequential places where the access index is the result of the previous access. Again, there is only one cycle in the dependency chain. Finally, for *random, mixed* access (7(e)), we access the memory at random places. However, we interleave accesses based on a random number generator and accesses based on the result of the previous access.

For rewired memory, we work on a flat array `a` that can be accessed directly at index i via `a[i]`. In the case of software-indirected memory, we have a directory `a` where a chunk has the size of a huge page and translate each access at index i to to `a[i / pageSize][i % pageSize]`. For the patters of Figures 7(a) to 7(d), the number of accesses equals 100% of the data. For the pattern of Figure 7(e), the number of independent accesses equals 20% of the data, where each independent access is followed by four dependent ones.

Table 2 shows the results for all five patterns. We can see that as expected, the direct access offered by rewired memory is in any case faster than going through the indirection. However, the amount of difference depends on the type of access pattern. The highest difference we observe in the case of sequential access with factors of 1.88x for independent and 2.12x for dependent access. In contrast to that, when performing any type of random access, the difference is overall less, ranging from 1.07x for independent access to 1.53x for mixed access. This is due to expensive cache-misses triggered by the random accesses both for the direct and the indirect cases, that overshadow the impact of the access pattern. The overall message is: software-indirection might be a valid alternative in terms of flexibility, however, it certainly has a negative impact on the performance, depending on the type of access pattern. Thus, a software-indirection should be used with caution and only, if the subsequent access patterns are known. This is in general not the case. In contrast to that, rewiring offers an equal flexibility without any negative impact of the access performance.

Access Pattern	Software-Indirected [s]	Rewired [s]	Speedup
7(a) random, independent	14.03	13.16	1.07x
7(b) sequential, independent	1.58	0.84	1.88x
7(c) random, dependent	113.40	106.99	1.05x
7(d) sequential, dependent	6.60	3.11	2.12x
7(e) random, mixed	58.33	38.19	1.53x

Table 2: ***Comparison of access patterns*** *on both software-indirected memory and rewired memory when using huge pages. Both methods map sequentially into a pool of initialized pages.*

5. APPLICATIONS

In Section 2 we recapped virtual memory, in Section 3 we introduced rewiring, and in Section 4 we micro-benchmarked it. Now, we are in the position to demonstrate the concept on the basis of concrete applications. Instead of implementing the technique in a full-fledged database management system, we want to use rewiring inside isolated applications, that represent core components of basically every DBMS. By this, we are able to analyze the individual impact of rewiring more carefully than plugging it in at several layers and places in a complete system at once. Of course, seeing rewiring integrated in a full-fledged system is the final goal of this research. However, it goes far beyond the scope of a single 12-page paper and we leave it open for future work, see Section 6.

To showcase rewiring, we start with a rewired vector, that is a natural candidate for our technique as it requires both high flexibility and access performance. We then present the benefit of this data structure exemplary by embedding it into our state-of-the-art partitioning algorithm, leading to a significant improvement. Afterwards, we explore snapshotting which is used for high-frequency indexing and realtime OLAP. Further, other promising applications are briefly discussed in Section 6.

5.1 Rewired Vector

Let us start simple and explore a data structure that is not only present at various layers in database systems, but widely used in basically any kind of software: the vector, a resizable array. In a DBMS, a vector-like structure is the fundamental component of every storage layer, representing tables or columns that grow and shrink under inserts and deletes. Obviously, it is crucial for the

(a) Random Independent (b) Sequential Independent (c) Random Dependent (d) Sequential Dependent (e) Random Mixed

Figure 7: ***Visualization of Access Patterns on an Array****. The sequence numbers show the order of access.*

storage-structure to provide high access performance and low update cost at the same time. Traditionally, there is a tradeoff between the flexibility and adaptiveness of a data structure and its provided access times. For instance, a plain array grants us direct access via a static offset, but has a fixed size. In contrast to that, a linked list can grow and shrink dynamically, but suffers from slower access of individual entries. The widely used vector structure lies between these two extremes. It offers direct access into a consecutive virtual memory area but has the ability to grow and shrink dynamically. Let us see how state-of-the-art vectors offer these properties.

5.1.1 Classical Vectors

The vector is easy to handle for the user as it adjusts its size transparently based on the fillstate (similar to a linked list), but at the same time offers a high access performance as the underlying data is guaranteed to be stored in a consecutive memory region. However, a price needs to be paid if the data structure runs out of internal space and needs to grow. In that case, different *resizing policies* are possible: (1) Allocate a consecutive virtual memory area of twice the size, physically copy all data from the old to the new region, and free the old one. This is the standard policy of `std::vector` from STL of C++. (2) Increase the size of the virtual memory area using the `mremap` system call. (3) Realize the vector as a linked list of memory chunks that are hidden through a software-indirection. From our current point of view, all these techniques seem suboptimal. Policy (1) leads to performing unnecessary physical copies, (2) is not compatible with a pool of pages, and (3) disallows storing all entries in a single continuous virtual memory area. In contrast to that, our rewired vector avoids all these problems entirely.

5.1.2 Rewired Vector

For the *rewired vector*, in order to double the capacity of the structure, we do not physically copy any of the data that is already stored in the vector, thus avoiding the issue of policy (1). Instead, we map the first half of a fresh virtual memory area of twice the size to the physical pages containing the old data and the second half to unused physical pages in the pool. The old memory area is unmapped. Of course, the same concept is applied in the opposite direction for shrinking the data structure. This solves both the problems of policy (2) and (3) as we are able to reuse pooled pages freely and provide a single consecutive virtual memory area storing all the data at any time. The simplicity shows how natural rewiring fits to the problem: we map and unmap physical pages on demand without giving up the conveniences of direct memory access. All this is possible with less than 40 lines of code to setup a fresh rewired vector and around 20 lines for the insert function.

5.1.3 Experimental Evaluation

To evaluate the structure, we compare the rewired vector with representatives of the resizing policies (1), (2), and (3). The `std::vector` of STL[3] represents policy (1), the *mremap vector* represents policy (2), and the *software-indirected vector* represents policy (3). We create all vectors with an initial capacity of 2MB. All tested methods are backed by huge pages. We then insert 1 billion entries of 8B each into the vectors, leading to a total dataset size of 7.45GB. To see the detailed behavior of the vectors, we measure the time for every batch of $100,000$ consecutive inserts. Figure 8(a) shows the times of the individual batches over the entire insertion sequence, alongside with a zoom-in of 5 million inserts. Additionally, Figure 8(b) shows the accumulated insertion time over the entire insertion sequence. All structures double their capacity as soon as they run out of space. As expected, `std::vector` suffers from expensive physical copying steps every time it has to double its internal memory. Over the entire sequence, a significant amount of the runtime is spent purely on the reallocation process. The last doubling to 8GB after around 536 million insertions physically copies 4GB of data from the old to the new memory region. The remaining three techniques avoid physical copying and thus do not show any significant runtime spikes. Nevertheless, there are differences in the insertion times observable, when we look at the zoom-in of Figure 8(a): the mremap vector as well as `std::vector` suffer from hard page faults whereas the rewired vector only from soft ones. The software-indirected vector also exploits the pool and thus offers high insertion throughput. However, it is slightly throttled by checking if page boundaries are crossed.

Overall, rewired vector can insert the entries in the simplest possible way and thus shows the best accumulated runtime. It improves throughput by 150% over `std::vector` and by 40% over mremap vector. Even over the software-indirected vector, that plays in a lower league as it does not keep the data consecutively, the pure insertion improvement is still 20%.

Let us now see a concrete application of the vector in the database context: the partitioning of a dataset. This algorithm requires the flexibility of enlarging the partitions on the fly. Besides, the further efficient processing of the partitions requires the data to be stored consecutively. Both is offered by our rewired vector.

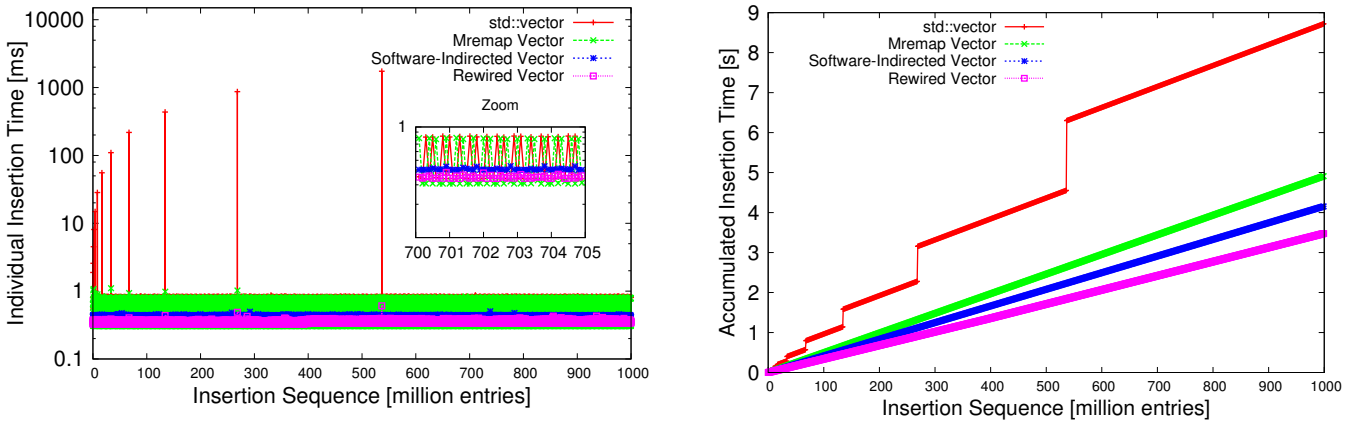
5.2 Rewired Partitioning & Sorting

Partitioning a dataset into k disjoint partitions based on a key is a fundamental task in data processing. It is widely used to divide work among threads or as an intermediate step in sorting [2], and join processing [3]. Despite of its simplicity, several optimizations can be applied to the base radix partitioning algorithm such as software-managed buffers [3, 14, 19], non-temporal streaming stores [3, 5, 14, 19], and clever placing of working variables [3, 14]. In recent previous work, we already thoroughly studied partitioning and the impact of the different techniques [18]. In this section we will take this as a baseline and explore how to create a rewired version of state-of-the-art partitioning, that manages to push the end-to-end performance even further.

5.2.1 Two-pass Partitioning

One of the most popular partitioning algorithms [3, 12, 18], coined *two-pass partitioning* in the following, aims at generating a consecutive and partitioned result area. The algorithm works as follows: Initially, we perform a first pass over the data to build a histogram. The histogram counts for each partition the number of entries that belong to it. Based on that, we can compute the start of each partition in the consecutive result area. Afterwards, we perform a second pass over the input data where we physically copy

[3]The `std::vector` test is written in C++, compiled using `g++`, and linked against our C-codebase compiled using `gcc`. When inserting the n elements, we call the C++ library *exactly once* passing a pointer to an array with the elements to insert to avoid overhead.



(a) Individual Insertion Times

(b) Accumulated Insertion Times

Figure 8: ***Insertion*** *of 1 billion random elements into a vector with an initial size of 2MB. We compare the* `std::vector` *of the STL, that physically copies the content into an area of twice the size when the capacity is reached with the mremap vector, that uses* `mremap` *to grow a private anonymous memory area. Further, we test a software-indirected vector as well as the rewired vector, that both claim pages from a page pool. All methods back their memory with huge pages.*

each entry to its destination partition. Many low-level optimizations may be applied to this base algorithm, see [18]. The major drawback of two-pass partitioning is implied by its name already: we need two complete passes over the input to perform the partitioning — costs that can be significant if the input is large and/or the table is wide in row-layout.

5.2.2 Chunked Partitioning

The second option, coined *chunked partitioning* in the following, partitions the dataset in a single pass without creating a histogram in the first place. Notice that chunked partitioning was also used in very recent work [12,14]. Instead of computing a histogram upfront to determine the exact partition sizes, we organize each partition as a list of chunks (respectively pages) that we create on demand. Each write to a chunk is preceded by a check for sufficient space in the current chunk. While chunked partitioning indeed partitions the input data in a single pass, it has a severe drawback — it does not create a consecutive result area, but only a list of chunks belonging to that partition. That list may lead to additional costs in further processing, as we will see in the evaluation.

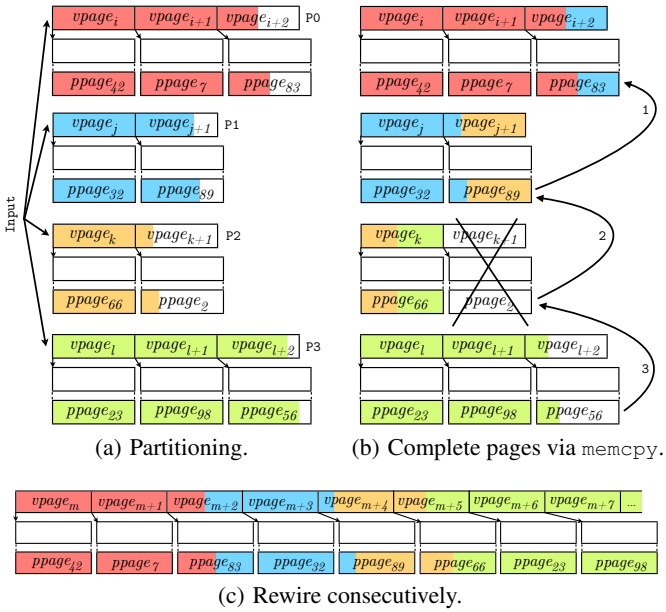
5.2.3 Mremap Partitioning

A third option that is positioned between the previously mentioned two approaches uses the `mremap` system call again, thus, we coin it *mremap partitioning*. Like chunked partitioning, it avoids the generation of a histogram pass. However, instead of forming the partitions out of manually linked chunks, we increase the size of the partitions adaptively page-wise by calling `mremap`. On one hand, this has the advantage that every partition is represented by a consecutive virtual memory area. On the other hand, it disables the ability of pooling. Furthermore, it is not possible to create a *single* consecutive virtual memory area over all partitions.

5.2.4 Rewired Partitioning

To overcome all these limitations, in the following, we propose an algorithm that combines the best properties of the previously mentioned techniques, while avoiding the disadvantages. We skip the generation of a histogram and thus of a complete pass while at the same time generating a perfectly consecutive result area.

We use chunked partitioning as the basis and modify it in a sense that each individual partition is represented using a rewired vector. In the partitioning phase, visualized in Figure 9(a), we add an entry to a partition by pushing it into the corresponding rewired vector. Each vector is configured with an initial capacity of a single page and increases its capacity not by doubling, as evaluated in Section 5.1, but by appending individual pages gathered from the pool. This ensures that the overhead in memory consumption remains small. Subsequent to the partitioning phase, we want to establish a single consecutive memory region containing the data of all partitions without any holes or gaps. This is done in two steps. First we have to fill up the last page of all partitions except the one of the very last partition. If the last page of the vector representing partition i has space left for k tuples, then we move k tuples from the end of the vector representing partition $i+1$ to vector i, if it contains a sufficient amount of entries. Figure 9(b) demonstrates the concept. The application of this algorithm might clear

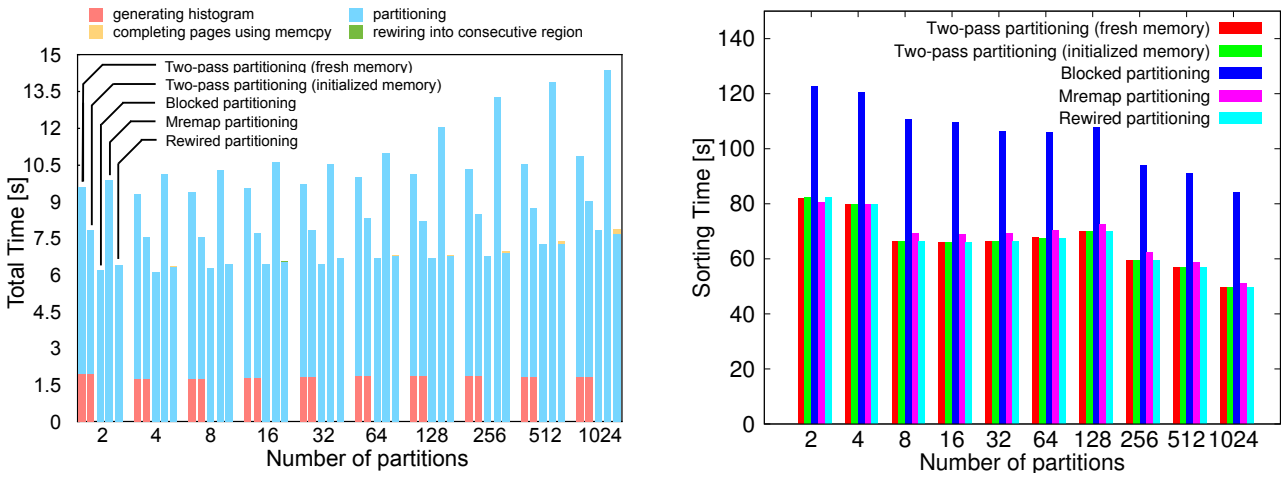


(a) Partitioning.

(b) Complete pages via `memcpy`.

(c) Rewire consecutively.

Figure 9: ***Rewired partitioning using chunks*** *partitions a dataset into a consecutive array* ***without a histogram generation pass****.*



(a) **Time to partition** the input.

(b) **Time to sort** the partitions locally.

Figure 10: ***Partitioning*** *an array of* 1 *billion entries out of place into* 2 *to* 1024 *partitions, which are then* ***sorted*** *in place. We divide the total partitioning time into generating the histogram, partitioning, completing the pages using* `memcpy`*, and rewiring into a consecutive region.*

pages entirely, as it can be seen in the example for $vpage_{k+1}$. In the second step, we are now able to rewire the remaining physical pages backing the rewired vectors into a single consecutive virtual memory area, as we show in Figure 9(c).

5.2.5 Experimental Evaluation

Let us now see how our new rewired partitioning performs in comparison to the traditional counterparts. We vary the number of requested partitions from 2 to 1024 in logarithmic steps and partition 1 billion entries of 8B each. The keys are uniformly and randomly distributed. We test two-pass partitioning generating a histogram (Section 5.2.1) both when partitioning into fresh and initialized memory. Furthermore, we evaluate classical chunked partitioning using a linked list (Section 5.2.2) as well as `mremap` partitioning, enlarging the partitions using the system call (Section 5.2.3). We compare these baselines agains our rewired partitioning (Section 5.2.4). In Figure 10(a), we display the end-to-end partitioning times achieved by the different algorithms. To get a more detailed view of the behavior, we additionally show the individual parts of the processing. As we can see, rewired partitioning, that outputs the result in form of a single consecutive memory area, offers basically the same runtime as chunked partitioning, that creates only a linked list of memory chunks. Rewired partitioning is significantly faster than two-pass partitioning due to the avoidance of the additional histogram generation. In comparison with two-pass partitioning into fresh memory, the improvement in throughput ranges from 49% for 2 partitions to 37% for 1024 partitions. Even when making the assumption of partitioning into initialized memory, the improvement still ranges from 22% to 14% for 2 respectively 1024 partitions. Interestingly, `mremap` partitioning is the slowest of all methods. In comparison, rewired partitioning offers an up to 83% higher throughput (1024 partitions). The relocation of virtual memory regions, that is performed in case an enlargement at the current place is not possible, turns out to be surprisingly expensive. For rewired partitioning, physically completing the last pages of the vectors and the subsequent rewiring into a consecutive area cause negligible costs. Even for 1024 partitions, where the last page of each partition is filled around 75%, this cost makes only 2.5% of the end-to-end time. The alerted reader might argue now that chunked partitioning is still a valid alternative to the rewired version. As we will see now, this depends heavily on how the produced partitions are further processed and whether this processing can be made aware of the list of chunks. Imaging the very common use-case of locally sorting the individual partitions to establish a globally sorted state using your favorite sorting algorithm. It is straight-forward to apply it onto the result of rewired partition, but it certainly has to be modified to apply it onto the chunked partitions. This can be (1) impossible if the algorithm is black-boxed or (2) very hard, depending on the type of algorithm. Extending the algorithm with a chunk-wise indirection is at least always possible if the code is accessible. However, this can have a tremendously negative effect, as demonstrated in Figure 10(b). The sorting time of chunked partitioning, working through the indirection, is significantly higher in all cases than the direct approach. Overall, only rewired partitioning manages to truly combine the best of both worlds: *flexibility* and *processing speed*.

5.3 Rewired Snapshotting

Previously, we have seen that the rewiring of memory is a very helpful tool to improve both data structures and data processing algorithms in terms of flexibility and performance. Another database application that heavily relies on these properties is snapshotting. A snapshot [1] is a read-only static view or copy of the database which can be used to serve long running analytical queries. It is transactionally consistent with the source database at snapshot creation. This technique is useful to create virtual memory snapshots which in turn can be used to separate long running OLAP queries and short running OLTP transactions. Researchers have proposed several strategies [11] to efficiently execute both query types concurrently ranging from using two different systems, to exploiting kernel features like COW. Systems like HyPer [8] exploit COW to snapshot transactional data lazily during query execution. While these systems are quite efficient, they rely on the operating system's fresh page allocation feature which is expensive as we learned in the previous sections. Consequently, we propose rewired snapshotting that completely eliminates page allocation overhead by utilising the pooling feature of rewired memory and compare it against the virtual memory aware snapshotting described as in HyPer [8].

5.3.1 Virtual Memory Aware Snapshotting

HyPer [8] uses the `fork` system call provided by the Linux kernel to implicitly snapshot transactional data. The forked process gets a copy of the parent process's page table, thereby a virtual

(a) Sequential-Independent Workload

# updates in-between snapshots ($\cdot 10^5$)	Update Throughput #updates per second ($\cdot 10^8$) implicit COW	rewired COW	Gain [%]
1000	3.24	3.71	14.50
100	2.41	3.67	52.28
10	2.19	3.70	68.94
1	1.07	2.10	96.26

(b) Random-Independent Workload

# updates in-between snapshots ($\cdot 10^5$)	Update Throughput #updates per second ($\cdot 10^5$) implicit COW	rewired COW	Gain [%]
1000	3160.00	3539.25	12.00
100	172.50	201.86	17.02
10	30.67	38.17	24.25
1	3.33	4.17	25.22

Table 4: ***Update throughput*** *when snapshotting.*

snapshot of the most recent transactional data. Each COW operation requests a fresh page from the kernel which is quite expensive as shown in Section 4.3. To see the effect of page allocation on COW, we micro-benchmark fork's implicit COW and compare it against the manual copying of a page. For this experiment, we allocate two separate arrays initialized with zeroes of size 1GB containing 8B entries. We back the first array by private anonymous huge pages and the second array by rewired memory. We update both arrays by modifying the first entry of each allocated page sequentially from start to end. The first array triggers an implicit COW for each modified page by allocating a fresh page. For the second array, we perform a so-called rewired COW (as explained in Section 5.3.2) to a pre-allocated pool page. We measure and compare the time for performing the implicit COW against the rewired COW.

Copy Strategy	Time [μs]
Implicit	594
Manual	442

Table 3: COW cost for implicit and manual page copying

Table 3 shows that implicit COW takes $594\ \mu s$ to copy and update entry for one page. On the other hand, rewired COW takes $442\ \mu s$ which is 25.5% less than its implicit counterpart. Obviously, this clearly motivates the need for page pooling during COW, as presented in the following.

5.3.2 Rewired Snapshotting

In this section, we present a pool-based snapshotting technique that avoids allocation overhead by reusing pages. We implement rewired snapshotting on top of `fork`, but avoid fresh page allocation during COW by maintaining a pool of pre-allocated huge pages which are file-backed as described is Section 3. To perform a COW for page p, we claim a page q from the pool and manually copy the content from p to q using `memcpy`. After this we rewire the claimed physical page q to the OLTP view. We will use view and snapshot interchangeably throughout this section. Figure 11 illustrates an example of how we perform rewired snapshotting step by step. We start with an OLTP view of the transactional data that can be updated directly without performing any COWs. When an OLAP query arrives at the end of `epoch 1`, we create a fresh snapshot by forking a new process and write protecting the OLTP view. Later, when the first write request arrives for a virtual page at the end of `epoch 2`, it triggers a segmentation fault. We handle these segfaults manually using signal handling API provided by the linux kernel. In the segfault handler, we perform rewired COW manually as explained earlier in this section. We take a fresh snapshot at the end of `epoch 5` by forking a new process from the process handling the OLTP queries. We attach all arriving OLAP queries to the most recent snapshot. At the end of `epoch 6`, after all OLAP queries using an old snapshot terminate, we garbage collect the unused pages of this snapshot for serving future COW requests as shown at the end of `epoch 7`.

The policy for rewired snapshotting is to snapshot transactional data after every k updates. If no OLAP query arrives after k updates, further snapshotting is delayed until the arrival of a next OLAP query. All OLAP queries arriving between update nk and $(n+1)k$ are attached to the existing snapshot taken after nk updates. A snapshot taken at any time t can be garbage collected after $2t$ seconds under the assumption that the longest running OLAP query terminates within t seconds. All write-protect permissions on transactional data are removed after the last OLAP query finishes, allowing the OLTP process to modify transactional data directly without performing any further COWs.

5.3.3 Experimental Evaluation

We evaluate the performance of rewired snapshotting by comparing its update throughput with virtual memory-aware snapshotting as implemented in HyPer [8]. Our test setup allocates two arrays each of size 1 GB containing 8 B entries, one backed by anonymous and other by pooled huge pages. We measure the update throughput for each of these arrays by varying the snapshotting frequency which refers to the number of updates applied to array before taking a fresh snapshot. Table 4 shows the throughput for sequential-independent and random-independent workload.

As expected, fresh page allocation cost has significant impact on performance of virtual memory aware snapshotting. With high snapshotting frequency which tends to perform more COWs, removing the page allocation overhead improves the update throughput by around 96% for sequential workload whereas by about 25% for random workload which is dominated by cache misses that overshadows the gain. Since a generic real-world workload lies somewhere between these two extremes, rewired snapshotting provides a substantial boost to OLTP update throughput especially if the snapshotting frequency is quite high and the OLAP workload sees a very recent view of the transactional data.

6. FUTURE WORK

We have seen some fundamental applications of rewiring. To demonstrate the impact of our technique, we picked the vector structure as well as the partitioning and sorting problem. All of these are fundamental use-cases both in database systems as well as generally in computer science. Further, we showed the impact on fork-based snapshotting, a naturally candidate to be enhanced by rewiring. Of course, there are several other promising applications where we believe that rewiring could be applied successfully. In future work, an entire DBMS can be build around or adapted to rewiring memory from the ground. Let us discuss these components and how they could be assembled in a single system.

On the storage layer, we can realize the managed tables and columns purely with rewired memory, as seen in the vector application. Thus, **inserts** and **deletes** are as cheap as possible to handle. Updates to a memory region can be collected on separate pages in a COW style and rewired into the base table from time to time (**differential updates**). The memory is claimed and released in a managed **pool**. Further, multiple different **virtual views** on the same physical data are possible. For instance, a view selecting rows on a table can directly map to only those physical pages, that contain qualifying rows. Connected to this, classical **data duplication** can be applied exploiting rewiring. If data is repeated, it is possible to store it physically only once and map to it from multiple places. Thus, it becomes possible to realize the inverse operation of

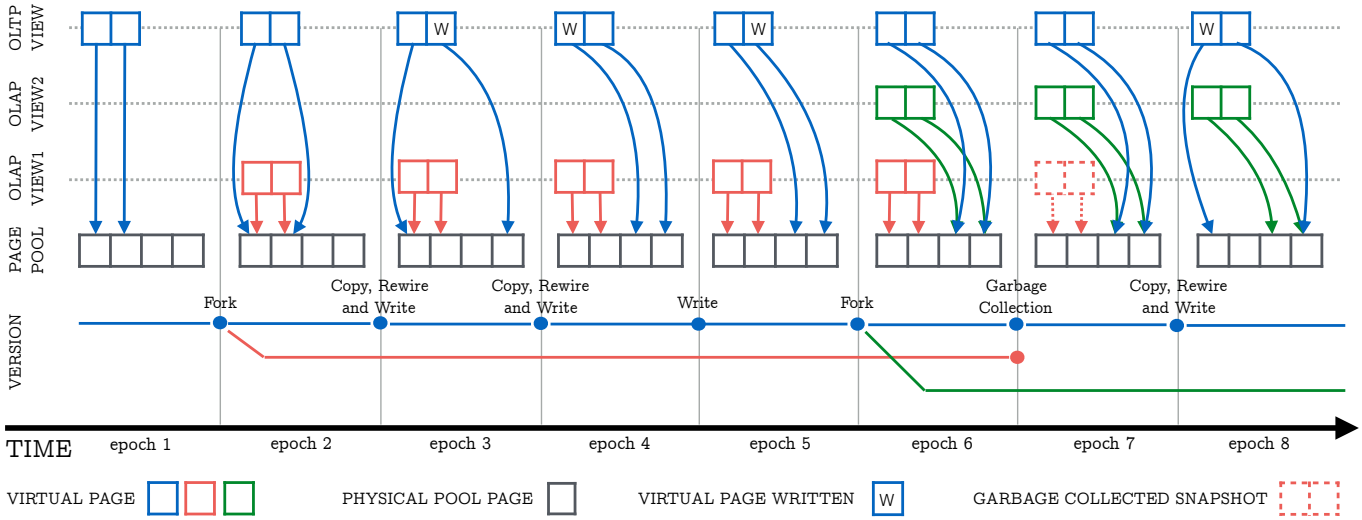


Figure 11: ***Rewired Snapshotting:*** *Throughput gain for pool-backed rewired snapshotting over implicit snapshotting using* `fork` *for different snapshotting frequencies (number or updates between snapshots).*

COW: merge-on-write (MOW)[4]. With respect to multi-socket machines, we can use rewiring to control the distribution of the memory across the difference memory regions for **NUMA-awareness**. Rewiring could be exploited to transparently swap NUMA-remote with NUMA-local pages, while the thread is working on the corresponding virtual memory area.

On the index layer, rewiring can be exploited at two places: firstly, there is a tremendous amount of read-optimized or even **read-only** index structures as for instance the CSS-tree [15] or FAST-tree [9]. Often, these structures are very hard to update. Massive physical copying or even a rebuild of the entire structure is necessary. Rewiring offers the chance to cheaply swap in and out parts of the structure in a transparent fashion. Secondly, rewiring can be used to speed up lookups in tree-based index structures by removing one level of indirection. It can also be used to enhance increasingly popular hashing methods [16]. For instance, the classical **extendible hashing** [6] uses a directory to indirect lookups into its buckets; this directory may be realized directly in form of the page table which completely removes the costs for that lookup.

Besides of that, the previously seen **snapshotting** mechanism can be used to enable fast concurrent processing of OLAP and OLTP queries. The seen rewired partitioning can be applied at many places, for instance to divide the data for **join-processing** [3, 17] or to split it into **horizontal partitions**. Obviously, rewiring can play a role at various places of a data management systems. Due to the enormous complexity, integrating these concepts in a full-fledged system will be a future project.

7. CONCLUSION

We have presented the basic toolbox to bridge the gap between the duality of *flexibility* and *access performance*. We reinterpreted the usage of memory mapped files and shared memory to introduce a convenient handle for physical memory in user-space. We evaluated the properties of the technique in depth under micro-benchmarks to learn the strengths and pitfalls and integrate the technique in a set of real-world applications. We showcased how easily rewiring improves the insertion throughput of a consecutive vector by 40% to 150%. By integrating the technique into state-of-the-art partitioning algorithms, we managed to improve the end-to-end throughput by 37% to almost 50%, while still offering a consecutive result. Finally, we presented a snapshotting mechanism exploiting rewired memory, that improves the update throughput by up to 96%.

Acknowledgements. We would like to thank Pankaj Khanchandani for contributions in early stages of this project. Further, we thank the anonymous reviewers for their helpful comments.

[4] `https://youtu.be/sLl-kyv-DCo`

8. REFERENCES

[1] M. E. Adiba et al. Database snapshots. In *VLDB 1980*, pages 86–91.
[2] V. Alvarez, F. M. Schuhknecht, J. Dittrich, et al. Main memory adaptive indexing for multi-core systems. In *DaMoN 2014*, pages 3:1–3:10.
[3] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *PVLDB*, 7(1):85–96, 2013.
[4] N. Douglas. User mode memory page allocation: A silver bullet for memory allocation? *CoRR*, abs/1105.1811, 2011.
[5] U. Drepper. What every programmer should know about memory, 2007.
[6] R. Fagin et al. Extendible hashing - A fast access method for dynamic files. *ACM Trans. Database Syst.*, 4(3):315–344, 1979.
[7] M. S. Johnstone and P. R. Wilson. The memory fragmentation problem: Solved? In *ISMM 1998*, pages 26–36.
[8] A. Kemper and T. Neumann. Hyper: A hybrid oltp & olap main memory database system based on virtual memory snapshots. In *ICDE 2011*, pages 195–206.
[9] C. Kim, J. Chhugani, N. Satish, et al. FAST: fast architecture sensitive tree search on modern cpus and gpus. In *SIGMOD 2010*, pages 339–350.
[10] T. Kissinger, B. Schlegel, D. Habich, and W. Lehner. Kiss-tree: smart latch-free in-memory indexing on modern architectures. In *DaMoN 2012*, pages 16–23.
[11] H. Mühe, A. Kemper, et al. How to efficiently snapshot transactional data: Hardware or software controlled? In *DaMoN 2011*, pages 17–26.
[12] I. Müller, P. Sanders, A. Lacurie, W. Lehner, and F. Färber. Cache-efficient aggregation: Hashing is sorting. In *SIGMOD 2015*, pages 1123–1136.
[13] E. Petraki, S. Idreos, and S. Manegold. Holistic indexing in main-memory column-stores. In *SIGMOD 2015*, pages 1153–1166.
[14] O. Polychroniou and K. A. Ross. A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort. In *SIGMOD 2014*, pages 755–766.
[15] J. Rao and K. A. Ross. Cache conscious indexing for decision-support in main memory. In *VLDB 1999*, pages 78–89.
[16] S. Richter, V. Alvarez, and J. Dittrich. A seven-dimensional analysis of hashing methods and its implications on query processing. *PVLDB*, 9(3):96–107, 2015.
[17] S. Schuh, X. Chen, and J. Dittrich. An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory. In *SIGMOD 2016*.
[18] F. M. Schuhknecht, P. Khanchandani, and J. Dittrich. On the Surprising Difficulty of Simple Things: the Case of Radix Partitioning. *PVLDB*, 8(9):934–937, 2015.
[19] J. Wassenberg and P. Sanders. Engineering a multi-core radix sort. In *Euro-Par 2011*, pages 160–169.